cleared with the CLEARI(<word_address>) in the instruction memory or with
CLEARD(<byte_address>) in the data memory.

The user can get an instruction or data memory dump with the command
DUMPIC or DUMPDC. The memory dumps are output to the files dump.ic and
dump.dc respectively.
Chapter VI.

FRISC-F Modules

In this chapter it is shown how the critical sections of the datapath can be efficiently implemented with differential current tree logic. The delays on the critical feed forward shifter and the ALU path are examined. Further, it is shown how the instruction decoder for the highly pipelined processor can be implemented. The circuits presented are for an implementation of FRISC-F as described in the architecture chapter. The FRISC-E version chips that have been routed with Merlyn-S are documented in [SL88,DL89,DCP89].

A. Datapath Module

Figure 71 shows a bit slice of the datapath. The datapath structure can be efficiently implemented with three level current tree logic. The shifter and feed forward logic for operands A and B can be implemented with only four input multiplexers per bit. The data signals are all routed on level one to minimize propagation delays. All of the registers can be directly implemented with data or master slave latch standard cells. The capability of merging two input gates with data or master slave latches is very effective in reducing power and propagation delays. If the signals on critical paths are connected to the level one input of these
latches the delay penalty is negligible. Only the loading at top of the tree is slightly
larger in a latch with combinational inputs increasing the propagation delay by
about 5ps. An XOR gate is merged with the ALU input latch OPB. The XOR gate
provides the optional inversion of operand B. A two input multiplexer is merged
with the ALU input latch OPA. This reduces the delays on the critical shift path.
The RES_EX pipeline register is part of the ALU logic since the XOR gate needed
for the sum generation can be merged with the output latch.

Most of the control signals do not have to be latched on the datapath. However, the signal invb and size signals need to be latched since they are sent
during the DE pipeline stage to control the ALU operand setup and the operand
shift and are needed one cycle later for the ALU flag generation. The ALU function
signals, f1 and f0, need to be latched on the datapath with master slave latches
at phase 1 since they must be stable until phase 4 of an ALU operation.

The critical timing of the datapath is associated with the ALU and the feed
forward and shifter path. At the beginning of phase 4 the RES_EX and RFA
registers are updated. The data from either of these registers might have to pass
through the feed forward multiplexer for operand A and pass through the shifter
into the ALU input latch OPA. The data must have arrived at the ALU input at the
beginning of phase 1. Hence, only 1ns is available for this path.

The ALU has 3ns to generate a result and to copy it into the output register
RES_EX. Since the ALU result might be fed forward immediately to the next ALU
operation a loop is generated. This loop is the most critical path because of the partitioning of the 32bit datapath into 8bit slices.

1. ALU Logic

The 32bit ALU is partitioned into four 8bit slices. The critical ALU operation is a 32bit add with carry passing through all four slices from the least significant to the most significant bit. The combinational ALU functions are not time critical since each output bit only depends upon the two input bits. Hence, only the adder part of the ALU is affected by the partitioning. Figure 72 shows the three different carry chain alternatives that were examined. The Ripple Carry chain can be implemented with the lowest number of gates but it is slow since the carry has to ripple through all 32bits from the Least Significant Bit (LSB) to the Most Significant Bit (MSB). The Carry Select ALU uses two carry chains. One chain generates the carries for an assumed input carry of one and the other for an assumed input carry of zero. The output carry signal is selected from the appropriate chain. Thus the fall through time through a slice is basically reduced to a multiplexer delay plus a receiver and driver delay if the carry chains had time to settle. The Carry Lookahead adder shown in Figure 72 uses carry lookahead over two bit blocks. The fanin limitation imposed by the three level current tree logic only allows the implementation of a two bit lookahead block with a reasonable number of gates. The resulting carry lookahead tree is three levels deep. The carry fall through time
Figure 72. ALU Carry Chain Implementation Options
of the carry lookahead adder is reduced to a gate delay plus a driver and receiver delay once the input carry generate and propagate signals have propagated in each slice to the top of the tree. The AND and OR gate at the top of the carry lookahead tree can be implemented with a single ANDOR current tree.

<table>
<thead>
<tr>
<th>ALU Type</th>
<th>Gates per Slice</th>
<th>Power per Slice</th>
<th>Delay 0ff/fanout</th>
<th>Delay 40ff/fanout</th>
<th>Delay 60ff/fanout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ripple Carry</td>
<td>32</td>
<td>324mW</td>
<td>4142ps</td>
<td>4456ps</td>
<td>4720ps</td>
</tr>
<tr>
<td>Carry Select</td>
<td>42</td>
<td>428mW</td>
<td>2304ps</td>
<td>2424ps</td>
<td>2550ps</td>
</tr>
<tr>
<td>Carry Select¹</td>
<td>43</td>
<td>434mW</td>
<td>2004ps</td>
<td>2134ps</td>
<td>2232ps</td>
</tr>
<tr>
<td>Carry Lookahead</td>
<td>74</td>
<td>682mW</td>
<td>2094ps</td>
<td>2220ps</td>
<td>2352ps</td>
</tr>
</tbody>
</table>

¹ Modified first slice with carry select over three and then five bits

**Table XXII. Comparison of 32Bit Adder Delays**

A summary of the ALU delays for a worst case 32bit add is shown in Table XXII. Table XXII shows also the number of core cells and the power dissipation for the different 8bit ALUs. The delays are based on FASTSIM simulations with Δt=2ps. The delays do not include the chip to chip interconnect delays (3x48ps). Different load capacitances per fanout were simulated to examine the load sensitivity of the different ALU types.

The Ripple Carry ALU is clearly the slowest but most power efficient ALU implementation. The Carry Select adder provides much better performance and dissipates only 32% more power than the simple Ripple Carry ALU. The Carry Lookahead ALU consumes more than twice the power of the Ripple Carry ALU
and provides only slightly better performance that the carry select adder. Hence, a trade off between power and performance clearly favors the Carry Select ALU. Further, the Ripple Carry and the Carry Select scheme have the same structure for each bit slice, hence the layout of these carry chains is much simpler than for the Carry Lookahead tree. The performance of the Carry Select ALU can be further improved by providing a special first slice with Carry Select over three bits and then five bits. This reduces the operand input to carry out delay from 9 gate delays to 5 gate delays saving about 300ps. Unfortunately this modification increases the fall through time and can thus only be applied to the first slice. Hence, this modification is costly, it requires the fabrication of a special first slice. Due to the low power dissipation and the high performance a Carry Select ALU was chosen for FRISC.

Figure 73 shows the 8bit Carry Select ALU. The ALU has only four functions: ADD, AND, XOR, and OR. The boolean functions are generated by the special cell ALUMAC. Subtraction is implemented by inverting operand B and the carry in of the first slice. This is equivalent to adding the two’s complement of operand B as shown below.

The output latch RES_EX is merged with the XOR gate needed for the generation of the sum. The carry in, C7, and the carry out, COUT, of the MSB have an additional fanout for the overflow and carry flag generation. multiplexers that generate the carry in for each bit are only enabled for additions.
Figure 73. Carry Select ALU Slice
claim:

\[ -(\text{opB+cin}) = \overline{\text{opB+cin}} \]

proof:

\[ \text{opB+cin+\overline{opB+cin} = } \sum_{i=0}^{n-2} (b_i + \overline{b_i}) \times 2^i - (b_{n-1} + \overline{b_{n-1}}) \times 2^{n-1} + (\text{cin+cin}) \]

\[ = \sum_{i=0}^{n-2} 1 \times 2^i - 1 \times 2^{n-1} + 1 \]

\[ = (2^{n-1} - 1 - 2^{n-1} + 1) = 0 \]

\[ \implies (\overline{\text{opB+cin}}) = -(\text{opB+cin}) \]

This saves a multiplexer for the routing of the ALUMAC signal to the output latch for AND, OR, and XOR functions. The carry out is forced low for boolean functions to simplify the overflow and carry flag logic.

The critical paths in the ALU are marked in Figure 73. On the first slice the path from operand A to the carry output to the next slice is critical. The carry in to carry out path is critical on slice two and three. On the last slice the carry in to MSB output is critical. The ALU has been simulated in SPICE since its performance is very critical. Table III shows the ALU delays obtained from SPICE simulations. The simulations indicate a worst case add delay of 2.792ns if the chip to chip carry signal interconnect is 8mm long. This indicates that the ALU operation can be performed in the available 3ns time slot if the clock skew can be controlled within ±100ps.
<table>
<thead>
<tr>
<th>Trutb Table: Cout, MSB of Sum, Overflow Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSB (opA)</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

\[ V = \text{Cin} \oplus \text{Cout} \]

9. Overflow Detection

Two's complement overflows during additions can be detected by comparing the signs of the input operands with the sum. An overflow occurred if both operands signs are equal but different from the sign of the sum. However, overflow detection can be implemented more efficiently. The truth table for the MSBs of the operands and the carry in and out of the MSB shows that an overflow can be detected with a two input XOR gate which compares the carry in and carry out of the MSB.
2. Shifter Logic

Figure 74 shows the multiplexer based implementation of the shifter. The shifter provides left and right shifts of one or two bit positions. The two bits that need to be exchanged between slices are transferred on two bit buses that connect adjacent slices. Tristate buses are used to reduce the number of I/O pads for the shifter.

FRISC-F supports arithmetic and extended shift operations. The two bits shifted in and out during extended precision shifts are stored in the C1 and C0 flags of the processor status word (PSW). The shifter reads these flags during phase 4 of the decode stage and phase 1 of the execution stage. The flags are updated at the beginning of phase 2 of the execution stage. Hence, the working copies of the shifter flags are associated with the EX stage!

After an exception the C1, C0 flags must be restored to the value they had when the last instruction completed. Instructions complete once they successfully passed the D2 stage and entered the DW stage. Hence, the shifter flags have to be restored to the value of the PSW_DW register after an exception. Figure 75 shows the implementation of the C1 and C0 shifter flags. If an instruction writes into the status word the write will occur at the beginning of the D1 stage. The write must modify the working copies of C1, C0 and the C1, C0 flags in the PSW_D1 register as well in order to restore the PSW to the correct value after an exception. Note that instruction that modify the PSW have a latency of one cycle.
Figure 76 shows the data movements during shift operations. The C1, C0 shifter flags are stored on the least significant slice. Extended shifts with C1, C0 are only supported for word size operands. Thus the C1 and C0 flags only need to be exchanged with slice 4. Sign extension is necessary on the most significant slice for arithmetic shifts. The most significant slice is slice 1 for byte operands, slice 2 for half word operands, and slice 4 for word operands. Shifter overflows must be detected for arithmetic left shifts. The shift overflow signal, shov, is delayed by one cycle and then ored with the ALU overflow signal. Shift or add overflows will cause an arithmetic trap unless the instruction specifically disables arithmetic traps (SIZ=11).
Figure 76. Data Movements for Shift Operations

The opa1 signal that determines whether a shift left or right is going to be performed is directly derived from an instruction field and is therefore available very early. Hence, the critical path is only associated with the data since it is only available at the beginning of phase 4. Only the short interconnect between adjacent datapath slices is on the critical path. On the long interconnect between the first slice and the last slice only the shifter flags C1 and C0 are transferred. This relaxes the timing constraints since the flags output signals are available at
the beginning of phase 2 and the input data is only required at the beginning of phase 2 of the next cycle.

<table>
<thead>
<tr>
<th></th>
<th>Fanout</th>
<th>$C_p$</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUX41H (feed forward)</td>
<td>5</td>
<td>400fF</td>
<td>168ps</td>
</tr>
<tr>
<td>TDR0 (tristate driver)</td>
<td>1</td>
<td>1pF</td>
<td>128ps</td>
</tr>
<tr>
<td>interconnect (max 4mm)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>to adjacent slice</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DR1 (receiver)</td>
<td>1</td>
<td>120fF</td>
<td>104ps</td>
</tr>
<tr>
<td>MUX21H (shift extend)</td>
<td>2</td>
<td>160fF</td>
<td>118ps</td>
</tr>
<tr>
<td>MUX41H (shifter)</td>
<td>1</td>
<td>80fF</td>
<td>114ps</td>
</tr>
<tr>
<td>DLMUX1h (OPA latch)</td>
<td>3</td>
<td>120fF</td>
<td>112ps</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td>768ps</td>
</tr>
</tbody>
</table>

**Table XXIII.** Delays on Critical Feed Forward and Shifter Path

Table XXIII shows the delays on the critical feed forward and shift path. Delays are based upon Fastsim simulations ($\Delta t=2$ps) and include estimated wiring capacitances. Despite the heavy loading the delay within the 1ns limit.

3. Negative and Zero Flag

Figure 77 shows the implementation of the Negative and Zero Flags. These status flags are set (SCC=1) or overwritten (SETPSW, MPSW) during phase two of the D1 stage. Like the C1 C0 flags they must be restored to the PSW_DW value after an exception. An additional complication arises from the fact that ALU
Figure 77. Negative and Zero Flags

instructions can have byte, half word, and word sized operands. Hence, Negative flag is generated on slice 1 for byte operands, on slice 2 for half word operands, and on slice 4 for word operands. Similarly the Zero flag is generated by the zero signal from slice 1 for byte operands, by a boolean OR of the zero signals from slice 1 and 2 for half word operands, and by a boolean OR of the
zero signals from all four slices for word sized operands. The Negative and Zero flags must finally be set on slice 4. A net with tristate drivers is used for the Negative flag transfer and a single ended wired OR connection is used for the Zero flag generation and transfer. The slice with the MSB will drive the net, N_, and all slice will copy the signal on this net into their copy of the Negative flag at the beginning of phase 2. The signal is latched to ensure that the signal sent onto the external net is stable. This is very important since single ended multi-driver nets have a tendency to ring and generate considerable switching noise.

Since all slices are identical a configuration of the slices is necessary. The two configuration inputs conf_1, conf_0 mark the physical slice as slice one, two, three or four. Each slice decodes whether it is the most significant slice and whether it should enable the zero output signal based upon the configuration and the operand size signals, size_1 and size_0. An efficient way for decoding the most significant slice, mssl, and the enable zero, ezero, signal is shown in Figure 77. The mssl and ezero signal must be delayed by one cycle, not counting stall cycles, before they are used for flag signal transfers since the operand size signals are received one cycle before the ALU operation completes to control the shifter.
Figure 78 shows the implementation of the Carry and Overflow flags. Like the other status flags they must be restored after an exception to the value in the DW stage. The Overflow flag is set if the shifter detected an overflow in the EX stage or the ALU detects an overflow at the end of the EX stage. The Overflow signals from the shifter and ALU are generated on different slices depending upon the operand size. The signals are transferred to slice 4 on nets with tristate drivers. The carry out of the MSB and the overflow signal from the ALU are again latched
before they are sent out of the chip. The Carry flag logic on slice 4 generates the carry in for slice 1. The Carry flag is clocked at the beginning of phase 1 to avoid a critical path for the carry in signal. This does not create a critical path for the carry flag generation since the output carry of the most significant slice is available at the end of phase 3 and 1ns is available for the transfer to slice 4. The carry input signal for the first slice is only enabled if the ecarry signal is set. The carry in must be inverted if operand B is inverted (invb=1). The ALU carry output signal must be inverted for the Carry flag if operand B was inverted since it actually represents a borrow. The invb1 signal is a copy of the invb signal delayed by one cycle. This delay is necessary because the invb is received on the datapath during the DE stage for the inversion of the carry in and ALU input latch OPB. The mssl, and ezero signals must also be delayed by one cycle before they are used for the ALU flag logic.

5. Program Counter Logic

Figure 79 shows the program counter (PC) logic. The program counter incrementer can be implemented with only two gates per bit. Low power gates can be used since the interconnect is short and the counter has a full 4ns cycle to generate the next address. The branch signal that controls the multiplexer is generated on slice 4 and transferred to all four slices and the instruction decoder. The input multiplexer for branch target address or next address select has been
Figure 19: Bit Slice of Program Counter

PC_I1  PC_I2  PC_DE  PC_EX  PC_D1  PC_D2  PC_DW


bra  5  6

p2x-stall


clip

p2x-stall

p2: clock phase 2
clip: carry in from previous bit slice
clip: carry out to next bit slice
clip: carry in of 64-bit slice
clip: select last 4 PC registers (Protected Mode)
clip: select last PC (PC_TW)
clip: clear PC_E1 to mask flushed instruction

bra: branch signal
stall: pipeline stall signal
po[1]: output to ALU
res_x[11]: output register of ALU
merged with the PC. 1 master slave latch. An AND gate has been merged with the PC_D1 master slave latch to provide a clear input. The program counter PC_D1 is cleared if a bubble is in the D pipeline stage.

6. Datapath Input/Output Signals

The datapath chip has 40 output, 66 input, and 66 power pads. A chip pad layout with 28x60 pads provides the necessary 172 pads. The die size is 4.4mm by 9.54mm with 80µm x 80µm pads. The estimated power dissipation of the datapath chip is 12W. The single ended drivers have their own VPP power rail to keep the switching noise away from the core with the standard cells. A high number of power connections is needed for the bipolar chip to avoid large voltage drops on the power rails and to reduce the power supply inductance. As a guide line two single ended drivers share one VPP and one VEE pad. Not all drivers can possibly switch at the same time. For example, the a_bus drivers switch during phase 4 and the d_buso drivers switch during phase 1. A list of the datapath I/O signals is shown below. By convention signal names with an '_' character mark single ended signals.
<table>
<thead>
<tr>
<th>SIGNAL</th>
<th>PINS</th>
<th>I/O</th>
<th>FROM/TO</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_bus</td>
<td>8</td>
<td>out</td>
<td>dc, ic</td>
<td>address bus</td>
</tr>
<tr>
<td>v_ia</td>
<td>1</td>
<td>out</td>
<td>ic</td>
<td>valid instruction address</td>
</tr>
<tr>
<td>v_dad</td>
<td>1</td>
<td>out</td>
<td>dc</td>
<td>valid data address</td>
</tr>
<tr>
<td>d_buso</td>
<td>8</td>
<td>out</td>
<td>dc</td>
<td>data output bus to data cache</td>
</tr>
<tr>
<td>e_int</td>
<td>1</td>
<td>out</td>
<td>id</td>
<td>arithmetic trap (Overflow)</td>
</tr>
<tr>
<td>clintal</td>
<td>2</td>
<td>out</td>
<td>dp4→dp1</td>
<td>carry in for LSB</td>
</tr>
<tr>
<td>cout</td>
<td>2</td>
<td>out</td>
<td>dp</td>
<td>ALU carry output to next slice</td>
</tr>
<tr>
<td>cpo</td>
<td>2</td>
<td>out</td>
<td>dp</td>
<td>PC counter carry output</td>
</tr>
<tr>
<td>u_o</td>
<td>1</td>
<td>out</td>
<td>sys</td>
<td>user output flag</td>
</tr>
<tr>
<td>N_</td>
<td>1</td>
<td>inout</td>
<td>dp</td>
<td>Negative flag</td>
</tr>
<tr>
<td>Z_</td>
<td>1</td>
<td>inout</td>
<td>dp</td>
<td>Zero flag</td>
</tr>
<tr>
<td>C_</td>
<td>1</td>
<td>inout</td>
<td>dp</td>
<td>Carry flag</td>
</tr>
<tr>
<td>V_</td>
<td>1</td>
<td>inout</td>
<td>dp</td>
<td>Overflow flag</td>
</tr>
<tr>
<td>bRa</td>
<td>1</td>
<td>inout</td>
<td>dp,id</td>
<td>branch signal</td>
</tr>
<tr>
<td>bmlD1</td>
<td>2</td>
<td>inout</td>
<td>dp</td>
<td>shift bit from to previous slice</td>
</tr>
<tr>
<td>bm2b0</td>
<td>2</td>
<td>inout</td>
<td>dp</td>
<td>shift bit from to previous slice</td>
</tr>
<tr>
<td>bpb6</td>
<td>2</td>
<td>inout</td>
<td>dp</td>
<td>shift bit from to next slice</td>
</tr>
<tr>
<td>bpb7</td>
<td>2</td>
<td>inout</td>
<td>dp</td>
<td>shift bit from to next slice</td>
</tr>
<tr>
<td>d_busi</td>
<td>8</td>
<td>in</td>
<td>dc</td>
<td>data input bus from data cache</td>
</tr>
<tr>
<td>lmem</td>
<td>8</td>
<td>in</td>
<td>id</td>
<td>immediate data input</td>
</tr>
<tr>
<td>ld_E1</td>
<td>1</td>
<td>in</td>
<td>id</td>
<td>load imm into dp4, dp3</td>
</tr>
<tr>
<td>rf_addr</td>
<td>5</td>
<td>in</td>
<td>id</td>
<td>register file address</td>
</tr>
<tr>
<td>w_ff</td>
<td>1</td>
<td>in</td>
<td>id</td>
<td>write register file</td>
</tr>
<tr>
<td>w_din</td>
<td>1</td>
<td>in</td>
<td>id</td>
<td>select DIN register as if input</td>
</tr>
<tr>
<td>w_dout</td>
<td>1</td>
<td>in</td>
<td>id</td>
<td>write data input register DIN</td>
</tr>
<tr>
<td>m_by_a</td>
<td>1</td>
<td>in</td>
<td>id</td>
<td>bypass memory for operand A</td>
</tr>
<tr>
<td>m_by_b</td>
<td>1</td>
<td>in</td>
<td>id</td>
<td>bypass memory for operand B</td>
</tr>
<tr>
<td>ff_ra</td>
<td>2</td>
<td>in</td>
<td>id</td>
<td>feed forward select for op A</td>
</tr>
<tr>
<td>ff_rz</td>
<td>2</td>
<td>in</td>
<td>id</td>
<td>feed forward select for op B</td>
</tr>
<tr>
<td>op_a</td>
<td>2</td>
<td>in</td>
<td>id</td>
<td>ALU operand A select</td>
</tr>
<tr>
<td>op_b</td>
<td>1</td>
<td>in</td>
<td>id</td>
<td>ALU operand B select</td>
</tr>
<tr>
<td>cin</td>
<td>2</td>
<td>in</td>
<td>dp</td>
<td>ALU carry input</td>
</tr>
<tr>
<td>cpi</td>
<td>2</td>
<td>in</td>
<td>dp</td>
<td>PC counter carry input</td>
</tr>
<tr>
<td>sl_e</td>
<td>1</td>
<td>in</td>
<td>id</td>
<td>select shifter as op A</td>
</tr>
<tr>
<td>sh_ex</td>
<td>1</td>
<td>in</td>
<td>id</td>
<td>extended shift</td>
</tr>
<tr>
<td>inv_b</td>
<td>1</td>
<td>in</td>
<td>id</td>
<td>invert operand B and carry</td>
</tr>
<tr>
<td>alu_op</td>
<td>2</td>
<td>in</td>
<td>id</td>
<td>ALU function select</td>
</tr>
<tr>
<td>size</td>
<td>2</td>
<td>in</td>
<td>id</td>
<td>ALU operand size select</td>
</tr>
<tr>
<td>e_carry</td>
<td>1</td>
<td>in</td>
<td>id</td>
<td>enable Carry for ALU add</td>
</tr>
<tr>
<td>w_addr</td>
<td>1</td>
<td>in</td>
<td>id</td>
<td>write ALU output into ADDR</td>
</tr>
<tr>
<td>w_paw</td>
<td>1</td>
<td>in</td>
<td>id</td>
<td>write ALU output into PSW_D1</td>
</tr>
<tr>
<td>u_i</td>
<td>1</td>
<td>in</td>
<td>id</td>
<td>user input flag</td>
</tr>
<tr>
<td>s_cc</td>
<td>1</td>
<td>in</td>
<td>id</td>
<td>set condition code in PSW_D1</td>
</tr>
<tr>
<td>c_c</td>
<td>4</td>
<td>in</td>
<td>id</td>
<td>condition code</td>
</tr>
<tr>
<td>ecx</td>
<td>1</td>
<td>in</td>
<td>id</td>
<td>exception signal</td>
</tr>
<tr>
<td>pc_rel</td>
<td>1</td>
<td>in</td>
<td>id</td>
<td>program counter select</td>
</tr>
<tr>
<td>cli_pc</td>
<td>1</td>
<td>in</td>
<td>id</td>
<td>clear program counter PC_D1</td>
</tr>
<tr>
<td>pc_lock</td>
<td>1</td>
<td>in</td>
<td>id</td>
<td>lock program counters</td>
</tr>
<tr>
<td>hold</td>
<td>1</td>
<td>in</td>
<td>id</td>
<td>tristate external output signals</td>
</tr>
<tr>
<td>stall_</td>
<td>1</td>
<td>in</td>
<td>id</td>
<td>pipeline stall</td>
</tr>
<tr>
<td>clk</td>
<td>2</td>
<td>in</td>
<td>id</td>
<td>clock signal</td>
</tr>
<tr>
<td>run_stop</td>
<td>1</td>
<td>in</td>
<td>id</td>
<td>phase generator startup signal</td>
</tr>
<tr>
<td>conf_</td>
<td>2</td>
<td>in</td>
<td>id</td>
<td>slice configuration signal</td>
</tr>
<tr>
<td>VPP</td>
<td>24</td>
<td>power</td>
<td>0V</td>
<td></td>
</tr>
<tr>
<td>VCC</td>
<td>18</td>
<td>power</td>
<td>0V</td>
<td></td>
</tr>
<tr>
<td>VEE</td>
<td>24</td>
<td>power</td>
<td>-5V</td>
<td></td>
</tr>
</tbody>
</table>

10. Datapath Signals
The instruction decoder decodes the instruction and generates the necessary datapath and memory control signals in the appropriate pipeline stage. The pipeline controller can not be designed effectively as a state machine since FRISC-F has seven pipeline stages and each of them can be either full or empty. A state machine with 128 states would be hard to design and very difficult to implement with a logic family that has a maximum fanin of three. However,
controllers for highly pipelined machines can be designed effectively if the controller itself is implemented as a pipeline with one section per pipeline stage.

1. Pipeline Controllers

Each stage of the pipeline controller controls the activities associated with the corresponding pipeline stage on the datapath. Since the activities performed in a pipeline stage only affect a small part of the hardware, the division of the controller into stages is similar to a divide and conquer design approach.

Figure 80 shows a general controller stage for an elastic pipeline. Each stage reads an input vector from the previous stage and generates an output vector for the next stage. A valid flag is associated with each input and output vector that indicates whether the vector data is valid. The output vector is also cleared if the valid bit is not set to mark it as a NOOP. Each stage generates the control signals for the corresponding datapath pipeline stage and can receive status signals from the datapath. For example, exception signals can be received from the datapath.

If a pipeline stage needs a shared resource, a resource request must be sent to a global resource arbiter. If the resources are available, the pipeline stage will receive the resource grant signals and advance unless a global pipeline stall is pending or the next pipeline stage is blocking. If the resource is not available because it has already been checked out or has not yet been generated (data
dependency) the pipeline stage is blocked and all the preceding pipeline stages will be blocked up to the first bubble or empty stage. If multiple stages compete for a resource the later stages of the pipeline controller will get higher priorities such that they can advance and finally complete. This resolves data dependencies and avoids deadlocks. The pipeline is elastic since even if a particular stage is blocked other stages can advance. This leads to the dynamic generation and deletion of pipeline bubbles. Each pipeline controller stage can abort itself or be flushed by an external signal.

2. FRISC-F Pipeline Stage Controllers

FRISC-F does not use an elastic pipeline since too many control signals would be required for its implementation. Further, the exception handling for an elastic pipeline would be more complex because of the dynamically generated bubbles.

Figure 81 shows the structure of the FRISC-F instruction decoder. The chip contains seven pipeline stage controllers, a simple two bit CPU state machine, and a trap encoder. The controllers for the l1 and l2 pipeline stages consist only of the valid flags. The DE stage receives a 32bit instruction and generates most of the datapath control signals. The DE stage will cause a global stall if the hit_i signal from the instruction cache is low because of a cache miss at the beginning of phase 2. This has an important implication. The instruction in the DE stage and
Figure 81. Instruction Decoder
hence the control signals can be invalid if the stall signal is active! The instruction register will be reclocked at the beginning of phase until a valid instruction has been read in unless the valid flag of the DE stage (v_de) is low marking a bubble. If the trap encoder asserts the trap and flush signals to indicate that an exception is pending the DE stage will force the input vector to be equivalent to a JUMP always trap_vector. The DE stage extracts the source A and source B register file addresses from the appropriate instruction fields and puts them on the srca and srcb buses. The source B address is sent to the register file during phase 1 and the source A address is sent during phase 2. The two register file addresses are also read in by the EX, D1, D2, and DW controllers for address tag comparisons.

While the DE stage reads in a 32bit input vector its output vector can be compressed. The output vector contains a 5bit destination address field with a valid flag that indicates whether the ALU result has to be written into the register file in the last pipeline stage. The valid flag also indicates whether the ALU result can be fed forward if it matches either the srca or srcb address. As the instruction advances in the pipeline the destination field becomes the address tag for the associated ALU result register. The output vector further contains bits that show whether the instruction is a LOAD or a STORE instruction for the DW stage. In addition, an arithmetic trap enable bit, a software trap enable bit, an enable clear protected mode, and a set protected mode information bit must be passed on to later stages.
The decoding of the instruction is not time critical because of the low level of encoding. Several fields like OPA, INV, SRCA, and IMM can be directly extracted from the instruction. The DE stage controller has between 2-4ns to decode the instruction. Since the opcode field is only 5bits wide and the gate delays are in the 100-200ps range this a safe timing constraint.

The EX stage controller generates the 3bit input/output control vector, io_cnt, and the data write signal, w_. It further generates the write data output register signal, w_dout, the write address register, w_addr, the write processor status word, w_psw, and clear_pc. The EX stage can flush the DE, I2, and I1 pipeline controller stages if the EXE field in the branch instruction was not set to 11. If a branch condition was matched the datapath assert the bra_ signal and the EX controller will flush the appropriate pipeline stages. If the valid destination address tag bit is set the controller generates the feed forward ALU result signals, ff_res_ex_a and ff_res_ex_b. These signals are set high if the destination address tag matches srca or srcb address respectively.

The D1 controller stage receives the arithmetic trap signal from the datapath and will set the at_d1_o bit in the output vector if the input vector had the enable arithmetic bit set. Like the EX stage the D1 stage controller generates feed forward signals ff_res_d1_a and ff_res_d1_b.

The D2 controller stage generates the arithmetic trap (atrap) and the software trap (strap) input signals for the trap encoder. The arithmetic trap signal
is activated in the current cycle to prevent the associated instruction from completing whereas the software trap signal is activated late for a trap in the next cycle to allow the instruction to complete and prevent a restart of the instruction. A restart of instructions that generate software traps would cause an infinite loop! Further, the D2 controller generates the set or clear protected mode signals for the CPU state machine. The state machine will respond to these signals only if the instruction can complete, no pipeline flush occurs. The trap signal has priority over the clear_pm signal. If the instruction in the D2 stage has a valid destination address the RES_D2 will be written into the register file at the beginning of the next cycle. If a load instruction is in the D2 stage the ld_in_dw signal is sent to the DW stage to signal that an empty write slot is available to write the data input register D_IN into the register file. Like the EX and D1 stages the D1 controller generates feed forward signals, ff_res_d2_a and ff_res_d2_b. The feed forward signals from the EX, D1, and D2 stages are sent to two priority decoders. The feed forward signal from the stage with the most recent data has priority!

The DW controller stage generates the stall_d signal if the hit_d signal is low at the beginning of phase 2 and a LOAD or STORE instruction has entered the DW stage. It further generates the write register file signal, w_rf, and the sel_d signal which controls whether the RES_D2 pipeline register or the data input register D_IN is transferred to the register file input port. The data input register address tag (DTAG) is only overwritten if another load instruction is entering the
stage. The DTAG is, however, invalidated if the corresponding register is overwritten by the instruction in the D2 stage or a load instruction is in the D2 stage and the D_IN register is written into the register file. The DTAG can not be flushed if the LOAD instruction that has set has completed the DW stage. If the DTAG is valid and the address matches the srca or srcb the memory bypass signals m_by_a and m_by_b are generated.

The whole instruction pipeline is stalled on instruction or data cache misses (stalli, stallid) or if the external halt signal is applied. No stalls due to internal resource conflicts occur in FRISC-F.

3. Instruction Decoder Input/Output Signals

The instruction decoder chip has 62 output, 46 input, and 64 power pads. A chip pad layout with 28x60 pads provides the necessary 172 pads. The die size is 4.4mm by 9.54mm with 80μm×80μm pads. The estimated power dissipation for the instruction decoder chip is 10W. The single ended drivers have their own VPP power rail to keep the switching noise away from the core with the standard cells. A large number of power connections is needed for the bipolar chip to avoid large voltage drops on the power rails and to reduce the power supply inductance. As a guide line two single ended drivers share one VPP and one VEE pad. Note that not all drivers can switch at the same time. A list of the datapath I/O signals is shown below. By convention signal names with an '_' character mark single ended
<table>
<thead>
<tr>
<th>SIGNAL</th>
<th>PINS</th>
<th>I/O</th>
<th>FROM/TO</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>exc</td>
<td>1</td>
<td>out</td>
<td>dp,ic,dc</td>
<td>exception signal</td>
</tr>
<tr>
<td>m_stall</td>
<td>1</td>
<td>out</td>
<td>ic,dc</td>
<td>pipeline stall (memory)</td>
</tr>
<tr>
<td>stall</td>
<td>1</td>
<td>out</td>
<td>dp</td>
<td>pipeline stall (datapath)</td>
</tr>
<tr>
<td>io_cnt</td>
<td>3</td>
<td>out</td>
<td>ic,dc</td>
<td>memory access control</td>
</tr>
<tr>
<td>w_</td>
<td>1</td>
<td>out</td>
<td>dc</td>
<td>write data cache</td>
</tr>
<tr>
<td>hold_</td>
<td>1</td>
<td>out</td>
<td>ic,dc</td>
<td>hold output signals tristate (low)</td>
</tr>
<tr>
<td>pc_sel</td>
<td>1</td>
<td>out</td>
<td>dp</td>
<td>select PC DE/PC DW signal</td>
</tr>
<tr>
<td>clr_pc</td>
<td>1</td>
<td>out</td>
<td>dp</td>
<td>clear PC_D1 to mark it as flushed</td>
</tr>
<tr>
<td>pc_lock</td>
<td>1</td>
<td>out</td>
<td>dp</td>
<td>lock last four PC registers</td>
</tr>
<tr>
<td>s_cc</td>
<td>1</td>
<td>out</td>
<td>dp</td>
<td>set condition code</td>
</tr>
<tr>
<td>c_c</td>
<td>4</td>
<td>out</td>
<td>dp</td>
<td>condition code</td>
</tr>
<tr>
<td>p_m</td>
<td>1</td>
<td>out</td>
<td>dp</td>
<td>protected mode signal</td>
</tr>
<tr>
<td>e_carry</td>
<td>1</td>
<td>out</td>
<td>dp</td>
<td>enable carry for ALU add</td>
</tr>
<tr>
<td>w_psw</td>
<td>1</td>
<td>out</td>
<td>dp</td>
<td>write ALU result into PSW_D1</td>
</tr>
<tr>
<td>w_addr</td>
<td>1</td>
<td>out</td>
<td>dp</td>
<td>write ALU result into ADDR</td>
</tr>
<tr>
<td>sh_</td>
<td>1</td>
<td>out</td>
<td>dp</td>
<td>select shifter as ALU op A</td>
</tr>
<tr>
<td>sh_ex</td>
<td>1</td>
<td>out</td>
<td>dp</td>
<td>extended precision shift</td>
</tr>
<tr>
<td>alu_op</td>
<td>2</td>
<td>out</td>
<td>dp</td>
<td>ALU function select</td>
</tr>
<tr>
<td>size</td>
<td>2</td>
<td>out</td>
<td>dp</td>
<td>ALU operand size</td>
</tr>
<tr>
<td>op_a</td>
<td>1</td>
<td>out</td>
<td>dp</td>
<td>operand A select</td>
</tr>
<tr>
<td>op_b</td>
<td>1</td>
<td>out</td>
<td>dp</td>
<td>operand B select (sel imm)</td>
</tr>
<tr>
<td>inv_b</td>
<td>1</td>
<td>out</td>
<td>dp</td>
<td>invert operand B and carry</td>
</tr>
<tr>
<td>imm</td>
<td>16</td>
<td>out</td>
<td>dp</td>
<td>immediate constant, trap vector</td>
</tr>
<tr>
<td>ld_hi</td>
<td>1</td>
<td>out</td>
<td>dp</td>
<td>load imm into upper 16bits</td>
</tr>
<tr>
<td>rf_addr</td>
<td>5</td>
<td>out</td>
<td>dp</td>
<td>register-file address</td>
</tr>
<tr>
<td>m_by_a</td>
<td>1</td>
<td>out</td>
<td>dp</td>
<td>memory bypass for operand A</td>
</tr>
<tr>
<td>m_by_b</td>
<td>1</td>
<td>out</td>
<td>dp</td>
<td>memory bypass for operand B</td>
</tr>
<tr>
<td>ff_ra</td>
<td>2</td>
<td>out</td>
<td>dp</td>
<td>feed forward select op A</td>
</tr>
<tr>
<td>ff_kb</td>
<td>2</td>
<td>out</td>
<td>dp</td>
<td>feed forward select op B</td>
</tr>
<tr>
<td>w_rf</td>
<td>1</td>
<td>out</td>
<td>dp</td>
<td>write register file</td>
</tr>
<tr>
<td>sel_d</td>
<td>1</td>
<td>out</td>
<td>dp</td>
<td>select DIN as rf input</td>
</tr>
<tr>
<td>w_din</td>
<td>1</td>
<td>out</td>
<td>dp</td>
<td>copy dbus1 into DIN</td>
</tr>
<tr>
<td>w_dout</td>
<td>1</td>
<td>out</td>
<td>dp</td>
<td>write OUT_D1</td>
</tr>
<tr>
<td>i_bus</td>
<td>32</td>
<td>in</td>
<td>ic</td>
<td>instruction bus</td>
</tr>
<tr>
<td>hit_i</td>
<td>1</td>
<td>in</td>
<td>ic</td>
<td>instruction cache hit</td>
</tr>
<tr>
<td>hit_d</td>
<td>1</td>
<td>in</td>
<td>dc</td>
<td>data cache hit</td>
</tr>
<tr>
<td>reset</td>
<td>1</td>
<td>in</td>
<td>sys</td>
<td>reset signal</td>
</tr>
<tr>
<td>error</td>
<td>1</td>
<td>in</td>
<td>sys</td>
<td>error trap signal</td>
</tr>
<tr>
<td>trap_i</td>
<td>1</td>
<td>in</td>
<td>ic</td>
<td>instruction trap, page fault</td>
</tr>
<tr>
<td>trap_d</td>
<td>1</td>
<td>in</td>
<td>dc</td>
<td>data trap, page fault</td>
</tr>
<tr>
<td>a_t</td>
<td>1</td>
<td>in</td>
<td>dp</td>
<td>arithmetic trap</td>
</tr>
<tr>
<td>int_</td>
<td>1</td>
<td>in</td>
<td>sys</td>
<td>interrupt signal</td>
</tr>
<tr>
<td>e_int</td>
<td>1</td>
<td>in</td>
<td>dp</td>
<td>interrupt enable signal</td>
</tr>
<tr>
<td>halt_</td>
<td>1</td>
<td>in</td>
<td>dp</td>
<td>processor halt signal</td>
</tr>
<tr>
<td>bra_</td>
<td>1</td>
<td>in</td>
<td>dp</td>
<td>branch signal</td>
</tr>
<tr>
<td>clk_</td>
<td>2</td>
<td>in</td>
<td>sys</td>
<td>clock signal</td>
</tr>
<tr>
<td>run_stop</td>
<td>1</td>
<td>in</td>
<td>sys</td>
<td>phase generator startup signal</td>
</tr>
<tr>
<td>VPo</td>
<td>24</td>
<td></td>
<td>power</td>
<td>0V</td>
</tr>
<tr>
<td>VCC</td>
<td>16</td>
<td></td>
<td>power</td>
<td>0V</td>
</tr>
<tr>
<td>VEE</td>
<td>24</td>
<td></td>
<td>power</td>
<td>-5V</td>
</tr>
</tbody>
</table>

11. Instruction Decoder Signals
signals

Figure 82. 400mil by 400mil Fabrication Reticle

The instruction decoder chip and the datapath chip can fit on a single step and repeat reticle as shown in Figure 82. Thus both FRISC-F dies could be fabricated in a single run.

While the decoding of the instruction is not time critical the signal distribution from the instruction decoder to all four datapath slice is critical. Figure 83 shows a worst case for the signal distribution. The FRISC-F timing allocates 1ns for signal distribution. The interconnect from the instruction decoder to the last datapath slice can be 35mm long for a worst case pad placement. Further, the signal received
Figure 83. Signal Distribution

on the datapath might have to run all the way across the chip (4mm run). Table XXIV shows the delays on the signal distribution path. The delays are extracted from FASTSIM simulations with $\Delta t = 2$ps. The delays are for a high to low signal transition since single ended drivers and receivers have larger propagation
<table>
<thead>
<tr>
<th>Gate</th>
<th>Fanout</th>
<th>C_r</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLMUX1H (latch on ID)</td>
<td>1</td>
<td>160ff</td>
<td>220ps</td>
</tr>
<tr>
<td>SER1 (single ended driver)</td>
<td>1</td>
<td>1pF</td>
<td>234ps</td>
</tr>
<tr>
<td>max interconnect (35mm)</td>
<td></td>
<td></td>
<td>228ps</td>
</tr>
<tr>
<td>SER1 (receiver on DP 4)</td>
<td>8</td>
<td>640ff</td>
<td>146ps</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td>828ps</td>
</tr>
</tbody>
</table>

**Table XXIV.** Instruction Decoder Signal Distribution Delays

delays for falling edges. The total delay on the signal distribution path is 828ps.
Chapter VII.

Computer Aided Design Tools

A lot of effort went into the development of a working CAD tool set for differential bipolar logic design. Since the Advanced Bipolar process was still in the development stage, the design rules were still changing. Hence, the physical layout had to be generated by automated place and route software to reduce the amount of work required for updating layouts. A standard cell layout strategy was finally chosen. A gate array implementation was not attempted since gate arrays have a lower circuit densities and hence lower performance. Schematic capture was needed since the complexity of the design required documentation in the form of schematics.

The differential current tree logic family has several restrictions and features that are not supported by most CAD tools since they are mainly targeted for CMOS designs. The differential current tree logic has signals with three different logic offset levels. Most standard cells have input level restrictions. Level violations cause saturation of bipolar devices in the current tree and will cause circuit failure. Therefore, level violations need to be detected as early as possible. Further, differential signals can be inverted simply by twisting the connection of the (true,false) signal pair at any input or output port of a standard cell. Thus, the designer must be able to indicate such a differential signal inversion in the
schematic. In addition, he should not be required to draw two wires in the schematic for each differential signal since that would increase schematic capture time and would make the schematics less readable. The router should place differential signal wires right next to each other to obtain equal loads on each of the two wires in a differential pair and to reduce signal crosstalk.

The following CAD tools have been used for the project because of their availability and capabilities:

**ORCAD** ;schematic capture software from Orcad

**FASTSIM** ;digital circuit simulator from Tektronix

**MERLYN-S** ;place and route software from Tektronix

**VERILOG** ;behavioral and gate level Simulator from Gateway

---

**Figure 84. Schematic with Differential and Single Ended Cells**
The schematic capture package, QUICKIC, a schematic capture and layout tool from Tektronix, had been tried initially. However, ORCAD is easier to use and faster than QUICKIC running on a Tektronix 6130 platform. The use of ORCAD has, however, also some drawbacks. ORCAD's schematic editor is very simple, it treats schematic symbols as simple graphics objects. It is also very tedious to use split buses in ORCAD as they frequently occur in partitioned systems. ORCAD does further not support cardinality. Thus, all elements of an 8bit component must be entered by the user and placed in a subcircuit to keep the size of the schematic down. The software packages that were finally used were running unfortunately on different platforms. ORCAD ran on IBM PCs, FASTSIM on SUN-3's, and MERLYN-S on VAX platforms under the VMS operating system. QUICKIC, FASTSIM, and MERLYN-S were donated by Tektronix for the FRISC project.

A program was needed to interface ORCAD with the FASTSIM simulator. Further, a schematic library for the Advanced Bipolar standard cells had to be created. The program TOTEKSIM was written to convert an Electronic Design Interchange Format (EDIF) netlist from ORCAD to a differential netlist in TEKSIM format for the FASTSIM simulator. The EDIF netlist format was chosen since it is the only hierarchical netlist format supported by ORCAD. In addition to the netlist format conversion TOTEKSIM expands differential signal nets into two differential nodes with the appropriate twist for virtual inverters. These features had to be part of the netlist convertor since the schematic capture package does not support
differential current tree logic design. Further, TOTEKSIM flags level violations, connections of differential signals to single ended ports, and unconnected module ports. The port properties of each standard cells must be declared in the definition file, stdcells.def. The following port properties must be declared: input/output, single ended/differential, and permissible input levels. An example is shown below.

```plaintext
(std_cell_library
  (std_cell or2sm (nodes ot :1 of :1 i1 :1 i2 :1))
  (std_cell or3sm (nodes ot :1 of :1 i1 :1 i2 :1 i3 :1))
  (std_cell or2lm (nodes o :1 i1rp :3 i1 :1 i2 :2))
  (std_cell buf2m (nodes o :2 i1 :1-3-))
  (std_cell vinv (nodes o i))
  (std_cell vstod (nodes o i_s i_s)))
```

12. Port Declaration for Standard Cells

A port level can be declared as unrestricted, as a fixed level, as a range of levels, or as a level relative to another port level. By convention any net or port with a '_' character in its name is declared as single ended. If the ports are properly defined single ended and differential cells can be mixed in the schematic as shown in Figure 84. Differential interconnect is drawn with a single wire. A differential signal is inverted with a virtual inverter symbol, VINV, as shown in Figure 84. The program uses the simple net integrity rule that all ports on a net must be either differential or single ended. A virtual single ended to differential converor, VSTOD, is provided to connect the complementary output of a single ended standard cell to differential ports. The virtual cells, VINV and VSTOD, are only provided to guide the netlist generator. These cells appear in the netlist only
as comments as shown in Figure 85. TOTEKSIM can also generate output in PSPICE format. PSPICE is a device level simulator from MICROSIM that runs on IBM PC platforms. The manual page for TOTEKSIM is listed in the appendix.

```plaintext
;NETLIST OF sample_sch
;sample: TimeStamp 22/9/1990

model sample subckt: nodes=(i1_se i2_se i3_se i4_se idif idifb o_se $ odifb odif)
g3 l1 l1b i3_se i2_se i1_se or3sm:
g2 odif odifb n12E n12 $ l1b l1 or21m:
b1 n12b n12 idif idifb buf2m:
v2 l1 l1 l1b vstod
v1 odif n10 winv
g1 o_se nc0. l1 i4_se or2sm:
endm sample
```

13. Netlist created by TOTEKSIM

In order to model the differential current tree logic with the FASTSIM digital simulator an extension of the simulator was necessary. Current switch and other primitives were added through the C language interface. This allowed modelling of the standard cells at the current switch level. A complete digital standard cell library was defined for the Advanced Bipolar differential current tree logic family. The delay parameters were extracted from SPICE simulations.

A collection of programs, called FLATTEN, was developed by graduate students Saheb Lahouar, Doug Lopata, and Doug Pricer. Its main task is the generation of the flattened preprocessor file for MERLYN-S. Later on a program called Wire Cutter was written by Paul Charlton. It modifies the input netlist and
the physical database to force MERLYN-S to route each differential interconnect with a fat wire that is cut into two the differential wires in a post processing phase. Thus, the two wires of a differential pair run always right next to each other!

Figure 85. CAD Tooling Stream at RPI

MERLYN-S is capable of extracting interconnect capacitances from the layout. The node capacitances are then fed back to the FASTSIM simulator to
evaluate the performance impact of the interconnect capacitance and to check for
circuit failure caused by excessive node capacitances on critical timing signals.
This process is also called back annotation. Back annotation creates an "infinite"
loop as shown in Figure 85. The layouts have shown that MERLYN-S can not be
effectively guided in the placement stage by wire priorities. Hence, time consuming
hand placement is necessary, especially for the critical ALU path.

For the test circuits designed by the author while working at Tektronix in
Beaverton, Oregon, a different tooling stream was used. TSPICE, a Tektronix
SPICE device level simulator, was used for circuit simulations and the QUICKIC
package from Tektronix was used for physical layout. QUICKIC checks the layout
against the SPICE netlist as it is created and thus minimizes the risk of layout
errors. The package has, however, currently no automatic place and route
capability. Further, the built in Design Rule Checker (DRC) could not handle all the
design rules for the Advanced Bipolar process. Therefore, a final DRC had to be
performed with MASKCAP, an old Fortran package from Phenix Data Systems that
is currently owned by ECAD.

VERILOG, a behavioral and gate level simulator from GATEWAY has been used
for the definition of a Register Transfer Model (RTL) of the FRISC-F processor.
The VERILOG simulator was donated for the FRISC project by GATEWAY. The
company has been meanwhile been bought by Cadence.
Figure 86. CAD Tooling Stream used for Test Circuits
Conclusions

FRISC can have no hardware features that can not be effectively partitioned or implemented at a low hardware cost. Despite these restrictions many features can be supported. The processor has 32 general purpose registers and a special purpose register for status information. The processor has a Harvard architecture with a shared address bus to provide parallel access to instructions and data with minimal interconnect. The instruction and data address transfers are pipelined to maximize the time available for accessing cache memory. The instruction set is very regular and all instructions are executed in a single cycle. However, load and branch instructions have a latency of three cycles. The processor is highly pipelined to boost peak performance and relax timing constraints for cache memory implementation. Interrupts and traps are vectored to provide fast interrupt response for real time applications.

The predicted peak throughput of 250MIPS for an Advanced Bipolar implementation is high even when compared to the predicted peak throughput of 200MIPS for the fastest GaAs RISC processor to be implemented on a single chip with Hetrojunction Bipolar Integrated Injection Logic (HIIL). The low integration levels of Advanced Bipolar circuits can be overcome to a large degree by dense multichip packaging. The high throughput makes FRISC attractive for real time controller applications or for high performance workstations. However, a math coprocessor is necessary to support floating point computations.
The packaging cost for the partitioned implementation of FRISC is low since a dense multichip package is required in any case for the high speed computing node consisting of the processor and cache memories. Without dense packaging the long signal delays between the processor and the cache memories would reduce the available time for accessing memory and preclude the implementation of sufficiently large cache memories. Providing large cache memories is very important because cache misses have the most significant performance impact after load and branch latencies.

Code reorganization is desirable to increase the efficiency of the seven stage deep pipeline. The average number of cycles per instructions can be reduced to 1.88 resulting in a sustained performance of 133MIPS with 32kByte cache memories. The main disadvantages of highly pipelined processors are increased design complexity and high load and branch latencies. The instruction decoder and interrupt mechanism become more complex as the pipeline depth increases. However, very deep instruction pipelines are feasible with the pipeline controller structure described in chapter VI. The effectiveness of code reorganization make deep instruction pipelines an effective method to boost performance with a very small amount of hardware compared to the superscalar approach.
The high speed and flexibility make differential current tree logic attractive for logic design. Differential logic can be expected to be used frequently for the next generation of advanced circuit technologies because of its low signal swing and low switching noise even though twice as many interconnections have to be routed.
References


A. FRISC-F VERILOG Sources

1. Module: sys, File: frisc.v; Top Level Module

    /***************************************************************************/
    /*
    * Module: sys
    *
    * Description:
    * Top Module of FRISC-F Simulator
    * Defines User Tasks: HELP, ...... , QUIT
    * instantiates modules:
    *     frisc - FRISC Processor:
    *     id - instruction decoder
    *     dp - datapath
    *     dcache - data memory
    *     icache - instruction memory
    *     aldin - memory input data alignment
    *     aldout - memory output data alignment
    *
    * FRISC-F Simulator Project:
    * Copyright Hans J. Greub and RPI, Troy, September 1990
    *
    * Modification Log:
    */
    /***************************************************************************/
    'define tic 100

    module sys;

    wire v_iad;
    wire v_dad;
    wire [31:0] a_bus;
    wire [31:0] d_buso;
    wire [31:0] d_busi;
    wire [31:0] i_bus;
    reg reset_;
    reg int_; // dummy register
    reg step_; // dummy register
    reg step_phase; // dummy register
    reg trace; // dummy register
    reg debug; // dummy register
    reg [8*23:1] i_str; // dummy register
    reg [31:0] tracefile; // dummy register
    wire [1:0] cnt_dat,byte_addr; // control and byte address for data alignment
    wire [31:0] data_out,data_in; // data cache data input and output
    wire [31:0] icache_in; // instruction cache input port
    wire [2:0] io_cnt; // 3 bit data cache access control field
    integer icnt;

    // instruction cache
    cache icache (i_bus,icnt_dat,ibyte_addr,hit_i,trap_i,error_, exc_,
    icache_in,a_bus[29:0],2'b00),v_iad,m_stall,1'b0,3'b010,1'b1, clk,run_stop);
// data alignment hardware between processor and memory input port
align_min aldin (data_in,d_buso,io_cnt[1:0],clk,run_stop);

// data cache
cache dcache (data_out,cnt_dat,byte_addr,hit_d,trap_d,error_,exc_,
data_in,a_buš,v_dad,m_stall,w_,io_cnt,1'b0,clk,run_stop);

// data alignment hardware between memory output port and processor
align_mout aldout (d_busi,data_out,byte_addr,cnt_dat,clk,run_stop);

// processor
FRISC frisc (v_iad,v_dad,a_bus,io_cnt,w_,m_stall,exc_,d_buso,d_busi,
i_bus,hit_i,hit_d,reset_,error_,trap_d,trap_i,int_,
halt_,clk,run_stop);

clock_gen clkgen(clk,run_stop);
phase_gen phg (p1,p2,p3,p4,clk,run_stop);

task ADD;
inuput [4:0] dest;
inuput [4:0] srcsa;
inuput [4:0] srcsb;
begin
    wait(p4);
    force i_bus=(8'b10000000,dest,2'b00,4'b0100,srcsa,srcsb,3'b000);
    frisc.tmp=3;
    @(posedge p4) release i_bus;
end
task SUB;
inuput [4:0] dest;
inuput [4:0] srcsa;
inuput [4:0] srcsb;
begin
    wait(p4);
    force i_bus=(8'b10000000,dest,2'b00,4'b0100,srcsa,srcsb,3'b000);
    frisc.tmp=3;
    @(posedge p4) release i_bus;
end
task LDI;
inuput [4:0] dest;
inuput [15:0] imm;
begin
    wait(p4);
    force i_bus=(8'b011000010,dest,3'b000,imm);
    frisc.tmp=3;
    @(posedge p4) release i_bus;
end
task BRA;
inuput [3:0] cc;
inuput [15:0] imm;
begin
    wait(p4);
    force i_bus=(8'b01111010,cc,4'b0000,imm);
frisc.tmp=3;
\@ (posedge p4) release i_bus;
end
task JMP;
input [3:0] cc;
input [4:0] srca;
input [4:0] srcb;
begin
  wait(p4);
  force i_bus={8'b11111000, cc, 7'b0000000, srca, srcb, 3'b000};
  frisc.tmp=3;
  \@ (posedge p4) release i_bus;
end
task GETLPC;
input [4:0] dest;
begin
  wait(p4);
  force i_bus={8'b01000110, dest, 19'b0};
  frisc.tmp=3;
  \@ (posedge p4) release i_bus;
end
task SETPM;
begin
  wait(p4);
  force i_bus={8'b00101100, 4'b0, 4'b1100, 16'b1};
  frisc.tmp=3;
  \@ (posedge p4) release i_bus;
end
task CLRPM;
begin
  wait(p4);
  force i_bus={8'b01111110, 4'b1110, 4'b1000, 16'b000};
  frisc.tmp=3;
  \@ (posedge p4) release i_bus;
end
task STRAP;
begin
  wait(p4);
  force i_bus={8'b00111100, 4'b1110, 4'b0001, 16'b000};
  frisc.tmp=3;
  \@ (posedge p4) release i_bus;
end
task RESET;
begin
  wait(p4)
  reset_=1;
wait(p2);
wait(p4);
reset_==0;
end
dotask

task LDGO;
input [32*8:1] file;
reg [36*8:1] fname;
begin
wait(p2);
$readmemh(file,".ins"),icache.memory);
$readmemh(file,".dat"),dcache.memory);
wait(p4)
reset_==1;
wait(p2);
wait(p4);
wait(p2);
wait(p4);
reset_==0;
end
dotask

task BREAKI;
input [7:0] address;
begin
icache.dummy_con[address]=3;
end
dotask

task STALLI;
input [7:0] address;
begin
icache.dummy_con[address]=1;
end
dotask

task FAULTI;
input [7:0] address;
begin
icache.dummy_con[address]=2;
end
dotask

task CLEARI;
input [7:0] address;
begin
icache.dummy_con[address]=0;
end
dotask

task BREAKD;
input [9:0] address;
begin
dcache.dummy_con[address>>2]=3;
end
dotask
task STALLD;
input [9:0] address;
begin
  dcache.dummy_con[address>>2]=1;
end
downtask

task FAULTD;
input [9:0] address;
begin
  dcache.dummy_con[address>>2]=2;
end
downtask

task CLEARD;
input [9:0] address;
begin
  dcache.dummy_con[address>>2]=0;
end
downtask

task DUMPIC;
reg [31:0] mcd;
integer index;
begin
  mcd=$fopen("dump.ic")
  if (mcd==0) $display("***ERROR: could not open file dump.ic")
  else for (index=0;index<256;index=index+1)
  $fdisplay(mcd,"@'h%h 'h%h",index,icache.memory[index])
end
downtask

task DUMPDC;
reg [31:0] mcd;
integer index;
begin
  mcd=$fopen("dump.dc")
  if (mcd==0) $display("***ERROR: could not open file dump.dc")
  else for (index=0;index<256;index=index+1)
  $fdisplay(mcd,"@'h%h 'h%h",index,dcache.memory[index])
end
downtask

task RUN;
input [31:0] cycles;
integer i;
reg state1,state2;
begin
  state1=step;
  state2=step_phase;
  step=0;
  step_phase=0;
  for (i=0;i<cycles;i=i+1) @(negedge p4);
  step=state1;
  step_phase=state2;
  $stop;
end
downtask
task ERROR;
begin
  wait(p4)
  force error_i=1;
  wait(p2);
  wait(p4);
  release error_i;
end
dontask

task INT;
input int;
begin
  @(posedge p1) int_=int;
end
dontask

task TRAPD;
begin
  wait(p4)
  force trap_d=1;
  wait(p2);
  wait(p4);
  release trap_d;
end
dontask

task TRAPI;
begin
  wait(p4)
  force trap_i=1;
  wait(p2);
  wait(p4);
  release trap_i;
end
dontask

task GETPSW;
input [4:0] dest;
begin
  wait(p4);
  force i_bus=(8'b01100100,dest,19'b0);
  frisc.tmp=3;
  @(posedge p4) release i_bus;
end
dontask

task SETPSW;
input [4:0] srcs;
begin
  wait(p4);
  force i_bus=(8'b11011010,11'b1,srcs,8'b0);
  frisc.tmp=3;
  @(posedge p4) release i_bus;
end
dontask

task LOAD;
input [4:0] dest;
input [4:0] srca;
input [4:0] srcb;
bEGIN
    wait(p4);
    force i_bus=(8'bl0000000,dest,6'b0000111,srca,srcb,3'b000);
    frisc.tmp=3;
    @(posedge p4) release i_bus;
end
eNDTASK

task SHL1;
input [4:0] dest;
bEGIN
    wait(p4);
    force i_bus=(8'bl0100010,dest,6'bl11111,dest,8'b0);
    frisc.tmp=3;
    @(posedge p4) release i_bus;
end
eNDTASK

task SHL2;
input [4:0] dest;
bEGIN
    wait(p4);
    force i_bus=(8'bl0100000,dest,6'bl11111,dest,8'b0);
    frisc.tmp=3;
    @(posedge p4) release i_bus;
end
eNDTASK

task SHR1;
input [4:0] dest;
bEGIN
    wait(p4);
    force i_bus=(8'bl0100100,dest,6'bl11111,dest,8'b0);
    frisc.tmp=3;
    @(posedge p4) release i_bus;
end
eNDTASK

task SHR2;
input [4:0] dest;
bEGIN
    wait(p4);
    force i_bus=(8'bl0100110,dest,6'bl11111,dest,8'b0);
    frisc.tmp=3;
    @(posedge p4) release i_bus;
end
eNDTASK

task STORE;
input [4:0] srca;
input [4:0] srcb;
bEGIN
    wait(p4);
    force i_bus=(8'bl10100000,5'bl0101,6'b000111,srca,srcb,3'b000);
    frisc.tmp=3;
@ (posedge p4) release i_bus;
end
endtask

task XORPSW;
input [15:0] imm;
begin
    wait (p4);
    force i_bus={8'b00110000,8'b0,imm};
    frisc.tmp = 3;
    @ (posedge p4) release i_bus;
end
endtask

task STEP;
input in;
begin
    step = in;
    if (step) $stop;
end
endtask

task PSTEP;
input in;
begin
    step_phase = in;
    if (step_phase) $stop;
end
endtask

task TRACE;
input in;
begin
    trace = in;
    $stop;
end
endtask

task QUIT;
begin
    $display("->%0d instruction cycles have been simulated",phg.cycle);
    $finish(2);
end
endtask

task DEBUG_DC;
input in;
begin
    dcache.debug = in;
end
endtask

task DEBUG_IC;
input in;
begin
    icache.debug = in;
end
endtask

task DEBUG;
input in;
begnin
if (in)
begnin
debug=1;
frisc.debug=1;
frisc.dp.debug=1;
frisc.id.debug=1;
frisc.dp.debug=1;
icache.debug=1;
dcache.debug=1;
end
else
begnin
debug=0;
frisc.debug=0;
frisc.dp.debug=0;
frisc.id.debug=0;
frisc.dp.debug=0;
icache.debug=0;
dcache.debug=0;
end
endtask

task WINDOW;
begin
frisc.WINDOW;
end
endtask

task HELP;
begin
$display("\nPRISC-F SIMULATOR V1.0 COMMAND LEVEL TASKS\n");
$display("HELP");
$display("TRACE([1|0])");
$display("DEBUG([1|0])");
$display("STEP([1|0])");
$display("RUN(<num>)");
$display("RESET");
$display("ERROR");
$display("INT([1|0])");
$display("TRAPD");
$display("TRAPF");
$display("DUMPIC");
$display("DUMPCD");
$display("BREAKD(<num>)");
$display("STALLD(<num>)");
$display("FAULTD(<num>)");
$display("CLEARD(<num>)");
$display("BREAKI(<num>)");
$display("STALLI(<num>)");
$display("FAULTI(<num>)"); stall and page fault at instr. address <num>");
$display("CLEARI(<num>)"); clear break, stall, or page fault at instr. address <num>");
$display("WINDOW"); turn register window on");
$display("LDGO("<str>")"); load file: <str> and generate reset");
$display("QUIT"); quit");
$display("$input("<str>")"); read commands from file: <str>");
$display("\n->TO RUN A TASK ENTER: <TASK_NAME>;<CR>");
$display("->control-c stops the simulation and prompts user for input");
$display("->control-d or QUIT;<CR>"); at the prompt will end the program");
end
e ndtask

initial
begin
int_=0;
halt_=0;
icnt_=0;
if ($testplusargs("stepoff"); step=0;
else step=1;
tracefile=$fopen("trace.ins");
if ($testplusargs("traceoff"); trace=0;
else trace=1;
if ($testplusargs("debug"); DEBUG(1);
else DEBUG(0);
if ($testplusargs("window"); WINDOW;
HELP;
end
always @(negedge p4) if (step) $stop;
always @(phg.phase) if (step_phase) $stop;
always @(posedge p4) if (trace) begin
$display("$%0h : %s aop=%h aopb=%h res_ex=%h",
frisc.dp.pc.ex.display,frisc.id.itos(frisc.id.i_ex),frisc.dp.aopb,
frisc.dp.aopb,frisc.dp.res_ex);
if (-frisc.stall_) icnt=icnt+1;
end
endmodule

module FRISC (v_iad,v_dad,a_bus,io_cnt,w_,m_stall,exc_,d_buso,
d_busi,l_bus,Hit_i,Hit_d,reset_,error_,trap_d,
trap_i,halt_clk,Run_stop);
output v_iad; // valid instruction address (p4)
output v_dad; // valid data address (p4)
output [31:0] a_bus; // shared instruction / data bus (p4)
output [2:0] io_cnt; // data cache access control
output w_; // data cache write signal
output m_stall; // memory pipeline stall signal
output exc_; // exception signal
output [31:0] d_buso; // data output bus (p1)
input [31:0] d_busi; // data input bus (p2)
input [31:0] i_bus; // instruction bus
input hit_i; // instruction cache hit (p1)
input hit_d; // data cache hit (p1)
input reset_; // processor reset (p1)
input error; // system error, hardware failure trap (pl)
input trap_d; // data trap, page fault (pl)
input trap_i; // instruction bus trap, page fault (pl)
input int_; // user interrupt (pl)
input halt_; // halt processor (pl)
input clk; // 500MHz Clock;
input run_stop; // run_stop clock phase generator

wire [3:0] c_c;
wire [1:0] alu_op;
wire [15:0] imm;
wire [1:0] op_a;
wire [1:0] size;
wire [4:0] rf_addr;
wire [1:0] ff_ra;
wire [1:0] ff_rb;

id id (exc, stall_, m_stall, hold_, clr_pc, pc_sel, pc_lock, w_addr, s_cc,
c_c, e_carry, w_psw, alu_op, imm_, ldi_hi, op_a, op_b, inv_b, sh_, sh_ex,
size, rf_addr, ff_ra, ff_rb, m_by_a, m_by_b, w_rf, sel_d,
w_din, w_dout, w_io_cnt, clk, run_stop, i_bus, p_m, hit_i, hit_d,
reset, error, trap_d, trap_i, a_t, e_int, int_, Bra_, halt_);

dp dp (a_bus, d_buso, bra_e_int, a_t, v_iad, v dad, d busi, rf_addr, imm_, clk,
run_stop, alu_op, ff_ra, ff_rb, sh_, sh ex, op a, op b, m by a, m by b,
inv b, pc sel, clr pc, c c, s cc, w rf, sel d, w psw, w din, w dout,
w addr, exc_, stall_, pc lock, e carry, hold_, p m, size_, ldi hi);

integer tmp;
integer i;

reg [4:0] opcode
reg debug;

task WINDOW;
begin
$gr_regs("FRISC - USER VISIBLE STATE: cycle = %d phase = %d", id.phg.cycle, id.phg.phase,
"dout =%h din =%h TAG",
"", dp.dout_d1, dp.din_dw, (id.v_tag) ? id.tag:5'hxx,
"%s",
"R30 =%h R31 =%h", dp.rf[30], dp.rf[31],
"R28 =%h R29 =%h", dp.rf[28], dp.rf[29],
"R26 =%h R27 =%h", dp.rf[26], dp.rf[27],
"R24 =%h R25 =%h", dp.rf[24], dp.rf[25],
"R22 =%h R23 =%h", dp.rf[22], dp.rf[23],
"R20 =%h R21 =%h", dp.rf[20], dp.rf[21],
"R18 =%h R19 =%h", dp.rf[18], dp.rf[19],
"R16 =%h R17 =%h", dp.rf[16], dp.rf[17],
"R14 =%h R15 =%h", dp.rf[14], dp.rf[15],
"R12 =%h R13 =%h", dp.rf[12], dp.rf[13],
"R10 =%h R11 =%h", dp.rf[10], dp.rf[11],
"R8 =%h R9 =%h", dp.rf[8], dp.rf[9],
"R6 =%h R7 =%h", dp.rf[6], dp.rf[7],
"R4 =%h R5 =%h", dp.rf[4], dp.rf[5],
"SRCA=%h\", dp.rf[4], dp.rf[5],
);
R2 = `%h
R3 = `%h
SRCB = `%h ", dp.rf[ 2], dp.rf[
3], dp.rfaddrb,
"R0 = `%h
R1 = `%h
DEST = `%h ", dp.rf[ 0], dp.rf[
1], dp.rfaddrd,
"rfa = `%h
rfb = `%h ", dp.rfa, dp.rfb,
"aopa = `%h
aopb = `%h
imm = `%h
", dp.aopa, dp.aopb, dp.imm,
"res_ex = `%h
TAG = `%h
addr = `%h ", dp.res_ex, (id.v_res_ex)?id.i_ex[23:19]:5’hx, dp.addr,
"res_d1 = `%h
TAG = `%h
", dp.res_d1, (id.v_res_d1)?id.i_d1[23:19]:5’hx,
"res_d2 = `%h
TAG = `%h
", dp.res_d2, (id.v_res_d2)?id.i_d2[23:19]:5’hx,
"pc_i = `%h
psw: NCVZ=`b , (P, I, Cl, CQ)=`b ",
dp.pc_i, dp.psw_d1[31:28], (dp.psw_d1[3:2], dp.cl, dp.c0),
"pc_i2 = `%h
", dp.pc_i2,
"pc_d = `%h
", dp.pc_d,
"pc_ex = `%h
", dp.pc_ex,
"pc_d1 = `%h
", dp.pc_d1,
"pc_d2 = `%h
", dp.pc_d2,
"pc_dw = `%h
", dp.pc_dw
);
end
task
always @(exc_) if (debug)
 $display("%d0.p%d:exc=%b", id.phg.cycle, id.phg.phase, exc_);
always @(v.iad) if (debug)
 $display("%d0.p%d:v_iad=%b", id.phg.cycle, id.phg.phase, v_iad);
always @(v.vad) if (debug)
 $display("%d0.p%d:v_vad=%b", id.phg.cycle, id.phg.phase, v_vad);
always @(hit_i) if (debug)
 $display("%d0.p%d:hit_i=%b", id.phg.cycle, id.phg.phase, hit_i);
always @(hit_d) if (debug)
 $display("%d0.p%d:hit_d=%b", id.phg.cycle, id.phg.phase, hit_d);
always @(stall) if (debug)
 $display("%d0.p%d:stall=%b", id.phg.cycle, id.phg.phase, stall_);
always @(hold) if (debug)
 $display("%d0.p%d:hold=%b", id.phg.cycle, id.phg.phase, hold_);
always @(clr_pc) if (debug)
 $display("%d0.p%d:clr_pc=%b", id.phg.cycle, id.phg.phase, clr_pc);
always @(pc_sel) if (debug)
 $display("%d0.p%d:pc_sel=%b", id.phg.cycle, id.phg.phase, pc_sel);
always @(pc_lock) if (debug)
 $display("%d0.p%d:pc_lock=%b", id.phg.cycle, id.phg.phase, pc_lock);
always @(w.addr) if (debug)
 $display("%d0.p%d:w_addr=%b", id.phg.cycle, id.phg.phase, w_addr);
always @(a.bus) if (debug)
 $display("%d0.p%d:a_bus=%b", id.phg.cycle, id.phg.phase, a_bus);
always @(i.bus) if (debug)
 $display("%d0.p%d:i_bus=%b", id.phg.cycle, id.phg.phase, i_bus);
always @(d.buso) if (debug)
 $display("%d0.p%d:d_buso=%b", id.phg.cycle, id.phg.phase, d_buso);
always @(d.busi) if (debug)
 $display("%d0.p%d:d_busi=%b", id.phg.cycle, id.phg.phase, d_busi);
always @(s_cc) if (debug)
$display("%d0.p%d:s_cc=%b",id.phg.cycle,id.phg.phase,s_cc);
always @(c_c) if (debug)
  $display("%d0.p%d:c_c=%b",id.phg.cycle,id.phg.phase,c_c);
always @(e_car) if (debug)
  $display("%d0.p%d:e_car=%b",id.phg.cycle,id.phg.phase,e_car);
always @(w_psw) if (debug)
  $display("%d0.p%d:w_psw=%b",id.phg.cycle,id.phg.phase,w_psw);
always @(alu_op) if (debug)
  $display("%d0.p%d:alu_op=%b",id.phg.cycle,id.phg.phase,alu_op);
always @(imm_) if (debug)
  $display("%d0.p%d:imm=%h",id.phg.cycle,id.phg.phase,imm_);
always @(ldi_hi) if (debug)
  $display("%d0.p%d:ldi_hi=%b",id.phg.cycle,id.phg.phase,ldi_hi);
always @(op_a) if (debug)
  $display("%d0.p%d:op_a=%b",id.phg.cycle,id.phg.phase,op_a);
always @(op_b) if (debug)
  $display("%d0.p%d:op_b=%b",id.phg.cycle,id.phg.phase,op_b);
always @(inv_b) if (debug)
  $display("%d0.p%d:inv_b=%b",id.phg.cycle,id.phg.phase,inv_b);
always @(sh) if (debug)
  $display("%d0.p%d:sh=%b",id.phg.cycle,id.phg.phase,sh);
always @(sh_ex) if (debug)
  $display("%d0.p%d:sh_ex=%b",id.phg.cycle,id.phg.phase,sh_ex);
always @(size) if (debug)
  $display("%d0.p%d:size=%b",id.phg.cycle,id.phg.phase,size);
always @(rf_addr) if (debug)
  $display("%d0.p%d:rf_addr=%b",id.phg.cycle,id.phg.phase,rf_addr);
always @(ff ra) if (debug)
  $display("%d0.p%d:ff_ra=%b",id.phg.cycle,id.phg.phase,ff_ra);
always @(ff rb) if (debug)
  $display("%d0.p%d:ff_rb=%b",id.phg.cycle,id.phg.phase,ff_rb);
always @(m by_a) if (debug)
  $display("%d0.p%d:m_by_a=%b",id.phg.cycle,id.phg.phase,m_by_a);
always @(m by b) if (debug)
  $display("%d0.p%d:m_by_b=%b",id.phg.cycle,id.phg.phase,m_by_b);
always @(w rf) if (debug)
  $display("%d0.p%d:w_rf=%b",id.phg.cycle,id.phg.phase,w_rf);
always @(sel d) if (debug)
  $display("%d0.p%d:sel_d=%b",id.phg.cycle,id.phg.phase,sel_d);
always @(w din) if (debug)
  $display("%d0.p%d:w_din=%b",id.phg.cycle,id.phg.phase,w_din);
always @(w dout) if (debug)
  $display("%d0.p%d:w_dout=%b",id.phg.cycle,id.phg.phase,w_dout);
always @(halt_) if (debug)
  $display("%d5.p%d:halt=%b",id.phg.cycle,id.phg.phase,halt_);
always @(reset) if (debug)
  $display("%d0.p%d:reset=%b",id.phg.cycle,id.phg.phase,reset);
always @(p m) if (debug)
  $display("%d0.p%d:p_m=%b",id.phg.cycle,id.phg.phase,p_m);
always @(bra) if (debug)
  $display("%d0.p%d:bra=%b",id.phg.cycle,id.phg.phase,bra);
always @(w) if (debug)
  $display("%d0.p%d:w=%b",id.phg.cycle,id.phg.phase,w);
always @(io cnt) if (debug)
  $display("%d0.p%d:io_cnt=%b",id.phg.cycle,id.phg.phase,io_cnt);
always @(id.reset_cpu) if (debug)
  $display("%d0.p%d:reset_cpu=%b",id.phg.cycle,id.phg.phase,id.reset_cpu);
always @(int__) if (debug)
$display("%d.p%d:int_%b",id.phg.cycle,id.phg.phase,int_);
always @(error) if (debug)
    $display("%d0.p%d: error_%b",id.phg.cycle,id.phg.phase,error_);
always @(reset) if (debug)
    $display("%d0.p%d: reset_%b",id.phg.cycle,id.phg.phase,reset_);
always @(trap_d) if (debug)
    $display("%d0.p%d:trap_d=%b",id.phg.cycle,id.phg.phase,trap_d);
always @(trap_i) if (debug)
    $display("%d0.p%d:trap_i=%b",id.phg.cycle,id.phg.phase,trap_i);
always @(halt) if (debug)
    $display("%d0.p%d:halt_%b",id.phg.cycle,id.phg.phase,halt_);
always @(id.trap) if (debug)
    $display("%d0.p%d:trap=%b",id.phg.cycle,id.phg.phase,id.trap);
endmodule
Module: dp, File: lib/dp.v; Datapath

******************************************************************************

Module: dp

Description: FRISC-F 32bit Datapath

* FRISC-F Simulator Project:
* Copyright Hans J. Greub and RPI, Troy, September 1990
* Modification Log:

******************************************************************************

module dp (a_bus, d_buso, bra_, e_int, a_t, v_iad, v_dad, d_busi, rf_addr, imm_, clk,
  _run_stop, alu_op, ff_ra, ff_rb, sh_, sh_ex, op_a, op_b, m_by_a, m_by_b,
  inv_b, pc_sel, clr_pc, c_c, s_cc, w_rf, sel_d, w_psw, w_din, w_dout,
  w_addr, exc_, stall_, pc_lock, e_carry, hold_, p_m, size_, ld_hi);

p a r m e t e r

bits = 32, msb = 31, lsb = 0, rf_read = 800/\text{tic}, rf_write = 500/\text{tic},
  rf_asut = 500/\text{tic}, DRI\_delay = 400/\text{tic}, REC\_delay = 100/\text{tic};

output [(bits-1):0] a_bus; // addr bus out
output [(bits-1):0] d_buso; // data bus out
output bra_; // branch signal
output e_int; // enable interrupt
output a_t; // arithmetic trap
output v_iad; // valid instruction address on _a_bus
output v_dad; // valid data address on _a_bus

input [(bits-1):0] d_busi; // data bus in
input [4:0] rf_addr; // register file address
input [15:0] imm_; // immediate constant
input clk; // 500MHz system clock
input run_stop; // run/stop signal for system clock
input [1:0] alu_op; // ALU function select \{ ADD, AND, XOR, OR \}
input [1:0] ff_ra; // feed forward operand A \{oa, res_ex, res_d1, res_d2\}
input [1:0] ff_rb; // feed forward operand B \{ob, res_ex, res_d1, res_d2\}

input sh_; // shifter output is ALU operand A
input sh_ex; // shift extended
input [1:0] op_a; // operand a select
input op_b; // operand b select
input m_by_a; // memory bypass operand A
input m_by_b; // memory bypass operand B
input inv_b; // invert operand
input pc_sel; // \{pc_de, pc_dw\}
input clr_pc; // clear pc_ex register
input [3:0] c_c; // condition code
input s_cc; // set condition code
input w_rf; // write register file
input sel_d; // select din register as register file input \{res_d2, din\}
input w_psw; // write status register
input w_din; // write data input register
input w_dout; // write data output register
input w_addr;  // write address register
input exc_r;  // exception signal
input stall;  // pipeline stall signal
input pc_lock; // freeze pc_de, pc_d1, pc_d2, pc_dw
input e_carry; // enable carry
input hold;  // force all outputs to 0 (a_bus, d_bus)
input p_m;  // protected mode signal
input [1:0] size;  // size of alu operands {byte, half, word, word}
input ld_hi;  // load immediate constant into upper 16 bits of imm

event new_cycle;

wire pl, p2, p3, p4;  // clock phase signals
wire n, cout, z, v;  // status signals
wire in33, in32, inml, inmb2;  // shifter inputs for shift up and down
wire shov_de;  // shifter overflow signal
wire [(bits-1):0] ca, cb;  // internal bus
// output buses of alu, sh, rf, pc, psw
wire [(bits-1):0] sho, rfo, pco, psw0;
wire rw;  // rw signal for register file
wire [(bits-1):0] rfin;  // register file input port
wire [(bits-1):0] alures;  // alu output port
wire [4:0] regaddr;  // register file address port

// registers
reg [(bits-1):0] rf[0:31];  // register file
reg [(bits-1):0] res_ex, res_d1, res_d2;  // alu result registers
wire [(bits-1):0] alow, res_ex;
reg [(bits-1):0] din_dw, dout_ex, dout_d1;  // data I/O registers
reg [(bits-1):0] pc_i1, pc_i2, pc_de, pc_ex, pc_d1, pc_d2, pc_dw;  // program counters
reg [(bits-1):0] pc_ex_display;  // dummy register
reg [(bits-1):0] psw_d1, psw_d2, psw_d2_o;  // status word
reg [(bits-1):0] addr;  // address Register
reg [(bits-1):0] rfa, rfb;  // register file output registers
reg [(bits-1):0] acpa, acpb;  // alu input operand registers
reg [4:0] rfadda, rfaddrb, rfaddrd;  // register file address
reg hold1;  // latch for hold
reg edout;  // enable data register output on d_buso
reg cin;  // carry in for ALU
reg c0, c1;  // shifter overflow bits
reg [1:0] aluo1;  // latch for aluop
reg [1:0] size1;  // latch size
reg shov_ex;  // latch for shifter overflow
reg shov_d1;  // latch for shifter overflow
reg [3:0] branchcode;
reg invbo;

`define N psw_d1[31]
`define C psw_d1[30]
`define V psw_d1[29]
`define Z psw_d1[28]
`define F psw_d1[3]
`define I psw_d1[2]
`define C0 psw_d1[0]
`define C1 psw_d1[1]
integer index;

    // Output Drivers

    reg bra;
    assign #DRI_delay bra = bra;
    assign #DRI_delay e_int = I;
    reg viad;
    assign #DRI_delay v_iad = viad & holdl;
    reg vdad;
    assign #DRI_delay v_dad = vdad & holdl;
    reg at;
    assign #DRI_delay a_t = at;
    assign #DRI_delay d_buso = dout_d1 & holdl;
    assign #DRI_delay a_bus = addr & holdl;
    assign #DRI_delay imm1 = c1;
    assign #DRI_delay imm2 = c0;
    assign #DRI_delay in33 = c1;
    assign #DRI_delay in32 = c0;

    // Input Receivers

    wire [ (bits-1):0 ] dbusi;
    assign #REC_delay dbusi = d_busi;
    wire [4:0] rfaddr;
    assign #REC_delay rf_addr = rf_addr;
    wire [15:0] imm;
    assign #REC_delay imm = imm;
    wire [1:0] aluop;
    assign #REC_delay alu_op = alu_op;
    wire [1:0] ffra;
    assign #REC_delay ffra = ff_ra;
    wire [1:0] ffrb;
    assign #REC_delay ffrb = ff_rb;
    wire sh;
    assign #REC_delay sh = sh;
    wire shex;
    assign #REC_delay shex = sh_ex;
    wire [1:0] opa;
    assign #REC_delay op_a = op_a;
    wire opb;
    assign #REC_delay opb = op_b;
    wire mbya;
    assign #REC_delay mby_a = m_by_a;
    wire mbyb;
    assign #REC_delay mbyb = m_by_b;
    wire invb;
    assign #REC_delay invb = inv_b;
    wire pc.sel;
    assign #REC_delay pc_sel = pc_sel;
    wire clrpc;
    assign #REC_delay clrpc = clr_pc;
    wire [3:0] cc;
    assign #REC_delay cc = c_c;
    wire scc;
    assign #REC_delay scc = s_cc;
    wire wrf;
assign #REC_delay wrf=w_rf;
wire seld;
assign #REC_delay seld=sel_d;
wire wpsw;
assign #REC_delay wpsw=w_psw;
wire wdin;
assign #REC_delay wdin=w_din;
wire wdo;
assign #REC_delay wdo=w_dout;
wire waddr;
assign #REC_delay waddr=w_addr;
wire exc;
assign #REC_delay exc=exc_;
wire stall;
assign #REC_delay stall=stall_;
wire pclock;
assign #REC_delay pclock=pclock;
wire ecarry;
assign #REC_delay ecarry=ecarry;
wire hold;
assign #REC_delay hold=hold_;
wpm;
assign #REC_delay pm=p_m;
wire [1:0] Size;
assign #REC_delay size=size_;
wire ldhi;
assign #REC_delay ldhi=ld_hi;

reg write_ok; // dummy register
reg debug; // dummy register

// four phase clock generator
phase_gen pchg (p1,p2,p3,p4,clk,run_stop);
// arithemetic logical unit
alu
    all (n,cout,z,v,alures,aluop1,size1,aopa,aopb,cin);
// shifter [shift left by 2,shift left by 1,shift right by 1,shift right by 2]
shifter sh1 (shov_de,sho,opa,shex,size,in33,in32,oa,inm1,inm2);

//temporary
reg v_tag;
reg [4:0]tag;

initial
begin
    cin=0; // must be removed from final simulation file
    bra=1; // same
debug=1;
    for (index=0;index<32;index=index+1) rf[index]=index; // initialize rf
    // setup display
end

// Tasks
// Feed Forward Control
// use: ffra(p4),ff_ra(p4),mbya(p4),mbyb(p4)
// use: opa(p4),opb(p4)
// gen: oa(p4), ob(p4), rfa(p4)
task cont_ff;
begin
wait(p4);
   rfa = mbya ? din_dw : rfo;
   if (mbyb) rfb=din_dw;
   #REC_delay
   case (ffra)
      'b00: force oa=rfa;
      'b01: force oa=res_d2;
      'b10: force oa=res_d1;
      'b11: force oa=res_ex;
   endcase
   case (ffrb)
      'b00: force ob=rfb;
      'b01: force ob=res_d2;
      'b10: force ob=res_d1;
      'b11: force ob=res_ex;
   endcase
end
endtask

// Control Register File
/
/
// seld(p4), wrf(p1), rfaddr(p1,p2,p3), rfo(p4), din_dw(p4-p1), res_d2(p4-p1)
// gen: rfb(p3)
task cont_rf;
begin
   wait(p1);
   force rw=wrf;
   // transferring srcb
   wait(p2);
   force rw=1;
   rfaddrb=rfaddr;
   force regaddr=rfaddrb;
   wait(p3);
   rfb=rfo;
   rfaddr=rfaddr;
   force regaddr=rfaddr;
   wait(p4);
   rfaddrd=rfaddr;
   force regaddr=rfaddrd;
   force rfin = (seld) ? din_dw : res_d2;
end
endtask

// Program Counter Control
// controls the program counters (pc_i1, pc_i2, pc_de, pc_d1, pc_d2, pc_dw)
// use: pclock(p2), clrpc(p2), pcSEL(p4)
task cont_pc;
begin
   wait(p2);
   if (~stall & pclock)
      begin
         pc_dw=pc_d2;
         pc_d2=pc_d1;
         pc_d1=(clrpc)?0:pc_ex;
   end
end
pc_ex=pc_de;
end
if (~stall)
begin
    pc_ex_display=pc_de;
    pc_de=pc_i2;
    pc_i2=pc_i1;
    if (debug) $display("%0d.p%d: pc_i1 <- %h",phg.cycle,phg.phase,pc_i1);
    end
    if (~stall) pc_i1=(bra) ? aluo : (pc_i1+1);
    wait(p4)
    force pco = pcesel ? pc_dw : pc_de;
end
endtask

// ALU Control
// controls ALU, imm,res_ex,res_d1,res_d2, registers
// use: aluop(p1),sh(p4-p1),srep(p4-p1),invb(p4-p1),opb(p4-p1) ;imm(p4-p1)
//     ldhi(p4-p1);
task cont_alu;
begin
    wait(p1);
    aluop=p1; // latch ALU operation code in msl
    size=el; // latch ALU operand size in msl
    if (~stall) begin // only update ALU input latches if no pipeline stall
        if (sh) aopm=sho;
        else case (opa) .
            2'b00: aopm=oa;
            2'b01: aopm=0;
            2'b10: aopm=psw_d1;
            2'b11: aopm=pco;
        endcase
        case ((opb,invb))
            2'b00: aob=ob;
            2'b01: aob=ob;
            2'b10: aob=(ldhi) ? (imm,16'b0) : imm;
            2'b11: aob= ~(ldhi) ? (imm,16'b0) : imm;
        endcase
    end
    if (debug) $display("->aopm=%h, ->aob=%h",aopm,aob);
    wait(p2)
    if (~stall)
begin
    #1
    res_d2=res_d1;
    res_d1=res_ex;
end
wait(p4);
if (~stall) res_ex=alures;
    if (debug) $display("%0d.p%d: res_ex <- %h",phg.cycle,phg.phase,res_ex);
    // #1 $display("oa=%h,ob=%h, sho=%h,sh=%b, imm=%h,opb=%b,invb=%b, stall=%b",
    //     oa,ob,sho,sh,imm,opb,invb,stall);
    end
endtask
task cont_IO;
begin
  wait(p1);
  hold1=hold;
  if (wdout&~stall) begin
    dout_d1=dout_ex;
    if ((debug) $display("%0d.p%d: dout_d1 <- %h",phg.cycle,phg.phase,dout_d1);
  end
  if (~stall) dout_ex=ob;
  wait(p3);
  if (wdin) begin
    din_dw=dbusi;
    if ((debug) $display("%0d.p%d: din_dw <- %h",phg.cycle,phg.phase,din_dw);
  end
  wait(p4);
  if ((waddr|bra)&~stall) begin
    #REC_delay addr=aluo;
    if ((debug) $display("%0d.p%d: addr <- %h",phg.cycle,phg.phase,aluo);
  end
  if (~stall) begin
    viad=bra;
    vdad=waddr;
    if (viad&vdad) $display("ERROR: waddr & bra -> viad & vdad");
  end
endtask

// Processor Status Word Control
// use: invb(p1),ecarry(p1),exc(p1),wpsh(p1),scc(p1),sh(p1),stall(p1)
// cc(p2)
// gen: at(p1),bra(p3)

task cont_psw;
begin
  wait(p1);
  if (~stall) begin /* psw_d2_o contains psw as seen by last instr. to complete */
    if (~exc) psw_d2_o=psw_d2;
    psw_d2=psw_d1;
    branchcode=cc; //latch branch condition code
    psw_d1[1:0]={c1,c0};
    shov_d1=shov_ex;
    shov_ex=shov_d6&sh;
    at=(v|shov_d1);
  end
  if (shex&~stall) begin
    if (~opa[3]&sh) begin // shift left
      c0=oa[30];
      c1=oa[31];
      //display("->{c1,c0}={oa[31],oa[30]}=%b b oa=%h",c1,c0,oa);
    end
    else begin // shift right or no shift
c0=oa[0];
c1=oa[1];

//display("->{c1,c0}={oa[1],oa[0]}=%b b oa=%h",c1,c0,oa);
end
end
if (exc) begin
  psw_d1=psw_d2_o;
  {c1,c0}=psw_d2_o[1:0];
  $display("->restoring psw; psw_d1=%h",psw_d1);
end
else if (~stall) begin
  if (wpsw) begin
    psw_d1=alu_o;
    {c1,c0}=psw_d1[1:0];
    //display("%d.p%d: psw_d1 <- %h",phg.cycle,phg.phase,psw_d1);
    if (debug) $display("%0d.p%d: psw_d1 <- %h",phg.cycle,phg.phase,psw_d1);
  end
  else begin
    if (scc) begin
      'N=n;
      'V=(shov_d1[1] | v);
      'Z=z;
      if (debug) $display("%0d.p%d: NCVZ <- %b",phg.cycle,phg.phase,psw_d1[31:28]);
    end
    if (scc | shex) $display("->setting CC : NCVZ=%b Cl=%b,Co=%b shex=%b", { 'N, 'C, 'V, 'Z}, c1, c0, shex);
  end
end
if (~stall) invb_o=invb;
cin = invb ? ~(ecarry & 'C) : (ecarry & 'C);
//display("->cin=%b C=%b ecarry=%b invb=%b",cin, 'C, ecarry, invb);
wait(p2);
  'P=pm; //wait till p2 to catch current mode (changes during p1 on id)

wait(p3)
case (branchcode) // evaluate branch condition
  'b0000: bra="('N'\'V')+\'Z"; // less or equal (signed)
  'b0001: bra="(('N'\'V')+\'Z"; // greater (signed)
  'b0010: bra="('N'\'V"; // less (signed)
  'b0011: bra="('N'\'V"; // greater or equal (signed)
  'b0100: bra="('C'\'Z"; // lower or same (unsigned)
  'b0101: bra="('C'\'Z"; // higher (unsigned)
  'b0110: bra="\'C"; // higher or same (unsigned)
  'b0111: bra="\'C"; // lower (unsigned)
  'b1000: bra="\'N"; // negative (signed)
  'b1001: bra="\'N"; // plus (signed)
  'b1010: bra="\'Z"; // equal
  'b1011: bra="\'Z"; // not equal
  'b1100: bra="\'V"; // overflow (signed)
  'b1101: bra="\'V"; // no overflow (signed)
  'b1110: bra=1; // always
  'b1111: bra=0; // never
endcase
if (debug) $display("%0d.p%d: bra <- %b, cc=%b",}
// action
// register file read/write process

assign #rf_read rfo=rf[regaddr];

always
    begin @(regaddr)
        write_ok=0;
        if (~rw) $display("ERROR; register file address change while writing");
        #rf_asut write_ok=1;
    end

always
    begin @(negedge rw)
        if (write_ok) begin
            if (debug) $display("%0d.p%d: rf[%h] <= %h", 
                phg.cycle, phg.phase, regaddr, rfin);
            #rf_write rfo[rregaddr]=rfin;
        end
        else $display("***ERROR: register file address setup time violation");
    end

always
    begin
    fork
        cont_IO;
        cont_rf;
        cont_ff;
        cont_alu;
        cont_pc;
        cont_psw;
    join
    end
endmodule // datapath
3. Module: id, File: lib/id.v; Instruction Decoder

/***************************************************************/
* * Module: id *
* * Description: FRISC-F instruction Decoder *
* * FRISC-F Simulator Project: *
* * Copyright Hans J. Greub and RPI, Troy, September 1990 *
* * Modification Log: *
* *********************************************
module id (exc_, stall_, m_stall, hold_, clr_pc, pc_sel, pc_lock, w_addr, s_cc, c_c, e_carry, w_psw, alu_op, imm, idi_hi, op_a, op_b, inv_b, sh_, sh_ex, size_, rf_addr, ff_ra, ff_rb, m_by_a, m_by_b, w_rf,
  sel_d, w_din, w_dout, w_, io_cnt, clk, run_stop, instr, p_m, hit_i, hit_d,
  reset_error, trap_d, trap_i, a_t, e_int, int_, bra_, halt_);
parameter DRI_delay = 400/'tic, REC_delay = 100/'tic;

output exc_; // exception signal
output stall_; // pipeline stall signal
output m_stall; // memory sync
output hold_; // force d_buso and a_bus signals low (tristate)
output clr_pc; // clear pc_ex
output pc_sel; // select pc_de/pc_dw
output pc_lock; // lock pc_ex, pc_d1, pc_d2, pc_dw
output w_addr; // write address register (for IO instr)
output s_cc; // set condition code
output [3:0] c_c; // condition code
output e_carry; // enable carry input from PSW
output w_psw; // write ALU result to PSW
output [1:0] alu_op; // ALU operation code { ADD,AND,XOR,OR }
output [15:0] imm; // immediate constant for ALU
output idi_hi; // load immediate constant into upper half word of imm
  register
output [1:0] op_a; // select ALU operand A { rf, rf, pc, psw }
output op_b; // select immediate constant for ALU operand B
output inv_b; // invert ALU operand B
output sh_; // the output of the shifter is ALU operand A
output sh_ex; // extended shift operation
output [1:0] size_; // ALU operation size { Byte, Half Word, Word }
output [4:0] rf_addr; // Register file Address
output [1:0] ff_ra; // feed forward operand A { rf, res_dw, res_d1, res_d2 }
output [1:0] ff_rb; // feed forward operand B { rf, res_dw, res_d1, res_d2 }
output m_by_a; // feed forward ALU result for operand A
output m_by_b; // feed forward ALU result for operand B
output w_rf; // write register file
output sel_d; // select res_dw,din_dw for register write operation
output w_din; // write din dw register
output w_dout; // write dout d1 register
output w_; // write data cache
output [2:0] io_cnt; // i/o control signals
output p_m; // protected state signal

input clk; // System clock
input run_stop; // run/stop signal for four phase generator
input [31:0] instr; // instruction bus input
input hit_i; // instruction cache hit
input hit_d; // data cache hit
input reset_; // CPU reset signal
input error_; // System Error
input trap_d; // data cache trap
input trap_i; // instruction cache trap
input a_t; // arithmetic trap
input e_int; // enable interrupt
input int_; // interrupt signal
input bra_; // branch signal
input halt_; // cpu halt signal

// Instruction Opcodes

'define ALU 2'b10
'define ALUSH 3'b101

'define ADD 5'b10000
'define ADDSH 5'b10100
'define AND 5'b10001
'define ANDSH 5'b10101
'define XOR 5'b10010
'define XORSH 5'b10110
'define OR 5'b10111
'define ORSH 5'b10111

'define LD 5'b11000
'define LDSH 5'b11100
'define ST 5'b11010
'define STSH 5'b11110
'define SETPSW 5'b11011
'define JMP 5'b11111

'define NOOP 5'b00000
'define BRA 5'b01111
'define MPSW 3'b001
'define MPSWADD 5'b00100
'define MPSWAND 5'b00101
'define MPSWXOR 5'b00110
'define MPSWOR 5'b00111
'define GET 5'b01100
'define GETLPC 5'b01000

'define JMP_TRAP 32'b11111101011110001000000000000000

// default instruction after trap
// 32'b11111'1010'1111'0001'0000'0000'0000'0000

// JMP never

// Instruction Classes (recognized by the function match() )

'define ALU_CLASS 0 // {ADD,ADD,AND,ANDSH,XOR,XORSH,OR,ORSH}
'define IO_CLASS 1 // {LD,LDSH,ST,STSH}
'define LD_CLASS 2 // {LD,LDSH}
'define ST_CLASS 3 // {ST,STSH}
'define BRA_CLASS 4 // (JMP, BRA)
'define GET_CLASS 5 // (GET, GETL)

// Check if Opcode matches the specified instruction class
function match;
input [4:0] opcode;
input [2:0] instr;
begin
  case (instr)
    'ALU_CLASS: match=(opcode[4:3]=='ALU);
    'IO_CLASS : match=(opcode=='LD || opcode=='ST || opcode=='LDSH || opcode=='STSH);
    'LD_CLASS : match=(opcode=='LD || opcode=='LDSH);
    'ST_CLASS : match=(opcode=='ST || opcode=='STSH);
    'BRA_CLASS:match=(opcode=='JMP || opcode=='BRA);
    'GET_CLASS:match=(opcode=='GET || opcode=='GETLPC);
    default begin
      $display("ERROR: unknown instruction class; match(%h,%h)",opcode,instr);
      match=0;
    end
  endcase
endfunction

// Register Declarations

reg debug; // dummy register
reg pcunlock_ex,pcunlock_ex_o,pcunlock_d1; // latch

// valid input flags for pipeline stages {I1,I2,DE,EX,D1,D2,DW}
reg v_i1,v_i2,v_de,v_ex,v_d1,v_d2,v_dw;

// valid output flags for pipeline stages {I1,I2,DE,EX,D1,D2,DW}
reg v_i1_o,v_i2_o,v_de_o,v_ex_o,v_d1_o,v_d2_o,v_dw_o;

// input flag; valid result in res register
reg v_res_de,v_res_ex,v_res_d1,v_res_d2,v_res_dw;

// output flag; valid result in res register
reg v_res_d1_o,v_res_ex_o,v_res_d1_o,v_res_d2_o,v_res_dw_o;

// instruction registers for pipeline stages {DE,D1,D2,DW}
reg [31:0] i_de,i_ex,i_d1,i_d2,i_dw;
reg [31:0] i_de_o,i_ex_o,i_d1_o,i_d2_o,i_dw_o;

// tag for din_dw register & valid flag for tag field
reg [4:0] tag;
reg v_tag;

wire [2:0] trap_vector; // trap vector from trap handler
reg setup; // set protected state
reg at_d1,at_d1_o,atrap; // arithmetic trap pipeline reg
reg stFap;

// internal CPU control signals
reg flush; // pipeline flush signal
reg flush_de;
reg flush_i2;
reg flush_i1;
reg stall; // pipeline stall signal
reg stalli; // pipeline stall request due to instruction cache miss
reg stalld; // pipeline stall request due to data cache miss
reg clrpm; // return from protected state
reg halt_cpu; // halt signal
reg ioindw; // load/store instruction in DW stage
wire trap; // trap signal

// Pipeline Stage Communication Signals
reg load_in_d2;

// Source Operand A and B addresses for feed forward logic
wire [4:0] srca;
wire [4:0] srcb;

// Feed Forward Signals for d1,d2,dw Stages
reg ff_res_ex_a,ff_res_ex_b;
reg ff_res_d1_a,ff_res_d1_b;
reg ff_res_d2_a,ff_res_d2_b;

// Input Signals Receivers
wire #REC_delay bra = bra_;
wire #REC_delay hitd = hit_d;
wire #REC_delay hiti = hit_i;
wire #REC_delay reset = reset_;
wire #REC_delay int = int_;
wire #REC_delay e_int = e_int_;
wire #REC_delay at = a_t;
wire #REC_delay dtrap = trap_d;
wire #REC_delay itrap = trap_i;
wire #REC_delay error = error_;
wire #REC_delay halt = halt_;

// Output Signal Latches with I/O Drivers
reg [4:0] rfaddr;
assign #DRI_delay rf_addr = rfaddr;
reg [15:0] imm;
assign #DRI_delay imm = imm;
reg ldihi;
assign #DRI_delay ldi_hi = ldihi;
reg opb;
assign #DRI_delay op_b = opb;
reg invb;
assign #DRI_delay inv_b = invb;
reg ecarry;
assign #DRI_delay e_carry = ecarry;
reg [1:0] opa;
assign #DRI_delay op_a = opa;
reg [1:0] aluop;
assign #DRI_delay alu_op = aluop;
reg [3:0] cc;
assign #DRI_delay c_c = cc;
reg scc;
assign #DRI_delay s_cc = scc;
reg pcSEL;
assign #DRI_delay pc_sel = pcSEL;
reg clrpc;
assign #DRI_delay clr_pc=clrpc;
reg sh;
assign #DRI_delay sh_=sh;
reg shex;
assign #DRI_delay sh_ex=shex;
reg [1:0] size;
assign #DRI_delay size_=size;
reg waddr;
assign #DRI_delay w_addr=waddr;
reg wdout;
assign #DRI_delay w_dout=wdout;
reg wdin;
assign #DRI_delay w_din=wdin;
reg wpsw;
assign #DRI_delay w_psw=wpsw;
reg wrf;
assign #DRI_delay w_rf=wrf;
reg seld;
assign #DRI_delay sel_d=seld;
reg mbya;
assign #DRI_delay m_by_a=mbya;
reg mbyb;
assign #DRI_delay m_by_b=mbyb;
reg [1:0] ffra;
assign #DRI_delay ff_ra=ffra;
reg [1:0] ffrb;
assign #DRI_delay ff_rb=ffrb;
wire pm;
assign #DRI_delay p_m=pm;
reg hold;
assign #DRI_delay hold_=hold;
reg pclock;
assign #DRI_delay pc_lock=pclock;
wire exc;
assign #DRI_delay exc_=flush;
assign #DRI_delay stall_=stall;
reg stalled,stalled_o;
assign #DRI_delay m_stall=stalled;
reg w;
assign #DRI_delay w_=w;
reg [2:0] iOcnt;
assign #DRI_delay io_cnt=iOcnt;

// Internal Signals
wire p1,p2,p3,p4;

event end_of_cycle;

// Function for Instruction Field Extraction

function SHORT;
input [31:0] instr;
SHORT=instr[31];
endfunction

function [4:0] OPCODE;
input [31:0] instr;
OPCODE=instr[31:27];
endfunction

function SH;
input [31:0] instr;
SH=instr[29];
endfunction

function [1:0] ALUOP;
input [31:0] instr;
ALUOP=instr[28:27];
endfunction

function [1:0] OPA;
input [31:0] instr;
OPA=instr[26:25];
endfunction

function INVB;
input [31:0] instr;
INVB=instr[24];
endfunction

function [4:0] DEST;
input [31:0] instr;
DEST=instr[23:19];
endfunction

function [3:0] CC;
input [31:0] instr;
CC=instr[23:20];
endfunction

function [1:0] EXE;
input [31:0] instr;
EXE=instr[18:17];
endfunction

function [1:0] SIZE;
input [31:0] instr;
SIZE=instr[18:17];
endfunction

function OPB;
input [31:0] instr;
OPB=instr[16];
endfunction

function [2:0] IOCNT;
input [31:0] instr;
IOCNT=instr[15:13];
endfunction

function SCC;
input [31:0] instr;
SCC=instr[15];
endfunction
function RTN;
input [31:0] instr;
RTN=instr[19];
endfunction

function SPM;
input [31:0] instr;
SPM=instr[17];
endfunction

function LDH;
input [31:0] instr;
LDH=instr[18];
endfunction

function SHEX;
input [31:0] instr;
SHEX=instr[14];
endfunction

function C;
input [31:0] instr;
C=instr[13];
endfunction

function [4:0] SRCA;
input [31:0] instr;
begin
SRCA=instr[12:8];
end
endfunction

function [4:0] SRCB;
input [31:0] instr;
begin
SRCB=instr[7:3];
end
endfunction

function [7:0] SIMM;
input [31:0] instr;
SIMM=instr[7:0];
endfunction

function [15:0] LIMM;
input [31:0] instr;
LIMM=instr[15:0];
endfunction

// Function for Instruction Disassembly

// Convert Register Address to 3 char string
function [8*3:1] REG;
input [4:0] addr;
case (addr)
  00:REG="R0 ";
  01:REG="R1 ";
  02:REG="R2 ";
endcase
endfunction
03: REG="R3 ";
04: REG="R4 ";
05: REG="R5 ";
06: REG="R6 ";
07: REG="R7 ";
08: REG="R8 ";
09: REG="R9 ";
10: REG="R10 ";
11: REG="R11 ";
12: REG="R12 ";
13: REG="R13 ";
14: REG="R14 ";
15: REG="R15 ";
16: REG="R16 ";
17: REG="R17 ";
18: REG="R18 ";
19: REG="R19 ";
20: REG="R20 ";
21: REG="R21 ";
22: REG="R22 ";
23: REG="R23 ";
24: REG="R24 ";
25: REG="R25 ";
26: REG="R26 ";
27: REG="R27 ";
28: REG="R28 ";
29: REG="R29 ";
30: REG="R30 ";
31: REG="R31 ";
endcase
defunction

// Convert Source Operand A as specified in instruction to 5bit string (no shift)
defunction [8*5:1] MSRCA;
input [31:0] instr;
begin
case (OPA(instr))
  'b00: begin
    if (instr[31]) MSRCA=(REG(SRCA(instr))," ");
    else MSRCA=(REG(DEST(instr))," ");
  end
  'b01: MSRCA=" 0 ";
  'b10: MSRCA="PSW ";
  'b11: MSRCA=(OPCODE(instr)==`GETLPC) ? "LPC ":"CPC ";
endcase
defunction

// Convert Source Operand A as specified in instruction to 5bit string (shift)
defunction [8*5:1] MSRCASH;
input [31:0] instr;
begin
case (OPA(instr))
  'b00: MSRCASH=(REG(SRCA(instr)),"*4");
  'b01: MSRCASH=(REG(SRCA(instr)),"*2");
  'b10: MSRCASH=(REG(SRCA(instr)),"/2");
'b11: MSRCASH=(REG(SRCA(instr)),"/4");
endcase
end
endfunction

// Convert Nibble (4bits) to hex digit
function [8*1:1] HEX_DIGIT;
input [3:0] nibble;
begin
  case (nibble)
    'h0: HEX_DIGIT="0";
    'h1: HEX_DIGIT="1";
    'h2: HEX_DIGIT="2";
    'h3: HEX_DIGIT="3";
    'h4: HEX_DIGIT="4";
    'h5: HEX_DIGIT="5";
    'h6: HEX_DIGIT="6";
    'h7: HEX_DIGIT="7";
    'h8: HEX_DIGIT="8";
    'h9: HEX_DIGIT="9";
    'ha: HEX_DIGIT="a";
    'hb: HEX_DIGIT="b";
    'hc: HEX_DIGIT="c";
    'hd: HEX_DIGIT="d";
    'he: HEX_DIGIT="e";
    'hf: HEX_DIGIT="f";
  endcase
end
endfunction

// Convert Operand B as specified in instruction to 3 char string
function [8*4:1] MSRCB;
input [31:0] instr;
begin
  case (OPB(instr))
    'd0: MSRCB="","REG(SRCB(instr))");
    'd1: MSRCB=" ",HEX_DIGIT(instr[7:4]),HEX_DIGIT(instr[3:0])");
  endcase
end
endfunction

// Convert Operand B as specified in instruction to 3 char string
function [8*4:1] STOB;
input [31:0] instr;
begin
  STOB=" ",HEX_DIGIT(instr[2:0])");
end
endfunction

function [8*4:1] SSRCB;
input [31:0] instr;
begin
  if (INVB(instr))
    case (OPB(instr))
      'd0: SSRCB="-","REG(SRCB(instr))");
      'd1: SSRCB="- ",HEX_DIGIT(instr[7:4]),HEX_DIGIT(instr[3:0])");
    endcase
  else

case (OPB(instr))
  'b0: SSRCB="",REG(SRCB(instr));
  'b1: SSRCB=" ",HEX_DIGIT(instr[7:4]),HEX_DIGIT(instr[3:0]));
endcase
endfunction

// Convert 16bit Literal in Instruction to 5 char string
function [8*5:1] LIT16;
input [31:0] instr;
LIT16=" ",HEX_DIGIT(instr[15:12]),HEX_DIGIT(instr[11:8]),
       HEX_DIGIT(instr[7:4]),HEX_DIGIT(instr[3:0]));
endfunction

function [8*1:1] SIGN;
input [31:0] instr;
beg
if (INVb(instr)) SIGN="-";
else SIGN="+";
end
endfunction

function [8*1:1] POL;
input [31:0] instr;
beg
if (INVb(instr)) POL="-";
else POL=" ";
end
endfunction

// Convert Condition Code as specified in instruction to 3 char string
function [8*3:1] COND;
input [31:0] instr;
case (CC(instr))
  'b0000:COND="le";
  'b0001:COND="gt";
  'b0010:COND="lt";
  'b0011:COND="ge";
  'b0100:COND="ls";
  'b0101:COND="hi";
  'b0110:COND="\~C";
  'b0111:COND="\~C";
  'b1000:COND="\~N";
  'b1001:COND="\~N";
  'b1010:COND="\~Z";
  'b1011:COND="\~Z";
  'b1100:COND="\~V";
  'b1101:COND="\~V";
  'b1110:COND="\~l";
  'b1111:COND="\~0";
default COND="\~CC";
endcase
endfunction

// Disassemble Instruction into a 22 char string
function [8*23:1] itos;
input [31:0] instr;
begin
  case (OPCODE(instr))
      'ADD
        :itos="ADD", REG(DST(instr))," =
      ','MSRCA(instr),SIGN(instr),MSRCB(instr)," ");
      'ADDSH
        :itos="ADDSH", REG(DST(instr))," =
      ','MSRCA(instr),SIGN(instr),MSRCB(instr)," ");
      'AND
        :itos="AND", REG(DST(instr))," =
      ','MSRCA(instr)," &", SSRCB(instr)," ");
      'ANDSH
        :itos="ANDSH", REG(DST(instr))," =
      ','MSRCA(instr)," &", SSRCB(instr)," ");
      'XOR
        :itos="XOR", REG(DST(instr))," =
      ','MSRCA(instr)," ^", SSRCB(instr)," ");
      'XORSH
        :itos="XORSH", REG(DST(instr))," =
      ','MSRCA(instr)," ^", SSRCB(instr)," ");
      'OR
        :itos="OR", REG(DST(instr))," =
      ','MSRCA(instr)," |", SSRCB(instr)," ");
      'ORSH
        :itos="ORSH", REG(DST(instr))," =
      ','MSRCA(instr)," |", SSRCB(instr)," ");
      'LD
        :itos="LD", REG(DST(instr))," =
      ','MSRCA(instr),SIGN(instr),MSRCB(instr)," ");
      'LDSH
        :itos="LDSH", REG(DST(instr))," =
      ','MSRCA(instr),SIGN(instr),MSRCB(instr)," ");
      'S T
        :itos = { "" S T
      ','REG(SRCB(instr))"," =","MSRCA(instr),SIGN(instr),STOB(instr)," ");
      'S T S H
        :itos = { "" S T
      ','REG(SRCB(instr))"," =","MSRCA(instr),SIGN(instr),STOB(instr)," ");
      'SETPSW:itos="SETPSW PSW = ",MSRCA(instr),SIGN(instr),MSRCB(instr)};
      'JMP
        :itos="JMP", COND(instr)," PC =
      ','MSRCA(instr),SIGN(instr),MSRCB(instr)];
      'NOOP
        :itos="NOOP",
      'BRA
        :itos = "BRA", COND(instr)," PC =
      ','MSRCA(instr),SIGN(instr),LIT16(instr)];
      'MPSWADD:itos="MPSW PSW = ",MSRCA(instr),SIGN(instr),LIT16(instr)];
      'MPSWAND:itos="MPSW PSW = ",MSRCA(instr)," &", POL(instr),LIT16(instr)];
      'MPSWXOR:itos="MPSW PSW = ",MSRCA(instr)," ^", POL(instr),LIT16(instr)];
      'MPSWOR:itos="MPSW PSW = ",MSRCA(instr)," |", POL(instr),LIT16(instr)];
      'GET
        :itos="GET", REG(DST(instr))," =
      ','MSRCA(instr),SIGN(instr),LIT16(instr)];
      'GETLPC
        :itos="GETLPC", REG(DST(instr))," =
      ','MSRCA(instr),SIGN(instr),LIT16(instr)];
      default :itos="ILLEGAL",
  endcase
end

// Pipeline Stage Controllers

// Data Write STAGE CONTROLLER (DW)
// int in: load_in_d2,v_d2_o,v_res_d2_o,v_res_d2,flush,STALL,pm
// src_a,src_b,p1,p2,p3,p4
// ext in: hit_d(p2)
//         e x t o u t : wdin(p1),seld(p3),mbya(p3),mbyb(p3),rfaddr(p3),wrf(p4),pc_lock(p4)
// int out: ioindw(p1),stalld(p2)
task dw_stage;
reg [4:0] opcode;
reg load_in_dw;

begin
  wait(p1);
  if (~stall) begin // copy signals from d1 stage
    i_dw = i_d2_o;
    v_dw = v_d2_o;
  end
  opcode = OPCODE(i_dw);
  load_in_dw = match(opcode, 'LD_CLASS);
  ioin_dw = match(opcode, 'IO_CLASS);
  // enable write of data input register
  if (~stall | stalld) wdin = load_in_dw;
  else load_in_dw = 0;
  // if load is in dw stage set v_tag and update tag address for feed forward
  // when load enters from d2 stage v_tag is 0 since input data register
  // was just written into register file
  if (load_in_dw) begin
    v_tag = 1;
    tag = DEST(i_dw);
  end
  wait(p2);
  // stall pipeline if cache miss on I/O instruction
  stalld = hit & match(opcode, 'IO_CLASS);
  wait(p3);
  // check if memory bypass is necessary for operand A or B
  if (v_tag & srca == tag) mbya = 1;
  else mbya = 0;
  if (v_tag & srcb == tag) mbyb = 1;
  else mbyb = 0;
  // if load is in d2 stage write data input register into register file
  if (load_in_d2 & v_tag) begin
    rfaddr = tag;
    seld = 1;
  end
  else begin
    seld = 0;
    rfaddr = DEST(i_d2);
  end
  wait(p4);
  // enable register file write if data input register or res_d2 are ready
  if (seld | (v_res_d2 & flush & stall)) wrf = 1;
  else wrf = 0;
  // clear v_tag if v_tag was written into the register file
  // or load did not complete -> flush load in dw on exception!
  // or ALU instruction is overwriting the register <tag>
  i
  v_tag = 0;
end
task d2_stage;
reg [4:0] opcode;
begin
  // Data 2 STAGE CONTROLLER (D2)
  // int in:  v_d1_o, v_res_d1_o, flush, stall, srca, srcb, p1, p2, p3, p4
  // int out: ff_res_d2_a(p2), ff_res_d2_b(p2), load_in_d2(p1), v_res_d2
  //         atrap(p1), strap(p1), clrpm(p4), setpm(p4)

wait(p1)
if (~stall) begin // copy signals from d1 stage
  i_d2=i_d1_o;
  v_d2=v_d1_o;
  v_res_d2=v_res_d1_o;
  atrap=at_d1_o;
end
opcode=OPCODE(i_d2);
//#1 if (opcode[4:2]=="MFSW") atrap=1; //$$$$$ debug only!!!!!
load_in_d2=match(opcode, 'LD_CLASS);
wait(p2)
if (v_res_d2 && srca==DEST(i_d2)) ff_res_d2_a=1;
else ff_res_d2_a=0;
if (v_res_d2 && srcb==DEST(i_d2)) ff_res_d2_b=1;
else ff_res_d2_b=0;
wait(p4);
if (~flush)
  begin
    if (match(opcode, 'BRA_CLASS) && RTN(i_d2)) clrpm=1;
    else clrpm=0;
    if (opcode[4:2]=="MFSW && SPM(i_d2)") setpm=1;
    else setpm=0;
    // software trap if BRA taken and ST bit set
    if (opcode=='BRA && i_d2[16]) strap=1;
    else strap=0;
  end
  v_d2_o=v_d2_o&~flush;
  v_res_d2_o=v_d2_o&v_res_d2;
  i_d2_o=(v_d2_o) ? i_d2 : 0;
end
endtask

// Data 1 STAGE CONTROLLER (D1)
// int in: v_ex_o, v_res_ex_o, flush, stall, srca, srcb, p1, p2, p3, p4
// ext in: at(p4);
// int out: ff_res_d1_a(p2), ff_res_d1_b(p2)
task d1_stage;
begin
  wait(p1)
  if (~stall) begin // copy signals from ex stage
    i_d1=i_ex_o;
    v_d1=v_ex_o;
    v_res_d1=v_res_ex_o;
  end
  wait(p2)
  if (v_res_d1 && srca==DEST(i_d1)) ff_res_d1_a=1;
  else ff_res_d1_a=0;
  if (v_res_d1 && srcb==DEST(i_d1)) ff_res_d1_b=1;
  else ff_res_d1_b=0;
  wait(p4);
  v_d1_o=v_d1_o&~flush;
  v_res_d1_o=v_d1_o&v_res_d1;
  i_d1_o=(v_d1_o) ? i_d1 : 0;
  if (d1_o=(match(OPCODE(i_d1), 'ALU_CLASS) & v_d1_o && SIZE(i_d1)! = 2'b11) ? at
     : 0;
end
endtask
// EXECUTION STAGE CONTROLLER (EX)
// ext in: bra
// int in: v_de_o, v_res_de_o, flush, stall, src_a, src_b, p1, p2, p3, p4
// ext out: scc(p4), wpw(p4), waddr(p3), dout(p4), clrpc(p4),
// w(p4), iocnt(p4);
// flush_de(p4), flush_i2(p4), flush_i1(p4), v_ex_o(p4), v_dest_de(p4),
// i_ex_o(p4), v_res_ex_o(p4), ff_res_ex_a(p2), ff_res_ex_b(p2)
// task ex_stage;
// reg[4:0] opcode;
// reg[1:0] exe;
begin
wait(p1);
if (~stall) begin // copy signals from de stage
    i_ex = i_de_o;
    v_ex = v_de_o;
    v_res_ex = v_res_de_o;
end
wait(p2);
opcode=OPCODE(i_ex);
if (v_res_ex && src_a==DEST(i_ex)) ff_res_ex_a=1;
else ff_res_ex_a=0;
if (v_res_ex && src_b==DEST(i_ex)) ff_res_ex_b=1;
else ff_res_ex_b=0;
wait(p3);
waddr=match(opcode, 'IO_CLASS)&~flush;
wait(p4);
if (~stall) begin
    if (match(opcode, 'IO_CLASS)) iocnt=IOCNT(i_ex);
    w=match(opcode, 'ST_CLASS)&~flush;
end
wpw = (opcode==SETPSW || opcode[4:2]==MPSW) && ~flush) ? 1 : 0;
wout= match(opcode, 'ST_CLASS)&~flush;
if (match(opcode, 'ALU_CLASS)&~flush) scc=SCC(i_ex);
else scc=0;
if (match(opcode, 'BRA_CLASS)) begin
    // flush instruction in de, i2, i1 stages according to the following table
    // EXE | flush_de | flush_i2 | flush_i1
    // 00 | bra | bra | bra
    // 01 | 0 | bra | bra
    // 10 | 0 | 0 | bra
    // 11 | 0 | 0 | 0
    exe=EXE(i_ex);
    flush_de=(exe==2'b00) ? bra : 0;
    flush_i2=(exe[1]) ? 0 : bra;
    flush_i1=(exe==2'b11) ? 0 : bra;
if (debug) begin
    if (flush_de) $display("->flushing de stage");
    if (flush_i2) $display("->flushing i2 stage");
    if (flush_i1) $display("->flushing i1 stage");
end
if (~bra) begin
    if (debug) $display("disabling st and clrpm");
    if (opcode=='BRA) i_ex[16]=0; //disable software trap if no branch
    i_ex[19]=0; // disable return from protected mode if no branch
end
if (i_ex[19]) pcunlock_ex=1;
end
else begin
  flush_de=0;
  flush_i2=0;
  flush_i1=0;
end

if (flush) pcunlock_ex=0;
pclock=(flush&ioindw)|(pma~pcese~pcunlock_ex);
// clear pc if flushed instruction was in ex stage
cldrca~v_ex&(~pma)pcunlock_ex);
// generate output signal for dl stage
v_ex_o=v_ex&~flush;
i_ex_o=(v_ex_o) ? i_ex : 0;
v_res_ex_o=v_ex_o&v_res_ex;
end
detask

// DECODE STAGE CONTROLLER (DE)
// ext in: instr, hit_i
// i n t i n
// v_i2, flush, flush_de, stall, exc, trap_vector, ioindw(p1), p1, p2, p3, p4
// ext out: imm(p3), rghi(p3), rfaaddr(p1, p2), invb(p3), opa(p3), opb(p3),
// sh(p3), shex(p3), aluop(p4), ecarry(p4), cc(p4), pcese(p3)
// i n t o u t
// stalli(p2), v_de_o(p4), v_res_de_o(p4), i_de_o(p4), srca(p1), srcb(p1)
task de stage;
reg [4:0] opcode;
begin
wait(p1);
#REC_delay
if (~stall) v_de=v_i2_o;
if (~stall|stall) i_de = (v_de) ? instr:'JMP TRAP; // i_de = msff
  rfaaddr = SRCB(i_de);
  opcode=OPCODE(i_de);
  force srcb=SRCB(i_de);
  if (SHORT(i_de)) force srca=SRCA(i_de);
else force srca=DEST(i_de); // srca=dest for long immediate instructions
wait(p2);
  stallli=hitii&v_de;
  rfaaddr = srca;
wait(p3);
  opb=OPB(i_de)|~SHORT(i_de);
if (LDH(i_de) & (match(opcode, 'GET_CLASS') | (opcode[4:2]=='MPSW)))
    ldihi=1;
else ldihi=0;
if (exc) begin
  imm=(trap_vector, 2'b00);
  i_de[4:0]=imm[4:0]; // update IMM field to get correct trace on display
  if (debug) $display("->trap_vector=%b", trap_vector);
end
else
  if (SHORT(i_de)) begin
    imm=SIMM(i_de);
    if (match(opcode, 'ST_CLASS')) begin // stores have only 3bit imm
      imm=imm&7;
      opb=1;
    end
end
end
else imm=LIMM(i_de);
op=OPA(i_de);
invb=INVB(i_de);
if (opcode[4:2]==ALUSH || opcode==LDSH || opcode==STSH) begin
  sh=1;
  shex=SHEX(i_de) & (opcode[4:2]==ALUSH);
end
else begin
  sh=0;
  shex=0;
end
pcsel=opcode==GETLPC);
wait(p4);
// lock pc if exception pending and no load/store instruction in dw
// unlock pc if GETLPC executed in protected mode
if (match(opcode,ALU_CLASS) || opcode[4:2]==MPSW) aluop=ALUOP(i_de);
else aluop=0;
if (match(opcode,ALU_CLASS)) begin
  ecarr=C(i_de);
  size=SIZE(i_de);
end
else begin
  ecarr=0;
  size=2'b11;
end

end

if (exc) i_de[23:20]=4'b1110; // wait for flush_de set CC
if (~stall) if (~flush|flush_de) & (opcode==BRA || opcode==JMP)) begin
  cc=CC(i_de);
end
else cc=4'b1111;
v_de_o=(v_de~flush_de~flush)|exc;
i_de_o=(v_de_o) ? i_de : 0;
v_res_de_o=v_de_o & (match(opcode,ALU_CLASS)|match(opcode,GET_CLASS))
end
task endtask

// INSTRUCTION FETCH 2 STAGE CONTROLLER (I2)
// int in: v_i2_o, flush, flush_i2, stall, p1, p2, p3, p4
// int out: v_i2_o(p4)
task i2_stage;
begin
  wait(p4);
  if (~stall) v_i2=v_i1_o;
  wait(p4);
  #1 v_i2_o=v_i2~flush~flush_i2;
end
task endtask

// INSTRUCTION FETCH 1 STAGE CONTROLLER (I1)
// int in: flush, flush_i1, p1, p2, p3, p4
// int out: v_i1_o(p4)
task i1_stage;
begin
  wait(p1) v_i1=~flush;
  wait(p4);
  #1 v_i1_o=v_i1~flush~flush_i1;
end
detask

// four phase clock generator
phase_gen phg (p1,p2,p3,p4,clk,run_stop);
state_cont cpu(exc, pm, reset_cpu, trap, setpm, clrpm, halt_cpu, p1);
trap_encod te(trap, trap_vector, reset_cpu, reset, exc, error, dtrap,
              atrap, strap, itrap, int, eint, pm, p1, p2);

initial
begin
  v_tag=0;
  v_res_dw=$random;
  load_in_d2=$random;
  iocnt=$random&3;
end

always
begin
  wait (p2)
  halt_cpu=halt;
  wait (p3)
    // encode ffra and ffrb signals
    if (ff_res_ex_a) ffrb=2'b11;
    else if (ff_res_d1_a) ffrb=2'b10;
    else if (ff_res_d2_a) ffrb=2'b01;
    else ffrb=2'b00;
    if (ff_res_ex_b) ffrb=2'b11;
    else if (ff_res_d1_b) ffrb=2'b10;
    else if (ff_res_d2_b) ffrb=2'b01;
    else ffrb=2'b00;
    stalled_o=stalled;
    stalled=halt_cpu|((stalld|stalli)&~trap);
    stall=(stalled|stalled_o)&~trap;
    if (stall) $display("->Pipeline Stall (p3)");
    #1 flush=trap;
    if (flush) $display("->Pipeline Flush (p3)");
    hold=halt_cpu;
end
always
begin
fork
  i1_stage;
  i2_stage;
  de_stage;
  ex_stage;
  d1_stage;
  d2_stage;
  dw_stage;
  @(negedge p4) -> end_of_cycle;
join
end
endmodule
4. Module: cache, File: lib/cache.v; Cache Memory

/*****************************************************************************/
* Module: cache
*
* Description: 256x32bit Pipelined Memory
*   Controller has three pipeline stages:
*     READ (RD), WRITE (WR), OLD (OLD)
* *
* FRISC-F Simulator Project:
* Copyright Hans J. Greub and RPI, Troy, September 1990
* *
* Modification Log:
*
*****************************************************************************/

module cache (data_out,cnt_dat,byte_addr,hit_,page_fault,error_,
exc_,data_in,addr_,v_addr,m_stall,w_,io_cnt,inc_,clk,run_stop);

parameter Size=255,Width=31,Rec_delay=100/'tic,Dri_delay=400/'tic;
parameter Access_delay=3000/'tic,Type=0;
output [Width:0] data_out; // memory data output
output [1:0] cnt_dat; // data alignment control for data input
output [1:0] byte_addr; // byte address lowest two bits of address
output hit_; // cache hit
output page_fault; // page fault signal
output error_; // cache error (parity,alignment)
input exc_; // exception signal -> reset cache control
input [Width:0] data_in; // memory data input
input [Width:0] addr_; // memory data input
input v_addr; // valid address input signal
input m_stall; // pipeline stall signal
input w_; // memory write signal
input [2:0] io_cnt; // memory access & alignment control input
input inc_; // increment address
input clk; // system clock signal
input run_stop; // four phase clock startup signal

//define resources
integer index;
reg [8*7:1] type; // dummy register for cache type
reg [Width:0] memory[Size:0]; //dummy memory to set breakpoints, simulate cache misses, and page faults
reg [1:0] dummy_con[Size:0];
reg v_rd,v_wr,v_old,v_d; //valid control data registers
reg [Width:0] addr_rd,addr_wr,addr_old,addr_d; //address registers
reg [2:0] cnt_rd,cnt_wr,cnt_old,cnt_d; //access control registers
reg w_rd,w_wr,w_old,w_d; //write signal control registers
reg sticky_rd,sticky_wr; //sticky bit
reg [Width:0] data_wr; // data input register
reg [Width:0] memIn; // memory input register
wire [(Width-2):0] address=addr_rd[Width:2];
wire [Width:0] #Access_delay memOut=memory[address];
reg debug; // dummy control register
reg advanced;
event cachemiss,missandfault; //cache miss and cache miss with page fault events
// Input Receivers
wire #Rec_delay exc=exc;
wire #Rec_delay stall=m_stall;
wire [Width:0] #Rec_delay data_in=data_in;
wire [Width:0] #Rec_delay addr=addr_
wire #Rec_delay vaddr=v_addr;
wire #Rec_delay w=w_
wire [2:0] #Rec_delay iocnt=io_cnt;

// Output Drivers
reg pagefault;
assign #Dri_delay page_fault=pagefault;
reg error;
assign #Dri_delay error_=error;
reg [Width:0] dataout;
assign #Dri_delay data_out=dataout;
reg hit;
assign #Dri_delay hit_=hit;
assign #Dri_delay cnt_dat=cnt_wr[1:0];
assign #Dri_delay byte_addr=addr_wr[1:0];

// four phase clock generator
phase_gen phg (p1,p2,p3,p4,clk,run_stop);

/* task advance_pipeline */
task advance_pipeline;
begin
advanced=1;
addr_old=addr_wr;
addr_wr=addr_rd;
cnt_old=cnt_wr;
cnt_wr=cnt_rd;
w_old=w_wr;
w_wr=w_rd;
v_old=v_wr;
v_wr=v_rd;
sticky_wr=sticky_rd;
end
endtask

initial
begin
debug=1;
addr_rd=$random&127;
cnt_rd=$random&3;
cnt_wr=$random&3;
// initialize dummy control memory
for (index=0;index<=Size;index=index+1) dummy_con[index]=0;
@ (inc ) type="I-Cache" // Remote PC enabled -> Instruction Cache
else type="D-Cache";
end

// cache pipeline stage controller for Read Stage
always
begin @ (posedge p3)
advanced=0;
end
if (stall) begin // if (stall) reload operation in old stage
    /* copy last pipeline register into dummy register */
    v_d = v_old;
    w_d = w_old;
    cnt_d = cnt_old;
    addr_d = addr_old;
    /* circular pipeline advance
    first restart access that bombed out
    then wait till hit goes high and advance a second time */
    if (!(~sticky_rd & (sticky_rd & hit)) & ~sticky_wr) begin
        advance_pipe;
        sticky_rd = 1; // mark data as sticky
        v_rd = v_d;
        addr_rd = addr_d;
        cnt_rd = cnt_d;
        w_rd = w_d;
    end
    end
else begin // no stall
    advance_pipe;
    if (exc) begin // if (exception) flush all stages
        v_rd = 0;
        v_wr = 0;
        v_old = 0;
        w_old = 0;
        w_wr = 0;
        w_rd = 0;
        error = 0;
        pagefault = 0;
        hit = 1;
        disable miss;
        disable fault;
        sticky_rd = 0;
    end
    sticky_rd = 0; // no stall -> clear sticky bits
    sticky_wr = 0;
    if (vaddr) begin // if (vaddr & stall) load next operation
        v_rd = 1;
        addr_rd = addr;
        cnt_rd = iocnt;
        w_rd = w;
    end
    else if (inc) begin // if (inc &~ stall & vaddr) generate next operation
        v_rd = 1;
        addr_rd = (addr_rd & hfffffff) + 4; // generate next word address (RPC)
        cnt_rd = 3'b010; // word operation
        w_rd = 0; // read operation
    end
    else begin
        v_rd = 0;
        w_rd = 0;
    end
end // end no stall
if (w_wr & ~stall) data_wr = datain; // update data input if write is in WR stage
if (debug) $display("->%s address: addr_rd=%h datain=%h cnt_rd=%b w_rd=%b",type,addr_rd,datain,cnt_rd,w_rd);

if (v_wr&hit)
case (dummy_con[addr_wr[Width:2]])
  2'b00: hit=1;
  2'b01: begin
    hit=1; //set hit=1 in cache miss is disabled
    ->cachemiss;
    end
  2'b10: begin
    hit=1;
    ->missandfault;
    end
  2'b11: begin
    hit=1;
    $display("%s Breakpoint at address: %h",type,addr_wr[Width:2]);
    $stop;
    end
  default $display("***ERROR: in %s dummy memory @'h%h%b",type,addr_wr[Width:2],dummy_con[addr_wr[Width:2]]);
endcase

//latch data output
#1 if (advanced)
  if (hit) dataout=memout;
  else dataout=32'hxxxxxxxxx; // mark cache miss!
  if (debug) $display("->%s output = %h",type,dataout);

if (v_wr) begin // check address alignment
case (cnt_wr[1:0])
  2'b01: if (addr_wr[0]!=0)
    $display("***Warning:bad address alignment for 16bit data in %s",type);
  2'b10: if (addr_wr[1:0]!=0)
    $display("***Warning:bad address alignment for 32bit data in %s",type);
  endcase
end // end check address

// check if address is stable for write
if (w_wr&v_rd~stall) $display("***Error: Write followed by another Memory Operation in %s!",type);

if (hit&w_wr) begin // if (hit & write) write memory
  memin=memory[addr_wr[Width:2]];
  case (cnt_wr[1:0])
  2'b00: begin // byte access
case (addr_wr[1:0])
    2'b00: memin[7:0]=data_wr[7:0];
    2'b01: memin[15:8]=data_wr[15:8];
    2'b10: memin[23:16]=data_wr[23:16];
    2'b11: memin[31:24]=data_wr[31:24];
  endcase
end
2'b01: begin // half word access
        error=1;
        case (addr_wr[1])
            1'b0: memin[15:0]=data_wr[15:0];
            1'b1: memin[31:16]=data_wr[31:16];
        endcase
    end
2'b10: begin
        error=1;
        memin=data_wr;
    end
    default $display("***Error: Unknown Write Access Code in %s",type);
endcase
    wait(p2)
    if (debug) $display("-->overwriting location %h : old=%h, new=%h in %s",
        addr_wr[Width:2],memory[addr_wr[Width:2]],memin,type);
    memory[addr_wr[Width:2]]=memin;
end // end write memory
end

always
begin :miss
    integer i;
    @(cachemiss) // wait for a cache miss event
    hit=0;
    i=3+($random&3); // random stall
    $display("-->%s miss; stall for %0d cycles at %h",type,i,addr_wr[31:2]);
    while (i) @(posedge p2) i=i-1;
    hit=1;
    repeat (9) @(posedge p3); //prevent triggering for 9 cycles
end

always
begin :fault
    integer i;
    @(missandfault) // wait for a pagefault event
    hit=0;
    i=3+($random&3); // random stall
    if (debug) $display("-->%s miss; stall for %0d cycles, and page fault at %h",type,i,addr_wr[31:2]);
    while (i) @(posedge p3) i=i-1;
    pagefault=1;
    repeat (9) @(posedge p3); //prevent triggering for 6 cycles
end
endmodule
5. Module: align_{mi}, File: lib/align_{mi}.v; Data Aligner MI

*******************************************************************************
* * Module: align_min *
* * Description: Data Alignment at Memory Input *
* * FRTSC-F Simulator Project: *
* * Copyright Hans J. Greub and RPI, Troy, September 1990 *
* * Modification Log: *
*******************************************************************************
module align_min (data_out, data_in, cnt_dat, clk, run_stop);
parameter Rec_delay=1000/\tic, Dri_delay=400/\tic;
output [31:0] data_out; // output data
input [31:0] data_in; // input data
input [1:0] cnt_dat; // (byte, half word, word) select
input clk; // system clock
input run_stop; // startup signal for four phase clock

wire [31:0] #Rec_delay datain=data_in;
wire [1:0] #Rec_delay cntdat=cnt_dat;

reg [7:0] byte_i3, byte_i2, byte_i1, byte_i0; // four input bytes
reg [7:0] byte_o3, byte_o2, byte_o1, byte_o0; // four output bytes

assign #Dri_delay data_out=(byte_o3, byte_o2, byte_o1, byte_o0);

// four phase clock generator
phase_gen phg (p1, p2, p3, p4, clk, run_stop);

always
begin
@ (posedge p4)
  (byte_i3, byte_i2, byte_i1, byte_i0)=datain;
@ (posedge p1)
  case (cntdat)
    2’b00: (byte_o3, byte_o2, byte_o1, byte_o0)=(byte_i0, byte_i0, byte_i0, byte_i0);
    2’b01: (byte_o3, byte_o2, byte_o1, byte_o0)=(byte_i1, byte_i0, byte_i0, byte_i0);
    2’b10: (byte_o3, byte_o2, byte_o1, byte_o0)=(byte_i3, byte_i2, byte_i1, byte_i0);
    default: begin
      (byte_o3, byte_o2, byte_o1, byte_o0)=(byte_i3, byte_i2, byte_i1, byte_i0);
      $display("***ERROR: unknown alignment code: cntdat=%b", cntdat);
    end
  endcase
end
endmodule
6. Module: align_mout, File: lib/align_mout.v; Data Aligner MO

/** ******************************************************/
* Module: align_mout
* Description: Data Alignment at Memory Output
* FRISC-F Simulator Project:
* Copyright Hans J. Greub and RPI, Troy, September 1990
* Modification Log:
* ******************************************************/
module align_mout (data_out, data_in, byte_addr, cnt_dat, clk, run_stop);
// align data from processor to memory
parameter Rec_delay=100/'tic, Dri_delay=400/'tic;
output [31:0] data_out; // output data
input [31:0] data_in; // input data
input [1:0] byte_addr; // byte address
input [1:0] cnt_dat; // (byte, half word, word) select
input clk; // system clock
input run_stop; // startup signal for four phase clock

wire [31:0] #Rec_delay datain=data_in;
wire [1:0] #Rec_delay cntdat=cnt_dat;
wire [1:0] #Rec_delay byteaddr=byte_addr;

reg [7:0] byte_i3, byte_i2, byte_i1, byte_i0; // four input bytes
reg [7:0] byte_o3, byte_o2, byte_o1, byte_o0; // four output bytes

assign #Dri_delay data_out=(byte_o3, byte_o2, byte_o1, byte_o0);

// four phase clock generator
phase_gen phg (p1, p2, p3, p4, clk, run_stop);

always @ (posedge p4)
begin
{byte_i3, byte_i2, byte_i1, byte_i0}=datain;
// $display("->align_mout: datain=", {byte_i3, byte_i2, byte_i1, byte_i0});

@ (posedge p1)
case (cntdat)
2'b00: begin // byte access shift & sign extend
  case (byteaddr)
    2'b00: {byte_o3, byte_o2, byte_o1, byte_o0}=
    {byte_i0[7]}?24'hffffff:24'h0, byte_i0);
    2'b01: {byte_o3, byte_o2, byte_o1, byte_o0}=
    {byte_i1[7]}?24'hffffff:24'h0, byte_i1);
    2'b10: {byte_o3, byte_o2, byte_o1, byte_o0}=
    {byte_i2[7]}?24'hffffff:24'h0, byte_i2);
    2'b11: {byte_o3, byte_o2, byte_o1, byte_o0}=
    {byte_i3[7]}?24'hffffff:24'h0, byte_i3);
    default $display("bad byte address in align_mout");
  endcase
endcase

end
end
2'b01:begin // half word access shift & sign extend
case (byteaddr[1])
1'b0: {byte_o3,byte_o2,byte_o1,byte_o0} =
    {byte_i1[7]}?16'hfff:16'h0,byte_i1,byte_i0);
1'b1: {byte_o3,byte_o2,byte_o1,byte_o0} =
    {byte_i3[7]}?16'hfff:16'h0,byte_i3,byte_i2);
default $display("bad byte address [1] in align_mout");
endcase
end

2'b10: {byte_o3,byte_o2,byte_o1,byte_o0} = {byte_i3,byte_i2,byte_i1,byte_i0};
default begin
    {byte_o3,byte_o2,byte_o1,byte_o0} = {byte_i3,byte_i2,byte_i1,byte_i0};
    $display("Error unknown alignment code cnt_dat=$b",cnt_dat);
end
endcase
end
endmodule
7. Module: alu, File:lib/alu.v; ALU

/***************************************************************************/
* Module: alu
*
* Description: 32bit ALU with four functions {ADD,AND,XOR,OR}
* N,Z,C,V flags are set according to size
* \{8bit,16bit,32bit,32bit\}
* FRISC-F Simulator Project:
* Copyright Hans J. Greub and RPI, Troy, September 1990
*
* Modification Log:
*
/***************************************************************************/
module alu (N,C,Z,V,\text{res,func,size,opa,opb,\text{cin}});
parameter delay=2000/\text{tic-1,del_flag=500/\text{tic,msb=31};}
output \text{N; // Negative Flag}
output \text{C; // Carry Flag}
output \text{Z; // Zero Flag}
output \text{V; // Overflow Flag}
output \text{[31:0] \text{res; // result}}
input \text{[31:0] \text{opa; // operand A}}
input \text{[31:0] \text{opb; // operand B}}
input \text{[1:0] \text{func; \text{function select \{ADD,AND,XOR,OR\}}}}
input \text{[1:0] \text{size; // operand size \{byte, half word, word, word\}}}
input \text{\text{cin; // carry in}}

reg \text{[31:0] \text{dummy,\text{res}};}
reg \text{Z,N,C,V,enableV;}
reg \text{[31:0] \text{cdummy; // carry out for msb-1}}

always @\text{(func or opa or opb or cin or size)}
begin
\text{\#delay}
\text{case (size)}
\text{2'b00: \text{cdummy = opa[6:0]+opb[6:0]+cin;}}
\text{2'b01: \text{cdummy = opa[14:0]+opb[14:0]+cin;}}
\text{2'b10: \text{cdummy = opa[30:0]+opb[30:0]+cin;}}
\text{2'b11: \text{cdummy = opa[30:0]+opb[30:0]+cin;}}
\text{endcase}
\text{case (func)}
\text{2'b00: begin // ADD}
\text{enableV = 1;}
\text{res=opa+opb+cin;}
\text{case (size)}
\text{2'b00: \{C,dummy[7:0]\} = opa[7:0]+opb[7:0]+cin;}
\text{2'b01: \{C,dummy[15:0]\} = opa[15:0]+opb[15:0]+cin;}
\text{2'b10: \{C,dummy\} = opa+opb+cin;}
\text{2'b11: \{C,dummy\} = opa+opb+cin;}
\text{endcase}
\text{end}
\text{2'b01: begin // AND}
\text{C = 0;}
\text{res = opa \& opb;}
\text{enableV=0;}
\text{end}
2'b10: begin // XOR
    C = 0;
    res = opa ^ opb;
    enableV=0;
    end
2'b11: begin // OR
    C = 0;
    res = opa | opb;
    enableV=0;
    end
default: begin
    C=1'b1;
    res='bx;
    V=1'b1;
    end
endcase
#1
case (size)
2'b00:begin
    V = enableV & ( C ^ cdummy[7]);
    N = res[7];
    Z = #(del_flag-1) ~(|res[7:0]);
    end
2'b01:begin
    V = enableV & ( C ^ cdummy[15]);
    N = res[15];
    Z = #(del_flag-1) ~(|res[15:0]);
    end
2'b10:begin
    V = enableV & ( C ^ cdummy[31]);
    N = res[31];
    Z = #(del_flag-1) ~(|res);
    end
2'b11:begin
    V = enableV & ( C ^ cdummy[31]);
    N = res[31];
    Z = #(del_flag-1) ~(|res);
    end
endcase
endmodule
8. Module: shifter, File: lib/shifter.v; Shifter

/***************************************************************************/
* *
* Module: shifter
* *
* Description: 32 bit shifter
* four function shifter ( SHL2, SHL1, SHR1, SHR2 )
* ov flag set according to operand size (8bit, 16bit, 32bit, 32bit)
* *
* FRISC-F Simulator Project:
* Copyright Hans J. Greub and RPI, Troy, September 1990
* *
* Modification Log:
* */
******************************************************************************/
module shifter(ov, res, shc, shex, size, in33, in32, in, inm1, inm2);
parameter sh_del=200/'tic, ov_del=200/'tic;
output ov; // shifter overflow
output [31:0] res; // shifter output
input [1:0] shc; // shifter control ( SHL2, SHL1, SHR1, SHR2 )
input shex; // extended shift operation
input in33; // input bit 33 for down shift SHR2
input in32; // input bit 31 for down shifts ( SHR1, SHR2 )
input [31:0] in; // input bits 31-0
input [1:0] size; // size of operand
input inm1; // input bit -1 for up shifts ( SHL1, SHL2 )
inout inm2; // input bit -2 for up shift [ SHL2 ]
reg [31:0] dw; // dummy word
reg [15:0] halfword; // dummy half word
reg [7:0] byte; // dummy byte
wire [35:0] result; // output bus
wire v;
wire &ov_del ov=v;
wire [31:0] &sh_del res = result[33:2];
always @(shc or shex or size or in)
begin
  case (shc)
    'b00: begin
      force result=(in[31:0], shex&inm1, shex&inm2)<<2;
    endcase
    'b01: begin
      force result=(in[31:0], shex&inm1, 1'b0)<<1;
    endcase
  end
  case (size)
    2'b00: force v=~shex&(in[7:5]=3'b000 & in[7:5]=3'b111);
    2'b01: force v=~shex&(in[15:13]=3'b000 & in[15:13]=3'b111);
    2'b10: force v=~shex&(in[31:29]=3'b000 & in[31:29]=3'b111);
    2'b11: force v=~shex&(in[31:29]=3'b000 & in[31:29]=3'b111);
  endcase
end
'b10: begin
    dw=( (shex?in32:in[31]),in[31:0],1'b0,1'b0)>>1;
    case (size)
        2'b000: force
            result=(dw[31:8], (shex?in32:in[7]),in[7:0],1'b0,1'b0)>>1;
        2'b001: force
            result=(dw[31:16], (shex?in32:in[15]),in[15:0],1'b0,1'b0)>>1;
            2'b10: force result=( (shex?in32:in[31]),in[31:0],1'b0,1'b0)>>1;
            2'b11: force result=( (shex?in32:in[31]),in[31:0],1'b0,1'b0)>>1;
        endcase
        force v=0;
    end

'b11: begin
    dw=( (shex?{in33,in32}:{in[31],in[31]}),in[31:0],1'b0,1'b0)>>2;
    case (size)
        2'b000: force
            result=(dw[31:8], (shex?{in33,in32}:{in[7],in[7]}),in[7:0],1'b0,1'b0)>>2;
        2'b001: force
            result=(dw[31:16], (shex?{in33,in32}:{in[15],in[15]}),in[15:0],1'b0,1'b0)>>2;
        2'b10: force
            result=( (shex?{in33,in32}:{in[31],in[31]}),in[31:0],1'b0,1'b0)>>2;
        2'b11: force
            result=( (shex?{in33,in32}:{in[31],in[31]}),in[31:0],1'b0,1'b0)>>2;
        endcase
        force v=0;
    end
endcase
end

endmodule
9. Module: `trap_enc`, File: `lib/trap_enc.v`; Trap Encoder

```verbatim
trap_enc(in, out, priority, signal, trap, vector)
```

Description: trap signal encoder

- `priority`: signal index
- `signal`: signal type
- `trap`: trap signal
- `vector`: trap vector

- 7: highest reset 3'b000
- 6: restart 3'b001
- 5: error 3'b010
- 4: dtrap 3'b011
- 3: atrap 3'b100
- 2: strap 3'b101
- 1: itrap 3'b110
- 0: lowest int 3'b111

- in protected mode any trap other than reset gets mapped to restart trap
- reset signal clears protected mode (pm) signal

FRISC-F Simulator Project:
Copyright Hans J. Greub and RPI, Troy, September 1990

Modification Log:

module trap_enc(trap, trap_vector, reset_cpu, reset, exc, error, dtrap, atrap, strap, itrap, int, eint, pm, p1, p2);
output trap; // trap signal for CPU state machine
output [2:0] trap_vector; // trap vector
output reset_cpu; // reset signal for CPU state machine
input exc; // exception state
input reset; // reset signal
input error; // error signal
input dtrap; // data trap signal
input atrap; // arithmetic trap signal
input strap; // software trap signal
input itrap; // instruction trap
input int; // interrupt signal
input eint; // enable interrupt signal
input pm; // protected mode
input p1, p2; // clock phase signals
reg trap;
reg [2:0] trap_vector;
reg [2:0] vector;
reg reset, reset_cpu, reset_trap;
reg int1, int_trap;
reg error1, error_trap;

always
begin @ (posedge p2)
    // latch trap signals during p1
    reset_trap = reset_cpu;
    reset_cpu = reset;
```

```
int_trap=init1;
int1=init;
if (int) $display("int=1 eint=%b",eint);
error_trap=error1;
error1=error;
if (exc) trap_vector=vector; // latch trap vector generated p2-p1
if (exc) trap=0;
else if (reset_trap) begin
    trap=1;
    vector=3'b000;
    $display("->reset trap pending");
end
else if (error) begin
    trap=1;
    vector=3'b010;
    $display("->error trap pending");
end
else if (dtrap) begin
    trap=1;
    vector=3'b011;
    $display("->data trap pending");
end
else if (atrap) begin
    trap=1;
    vector=3'b100;
    $display("->arithmetic trap pending");
end
else if (strap) begin
    trap=1;
    vector=3'b101;
    $display("->software trap pending");
end
else if (itrap) begin
    trap=1;
    vector=3'b110;
    $display("->instruction trap pending");
end
else if (int_trap&eint&~pm) begin
    trap=1;
    vector=3'b111;
    $display("->user interrupt pending");
end
else trap=0;
if (pm&trap) vector=3'b001;
end
10. Module: state_co, File: lib/state_co; State Controller

/****************************************************
 */
*/
*/ Module: state_cont
*/
*/ Description: CPU state controller - states (normal, exception, protected)
*/
*/
*/ FRISC-F Simulator Project:
*/ Copyright Hans J. Greub and RPI, Troy, September 1990
*/
*/ Modification Log:
*/
******************************************************************************/

module state_cont (exc, pm, reset_cpu, trap, setpm, rtn, halt_cpu, p1);
output exc;  // exc pipeline state
output pm;  // protected state
input reset_cpu; // state machine reset signal
input trap;  // trap signal
input setpm; // set protected state
input rtn;  // return to normal state
input halt_cpu; // halt cpu state machine
input p1;    // clock phase 1

reg exc, pm;
reg normal, exception, protected;
event normal_state;
event exception_state;
event protected_state;

always @(posedge reset_cpu)
begin
    disable normal_block;
    disable exception_block;
    disable protected_block;
    pm'=0;
    @ (negedge reset_cpu) ->normal_state;
end

always @normal_state
begin:
    normal_block
    normal=1;
    $display("->normal state");
    pm=0;
    while (normal) begin
        normal=0;
        @(posedge p1) if (halt_cpu) normal=1;
        if (trap) ->exception_state;
        else if (setpm) ->protected_state;
        else normal=1;
    end
end

always @exception_state
begin:
    exception_block
    $display("->exception mode");
    exception=1;
pm=1;
exc=1;
while (exception) begin
    exception=0;
    @(posedge pl) if (halt_cpu) exception=1;
    else ->protected_state;
end
end

always @protected_state
begin :protected_block
    $display("->protected mode");
    protected=1;
    pm=1;
    exc=0;
    while (protected) begin
        protected=0;
        @(posedge pl) if (halt_cpu) protected=1;
        else if (trap) ->exception_state;
        else if (rtn) ->normal_state;
        else protected=1;
    end
end
endmodule
11. Module: clock_gen, File: lib/clock_gen.v; Clock Generator

 /******************************************************************************************
 * * Module: clock_gen
 * * Description: clock signal generator
 * * FRISC-F Simulator Project:
 * * Copyright Hans J. Greub and RPI, Troy, September 1990
 * * Modification Log:
 * ******************************************************************************************/
 module clock_gen (clk, run_stop);
 // Clock Generator
 // outputs run_stop signal to start up phase_gen modules
 parameter phase = 1000/'tic;
 output run_stop, clk;
 reg run_stop;
 reg clk;

 initial begin
    run_stop = 0;
    clk = 1;
    #(2*phase) run_stop = 1;
    #(2*phase) clk = 0;
    while (1) begin
        begin
            phase clk = 1;
            phase clk = 0;
        end
    end
endmodule
12. Module: phase_gen, File: lib/phase_gen.v; Phase Generator

/*******************************************************************************/
* *
* Module: phase_gen
* *
* Description: Four Phase Clock Generator
* to startup phase_gen:1) set run_stop low and clock high
* 2) wait one clock period
* 3) set run_stop high
* 4) wait one clock period
* 5) start clock pulses
* *
* FRISC-F Simulator Project:
* Copyright Hans J. Greub and RPI, Troy, September 1990
* *
* Modification Log:
* *
/*******************************************************************************/
module phase_gen(p1, p2, p3, p4, clk, run_stop);

input clk, run_stop;
output p1, p2, p3, p4;
reg p1, p2, p3, p4;
reg clock;
integer cycle;
reg [2:0] phase;

initial begin
  {p1, p2, p3, p4} = $random;
end

always begin
  wait (~run_stop) begin
    #1 {p1, p2, p3, p4} = 'b1000;
    phase = 1;
    cycle = 0;
    assign clock = 1;
    disable clocking;
    wait (run_stop);
    if (!clk) $display("bad clock startup sequence");
    wait (clk) assign clock = clk;
  end
  begin : clocking
    @(negedge clock) begin
      p1 = 0;
      p2 = 1;
      phase = 2;
    end
    @(posedge clock) begin
      p2 = 0;
      p3 = 1;
      phase = 0;
    end
  end
endmodule
```verilog
module counter;

reg phase; // Initialize the phase variable

 @(negedge clock) begin
    phase=3;
    p3=0;
    p4=1;
    phase=4;
end

 @(posedge clock) begin
    p4=0;
    p1=1;
    cycle=cycle+1;
    phase=1;
end

endmodule
```
B. FRISC-F Assembler Source, File: fas.c

/***************************************************************************/
Assembler for FRISC Processor

- assembles the input source file, generates a listing on standard output, and generates instruction and data output files for the VERILOG system task $read_memh()
- supports symbolic references and labels (two pass assembler)
- source file and command line options are case insensitive
- works together with the UNIX m4 macro processor (use -s option on m4)

Copyright Hans J. Greub & Rensselaer Polytechnic Institute
September 1990
/******************************************************************************/

#define MSDOS
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <string.h>
#include <stdio.h>
#include <ctype.h>
#include <malloc.h>
#include <stdlib.h>
#endif

#ifndef MSDOS
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <string.h>
#include <stdio.h>
#include <ctype.h>
#include <malloc.h>
#endif

#ifndef ANSI
/* declare function prototypes for ANSI C compiler */
char *get_token(char*);
char *get_line(char*);
void error(char*);
void print_buffer(char*);
void print_header(void);
void dis_as(long);
char *get_all(char*);
char *read_directive(char*);
char *get_opa(char*);
char *get_opb(char*);
char *get_dest(char*);
char *get_instruction(char*);
char *get_directive(char*);
char *get_label(char*);
char *get_condition(char*);
char *get_options(char*);
int find_symbol(char*);
int add_symbol(char*);
void print_instr(int);
#else
char *get_token();
char *get_line();
void error();
void print_buffer();
void print_header();
void dis_as();
char *get_all();
char *readDirective();
char *get_opa();
char *get_opb();
char *get_dest();
char *get_instruction();
char *get_directive();
char *get_label();
char *get_condition();
char *get_options();
int find_symbol();
int add_symbol();
void print_instr();
#endif

/* allocate space for line & string buffers */
#define BUFFER 128
char buffer[BUFFER];
char line_buf[BUFFER];
char ofile_name[BUFFER]="frisc";
char obuf[BUFFER];
/* declare global pointer to buffers */
char *buf,*bp,*end;
/* last token buffer */
char last_token[BUFFER/2];
/* global buffer for token */
char token[BUFFER/2];
char *comment;
/* pointer to input file name */
char *file_name;
char *endptr;
/* pointer to output file name */
char *ofile=ofile_name;

/* variables for instruction fields */
long sh;
long shex;
long opa;
long opb;
long srca;
long srb;
long dest;
long imm;
long ldhi;
int symbol; /* dummy field to flag symbolic literals */
long spm;
long rtm;
long size;
long strap;
long io;
long ex;
long lpc;
long cc;
long scc;
long carry;
long invb;
long aluop;
long code;
long value;

int ip=0; /* instruction pointer */
int ap=1; /* alias pointer */
int dp=0; /* data pointer */
int lp=0; /* label pointer */
int move=0;
long instr;

#define IP_MIN 0
#define DP_MIN 0
#define AP_MAX 512 /* maximum number of symbolic variables and labels */

int DP_MAX=256; /* size of data memory in words */
int IP_MAX=256; /* size of instruction memory in words */

#define CR \n'
/* set default register for stack pointer */
int SP=31;
/* set default register for temporary destination register for CMP instr. */
int RTMF=30;
int IO_DEFAULT=2;
int symbol_table;
FILE *infile; /* file pointer to input file */
FILE *outfile; /* file pointer to output files */
int eof=0;

/* destination register codes for special registers */
#define CPC 32
#define PSW 33

/* 5 bits instruction opcodes */
#define SETPSW 0x1b
#define ADD 0x10
#define AND 0x11
#define XOR 0x12
#define OR 0x13
#define LD 0x18
#define ST 0x1a
#define JMP 0x1f
#define NOOP 0
#define BRA 0x0f
#define MPSW 0x04
#define PSWADD 0x04
#define PSWAND 0x05
#define PSWXOR 0x06
#define PSWOR 0x07
#define GET 0x0c
#define GETLPC 0x08
/* starting positions for fields within instruction */
#define CODE 27
#define OPA 25
#define OPB 16
#define DEST 19
#define SRCA 8
#define SRCB 3
#define EX 17
#define CC 20
#define LDHI 18
#define SCC 15
#define SPM 17
#define RTN 19
#define SIZE 17
#define INV 24
#define SHEX 14
#define STRAP 16
#define CARRY 13
#define IOCNT 13
#define SH 29
#define LPC 29

/* weight of option enable flags within dummy option enable field */
#define EC 1
#define ESCC 2
#define EEX 4
#define EIO 8
#define ESIZE 16
#define EST 32
#define ESPM 64
#define ERTN 128
#define ELDHI 256
#define ESHEX 512
long enable;
long option;

typedef struct Instruction {
    int symbol;
    char *name;
    long code;
} Instruction;

#define UNDEF 0
#define DLABEL 1
#define ILABEL 2

typedef struct Alias {
    int type;
    char *name;
    long address;
} Alias;

typedef union DATA {
    char b[4];
    short s[2];
    long w;
} DATA;
int errors=0;
int line_number=0;
int debug=0;

Alias alias[AP_MAX+1]; /* array for symbolic literals and labels */
DATA *wdata;      /* pointer to data array */
Instruction *instruction; /* pointer to instruction array */

/* macro to align pointer p on a b byte boundary */
#define align(p,b) (((p)+(b)-((p)+((p)+(b)-((p)+(b))))))

/* string copy */
#ifdef SYS5
char *strdup(s)
char *s;
{
    char *n,*p;
    if (s==(char *)NULL) {
        fprintf(stderr, "***ERROR: call of _strdup with null pointer as argument
";
    return s;
}
p=n=(char*)malloc(strlen(s)+1);
if (n==(char *)NULL) error("out_of_memory");
while (*(++p)= *s) s++;
return n;
}
#endif
/* main program output=stdout and frisc.dat and frisc.ins */

void main(argc,argv)
int argc;
char *argv[];
{
int i,b,ins,symb;
long offset;
#ifdef SYS5
if (sbrk(1024*1024)==-1) fprintf(stderr,"***ERROR: allocating memory
");
#endif
if (argc<2) {
    info:
    fprintf(stderr,"FRISC-F ASSEMBLER\n"");
    fprintf(stderr,"COPYRIGHT Hans J. Greub & Rensselaer Polytechnic
");
    fprintf(stderr,"usage: fas <file> <options>\n"");
    fprintf(stderr,"-symbol ;list symbol table\n"");
    fprintf(stderr,"-debug ;run disassembler\n"");
    fprintf(stderr,"-tmp=r<0.31> ;set temporary register for\n"");
    fprintf(stderr,"-sp=r<0.31> ;set stack pointer\n"");
    fprintf(stderr,"-ic=<size> ;set instruction memory size\n"");
    fprintf(stderr,"-dc=<size> ;set data cache size\n"");
    fprintf(stderr,"-o=<name> ;set output file name\n"");
    exit(1);
}
/* open input file */
infile=fopen(argv[1],"r");
if (infile==(FILE*)NULL) {
    fprintf(stderr,"*** ERROR; could not open input file %s\n",argv[1]);
    exit(1);
}
/* set file name for error messages */
file_name=strdup(argv[1]);
/* process command line options */
i=2;
while (i<argc) {
    if (!strcmp(argv[i],"-sp",3) || !strcmp(argv[i],"-SP",3)) {
        if (*argv[i]+3)!='=' goto info;
        if (*argv[i]+4)!='r' && *(argv[i]+4)!='R') goto info;
        b=atoi(argv[i]+5);
        if (b>31) fprintf(stderr,"***ERROR; stack pointer = r[0..31]\n\n");
        SP=b+31;
    }
    else if (!strcmp(argv[i],"-rtmp",5) || !strcmp(argv[i],"-RTMP",5)) {
        if (*argv[i]+5)!='=' {
            printf("*(argv[i]+5)=%c",*(argv[i]+5));
            goto info;
        }
        if (*argv[i]+6)!='r' && *(argv[i]+6)!='R') goto info;
        b=atoi(argv[i]+7);
        if (b>31) fprintf(stderr,"***ERROR; temporary register = r[0..31]\n\n");
        RTMP=b+31;
    }
    else if (!strcmp(argv[i],"-o",3) || !strcmp(argv[i],"-O",3)) {
        ofile=strdup(argv[i]+3);
    }
    else if (!strcmp(argv[i],"-debug") || !strcmp(argv[i],"-DEBUG"))
        debug=1;
    else if (!strcmp(argv[i],"-symbol") || !strcmp(argv[i],"-SYMBOLE"))
        symbol_table=1;
    else if (!strcmp(argv[i],"-dc",4) || !strcmp(argv[i],"-DC",4))
        DP_MAX=atoi(argv[i]+4,&endptr,NULL);
    else if (!strcmp(argv[i],"-ic",4) || !strcmp(argv[i],"-IC",4))
        IP_MAX=atoi(argv[i]+4,&endptr,NULL);
    else {
        fprintf(stderr,"\n***ERROR; unknown option %s\n\n",argv[i]);
        goto info;
    }
    i++;
}
/* allocate memory for instruction and data cache size */
instruction=(Instruction*)malloc(IP_MAX*sizeof(Instruction));
if (instruction==(Instruction*)NULL) {
    fprintf(stderr,"***ERROR; could not allocate memory for %d instructions\n",IP_MAX+1);
    exit(1);
}
data=(DATA*)malloc(DP_MAX*sizeof(DATA));
if (data==(DATA*)NULL) {
    fprintf(stderr,"***ERROR; could not allocate memory for %d data items\n",DP_MAX+1);
    exit(1);`
/* initialize instruction and data arrays */
for (i=0;i<DP_MAX;i++) wdata[i].w=0;
for (i=0;i<AP_MAX;i++) alias[i].type=UNDEF;
DP_MAX=DP_MAX<<2;
for (i=0;i<IP_MAX;i++) {
    instruction[i].code=0;
    instruction[i].name="";
    instruction[i].symbol=0;
}
/* start processing of input file */
buf=get_line(buffer);
while (!eof) {
    buf=get_token(buf);
    if (*buf==':') { /* process label */
        buf=get_label(buf);
        if (*buf==':') buf=get_token(buf);
        buf=get_instruction(buf);
    } else if (*buf=='/') { /* process assembler directive */
        buf=get_directive(buf);
    } else if (*buf=='/' && *buf=='*') { /* ignore C style comments */
        while (!eof && (*buf=='*' && *buf=='/')) buf=get_token(buf);
        buf=get_token(buf);
    }
/* process line and file sync info inserted by m4 preprocessor */
    if m4 is run with -s flag
else if (*buf=='#') {
    buf=get_token(buf);
    if (strcmp(token,"line")) {
        error("expected #line");
        buf=get_line(buffer);
    }
else {
    buf=get_token(buf);
    line_number=atoi(token);
    if (*buf==':') {
        buf=get_token(buf);
        buf=get_token(buf);
        file_name=strdup(token);
        while (*buf!='#') buf=get_token(buf);
        buf=get_token(buf);
    }
}
else {
    buf=get_instruction(buf); /* process instruction */
}
}
if (symbol_table) { /* print symbol table */
    printf("== INSTRUCTION LABELS --\n\n");
    for (ins=1;ins<ap;ins++)
        if (alias[ins].type==ILABEL) printf("@'h%%.8lx
%s\n",alias[ins].address,alias[ins].name);
    printf("n-- DATA LABELS --\n\n");
    for (ins=1;ins<ap;ins++)

if (alias[ins].type==DLABEL) printf("%08lx : %08lx
",alias[ins].address,wdata[alias[ins].address>>2].w,alias[ins].name);
printf("\n");

for (ins=1;ins<ap;ins++)
if (alias[ins].type==UNDEF) {
  fprintf(stderr,"\n***ERROR: label %s is referenced but not defined\n",alias[ins].name);
  errors++;
}

/* open data output file (default frisc.dat) */
ofile=strcat(ofile_name,".dat");
outfile=fopen(ofile,"w");
if (outfile==(FILE*)NULL) {
  fprintf(stderr,"***ERROR: could not open output file: %s\n",ofile);
  exit(1);
}

/* dump data array in VERILOG format for $readmemh system task */
printf("DATA MEMORY (all other locations are initialized to zero)\n\n");
for (ins=0;ins<(DP_MAX>>2);ins++) {
  if (wdata[ins].w) printf("%08lx %08lx\n",ins<<2,wdata[ins].w);
  fclose(outfile);
  *
}

/* open instruction output file (default frisc.ins) */
ofile=fopen(ofile_name,"w");
if (outfile==(FILE*)NULL) {
  fprintf(stderr,"***ERROR: could not open output file: %s\n",ofile);
  exit(1);
}

/* dump data array in VERILOG format for $readmemh system task */
printf("INSTRUCTION MEMORY\n\n");
for (ins=0;ins<INS_MAX;ins++) {
  if (symb=instruction[ins].symbol) { /* process symbolic literals */
    if (symb<0) { /* label reference */
      offset=alias[-symb].address-ins;
      if (offset>=0) { /* positive branch offset */
        if (instruction[ins].code<0) { /* instruction has 16bit literal */
          if (offset&0xff000000) {
            fprintf(stderr,"ERROR: branch offset overflow; %s\n",instruction[ins].name);
            printf("ERROR: branch offset overflow; %s\n",instruction[ins].name);
          }
          offset=offset&0xfffff;
        } else {
          if (offset&0xffffffff) { /* instruction has 8bit literal */
            fprintf(stderr,"ERROR: branch offset overflow; %s\n",instruction[ins].name);
            printf("ERROR: branch offset overflow; %s\n",instruction[ins].name);
          }
        }
      }
    } else {
      if (offset&0xffffffff) { /* negative branch offset */
        offset= -offset;
      }
    }
  }
}
if (instruction[i].code<0) { /* instruction has 16bit literal */
    if (offset&0xff0000) {
        printf(stderr,"ERROR: literal overflow; %s\n",instruction[i].name);
        printf("ERROR: literal overflow; %s\n",instruction[i].name);
    }
    offset=offset&0xffff;
}
else { /* instruction has 8bit literal */
    if (offset&0xffffffff00) {
        printf(stderr,"ERROR: literal offset overflow; %s\n",instruction[i].name);
        printf("ERROR: literal offset overflow; %s\n",instruction[i].name);
    }
    offset=offset&0xff;
}
instruction[ins].code=instruction[ins].code|((long)1<<INV)|offset;
}
else { /* process symbolic literals */
    offset=alias[symb].address;
    if (instruction[ins].code<0) { /* instruction has 16bit literal */
        if (offset&0xffffffff0000) {
            printf(stderr,"ERROR: literal overflow; %s\n",instruction[i].name);
            printf("ERROR: literal overflow; %s\n",instruction[i].name);
        }
        offset=offset&0xffff;
    }
else { /* instruction has 8bit literal */
        if (offset&0xffffffff00) {
            printf(stderr,"ERROR: literal offset overflow; %s\n",instruction[i].name);
            printf("ERROR: literal offset overflow; %s\n",instruction[i].name);
        }
        offset=offset&0xff;
    }
    instruction[ins].code=instruction[ins].code|offset;
}
/* generate output in VERILOG $readmemh format */
fprintf(outfile,"%4.4x %8.8lx\n",ins,instruction[ins].code);
/* print binary instruction code and corresponding source line */
if (*instruction[ins].name!=(char)NULL) print_instr(ins);
}
exit(errors);

/* get and process assembler directives */
char *get_directive(buf)
char *buf;
{
    long negative;
    if (!strcmp(token,".dp")) {
        if (*buf=='+') {

buf=get_token(buf);
buf=get_token(buf);
    dp=dp+strtol(token,&end,0);
    if (dp>DN_MAX || dp<0) error("data address out of range");
}
else if (*buf	==`-`) {
    buf=get_token(buf);
    buf=get_token(buf);
    dp=dp-strtol(token,&end,0);
    if (dp>DN_MAX || dp<0) error("data address out of range");
}
else if (isdigit(*buf)) {
    buf=get_token(buf);
    dp=strtol(token,&end,0);
    if (dp>DN_MAX || dp<0) error("data address out of range");
}
else error("syntax");
}
else if (!strcmp(token,".ip")) {
    if (*buf	==`+`) {
        buf=get_token(buf);
        buf=get_token(buf);
        ip=ip+strtol(token,&end,0);
        if (ip>IP_MAX) error("instruction address out of range");
    }
    else if (*buf	==`-`) {
        buf=get_token(buf);
        buf=get_token(buf);
        ip=ip-strtol(token,&end,0);
        if (ip<0) error("instruction address out of range");
    }
    else if (isdigit(*buf)) {
        buf=get_token(buf);
        ip=strtol(token,&end,0);
        if (ip>IP_MAX || ip<0) error("instruction address out of range");
    }
    else error("syntax");
}
else if (!strcmp(token,".long")||!strcmp(token,".word")) {
    buf=get_token(buf);
    if (ap!=find_symbol(token)) {
        fprintf(stderr,"***ERROR;symbol %s has multiple definition\n",token);
        errors++;
    }
    alias[ap].name=strdup(token);
    if (alias[ap].type!=UNDEF) error("redefinition of a label");
    alias[ap].type=DLABEL;
    dp=align(dp,4);
    alias[ap].address=dp;
    if (isdigit(*buf)) || *buf	==`-`) do {
        buf=get_token(buf);
        if (*token	==`-`) {
            buf=get_token(buf);
            negative=-1;
        }
    }
    else negative=1;
    if (*token	==`,'`) buf=get_token(buf);
    wdata[dp>>2].w=negative*strtol(token,&end,0);
dp=dp+4;
    if (dp>=DP_MAX) error("data address out of range");
} while (*buf!=(char)NULL && *buf!=';');
ap++;
if (ap>=AP_MAX) error("symbol table overflow");
} else if (!strcmp(token,".short")) {
    buf=get_token(buf);
    if (*token==',') buf=get_token(buf);
    if (ap!=find_symbol(token)) {
        fprintf(stderr,"***ERROR; symbol %s has multiple definition\n",token);
        errors++;
    }
    alias[ap].name=strdup(token);
    if (alias[ap].type!=UNDEF) error("redefinition of a label");
    alias[ap].type=DLABEL;
    dp=align(dp,2);
    alias[ap].address=dp;
    if (isdigit(*buf) || *buf=='-') do {
        buf=get_token(buf);
        if (*token=='-') {
            buf=get_token(buf);
            negative=-1;
        } else negative=1;
        value=strtol(token,&end,0);
    } else negative=1;
    value=strtol(token,&end,0);
    #ifdef SUN
    wdata[dp>>2].s[1-((dp>>1)&2)]=(short)negative*strtol(token,&end,0);
    #else
    wdata[dp>>2].s[(dp>>1)&2]=(short)negative*strtol(token,&end,0);
    #endif
    dp=dp+2;
    if (dp>=DP_MAX) error("data address out of range");
} while (*buf!=(char)NULL && *buf!=';');
ap++;
if (ap>=AP_MAX) error("symbol table overflow");
} else if (!strcmp(token,".char")||!strcmp(token,".byte")) {
    buf=get_token(buf);
    if (ap!=find_symbol(token)) {
        fprintf(stderr,"***ERROR; symbol %s has multiple definition\n",token);
        errors++;
    }
    alias[ap].name=strdup(token);
    if (alias[ap].type!=UNDEF) error("redefinition of a label");
    alias[ap].type=DLABEL;
    alias[ap].address=dp;
    if (isdigit(*buf) || *buf=='-') do {
        buf=get_token(buf);
        if (*token=='-') {
            buf=get_token(buf);
            negative=-1;
        } else negative=1;
        value=strtol(token,&end,0);
    } else negative=1;
    value=strtol(token,&end,0);
    #ifdef SUN
    wdata[dp>>2].b[3-(dp&4)]=(char)negative*strtol(token,&end,0);
    #else
    wdata[dp>>2].b[3-(dp&4)]=(char)negative*strtol(token,&end,0);
    #endif
wdata[dp>>2].b[dp%4]=(char)negative*strtol(token,&end,0);
#endif

dp++;
    if (dp>DP_MAX) error("data address out of range");
} while (*buf!=(char)NULL && *buf!=';');
ap++;
if (ap>AP_MAX) error("symbol table overflow");
#endif
if (!strcmp(token,".str")||!strcmp(token,".string")) {
    buf=get_token(buf);
    if (ap!=find_symbol(token)) {
        fprintf(stderr,"***ERROR:symbol %s has multiple definition\n",token);
        errors++;
    }
    alias[ap].name=strdup(token);
    if (*buf!='"') error("expected quoted string");
    end=buf+1;
    while (*end!='"' && *end!=(char)NULL) end++;
    if (*end==(char)NULL) error("newline in string constant");
    else *end=(char)NULL;
    if (alias[ap].type!=UNDEF) error("redefinition of a label");
    alias[ap].type=DLABEL;
    alias[ap].address=dp;
    buf=buf+1;
    if (*buf!=(char)NULL) do {
        #ifdef SUN
            wdata[dp>>2].b[3-(dp%4)]= *(buf++);#else
            wdata[dp>>2].b[dp%4]= *(buf++);
        #endif
        dp++;
        if (dp>DP_MAX) error("data address out of range");
    } while (*buf!=(char)NULL);
    ap++;
    if (ap>AP_MAX) error("symbol table overflow");
} else if (!strcmp(token,".end") ) eof=1;
else error("unknown directive");
    buf=get_line(buffer);
    return buf;
}

/* find symbol in symbol table */
int find_symbol(string)
    char *string;
{
    int p=1;
    while (p<ap) {*
        if (!strcmp(string,alias[p].name)) return p;
        p++;
    }
    return p;
}

/* get destination field */
char *get_dest(buf)
    char *buf;

buf=get_token(buf);
if (*token=='r') {
    dest=atoi(token+1);
    if (dest>31) error("register number out of range");
} else if (!strcmp(token,"psw")) dest=PSW;
else if (!strcmp(token,"sp")) dest=SP;
else if (!strcmp(token,"cpc")) dest=CPC;
else error("unknown destination");
if (*buf!=='='') error("expected = ");
buf=get_token(buf);
return buf;
}

/* get operand A of instruction */
char *get_opa(buf)
char *buf;
{
    int mult;
srca=0;
sh=0;
opa=0;
lpc=0;
buf=get_token(buf);
if (*token=='r') {
    srca=atoi(token+1);
    if ((srca>31) || (srca<0)) error("register number out of range");
    if ( (*buf=='/' || *buf=='*') && (*(buf+1)=='0' && *(buf+1)<='9') ) {
        if (*buf=='*') mult=1;
        else mult=0;
        sh=1;
        buf=get_token(buf);
    if (*token=='*' || *token=='/') buf=get_token(buf);
    if (mult) {
        if (!strcmp(token,"4")) opa=0;
        else if (!strcmp(token,"2")) opa=1;
        else error("divide constant out of range");
    }
    else {
        if (!strcmp(token,"2")) opa=2;
        else if (!strcmp(token,"4")) opa=3;
        else error("multiply constant out of range");
    }
} else if (!strcmp(token,"sp")) srca=SP;
else if (!strcmp(token,"0")) opa=1;
else if (!strcmp(token,"psw")) opa=2;
else if (!strcmp(token,"cpc")) opa=3;
else if (!strcmp(token,"lpc")) {
    if (code!=GET && !move) error("illegal operand LPC");
    opa=3;
lpc=1;
}
else error("unknown source A");
return buf;
/* get operand B of instruction */
char *get_opb(buf)
char *buf;
{
srcb=0;
opb=0;
imm=0;
buf=get_token(buf);
if (*token=="4" || *token=="~" || *token=="\"" || *token=="&" || *token=="|"
|| *token==",
buf=get_token(buf);
if (*token==")")
buf=get_token(buf);
else if (*token=="r") srcb=atoi(token+1);
else if (isdigit(*token)) {
    opb=1;
    imm=strtol(token,&end,0);
}
else if (*token=="@") {
    buf=get_token(buf);
    symbol=add_symbol(token);
    opb=1;
}
else if (!strcmp(token,"sp")) srcb=SP;
else if (*token=="#") {
    buf=get_token(buf);
    if (isdigit(*token)) {
        opb=1;
        imm=strtol(token,&end,0);
    } else error("expected number after ");
} else error("unknown source B");
return buf;

/* declare separators for get_token() */
char sep[]=" ()];:##-&|"~{"!?\["\]%t@";

/* tokenizer */
char *get_token(buf)
char *buf;
{
    int i,l;
    char ch;
    char *str,*ptr;
    char terminator=(char)255;
terminator=(char)255;
/* printf("get_token("%s")\n",buf); */
l=strlen(sep);
while (*buf==")" buf++;
/* printf("get_token("%s")\n",buf); */
    if (*buf==(char)NULL || *buf==")") {
buf = get_line(buffer);
if (!eof) return get_token(buf);
else {
    if (strcmp(last_token, ".end")) error("unexpected end of file");
    exit(errors+1);
}
}
strncpy(last_token, token);
/* printf(\"Buf=%s\n\",buf); */
str = token;
while (terminator == (char)255) {
    ch = *(buf++);
    if (isascii(ch)) {
        if (isupper(ch)) {
            ch = tolower(ch);
            /* printf(\"ch=%c tolower(*ch)=%c\n\",ch,tolower(ch)); */
        }
    }
    *(str++) = ch;
    ptr = sep;
    for (i = 0; i <= l; i++) if (ch == *(ptr++)) { i = 100; terminator = ch; }
}
if ((str-token) == 1) token[1] = (char) NULL;
else {
    *(--str) = (char) NULL;
    buf--;
}
/* printf("token = %s\n",token); */
while (*buf == '.') buf++;
return buf;
}

char *get_next(buf)
char *buf;
{
    char *end, *res;
    res = fgets(buf, BUFFER, infile);
    line_number++;
    if (res == (char *) NULL) return res;
    end = buf + strlen(buf);
    if (*((--end) == CR) * (end) == (char) NULL;
        while (*(--end) == ' ');
    if (*((end) == '$') {
        res = get_next(end);
        if (res == (char *) NULL) return end;
        else return res;
    }
    return end;
}

/* get next source line */
char *get_line(buf)
char *buf;
{
    char *start, *res;
    res = fgets(buffer, BUFFER, infile);
    /* printf("line=%s\n",buffer); */
line_number++;
if (res==(char*)NULL)
    ecf=1;
    return res;

while (*res!==(char)NULL) {
    switch (*res) {
    case 't': *res=' '; 
                break;
    case 'r': *res=0; 
                break;
    case 'n': *res=0; 
                break;

    res++;
    }
    start=buffer;
    res=start+strlen(start)-1;
    while (*(res--)==(char)NULL) 
    while (*(start--)) start++;
    strncpy(obuf,start,80);
    if (*(start)==';') return get_line(buf);
    else if (*(start==(char)NULL) return get_line(buf);
    return start;
}

/* print error message and line and file info */
void error(s)
char *s;

fprintf(stderr,"*** ERROR on line %d, file %s: %s
",line_number,file_name,s);
fprintf(stderr,"*** %s
",buffer);
printf("*** ERROR on line %d, file %s: %s
",line_number,file_name,s);
printf("*** %s
",obuf);
printf("token = %s, previous token = %s\n",token, last_token);
fprintf(stderr,"token = %s, previous token = %s\n",token, last_token);
errors++;
if (errors>10) exit(errors);
}
/* get and process label */
char *get_label(buf)
char *buf;
{
int i;
i=find_symbol(token);
if (i==ap) alias[ap++].name=strdup(token);
alias[i].address=ip;
if (alias[ap].type!=UNDEF) error("redefinition of a label");
alias[i].type=ILABEL;
buf=get_token(buf);
return buf;
}

/* get condition code for BRANCH and JUMP instructions */
char *get_condition(buf)
char *buf;
{
buf=get_token(buf);
if (!strcmp(token,"_le")) cc=0;
else if (!strcmp(token,"_gt")) cc=1;
else if (!strcmp(token,"_lt")) cc=2;
else if (!strcmp(token,"_ge")) cc=3;
else if (!strcmp(token,"_le")) cc=4;
else if (!strcmp(token,"_ht")) cc=5;
else if (!strcmp(token,"_c") ) cc=6;
else if (!strcmp(token,"_nc") ) cc=7;
else if (!strcmp(token,"_n") || !strcmp(token,"_mi")) cc=8;
else if (!strcmp(token,"_nn") || !strcmp(token,"_pl") ) cc=9;
else if (!strcmp(token,"_z") || !strcmp(token,"_eq") ) cc=10;
else if (!strcmp(token,"_nz") || !strcmp(token,"_ne") ) cc=11;
else if (!strcmp(token,"_v")) cc=12;
else if (!strcmp(token,"_nv") ) cc=13;
else if (!strcmp(token,"_l") ) cc=14;
else if (!strcmp(token,"_0") ) cc=15;
else error("unknown condition code");
return buf;

/* add a symbol to the array, check if it already exists */
int add_symbol(token)
char *token;
{
    int i=1;
    while (i<ap) {
        if (!strcmp(alias[i].name,token)) return i;
        i++;
    }
    alias[i].name=strdup(token);
    ap++;
    if (ap>AP_MAX) error("symbol table overflow");
    return i;
}

/* get options */
/* print error message if option is not enabled */
char *get_options(buf)
char *buf;
{
    buf=get_token(buf);
    if (*token!='/') error("expected /option");
    else buf=get_token(buf);
    if (!strcmp(token,"c") ) {
        if (enable&EC) option=option|EC;
        else error("bad option");
    }
    else if (!strcmp(token,"she") ) {
        if (enable&ESHEX) option=option|ESHEX;
        else error("bad option");
    }
    else if (!strcmp(token,"clrmp") ) {
        if (enable&ERTN) option=option|ERTN;
        else error("bad option");
    }
    else if (!strcmp(token,"setpm") ) {
        if (enable&ESPM) option=option|ESPM;
    }
else error("bad option");
}
else if (!strcmp(token,"byte") || !strcmp(token,"char")) {
    if (enable&ESIZE) size=0;
    else error("bad option");
} else if (!strcmp(token,"short")) {
    if (enable&ESIZE) size=1;
    else error("bad option");
} else if (!strcmp(token,"word") || !strcmp(token,"long")) {
    if (enable&ESIZE) size=2;
    else error("bad option");
} else if (!strcmp(token,"noat")) {
    if (enable&ESIZE) size=3;
    else error("bad option");
} else if (!strcmp(token,"ldhi")) {
    if (enable&ELDHI) option=option|ELDHI;
    else error("bad option");
} else if (!strcmp(token,"scc")) {
    if (enable&ESCC) option=option|ESCC;
    else error("bad option");
} else if (!strcmp(token,"trap")) {
    if (enable&EST) option=option|EST;
    else error("bad option");
} else if (!strcmp(token,"ex")) {
    if (!enable&EEX) error("bad option");
    buf=get_token(buf);
    if (*token=='=') buf=get_token(buf);
    if (*token=='0' || *token=='1' || *token=='2' || *token=='3') ex=
    *token-'0';
    else error("bad option; /ex=[0..3]".);
} else if (!strcmp(token,"io")) {
    if (!enable&EIO) error("bad option");
    buf=get_token(buf);
    if (*token=='=') buf=get_token(buf);
    if (isdigit(*token)) {
        io=strtol(token,&end,0);
        if (io&0xffffffff8) error("io option out of range");
        io=io&7;
    }
} else if (*token=='/') {
    buf=get_token(buf);
    if (!strcmp(token,"byte") || !strcmp(token,"char")) io=0;
    else if (!strcmp(token,"short")) io=1;
    else if (!strcmp(token,"word") || !strcmp(token,"long")) io=2;
    else error("bad option; /io=[0-7]".);
} else { error("unknown option");
}
return get_line(buffer);
}
if (*buf==';') return buf;
else return get_options(buf);

/* get and code instruction */
char *get_instruction(buf)
char *buf;
{
    int neg;
    instr=0;
    symbol=0;
    move=0;
    ipc=0;
    src=0;
    srcb=0;
    invb=0;
    imm=0;
    instruction[ip].name=strdup(obuf);
    if (!strcmp(token,"setpsw")) {
        code=SETPSW;
        buf=get_dest(buf);
        if (dest!=PSW) error("bad destination, expected PSW");
        dest=0;
        option=0;
        buf=get_opa(buf);
        if (sh) error("no shifts for setpsw");
        if (*buf=='+') {
            buf=get_opb(buf);
        }
        else if (*buf=='-') {
            buf=get_opb(buf);
            invb=1;
        }
        else if (*buf=='/ || *buf=='/') opb=1;
        else error("illegal operator");
        instruction[ip].symbol=symbol;
        if (imm>255) error("literal overflow");
    }
    instr=(code<<CODE)|(opa<<OPA)|(invb<<INV)|(dest<<DEST)|(opb<<OPB)|(src<<SRCA)|(srcb<<SRCB)|imm;
    instruction[ip].code=instr;
    /*
    print_instr(ip);
    */
}
else if (!strcmp(token,"bra") || !strcmp(token,"branch")) {
    code=BRA;
    cc=14;
    ex=0;
    option=0;
    if (*buf==')') buf=get_condition(buf);
    if (*buf=='@') {
        buf=get_token(buf);
if (*token=="0") buf=get_token(buf);
symbol= -add_symbol(token);
op=3;
invb=0;
imm=0;
}
else {
    buf=get_dest(buf);
    if (dest!="PC") error("bad destination, expected CPC");
dest=0;
    buf=get_opa(buf);
    if (*buf=='+') {
        buf=get_opb(buf);
    }
    else if (*buf=='-') {
        buf=get_opb(buf);
        invb=1;
    }
    else if (*buf==':' || *buf=='/') opb=1;
    else error("illegal operator");
}
if (*buf==';' || *buf=='/') opb=1;

if (*stricmp(token,"jmp") || !stricmp(token,"jump")) {
    code=3MF;
    cc=14;
    ex=0;
    option=0;
    if (*buf=='.') buf=get_condition(buf);
    if (*buf=='@') {
        buf=get_token(buf);
        if (*token=="@") buf=get_token(buf);
        symbol= -add_symbol(token);
        opa=3;
        opb=1;
        invb=0;
        srca=0;
        srcb=0;
        imm=0;
    }
else {
buf = get_dest(buf);
if (dest! = CPC) error("bad destination, expected CPC");
dest = 0;
buf = get_opa(buf);
if (*buf = ‘+’) {
    buf = get_opb(buf);
}
elif (*buf = ‘-‘) {
    buf = get_opb(buf);
    invb = 1;
}
elif (*buf = ‘;’ || *buf = ‘/‘) imm = 0;
else error("illegal operator");
}
if (*buf = ‘:’) {
    enable = EX | ER TN;
    buf = get_options(buf);
}
instruction[ip].symbol = symbol;
if (imm > 255) error("literal overflow");
rt n = (option & ER TN) ? 1: 0;

instr = (code << CODE) | (opa << OPA) | (invb << INV) | (cc << CC) | (ex << EX) | (rt n << RTN) | (opb << OPB) | (src a << SRC A) | (srcb << SRC B) | imm;
instruction[ip].code = instr;
/*
   i f ( d e b u g )
   printf("code=%ld, opa=%ld, invb=%ld, cc=%ld, opb=%ld, srca=%ld, srcb=%ld, symbol=%ld, imm=%ld\n", code, opa, invb, cc, opb, srca, srcb, symbol, imm);
   printf_instr(ip);
   */
else if (!strcmp(token, "add") || !strcmp(token, "xor") || !strcmp(token, "or")) {
    if (!strcmp(token, "add")) code = ADD;
    else if (!strcmp(token, "and")) code = AND;
    else if (!strcmp(token, "xor")) code = XOR;
    else if (!strcmp(token, "or")) code = OR;
    size = 2;
    option = 0;
    buf = get_dest(buf);
    if (dest > 31) error("bad destination, expected register [0..31]");
    buf = get_opa(buf);
    if (code == ADD) {
        if (*buf = ‘+’) {
            buf = get_opb(buf);
        } else if (*buf = ‘-‘) {
            buf = get_opb(buf);
            invb = 1;
        } else if (*buf = ‘;’ || *buf = ‘/‘) opb = 1;
        else error("illegal operator");
    } else if (code == AND) {
        if (*buf = ‘&’) buf = get_opb(buf);
        else error("illegal operator");
    }
if (code==XOR) {
  if (*buf=='^') buf=get_opb(buf);
  else error("illegal operator");
}
if (code==OR) {
  if (*buf=='^') buf=get_opb(buf);
  else error("illegal operator");
}
if (*buf!=';') {
  enable=ESCC|EC|ESIZE|ESHEX;
  buf=get_options(buf);
}
if (sh) code=code|4;
  scc=(ESCC&option)?1:0;
  carry=(EC&option)?1:0;
  shex=(ESHEX&option)?1:0;
  instruction[ip].symbol=symbol;
  if (imm>255) error("literal overflow");

instr=(code<<CODE) | (opa<<OPA) | (invb<<INV) | (dest<<DEST) | (size<<SIZE) | (opb
<<OPB) | (scc<<SCE) | (shex<<SHEX) | (carry<<CARRY) | (srca<<SRCA) | (srcb<<SRCB) | imm;
  instruction[ip].code=instr;
  /*
  print_instr(ip);
  i  f  (  d  e  b  u  g  )
  printf("code=%ld,op=%ld,invb=%ld,dest=%ld,size=%ld,opb=%ld,scc=%ld;shex=
=%ld,c=%ld,srca=%ld,srcb=%ld,symbol=%d,imm=%x\n\n",
    code,opa,invb,dest,size,opb,scc,shex,carry,srca,srcb,symbol,imm);
  */
else if (!strcmp(token,"ld") || !strcmp(token,"load")) {
  code=LD;
  option=0;
  ic=IO_DEFAULT;
  buf=get_dest(buf);
  if (dest>31) error("bad destination, expected register [0..31]");
  if (*buf='[') buf=get_token(buf);
  buf=get_opa(buf);
  if (*buf='+') {
    buf=get_opb(buf);
  }
  else if (*buf='-') {
    buf=get_opb(buf);
    invb=1;
  }
  else if (*buf==';' || *buf=='/' || *buf==']') {
    opb=1;
    if (*buf==']') buf=get_token(buf);
  }
  else error("illegal operator");
  if (*buf=='])') buf=get_token(buf);
  if (*buf=';') {
    enable=EIO;
    buf=get_options(buf);
  }
  if (sh) code=code|4;
  instruction[ip].symbol=symbol;
if (imm>255) error("literal overflow");

instr=(code<<CODE)|(opb<<OPB)|(srcb<<SRCB)|imm;

instruction[ip].code=instr;

/*
  i f  ( d e b u g )
 printf("code=%ld,opb=%ld,opb=%ld,dest=%ld,io=%ld,opb=%ld,srcb=%ld,imm=%ld,";
   code,opb,invb,dest,io,opb,srcb,srcb,symbol,imm);
 */
else if (!strcmp(token,"st") || !strcmp(token,"store")) {
  code=ST;
  option=0;
  io=IO_DEFAULT;
  buf=get_dest(buf);
  if (dest>=31) error("bad destination, expected register [0..31]");
  if (*buf=='+') buf=get_token(buf);
  buf=get_opa(buf);
  if (*buf=='-') {
    buf=get_opa(buf);
    invb=1;
  }
  else if (*buf==';' || *buf=='/' || *buf==']') {
    opb=1;
    if (*buf==']') buf=get_token(buf);
  }
  else error("illegal operator");
  if (*buf=='/') buf=get_token(buf);
  if (*buf=='-') {
    enable=IO;
    option=0;
    buf=get_options(buf);
  }
  if (!opb) error("illegal operand for STORE");
  if (symbol) error("illegal operand for STORE");
  if (imm>7) error("only 3bit literal for STORE");

instruction[ip].symbol=symbol;

instr=(code<<CODE)|(opb<<OPB)|(invb<<INV)|(io<<IOCNT)|(opb<<OPB)|(srcb<<SRCA)|(dest<<SRCA)|imm;

instruction[ip].code=instr;

/*
  i f  ( d e b u g )
 printf("code=%ld,opb=%ld,invb=%ld,opb=%ld,srca=%ld,srcb=%ld,imm=%ld,";
   code,opb,invb,opb,srca,srcb,srcb,imm);
 */
else if (!strcmp(token,"noop")) {
  instruction[ip].code=0;
  instruction[ip].symbol=0;
}
else if (!strcmp(token, "get")) {
    code=GET;
    option=0;
    buf=get_dest(buf);
    if (dest>31) error("bad destination, expected register [0..31]");
    buf=get_opa(buf);
    if (lpc) code=GETLPC;
    if (opaa==0 & dest!=srca) error("dest != srca");
    if (*buf=='+') {
        buf=get_opb(buf);
    }
    else if (*buf=='-') {
        buf=get_opb(buf);
        invb=1;
    }
    else if (*buf==';' || *buf=='/') opb=1;
    else error("illegal operator");
    if (*buf==';') {
        enable=ELDHII;
        buf=get_options(buf);
    }
    instruction[ip].symbol=symbol;
    ldhi=(option&ELDHI)?1:0;
    if (imm&0xffff0000) error("literal overflow");

    instr=(code<<CODE) | (opaa<<OPA) | (invb<<INV) | (dest<<DEST) | (ldhi<<LDHI) | imm;
    instruction[ip].code=instr;
    /*
    int f ( d e b u g )
    printf("code=%ld,opaa=%ld,invb=%ld,ldhi=%ld,symbol=%d,imm=%x\n\n",
        code,opaa,invb,ldhi,symbol,imm);
    */
    )
}

else if (!strcmp(token, "pswadd") || !strcmp(token, "pswadd") || !strcmp(token, "pswadd")) {
    if (!strcmp(token, "pswadd")) code=PSWADD;
    else if (!strcmp(token, "pswadd")) code=PSWADD;
    else if (!strcmp(token, "pswadd")) code=PSWADD;
    else if (!strcmp(token, "pswadd")) code=PSWADD;
    option=0;
    buf=get_dest(buf);
    if (dest!=PSW) error("bad destination, expected PSW");
    dest=0;
    buf=get_opa(buf);
    if (opaa==0) dest=srca;
    if (code==PSWADD) {
        if (*buf=='+') {
            buf=get_opb(buf);
        }
        else if (*buf=='-') {
            buf=get_opb(buf);
            invb=1;
        }
        else if (*buf==';' || *buf=='/') imm=0;
        else error("illegal operator");
    }

}
if (code==PSWAND) {
    if (*buf=="&") buf=get_opb(buf);
    else if (*buf==";" || *buf=="/") imm=0;
    else error("illegal operator");
}
if (code==PSWXOR) {
    if (*buf=="^") buf=get_opb(buf);
    else if (*buf==";" || *buf=="/") imm=0;
    else error("illegal operator");
}
if (code==PSWOR) {
    if (*buf=="|") buf=get_opb(buf);
    else if (*buf==";" || *buf=="/") imm=0;
    else error("illegal operator");
}
if (*buf!=";") {
    enable=ESPM|ELDHI;
    buf=get_options(buf);
}
if (sh) error("no shifts for MFSW instructions");
spm=(option&ESPM)?1:0;
ldhi=(option&ELDHI)?1:0;
imm=imm&0xffff;

instr=(code<<CODE)|(opase<OPA)|(invb<INV)|(dest<<DEST)|(ldhi<<LDHI)|(spm<<SPM)|imm;
    instruction[ip].code=instr;
    /*
    print_instr(ip);
        i f   ( d e b u g )
    printf("code=%ld,opase=%ld,invb=%ld,dest=%ld,spm=%ld,ldhi=%ld,symbol=%c,imm=%ld\n\n",
            code,opase,invb,dest,spm,ldhi,symbol,imm);
    */
}
!
e l s e
!
strcmp(token,"inc") || strcmp(token,"dec") || strcmp(token,"clr") || strcmp(token,"shl") ||

strcmp(token,"shr") || strcmp(token,"shl") || strcmp(token,"shr") ||

code=ADD;
    opa=0;
    option=0;
    if (!strcmp(token,"sh",2)) {
        code=ADD;
        if (!strcmp(token,"shl")) opa=1;
        else if (!strcmp(token,"shr")) opa=2;
        else if (!strcmp(token,"shr") opa=3;
    }
    if (!strcmp(token,"dec")) invb=1;
    else invb=0;
    if (!strcmp(token,"clr") || !strcmp(token,"clear") { 
        invb=1;
        opb=0;
        imm=0;
    }
else {
    opb=1;
    if (!strcmp(token, "sh", 2)) imm=0;
    else imm=1;
}
size=2;
option=0;
buf=get_token(buf);
if (*token=='r') {
    dest=atoi(token+1);
    if (dest>31) error("register number out of range");
} else if (!strcmp(token, "sp")) dest=SP;
else error("bad destination, expected register [0..31]");
src=dest;
srcb=(opb)?(long)0:dest;
if (*buf!=';') {
    enable=ESCC|ESIZE|ESHEX;
    option=0;
    size=2;
    buf=get_options(buf);
}
svc=(ESCC?option)?1:0;
sshx=(option|ESHEX)?1:0;
instruction[ip].symbol=symbol;
if (imm>255) error("literal overflow");

instr=(code<<CODE)|(opa<<OPA)|(invb<<INV)|(dest<<DEST)|(size<<SIZE)|(opb
<<OPB)|(scc<<SCC)|(shex<<SHEX)|(src<<SRCA)|(srcb<<SRCB)|imm;
instruction[ip].code=instr;

/*
  I F ( d e b u g )
printf("code=%ld,opa=%ld,invb=%ld,dest=%ld,size=%ld,opb=%ld,scc=%ld,shex
=\%ld,src=%ld,srcb=%ld,\symbol=%ld,imm=%lx\n\n",
    code,opa,invb,dest,size,opb,scc,shex,src,srcb,symbol,imm);
  */
}
else {
    if (!strcmp(token, "mov") || !strcmp(token, "move") || !strcmp(token, "cmp") || !str
    cmp(token, "compare")) {
        code=ADD;
        neg=0;
        opa=0;
        option=0;
        size=2;
        if (!strcmp(token, "cmp") || !strcmp(token, "compare")) {
            move=0;
            buf=get_opa(buf);
            if (*buf!=';' && *buf!='-') error("expected , or - ");
            else buf=get_opb(buf);
            invb=1;
            dest=RTMP;
        }
    else { // mov */
        move=1;
        buf=get_dest(buf);
        if (dest>31) error("bad destination");
    }
opa=1;
srca=0;
sh=0;
shex=0;
if (*buf=='-') neg=1;
if ((*buf>='0' && *buf<='9') || *buf=='#' || *buf=='@' || *buf=='~' || *buf=='-')
    buf=get_opb(buf);
else {
    opb=1;
    imm=0;
    srcb=0;
    buf=get_opa(buf);
}
if (neg) invb=1;
if (*buf!=';') {
    enable=ESCC|ESIZE|ESHEX|ELDHI;
    buf=get_options(buf);
}
if ((option&ELDHI && !move)) {
    error("bad option on CMP instruction");
    option=option&(~ELDHI);
}
if (move) scc=(ESCC&option)?1:0;
else scc=1;
instruction[ip].symbol=symbol;
if (imm!=(imm&0xffff)) { /* constant has more than 16bits */
    /* can it be fixed by an inversion ? */
    if ((imm&0xffff0000)==0xffff0000 && move) {
        imm=imm;
        invb=invb^1;
        if (imm&0xffff0000) error("16bit literal overflow");
    } else if ((imm&0x0000ffff)==0 && move) {
        imm=imm>>16;
        option=option|ELDHI;
    } else error("16bit literal overflow");
}
imm=imm&0x0000ffff;
if (imm>255 || option&ELDHI || lpc) {
    if (move && !(option&ESCC)) {
        if (imm>0xffff) error("16bit literal overflow");
    else {
        if ((option&ESCC) || size!=2 || !opb) error("bad option on MOVE (GET) with 16bit literal");
        if ((option&ELDHI) && opb) option=ELDHI;
        else option=0;
        size=0;
        opb=0;
        if (lpc) code=GETLPC;
        else code=GET;
    }

    } else error("literal overflow");
}
if (sh) code=code|4;
shex=(ESHEX?option)?1:0;
ldhi=(ELDHIC?option)?1:0;

instr=(code<<CODE)|(opb<<OFA)|(invb<<INV)|(dest<<OFS)|(siz

e<<SIZE)|(opb<<OFP)|(scc<<SCC)|(shex<<SHEX)|(srcb<<SRCA)|(srcb<<SRCB)|imm;
instruction[ip].code=instr;
/*
 print_instr(ip);
 i f( d e b u g )
 printf("code=%ld,opb=%ld,invb=%ld,dest=%ld,size=%ld,opb=%ld,scc=%ld,shex=
%ld,srcb=%ld,symbol=%d,imm=%x\n\n",
 code,opb,invb,dest,size,opb,scc,shex,srcb,symbol,imm);
 */
}
else error("unknown instruction");
ip++; 
if (*buf!=';') error("expected ",ip);
buf=get_line(buffer); 
if (ip>IP_MAX) error("instruction pointer out of range");
return buf;
}

/* print instruction code in binary to stdout */
void print_instr(ip)
int ip;
{
 int i;
 long instr=instruction[ip].code;
 printf("\x%4.4x : \',ip);
 for (i=32;i>=0;i=--)
 { 
 if (instr&((long)1<<i)) printf("1");
 else printf("0");
 if ((i&4)==0 && i!=0) printf(" ");
 }
 printf("\n");
 if (debug) dis_as(instr);
 printf(" %s \n",instruction[ip].name);
}

/* print destination register of instruction */
void print_dest(instr)
long instr;
{
 long reg;
 reg=(instr&0x00f80000)>>DEST;
 printf("R%ld=",reg);
}

/* print destination register of STORE instructions */
void print_dest_st(instr)
long instr;
{
 long reg;
 reg=(instr&0x000000f8)>>SRCB;
 printf("R%ld=",reg);
}

/* print operand A of Instruction (no shift)*/
void print_opa(instr)
long instr;
{
    switch ((int)((instr&0x06000000)>>OPA))
    {
    case 0: if (instr<0) printf("R%d",(instr&0x00001f00)>>SRCA);
            else printf("R%d",(instr&0x00f00000)>>DEST);
                    break;
    case 1: printf("0");
            break;
    case 2: printf("PSW");
            break;
    case 3: if ((instr>>CODE)==GETLPC) printf("LPC");
            else printf("CPC");
                    break;
    default:fprintf(stderr,"ERROR in print_opa()\n");
                    break;
    }
}

/* print operand A of instruction with shift enabled */
void print_opash(instr)
long instr;
{
    switch ((int)((instr&0x06000000)>>OPA))
    {
    case 0: printf("R%d*4",(instr&0x000f0000)>>SRCA);
            break;
    case 1: printf("R%d*2",(instr&0x000f0000)>>SRCA);
            break;
    case 2: printf("R%d/2",(instr&0x000f0000)>>SRCA);
            break;
    case 3: printf("R%d/4",(instr&0x000f0000)>>SRCA);
            break;
    default:fprintf(stderr,"ERROR in print_opash()\n");
            break;
    }
}

/* print condition code of BRANCH and JUMP instructions */
void print_cc(instr)
long instr;
{
    switch ((int)((instr&0xf000000)>>CC))
    {
    case 0: printf("_le ");
            break;
    case 1: printf("_gt ");
            break;
    case 2: printf("_lt ");
            break;
    case 3: printf("_ge ");
            break;
    case 4: printf("_ls ");
            break;
    case 5: printf("_ht ");
            break;
    case 6: printf("_c ");
break;
case 7: printf("_nc ");
break;
case 8: printf("_mi ");
break;
case 9: printf("_pl ");
break;
case 10: printf("_eq ");
break;
case 11: printf("_ne ");
break;
case 12: printf("_v ");
break;
case 13: printf("_nv ");
break;
case 14: printf("_l ");
break;
case 15: printf("_0 ");
break;
}

/* print operand B of instruction */
void print_opb(instr)
long instr;
{
int code;
if (instr>0) printf("%4.4lx", instr&0xfffff);
else {
  code=(instr>>CODE)&31;
  if (code==ST || code==(ST|4)) printf("%1.1lx", instr&0x7);
  else {
    if (instr&((long)1<OPB)) printf("%2.2lx", instr&0xff);
    else printf("R%ld", (instr&0xf8)>>SRCB);
  }
}
}

/* disassembler main routine (disassembles argument) */
void dis_as(instr)
long instr;
{
int code;
printf("->");
code=(instr>>CODE)&31;
switch (code)
{
case ADD : printf("ADD ");
  print_dest(instr);
  print_opa(instr);
  if (instr&((long)1<INV)) printf("-");
  else printf("+");
  print_opb(instr);
  switch ((int)((instr&0x0060000)>>SIZE)) {
    case 0: printf("/B");
      break;
    case 1: printf("/S");
      break;
  }
}

/* print operand B of instruction */
void print_opb(instr)
long instr;
{
int code;
if (instr>0) printf("%4.4lx", instr&0xfffff);
else {
  code=(instr>>CODE)&31;
  if (code==ST || code==(ST|4)) printf("%1.1lx", instr&0x7);
  else {
    if (instr&((long)1<OPB)) printf("%2.2lx", instr&0xff);
    else printf("R%ld", (instr&0xf8)>>SRCB);
  }
}

/* disassembler main routine (disassembles argument) */
void dis_as(instr)
long instr;
{
int code;
printf("->");
code=(instr>>CODE)&31;
switch (code)
{
case ADD : printf("ADD ");
  print_dest(instr);
  print_opa(instr);
  if (instr&((long)1<INV)) printf("-");
  else printf("+");
  print_opb(instr);
  switch ((int)((instr&0x0060000)>>SIZE)) {
    case 0: printf("/B");
      break;
    case 1: printf("/S");
      break;
  }
}
case 2: printf(" /W");
    break;
  case 3: printf(" /NOAT");
    break;
}
if (instr&((long)1<<SCC)) printf(" /SCC");
if (instr&((long)1<<SHEX)) printf(" /SHEX");
if (instr&((long)1<<CARRY)) printf(" /C");
break;
  case ADD|4:
    printf("ADD ");
    print_dest(instr);
    print_opash(instr);
    if (instr&((long)1<<INV)) printf("-");
    else printf("+");
    print_opb(instr);
    switch ((int)((instr&0x00060000)>>SIZE)) {
    case 0: printf(" /B");
        break;
    case 1: printf(" /S");
        break;
    case 2: printf(" /W");
        break;
    case 3: printf(" /NOAT");
        break;
}
if (instr&((long)1<<SCC)) printf(" /SCC");
if (instr&((long)1<<SHEX)) printf(" /SHEX");
if (instr&((long)1<<CARRY)) printf(" /C");
break;
  case AND:
    printf("AND ");
    print_dest(instr);
    print_opa(instr);
    if (instr&((long)1<<INV)) printf("&~");
    else printf("&");
    print_opb(instr);
    switch ((int)((instr&0x00060000)>>SIZE)) {
    case 0: printf(" /B");
        break;
    case 1: printf(" /S");
        break;
    case 2: printf(" /W");
        break;
    case 3: printf(" /NOAT");
        break;
}
if (instr&((long)1<<SCC)) printf(" /SCC");
if (instr&((long)1<<SHEX)) printf(" /SHEX");
if (instr&((long)1<<CARRY)) printf(" /C");
break;
  case AND|4:
    printf("AND ");
    print_dest(instr);
    print_opash(instr);
    if (instr&((long)1<<INV)) printf("&~");
    else printf("&");
    print_opb(instr);
    switch ((int)((instr&0x00060000)>>SIZE)) {
    case 0: printf(" /B");
        break;
case 1: printf(" /S");
    break;
case 2: printf(" /W");
    break;
case 3: printf(" /NOAT");
    break;
}
if (intr&((long)1<<SCC)) printf(" /SCC");
if (intr&((long)1<<SHEX)) printf(" /SHEX");
if (intr&((long)1<<CARRY)) printf(" /C");
    break;
case XOR:
    printf("XOR ");
    print_dest(intr);
    print_opa(intr);
    if (intr&((long)1<<INV)) printf(" ^- ");
    else printf(" ^ ");
    print_opb(intr);
    switch ((int)((intr&0x00060000)>>SIZE)) {
        case 0: printf(" /B");
            break;
        case 1: printf(" /S");
            break;
        case 2: printf(" /W");
            break;
        case 3: printf(" /NOAT");
            break;
    }
    if (intr&((long)1<<SCC)) printf(" /SCC");
    if (intr&((long)1<<SHEX)) printf(" /SHEX");
    if (intr&((long)1<<CARRY)) printf(" /C");
    break;
case XOR4:
    printf("XOR ");
    print_dest(intr);
    print_opash(intr);
    if (intr&((long)1<<INV)) printf(" ^- ");
    else printf(" ^ ");
    print_opb(intr);
    switch ((int)((intr&0x00060000)>>SIZE)) {
        case 0: printf(" /B");
            break;
        case 1: printf(" /S");
            break;
        case 2: printf(" /W");
            break;
        case 3: printf(" /NOAT");
            break;
    }
    if (intr&((long)1<<SCC)) printf(" /SCC");
    if (intr&((long)1<<SHEX)) printf(" /SHEX");
    if (intr&((long)1<<CARRY)) printf(" /C");
    break;
case OR:
    printf("OR ");
    print_dest(intr);
    print_opa(intr);
    if (intr&((long)1<<INV)) printf(" |- ");
    else printf(" | ");
    print_opb(intr);
    switch ((int)((intr&0x00060000)>>SIZE)) {
case 0: printf(" /B");
    break;
case 1: printf(" /S");
    break;
case 2: printf(" /W");
    break;
case 3: printf(" /NOAT");
    break;
}
if (instr & ((long)1<<SCC)) printf(" /SCC");
if (instr & ((long)1<<SHEX)) printf(" /SHEX");
if (instr & ((long)1<<CARRY)) printf(" /C");
break;
case OR|4 : printf("OR ");
    print_dest (instr);
    print_opash (instr);
    if (instr & ((long)1<<INV)) printf("|~");
    else printf("|");
    print_opb (instr);
    switch ((int)((instr&0x00060000)>>SIZE)) {
        case 0: printf(" /B");
                break;
        case 1: printf(" /S");
                break;
        case 2: printf(" /W");
                break;
        case 3: printf(" /NOAT");
                break;
    }
if (instr & ((long)1<<SCC)) printf(" /SCC");
if (instr & ((long)1<<SHEX)) printf(" /SHEX");
if (instr & ((long)1<<CARRY)) printf(" /C");
break;
case LD : printf("LD ");
    print_dest (instr);
    printf("[");
    print_opa (instr);
    if (instr & ((long)1<<INV)) printf("-");
    else printf("+");
    print_opb (instr);
    printf("] /I0=\%ld", (instr&0xe000)>>IOCNT);
    break;
case LD|4 : printf("LD ");
    print_dest (instr);
    printf("[");
    print_opash (instr);
    if (instr & ((long)1<<INV)) printf("-");
    else printf("+");
    print_opb (instr);
    printf("] /I0=\%ld", (instr&0xe000)>>IOCNT);
    break;
case ST : printf("ST ");
    print_dest_st (instr);
    printf("[");
    print_opa (instr);
    if (instr & ((long)1<<INV)) printf("-");
    else printf("+");
    print_opb (instr);
printf("\[IO=%ld", (instr&0xe000) >> IOCNT);
break;

\textbf{case ST|4}:
printf("ST ");
print_dest_st(instr);
printf("[");
print_opa(instr);
if (instr&((long)1<<INV)) printf("-");
else printf("+");
print_opb(instr);
printf("] /IO=%ld", (instr&0xe000) >> IOCNT);
break;

\textbf{case SETPSW}:
printf("SETPSW PSW=");
print_opa(instr);
if (instr&((long)1<<INV)) printf("-");
else printf("+");
print_opb(instr);
break;

\textbf{case JMP}:
printf("JMP ");
print_cc(instr);
printf(" CPC=");
print_opa(instr);
if (instr&((long)1<<INV)) printf("-");
else printf("+");
print_opb(instr);
printf(" /EX=%ld", (instr&0x60000) >> EX);
if (instr&((long)1<<RTN)) printf(" /CLRPM");
break;

\textbf{case BRA}:
printf("BRA ");
print_cc(instr);
printf(" CPC=");
print_opa(instr);
if (instr&((long)1<<INV)) printf("-");
else printf("+");
print_opb(instr);
printf(" /EX=%ld", (instr&0x60000) >> EX);
if (instr&((long)1<<RTN)) printf(" /CLRPM");
if (instr&((long)1<<STRAP)) printf(" /TRAP");
break;

\textbf{case GET}:
printf("GET ");
print_dest(instr);
print_opa(instr);
if (instr&((long)1<<INV)) printf("-");
else printf("+");
print_opb(instr);
if (instr&((long)1<<LDHI)) printf(" /LDHI");
break;

\textbf{case GETLPC}:
printf("GET ");
print_dest(instr);
print_opa(instr);
if (instr&((long)1<<INV)) printf("-");
else printf("+");
print_opb(instr);
if (instr&((long)1<<LDHI)) printf(" /LDHI");
break;

\textbf{case PSWADD}:
printf("PSWADD PSW=");
print_opa(instr);
if (instr&((long)1<<INV)) printf("-");
else printf("+");
print_opb(instr);
if (instr&((long)1<<SPM)) printf(" /SETPM");
break;
case PSWAND :printf("PSWAND PSW=");
    print_opa(instr);
    if (instr&((long)1<<INV)) printf("&~");
    else printf("&");
    print_opb(instr);
    if (instr&((long)1<<SPM)) printf(" /SETPM");
    break;
case PSWXOR :printf("PSWXOR PSW=");
    print_opa(instr);
    if (instr&((long)1<<INV)) printf("^~");
    else printf("^");
    print_opb(instr);
    if (instr&((long)1<<SPM)) printf(" /SETPM");
    break;
case PSWOR :printf("PSWOR PSW=");
    print_opa(instr);
    if (instr&((long)1<<INV)) printf("|~");
    else printf("|");
    print_opb(instr);
    if (instr&((long)1<<SPM)) printf(" /SETPM");
    break;
case NOOP :printf("NOOP");
    break;
default :printf("illegal instruction");
    fprintf(stderr,"illegal instruction for disassembly");
    break;
}
printf(";\n");}
C. TOTEKSIM Man Page

TOTEKSIM

COPYRIGHT Hans J. Greub & Rensselaer Polytechnic Institute

Toteksim is a program that converts a hierarchical netlist in EDIF (Electronic Design Interchange Format) as generated by ORCAD’s netlist utility to TEKSIM or PSPICE format. Toteksim can further generate netlists for schematics that have both differential and single ended standard cells if the user follows the conventions described below and declares the I/O port properties of the standard cells in the stdcell.def file. Moreover, it can detect level violations in ECL circuits and illegal connections of differential signal to single ended ports.

SYNOPSIS

toteksim <EDIF file> [switches] > <output file>

DESCRIPTION

TOTEKSIM reads in an EDIF netlist as generated by ORCAD’s netlist utility and a user defined standard cell list and outputs a hierarchical netlist in either TEKSIM or PSPICE format. The EDIF netlist generated by ORCAD is not semantically correct if the /h switch is used to avoid multiple listings of the same subcircuit and to reduce processing time for a complex hierarchy. However, if the user follows the suggested sheet name conventions described below, the above problem can be evaded.

The standard cell list can be either appended to the EDIF input file or it can be read from a file with the /s switch. The standard cell list describes the input/output ports of each standard cell or any library part the user likes to include in the schematics. Further, the ECL level of the I/O nodes and the type of the port (single ended or differential) is declared in the standard cell list.

TOTEKSIM will expand differential nets into two nets (true, false) and invert signals by twisting the (true, false) signals. This kind of inversion of a differential signal is shown in the schematic by placing a virtual inverter (vinv). See also netlist expansion. Further, TOTEKSIM can detect level violations in an ECL circuit and detect an illegal connection of a differential signal to a single ended port. A virtual single ended to differential converter (vstod) is provided to drive a differential net with two complementary single ended signals.

Toteksim will insert nc nodes on standard cells or library parts if not all ports are connected. With the /v switch a warning message is issued if a unconnected port is detected. The user can explicitly label unconnected ports with the special net names nc or nc_.

The converted netlist is sent to standard output. Error and warning messages are sent to both standard error and standard output. Standard output can be redirected to a file...
Toteksim uses a fully dynamic data structure and uses the large memory model. It can handle therefore large designs. It takes less than one minute to convert a 150kByte EDIF netlist to a 40kByte TEKSIM netlist. Since the generation of the EDIF netlist takes much longer the time penalty for using to toteksim is minimal. If toteksim runs out of memory it should die gracefully. Toteksim is not case sensitive. It converts the input file to all lower case.

SWITCHES

/c subcircuit names, the title of the sheet, and the time stamp of the last sheet modification are printed as comments. Further, virtual elements are included in the netlist as comments.
/e expand differential nodes.
/l expand differential nets and check for level violations and illegal connections of differential signals to single ended ports.
/r read standard cell definitions from a file. The default file name is stdcells.def. However, the user can specify an other file name including a path with /r <file name>. If the /r switch is not given toteksim expects the standard cell list to be appended to the EDIF input file.
/k keep instances type names and subcircuit names as output by ORCAD's netlist utility. Without the /k switch TOTEKSIM strips the <suffix> from instance type names (part value) and subcircuit names.
/n marks all unconnected ports with '#'.
/p output netlist in PSPICE format
/q suppress .END or endc statements
/s output top level circuit as a subcircuit.
/t list standard cells and show I/O port levels.
/v verbose output. With the /v switch toteksim issues additional warning messages and shows the level, number of drivers, type (single ended/differential), and local fanout for each net of a subcircuit. Use this switch to find problems in the netlist or toteksim.

EDIF INPUT FILE

The EDIF input file must be generated with /p switch on the netlist utility. The following commands are suggested for the generation of the EDIF netlist with ORCAD 3.2:
annotate <sheet name>.sch /u/m/h/q
netlist <sheet name>.sch <sheet name>.net /p/h/q

The first command performs an incremental annotation. It generates unique part names and writes them back to the sheets. After this step the hierarchical design can be plotted with the command: plotall <sheet name>.sch /h showing the annotated part names and processing each sheet only once. The second command generates the EDIF netlist that can then be fed to TOTEKSIM. With the /h switch each sheet will be processed only once saving you time, money, and disk space. The /h switch is though not required toteksim will ignore multiple definitions of the same subcircuit and issue a warning message. To make the netlist generation easier the batch file net.bat was written. It will run the annotate, netlist, and toteksim utilities for you.

NET NAMES

Any net that is labeled with a string that contains an underbar '_' is forced to be single ended. Without an underbar character in its name the net is considered to be differential unless all the ports on the net are single ended. The user is, therefore, not forced to declare single ended nets. It is however highly recommended for schematics that include both single ended and differential nets. Toteksim will issue an error message if a net is connected to both single ended and differential ports. With the /v switch toteksim outputs a warning if the net name does not declare the net type properly.

Note: net names should not start with an underbar and can have only characters from the following sets [a-z], [A-Z], [0-9] else ORCAD will rename the net!

TOTEKSM generates net names of the type n<integer> for nets that are not labeled in the schematic. The node counter is reset for each subcircuit in order to keep node number changes after a modification of a sheet local to the corresponding subcircuit. If the user labels unconnected ports with nc or nc_TOTEKSM generates nc port names like nc<integer> or nc <integer> for single ended nc ports. If the output format is TEKSIM a dot will be added to nc node names to keep FASTSIM from issuing error messages.

Toteksim will output an error message if the schematic contains any of the following reserved signal labels:

RESERVED NET NAMES

n<integer>
nc<integer>
nc_<integer>

Any net or node label with v<c><c> where <c> is a single character a global net. Example vbb, vcc, vee, vss. Global nets are listed in the netlist file like G_vcc for TEKSIM or $G_vcc for PSPICE. This feature is provided since
ORCAD does not support global nets. It lists all power connections as I/O ports in EDIF.

**SPECIAL NET NAMES**

- nc     differential no connection; output nc<integer>
- nc_<c><c>  single ended no connection; output nc_<integer>
- v_<c><c>  global node; output G_v_<c><c> or $G_v_<c><c>

**SHEET SYMBOLS NAMES**

ORCAD uses the sheet symbol name to generate both the subcircuit instance name (reference) and the instance type (part value) instead of using the file name of the sheet as the part value. It is therefore up to the user to verify that all sheets symbols that reference the same file have the same sheets symbol name prefix. Since the netlist utility fails to generate unique instance names when the /h switch is specified the user must use the following sheet symbol naming convention:

sheet name = <prefix>_<instance_name>

The prefix will become the subcircuit model name. A good rule is to use the sheet file name without extension for the prefix as long as you do not use the meaningless default file name generated by ORCAD. The instance name (reference) should uniquely define the particular instance of the subcircuit in the current sheet.

The order of the ports in the netlist for a subcircuit is determined by a string sort. Example: a subcircuit with ports: b q a a b b b would show the following port order: a b q a a b b.

A subcircuit instance name must begin with an ‘x’ in PSpICE. Therefore, toteksim will insert an ‘x’ if the sheet symbol name <prefix> does not begin with an ‘x’.

**PARTS WITH NUMERICAL PART VALUE**

Parts which have a numerical part value in SPICE like resistors, capacitors, inductors, and independent voltage or current source are assigned the part value x_<part value> in EDIF. Toteksim recognizes these parts and converts them to PSpICE or TEKSIM format. However, the part reference must start with a character that specifies the part type (r=resistor, c=capacitor, l=inductor, i=current source, v=volts source) even if the TEKSIM output format is used.

Since ORCAD's netlist utility replaces characters like ",","=" with an underscore the part value must currently be an integer. However, the integer value can have a scale factor attached. Only the first character is though transferred to the netlist! The only exception is the scale factor meg. Toteksim will multiply the part value by 1000 and replace the scale factor with the character 'k'.
DIFFERENTIAL NET EXPANSION

With the /e or /l switch toteksim will expand differential nets. A differential net <name> is expanded into the nets <name>, <name>b>. If this net expansion would cause two signals to be connected together that are not connected in the schematic toteksim will rename one of the nets and issue a warning message. Example:

ab,abb,b,bb ------> (ab,abb),(abb1,abb1b),(b,bb),(bb1,bb1b)
;*** Warning; net ab was renamed to abb1
;*** Warning; net bb was renamed to bb1

With the /l switch toteksim will also flag connections of single ended ports to differential nets and level violations. If the user wants to use the complementary outputs of a single ended standard cell to drive a differential net he must use a virtual single ended to differential converter vstod. The vstod has a differential output and two single ended input ports.

v1 out iplus, iminus, vstod:

This virtual element will only be listed in the netlist as a comment (/c or /v). It tells toteksim that the two single ended input signals which must be complementary are used to drive the differential net at the output of the vstod. Toteksim will rename the output net to (iplus, iminus) unless the output is an I/O net in which case it will rename the input net iplus = out, iminus = outb. If this renaming causes the loss of a user defined net label a warning message is output.

A differential signal can be inverted at no cost and with zero delay by simply switching the true and false signal connection on a differential port. Such an inversion can be indicated in the schematic by placing a virtual inverter vin. Totebox will invert all the differential ports connected to the output of the vin. This virtual element will also only be listed in the netlist as a comment (/c or /v). Toteksim will rename the output net to the input net name unless the output net is an I/O net in which case it will rename the input net. Again if this renames a net labeled by the user a warning message is issued. Note, virtual elements can not be connected in series.

The I/O ports of a standard cell can also be inverted by adding an integer qualifier to the part value. If bit n of the qualifier is set the differential port n will be inverted. Example:

and21m:12 -port 2 and port 3 will be inverted.

STANDARD CELL LIST

The standard cell list contains a listing of all standard cells with a listing of the I/O ports and an optional level specification. An error message is output if a port is found that is not listed in standard cell list.
The information in the standard cell file is necessary for level checking and for detecting unconnected ports. Port names that starts with the character 'o' are outputs. A level can be specified after the port name.

Example:
```
( std_cell and2lm (nodes o:1 i2:1rp_3 in1:1-2) )
( std_cell or2sm (nodes o_:1 ob_:1 i1_:1 i2_:2 ) )
```

The first node on cell and2lm is declared as an output at level 1. The second node is an input one level higher than node 3. The third node is an input which can have any level in the range 1-2. Cell or2sm is a single ended dual input or.

Levels 1 to 6 are available. Currently levels 1-3 are on chip signal levels. Level 4 is used for differential off chip signals and level 5 is used for single ended off chip signals.

Note, the last character in the standard cell name describes the power of the output driver (h = high, m = medium, l = low). The above standard cell I/O port description will also match and2lh and and2ll standard cells.

**BUGS**

Input and output nets of virtual elements are not fused together and are listed separately with the same name if the /v switch is specified.

Toteksim can not pass variables from the part fields to the netlist because ORCAD’s netlist utility converts dots and equal signs as well as ‘()’ to ‘’ and does not allow you to pass literals from the key field configuration menu to an EDIF netlist.

Please let ORCAD know if you are interested in passing variables. If enough users complain they might do something about it. Currently they claim the modified appearance of part fields in the EDIF netlist is just an undocumented ‘feature’!

ORCAD’s netlist utility outputs all subcircuits at a global level.

Please send bug reports or comments by EMAIL to:

hgreub@unix.cie.rpi.edu

**FILES:**

toteksim.exe, stdcells.def