Technical Report: FRISC-91-1

FRISC
A Fast Reduced Instruction Set Computer for Implementation with Advanced Bipolar and Hybrid Wafer Scale Technology

Hans J. Greub

Rensselaer Polytechnic Institute
February 1991

This work has been sponsored in part by the Defense Advanced Research Projects Agency under contract DARPA/ARO DAAL03-90-G0187. The views, opinions, and/or findings contained in this paper should not be construed as an official position or policy.
# Contents

List of Figures ................................................. v

List of Tables ................................................. viii

Inserts ......................................................... ix

Acknowledgement .............................................. x

## Abstract ..................................................... xi

Chapter I.  
Introduction .................................................. 1

Chapter II.  
Advanced Bipolar Circuit Technology .......................... 17  
   A. Advanced Bipolar Process .................................. 17  
   B. Current Switch ............................................. 19  
   C. Differential Current Tree Logic ............................ 23  
   D. Emitter Followers and Buffers ............................. 31  
   E. I/O Circuits ................................................ 35  
   F. ALU Circuit ................................................ 38  
   G. 32x8bit Register File .................................... 43  
   H. Digital Delay Line ....................................... 48  
   I. Standard Cell Library .................................... 53

Chapter III.  
Modeling of Differential Current Tree Logic .................. 56  
   A. Digital Current Switch Model .............................. 58  
   B. Modelling of Current Trees ................................ 64  
   C. Modelling Accuracy ....................................... 66

Chapter IV.  
Architecture ................................................... 69  
   A. System Architecture .................................... 69  
   B. Processor Architecture .................................. 72  
   C. Pipelining .................................................. 79  
   D. Timing Constraints / Critical Paths ...................... 86  
   E. Load and Branch Latencies ................................ 96  
   F. Sustained Performance ................................... 101
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>G. Partitioning</td>
<td>106</td>
</tr>
<tr>
<td>H. Instruction Set</td>
<td>109</td>
</tr>
<tr>
<td>I. FRISC Macro Assembler</td>
<td>127</td>
</tr>
<tr>
<td>1. FRISC Assembler Syntax</td>
<td>127</td>
</tr>
<tr>
<td>2. Assembler Command Line Options</td>
<td>130</td>
</tr>
<tr>
<td>3. M4 Macro Preprocessor</td>
<td>130</td>
</tr>
<tr>
<td>4. Sample Program</td>
<td>131</td>
</tr>
<tr>
<td>J. Input / Output Interface</td>
<td>133</td>
</tr>
<tr>
<td>K. Interrupt Handling</td>
<td>137</td>
</tr>
<tr>
<td>L. Perspective</td>
<td>145</td>
</tr>
</tbody>
</table>

Chapter V.
FRISC-F Simulator                  150
A. Model Structure and Basic Features 150
B. User Interface                   155

Chapter VI.
FRISC-F Modules                    160
A. Datapath Module                 160
1. ALU Logic                       163
2. Shifter Logic                   170
3. Negative and Zero Flag          174
4. Carry and Overflow Flag         177
5. Program Counter Logic           178
6. Datapath Input/Output Signals   180
B. Instruction Decoder             182
1. Pipeline Controllers            183
2. FRISC-F Pipeline Stage Controllers 184
3. Instruction Decoder Input/Output Signals 189

Chapter VII.
Computer Aided Design Tools        

Conclusions                       

References                        205

Appendix                          210
A. FRISC-F VERILOG Sources ............................................. 211
   1. Module:sys, File:frisc.v; Top Level Module ................... 211
   2. Module:dp, File:lib/dp.v; Datapath .............................. 225
   3. Module:id, File:lib/id.v; Instruction Decoder .................. 234
   4. Module:cache, File:lib/cache.v; Cache Memory .................. 252
   5. Module:align_mi, File:lib/align_mi.v; Data Aligner MI ........... 257
   6. Module:align_mo, File:lib/align_mo.v; Data Aligner MO .......... 258
   8. Module:shifter, File:lib/shifter.v; Shifter ....................... 262
   9. Module:trap_enc, File:lib/trap_enc.v; Trap Encoder .............. 264
  10. Module:state_co, File:lib/state_co; State Controller ............. 266
  11. Module:clock_ge, File:lib/clock_ge.v; Clock Generator .......... 268
  12. Module:phase_ge, File:lib/phase_ge.v; Phase Generator .......... 269

B. FRISC-F Assembler Source, File:fas.c ................................. 271
C. TOTENXSIM Man Page .................................................. 306
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Wafer Scale Hybrid</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Signal Degradation on Lossy Transmission Lines</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>FRISC and Cache Memories on Wafer Scale Hybrid</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>Device Structure (from [TY88])</td>
<td>17</td>
</tr>
<tr>
<td>5</td>
<td>Bipolar Current Switch</td>
<td>19</td>
</tr>
<tr>
<td>6</td>
<td>CML and ECL Buffer Delays versus Switching Current</td>
<td>20</td>
</tr>
<tr>
<td>7</td>
<td>Characteristic of Current Switch ($R_t=625\Omega, I_s=400\mu A$)</td>
<td>22</td>
</tr>
<tr>
<td>8</td>
<td>Differential Three Input AND Gate</td>
<td>24</td>
</tr>
<tr>
<td>9</td>
<td>Single Ended OR Gate</td>
<td>25</td>
</tr>
<tr>
<td>10</td>
<td>Carry Propagate Gate</td>
<td>26</td>
</tr>
<tr>
<td>11</td>
<td>Differential Four Input Multiplexor</td>
<td>27</td>
</tr>
<tr>
<td>12</td>
<td>Differential Latch with XOR Inputs</td>
<td>28</td>
</tr>
<tr>
<td>13</td>
<td>Basic and Improved Emitter Followers</td>
<td>32</td>
</tr>
<tr>
<td>14</td>
<td>Delays of ECL Buffers</td>
<td>33</td>
</tr>
<tr>
<td>15</td>
<td>Super Buffer SBUF1H</td>
<td>34</td>
</tr>
<tr>
<td>16</td>
<td>I/O Drivers &amp; Receivers</td>
<td>36</td>
</tr>
<tr>
<td>17</td>
<td>Layout of I/O Test Circuits</td>
<td>37</td>
</tr>
<tr>
<td>18</td>
<td>8bit Carry Select ALU Slice</td>
<td>39</td>
</tr>
<tr>
<td>19</td>
<td>Layout of ALU Test Circuit</td>
<td>41</td>
</tr>
<tr>
<td>20</td>
<td>8bit ALU Slice Signal Delays</td>
<td>42</td>
</tr>
<tr>
<td>21</td>
<td>32x8bit Register File</td>
<td>44</td>
</tr>
<tr>
<td>22</td>
<td>Memory Signals for 1ns Read and Write Cycles</td>
<td>45</td>
</tr>
<tr>
<td>23</td>
<td>Layout of Register File Circuit</td>
<td>46</td>
</tr>
<tr>
<td>24</td>
<td>Digital Delay Element</td>
<td>48</td>
</tr>
<tr>
<td>25</td>
<td>Differential Schmitt Trigger</td>
<td>49</td>
</tr>
<tr>
<td>26</td>
<td>Characteristic of Delay Element</td>
<td>50</td>
</tr>
<tr>
<td>27</td>
<td>Layout of Digital Delay Line Test Circuit</td>
<td>52</td>
</tr>
<tr>
<td>28</td>
<td>Digital Current Switch Model</td>
<td>58</td>
</tr>
<tr>
<td>29</td>
<td>Simple Data Latch</td>
<td>62</td>
</tr>
<tr>
<td>30</td>
<td>SPICE and Digital XOR3T Model</td>
<td>64</td>
</tr>
<tr>
<td>31</td>
<td>SPICE and Digital AND3T Model</td>
<td>65</td>
</tr>
<tr>
<td>32</td>
<td>Glitch of Three Input AND Tree</td>
<td>66</td>
</tr>
<tr>
<td>33</td>
<td>FRISC Wafer Scale System</td>
<td>69</td>
</tr>
<tr>
<td>34</td>
<td>FRISC Architecture</td>
<td>73</td>
</tr>
<tr>
<td>35</td>
<td>User Visible State</td>
<td>78</td>
</tr>
<tr>
<td>36</td>
<td>Instruction Pipeline</td>
<td>80</td>
</tr>
<tr>
<td>37</td>
<td>Cache and Instruction Pipelines</td>
<td>83</td>
</tr>
<tr>
<td>38</td>
<td>Usage of Resources</td>
<td>87</td>
</tr>
<tr>
<td>39</td>
<td>Dataflow during I1,I2 Pipeline Stages</td>
<td>88</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>79</td>
<td>Bit Slice of Program Counter</td>
<td>179</td>
</tr>
<tr>
<td>80</td>
<td>Controller for a Pipeline Stage</td>
<td>182</td>
</tr>
<tr>
<td>81</td>
<td>Instruction Decoder</td>
<td>185</td>
</tr>
<tr>
<td>82</td>
<td>400mil by 400mil Fabrication Reticle</td>
<td>191</td>
</tr>
<tr>
<td>83</td>
<td>Signal Distribution</td>
<td>192</td>
</tr>
<tr>
<td>84</td>
<td>Schematic with Differential and Single Ended Cells</td>
<td>195</td>
</tr>
<tr>
<td>85</td>
<td>CAD Tooling Stream at RPI</td>
<td>199</td>
</tr>
<tr>
<td>86</td>
<td>CAD Tooling Stream used for Test Circuits</td>
<td>201</td>
</tr>
</tbody>
</table>
List of Tables

Table I. How to Boost Performance of RISCs ........................................... 12
Table II. GST1 Minimal NPN Device Parameters ........................................... 18
Table III. Worst Case Silicon Delays for 32bit Add ........................................... 42
Table IV. Key Memory Parameters .......................................................... 47
Table V. Typical Logic Delays .............................................................. 54
Table VI. Interconnect Delays ............................................................... 55
Table VII. Parameters of Current Switch Model ............................................ 63
Table VIII. SPICE and FASTSIM Results without Interconnect Delays ................. 67
Table IX. SPICE and FASTSIM Results with Interconnect Delays ......................... 68
Table X. Scheduling of ALU and Branch Instructions ....................................... 84
Table XI. Scheduling of Load/Store Instructions ............................................ 86
Table XII. Timing Constraints .............................................................. 96
Table XIII. Branch Fill in Probabilities ...................................................... 98
Table XIV. Branch Performance ............................................................ 100
Table XV. Dynamic Instruction Mix ........................................................ 102
Table XVI. Cache Miss Ratio .............................................................. 102
Table XVII. System Memory Parameters ................................................... 103
Table XVIII. Number of Cycles per Instruction ............................................. 105
Table XIX. Sustained Performance .......................................................... 106
Table XX. FRISC-F Input/Output Signals .................................................. 136
Table XXI. Trap Vectors .................................................................. 144
Table XXII. Comparison of 32Bit Adder Delays ............................................ 165
Table XXIII. Delays on Critical Feed Forward and Shifter Path ......................... 174
Table XXIV. Instruction Decoder Signal Distribution Delays ............................. 193
<table>
<thead>
<tr>
<th>Insert</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Branch Execution Schemes</td>
<td>99</td>
</tr>
<tr>
<td>2. Instruction Set Overview</td>
<td>109</td>
</tr>
<tr>
<td>3. Long Immediate Instructions</td>
<td>110</td>
</tr>
<tr>
<td>4. Synthesized Instructions</td>
<td>113</td>
</tr>
<tr>
<td>5. Restart of Instruction Pipeline</td>
<td>143</td>
</tr>
<tr>
<td>6. Sample Instruction Trace</td>
<td>152</td>
</tr>
<tr>
<td>7. Format of User Visible State Display</td>
<td>154</td>
</tr>
<tr>
<td>8. Simulator Help Message</td>
<td>155</td>
</tr>
<tr>
<td>9. Overflow Detection</td>
<td>169</td>
</tr>
<tr>
<td>10. Datapath Signals</td>
<td>181</td>
</tr>
<tr>
<td>11. Instruction Decoder Signals</td>
<td>190</td>
</tr>
<tr>
<td>12. Port Declaration for Standard Cells</td>
<td>197</td>
</tr>
<tr>
<td>13. Netlist created by TOTEKSIM</td>
<td>198</td>
</tr>
</tbody>
</table>
Acknowledgement

The author would like to thank Professor J. McDonald for the guidance and encouragement he provided. Further, he would like to thank Ted Creedon, formerly with Tektronix, for his strong support of the FRISC project. The many technical discussion with Val Garuts, Einar Traa, and Arnie Frisch at Tektronix are also gratefully acknowledged.

Special thanks are also due to the many students that worked on the FRISC project at Rensselear Polytechnic Institute. The author would like to thank especially Saheer Lahouar, Doug Lopata, Doug Pricer, and Paul Charlton for the long hours they spent on the FRISC project. Further, the author would like to thank Bob Philhower for proof reading the manuscript. His many suggestions were very helpful.

The author would like to thank Tektronix for their support and the fabrication of four Advanced Bipolar test circuits. Partial support from DARPA is also gratefully acknowledged. Finally, the author would like to thank the Fulbright program for the scholarship received during his first year at Rensselear Polytechnic Institute.
Abstract

A high speed differential logic family based on current switch trees has been developed. The standard cell logic family is targeted for the Advanced Bipolar Process GST1 under development at Tektronix. Modelling techniques have been developed and Computer Aided Design tools have been modified or extended to support the design with this high speed bipolar logic family and its particularities.

A highly pipelined 32bit Fast Reduced Instruction Set Computer (FRISC) architecture has been developed for a partitioned implementation on a Wafer Scale Hybrid (WSH). The processor has been designed such that it can be partitioned into it slices that can be fabricated with high speed circuit technologies that do not have VLSI capability because of yield and power limitations. However, a dense WSH multichip package with high density interconnect is necessary to reduce the packaging delays introduced by the partitioning.

The FRISC architecture is described and the effect of its seven stage instruction pipeline on the hardware implementation and performance are discussed. The implementation of the critical hardware sections with Advanced Bipolar current tree logic is presented. SPICE and logic simulations of the critical delay paths show that a 4ns cycle time can be achieved which would result in a peak throughput of 250MIPS.
Chapter I.

Introduction

In the early 1980’s Wafer Scale Integration and Wafer Scale Packaging research was initiated at Rensslelar Polytechnic Institute. These dense packaging strategies were originally targeted for GaAs systems since GaAs circuits have low yield and high power dissipation but provide much higher switching speeds than CMOS circuits. Hence, electronic systems need to be partitioned into small integrated circuits that can be fabricated with high yield. However, the high speed inherent in the GaAs devices will only be visible at the system level if the packaging is very dense since partitioning adds driver, interconnect, and receiver delays on critical delay path. Packaging individual dies and mounting them on a printed circuit board would spread out the system over a large area and would therefore significantly reduce performance. Since GaAs systems have to compete with VLSI implementations in mature CMOS technologies the speed advantage of the device technology must be clearly visible at the system level to warrant the higher fabrication cost.

The dies are tested on the wafer and the interconnect is built on top of the wafer for Wafer Scale Integration. High speed routing software is required with programmable lithography as provided by an electron beam machine since the
location of the functioning dies and hence the interconnect changes from wafer to wafer. The processing steps for the interconnect on top of the wafer must have nearly perfect yield or repair capability to obtain acceptable wafer yield. Processing should further only require low temperature steps since some of the working dies might be damaged during high temperature processing steps. These restrictions and the high yield requirements make Wafer Scale Integration very challenging [JFM84]. However, the same technology can be used for Wafer Scale Hybrid (WSH) packaging [BJD86].

The wafers are diced and the functional dies are mounted on a hybrid with wafer scale interconnect for Wafer Scale Hybrid packaging. The WSH package is also known in industry as Thin Film Multichip Package. Since all WSH substrates are identical mask based lithography can be employed and mass production is less expensive. Further, the yield requirements are not as high as for Wafer Scale Integration since the silicon or aluminum substrates typically used are inexpensive compared to fully processed wafers.

The WSH package has important advantages over printed circuit board or Multi-Layer Ceramic (MLC) packages. The polyimide polymer that is used as dielectric material has a low dielectric constant of 3.2 resulting in a fast signal propagation of 17cm/ns and a low interconnect capacitance of 1.2pF/cm. The low dielectric constant together with the integrated circuit type fabrication process allows to use thin layers and fine line geometries for 50Ω transmission lines. The
Terminator

Die

Bypass Capacitor

Via Stud

Thermal Column

VCC Plane
Dielectric Layer
VEE Plane
Dielectric Layer
X Signal Layer
Dielectric Layer
VT Plane
Dielectric Layer
Y Signal Layer
Dielectric Layer
Metal Substrate (VSS)
dielectric and metal layers are typically 5-10μm thick. A crosssection of a Wafer Scale Hybrid is shown in Figure 1.

Only two wiring layers are needed for most applications because of the high wiring pitch (25-40μm). The wiring pitch is basically determined by the amount of crosstalk that can be tolerated and not by the process resolution. The ground planes for the 50Ω transmission lines are also used for power distribution. The transmission lines must be terminated with 50Ω resistors to avoid reflections. The transmission lines are slightly lossy because of the DC resistance and the skin effect resistance. At 10GHz the skin depth is only 0.82μm for aluminum wires. The Laplace transform for the signal received at the end of a transmission line with length l, width w, and height h can be derived from the telegraphers equations. However, the skin effect resistance (α'√s) must be included since it dominates at high frequencies.

\[
V_z(s) = V_a(s) * e^{-\gamma(s) * l} \]
\[
\gamma(s) = \sqrt{(R' + sL') + \alpha'(s) \sqrt{s}} * s * C' \]
\[
\alpha' = \frac{\sqrt{\mu_0 \sigma / 2}}{2 * h * w} \]

The minimal interconnect bandwidth can be estimated from the signal rise time using a single dominant pole approximation.

\[
f_{\text{signal_min}} = \frac{0.35}{\tau_{\text{rise}}} \]
The bandwidth of the interconnect can be adjusted by using thicker dielectric layers and thicker metal layers. Increasing the crosssection of the transmission lines will reduce the DC loss as well as the skin effect loss. The required interconnect bandwidth for signals with 100ps rise times is 3.5GHz.

**Figure 2. Signal Degradation on Lossy Transmission Lines**

Figure 2 shows the signal degradation for a 1V step on 5cm long aluminum transmission lines with a crosssection of 5μm×5μm and 10μm×10μm in a polyimide dielectric. Vr0 shows the signal at the receiver end of an ideal transmission line. The trace Vr1 is for a transmission line with a 10μm×10μm crosssection and Vr2 is
for a transmission line with a 5μm×5μm crossection. The impulse responses show a sharp signal rise after an ideal transmission delay. However, the losses reduce the signal amplitude and cause distortion. The signal degradation for 5cm long transmission lines with a crossection of 5μm×5μm is already unacceptable if the threshold level is at 0.5V.

Power bypass capacitors are needed to store charge locally since the power supply inductance is in general too high to respond to fast switching transients. The bypass capacitor(s) could be built into the WSH structure using a thin dielectric with a high dielectric constant for the capacitor [HJG86]. The terminating resistors could also be implemented in a nichrome layer. These modifications would avoid the cost for mounting a large number of bypass capacitors and terminating resistors and increase package density.

Since polyimide is a bad heat conductor thermal columns are needed to dissipate the power from the backside of the integrated circuits to the hybrid substrate. The substrate material has a high thermal conductivity and serves as a cold plate. The preferred material for the substrate and interconnect is aluminum since it has almost the same thermal expansion coefficient as polyimide. thermal resistance of the columns is quite small since they are very (≈100μm). The dies are mounted almost directly on top of a heat sink!

The electrical connections from the hybrid to the die can be made with wire bonds as shown in Figure . However, the high inductance of bond wire causes
voltage drops on power supply voltages. Any changes in current cause $\Delta I$ noise. Many parallel power connections are needed to limit the $\Delta I$ noise. By using tab bonding or a flip chip die mount and solder bump technology as in IBM's Thermal Conduction Module (TCM) [AJB82], the $\Delta I$ noise can be significantly reduced and the number of signal connections per die can be increased. However, the heat removal from flipped dies is more difficult resulting in an expensive package. Spring loaded pistons are used in IBM's TCM to remove the heat from the backside of the dies.

The packaging research led to the Fast Reduced Instruction Set Computer (FRISC) project. What type of processor can be effectively implemented with dense Wafer Scale Hybrid packaging and advanced circuit technology? Does the performance of such a system still reflect the high speed of the circuit technology and thus warrant the high implementation cost?

In October 1975 the 801 minicomputer project was started at IBM [GR83]. The 801 minicomputer is considered to be the first Reduced Instruction Set Computer (RISC). By studying the frequency with which instructions are executed it was found that many of the complex instructions especially memory to memory instructions are almost never executed and that simple instruction like Load Register, Store Register, and Register to Register instructions account for 70% of the dynamic instruction mix. This implies that it is not worth implementing an
instruction if its implementation affects the execution time of frequently used instructions unless the frequency of use justifies it.

The following RISC principles were formulated. Only a simple instruction set is implemented using hardwired control. The hardware resources for the microsequencer and the large microcode ROM needed in Complex Instruction Set Computer (CISC) architectures is thereby eliminated freeing hardware resources for the implementation of a large register file. Complex instructions are implemented as a sequence of primitive instructions by the compiler. The compiler should further optimize the usage of the large register file and reduce or prevent pipeline interlocks. The processor hardware is simpler and faster if no pipeline interlocks must be detected and resolved at run time. All instructions execute in a single cycle. This allows streamlining and pipelining of the instruction execution and thus increases performance. Further, RISC processors have a load/store architecture, only LOAD/STORE instructions can access external memory.

The 801 minicomputer project was considered a success but the Emitter Coupled Logic (ECL) machine was never offered commercially despite its high performance. However, the experience gained from the 801 project was later on applied to the design of the IBM's RISC workstations.

In 1980 the Berkeley RISC project was started with RISC I being completed in 1982. The goal of the project was to fit a full 32bit microprocessor onto a single 4μm NMOS chip at time when this was not feasible for CISC architectures
[DAP82]. The RISC performance compared favorably with CISC processors, outperforming the 16bit 68000, Z8000, and even the 32bit VAX 11-780. A faster version called RISC II was designed by Manolis G. Katevenis and Bob Sherburne [RWS84] and fabricated in 1983. The RISC II processor is also known as the Berkeley RISC. It features a barrel shifter and a large register file with 138 registers divided into 8 overlapping windows with 22 registers. The multiple register file windows speed up procedure calls and context switches.

Another early RISC processor is Stanford’s Microprocessor without Interlocked Pipe Stages (MIPS) [JH83]. The high performance of RISC processors and the short design time led to the development of many different architectures. Most semiconductor and computer companies offer today RISC processors besides their CISC processors; Hewlett Packard: Precision Architecture, Motorla: MC88000, Fairchild: Clipper, Sun: SPARC, Intel: i860, DEC: MIPS, and IBM: RS6000.

The low hardware cost make RISC architectures easier to implement with low integration level circuits than CISC architectures. Further, the high performance and pipelined instruction execution make RISC very attractive for high speed circuit technologies. Since a lot of software was already available for early RISC processors like RISC II and MIPS the question was whether these processors could be effectively partitioned and implemented with LSI circuits. Unfortunately the existing architectures contain features that can not be effectively partitioned or
implemented because of I/O pin or power limitations. A large windowed register file can not be implemented because of the process yield and the high power dissipation of bipolar memories. A barrel shifter can not be effectively partitioned because of its high connectivity. The partitioning also forces the alignment of certain instruction fields with bit slices boundaries. Hence, code compatibility is not feasible. However, the RISC principles used for the design of these early processor can be applied for the design of a new architecture.

An initial study of the performance for an Enhancement/Depletion Mode GaAs 1μm MESFET RISC showed that the partitioning still had a significant impact on performance even with dense Wafer Scale packaging [JFM86]. Even if the chip to chip interconnect delays are small the driver and receiver delays introduced by partitioning have a significant performance impact. The process yield must allow the implementation of 8bit slices (LSI circuits) in order to get good performance. Later on our research group got access to design information for the Advanced Bipolar Process under development at Tektronix. The Advanced Bipolar devices have better performance than 1μm MESFET's. In addition the high drive capability of bipolar transistors provides low I/O delays, a crucial parameter for partitioned systems. Thus FRISC was finally targeted for Tektronix's Advanced Bipolar process.

With Advanced Bipolar technology a short cycle time of 4ns seems feasible. Since RISC processors can fetch an instruction or data word in every
cycle high speed cache memories must be provided that can deliver an
instructions or data word every 4ns and isolate the processor from slow main
memory. The cache memories must be implemented very close to the processor
to avoid large signal transfer delays. In addition the cache memory access
including address transfer, memory read/wire, and data transfer needs to be
pipelined to reduce the timing constraints on the cache memories. Even if the
processor can be implemented on a single VLSI chip a dense multichip package
for the processor and the cache memories is indispensable.

By relaxing the timing constraints cache memories with higher density and
lower power dissipation can be used. This allows the implementation of larger and
thus more effective caches. Two pipeline stages must be allocated for pipelined
data or instruction memory access providing a full cycle for cache memory read
access. Processors operating at very short cycle times have therefore in general
deeper instruction pipelines. This can be observed in the VLSI GaAs RISC
processors [BAN87] under development for the Department of Defense. The GaAs
Heterojunction Integrated Injection Logic (HIIL) processor that is under joint
development by Texas Instruments and Control Data Corporation has six pipeline
stages with a targeted cycle time of 5ns [VM90].
Contributions from the following areas help to boost the performance of FRISC:

<table>
<thead>
<tr>
<th>Contribution</th>
<th>Speedup over RISC II</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dense Packaging</td>
<td>1</td>
</tr>
<tr>
<td>Advanced Circuit Technology</td>
<td>(3ns/90ps)</td>
</tr>
<tr>
<td>High Speed Logic Family</td>
<td>2/1</td>
</tr>
<tr>
<td>Architecture / Pipelining</td>
<td>7/3</td>
</tr>
</tbody>
</table>

Table I. How to Boost Performance of RISCs

The estimated speedup over RISC II is also shown to indicate the weight of a contribution. The speedup is based upon a comparison with the RISC II from Berkeley because this processor has been well documented in literature [MGK85]. RISC II was implemented with a 4μm NMOS technology and has a peak throughput of 2 Million Instructions per Second (MIPS).

The dense packaging does not provide a speedup for the processor itself but it helps to bridge the gap between the complexity of the circuits that can be fabricated and complexity of the processor. The fastest Advanced Bipolar inverters have a delay of 90ps whereas the fastest NMOS inverters used in RISC II have a propagation delay of 3ns. Hence, without considering the need for partitioning a speedup of 33 could be expected. By comparing the implementation of the critical ALU path in both processors a speedup factor of 2 is estimated for the
flexibility of the current tree logic family developed for FRISC. Only one gate is needed for carry propagation from one bit to the next whereas two NOR gates are necessary in RISC II. FRISC has a 7 stage pipeline whereas RISC II has only a 3 stage pipeline. Hence, the pipeline speedup factor is 7/3. All the speedup factors result in an estimated peak throughput of 311MIPS for FRISC. However, extensive simulations of the critical paths of FRISC indicate a performance of only 250MIPS. This is not surprising since the speedup factor for the Advanced Bipolar Circuit technology did not reflect that the yield and power dissipation force the partitioning of the processor. However, the circuit technology definitely provides the most significant speedup. The hardest speedup factor to obtain is the pipeline speedup. The design complexity increases significantly with deeper pipelines because more instructions are executed in parallel.

A FRISC processor with two 32kByte cache memories can be implemented on a 8cm by 8cm hybrid as shown in Figure 3 using 16kBit BICMOS memories. The BICMOS memories have an access time of 3.5ns, a power dissipation of only 500mW, and a small die size of 2.5mmx3.1mm [MS89]. The actual processor core consisting of an instruction decoder chip and four 8bit datapath slices is quite small (32mmx12mm). The gap between adjacent chips is 2mm. The 5mm gaps between the different blocks provide space for bypass capacitors and the implementation of nichrome terminating resistors.
Figure 3. FRISC and Cache Memories on Wafer Scale Hybrid

The two 32kByte cache memories occupy most of the area. Each cache memory consists of 26 memory dies and two interface chips. The estimated power dissipation for the processor core is 58W. The total power dissipation is highly dependent upon the cache memory technology. Equivalent ECL memories with a
capacity of 16kBit and an access time of 3.5ns dissipate 2W [NH86]. Pipelined 4kBit GaAs memories from GigaBit Logic have an access time of only 3ns but have a power dissipation of 3.9W. Hence, high density BICMOS memories are needed to be able to implement the processor with two 32kByte cache memories on a WSH while keeping the total power dissipation below 100W. With low power BICMOS memories the average power dissipation is below 2W/cm². However, the power dissipation of the datapath chips (12W) is too high for air cooling. The hybrid substrate (cold plate) must be kept about 20°C below the maximum junction temperature of 100°C because of the thermal resistance of the epoxy used for die attachment and the small resistance of the thermal columns.

The following chapters describe the differential current tree logic family, the FRISC architecture, and the gate level implementation of the datapath in detail. The chapter II describes Tektronix’s Advanced Bipolar process and the standard cell library as well as the implementation of critical circuits like drivers and receiver, the Arithmetic Logical Unit (ALU), and the register file. Further, the four test circuits that have been designed to test the most critical circuits of the processor are shown. Chapter III describes how the differential current trees can be accurately modelled on a digital simulator. The chapter IV gives a detailed description of the FRISC architecture. Specifically, the instruction pipeline, the instruction set, instruction latencies, interrupt handling, critical delay path, and the sustained performance are discussed. In addition the FRISC assembler syntax is defined.
The chapter V describes the Register Transfer Model (RTM) of a FRISC-F system implemented on the VERILOG behavioral simulator. The features of the model and the user interface are presented. Chapter VI describes the implementation of the datapath slices with differential current tree logic and shows the critical delays. Further, it shows how the instruction decoder for a highly pipelined processor can be implemented. Chapter VII shows which CAD tools have been used and the modifications that have been made for the FRISC project. The source for the assembler and the RTM model of FRISC are listed in the appendix.
Chapter II.

Advanced Bipolar Circuit Technology

A. Advanced Bipolar Process

![Device Structure](image)

Figure 4. Device Structure (from [TY88])

The GST1 Advanced Bipolar NPN transistor devices are built with a self aligned polysilicon emitter-base process with a coupling base implant. This results in shallow emitter and base junction depths [HKP86]. The 1μm trench isolation reduces the collector to substrate capacitance and increases device density. The smallest transistors have an emitter width of 0.6μm and can be placed on an 8μm x 12μm grid. A self aligned titanium silicide layer on top of the polysilicon layer
used for emitter and collector contacts reduces the sheet resistance to $1\Omega\square$ and provides thereby an additional layer for short interconnect. Without the silicide layer the polysilicon layer is resistive and yields a space efficient implementation of resistors. Two gold metal layers with a 4μm pitch are available for interconnect. Figure 4 shows the structure of GST1 devices and polysilicon resistors. The Advanced Bipolar NPN devices have a maximum $f_t$ of 15.5GHz [TY88]. Propagation delays of 55ps/stage have been measured in ring oscillators. Further, dual 4-bit analog to digital converters with a performance of .5Gs/s have been demonstrated [VEG88]. The dimensions and key parameters of the smallest GST1 device are summarized in the following table.

<table>
<thead>
<tr>
<th>GST1 Minimal NPN Device Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
</tr>
<tr>
<td>Emitter Size</td>
</tr>
<tr>
<td>Current Gain $h_{fe}$</td>
</tr>
<tr>
<td>E-B Capacitance</td>
</tr>
<tr>
<td>B-C Capacitance</td>
</tr>
<tr>
<td>C-S Capacitance</td>
</tr>
<tr>
<td>Cut-Off Frequency $f_{t}, Vo=0.85V, T=300K$</td>
</tr>
</tbody>
</table>

Table II. GST1 Minimal NPN Device Parameters
B. Current Switch

\[ \text{Figure 5. Bipolar Current Switch} \]

Figure 5 shows a current switch, the basic building block for bipolar current tree logic. The input current into the common emitter node is switched left or right depending upon the two base voltages. Using a simplified Ebers-Moll model for the bipolar transistor the DC characteristics of a current switch buffer can be expressed in a closed form [RLT88] as shown in Figure 5. However, the parasitic emitter and base resistances should be considered in the analysis to obtain a good fit to measured data. Unfortunately the analysis does not yield a closed form solution even if only the major parasitic elements are included.
Figure 6. CML and ECL Buffer Delays versus Switching Current

Figure 6 shows the delay of current switch buffers with minimal GST1 devices as a function of switching current and with a fanout of one. The delay without and with 500\(\mu\)m of interconnect capacitance are shown. The switching current was fixed at 400\(\mu\)A by a trade off between switching speed versus power. With a logic swing of \(\pm 250\)mV and a fanout of one a Current Mode Logic (CML) buffer has a delay of 69ps and an Emitter Coupled Logic (ECL) buffer with an emitter follower current of 800\(\mu\)A has a delay of 81ps. The CML buffer dissipates only 2mW but has a propagation delay sensitivity \(R_s\) of 400\(\Omega\). The delay sensitivity
$R_s$ multiplied by the load capacitance $C_{load}$ gives the incremental propagation delay due to capacitive or interconnect loading. A linear delay dependence upon capacitive loading is a good approximation for ECL type circuits.

$$T_{delay} = T_0 + R_s * C_{load}$$

The ECL buffer has a power dissipation of 10mW with an $R_s$ of only 119$\Omega$. To reduce power CML is used within standard cells where the interconnect length is short. ECL output drivers are used to drive interconnect between standard cells. A low delay sensitivity is important for standard cells since interconnect loading can vary in a wide range. Further, emitter followers not only lower the delay sensitivity but also shift the signal level down by one Vbe drop. Three offset levels are needed to avoid saturation in a tree with three current switch levels.

Figure 7 shows the DC transfer curve of a current switch. The nominal voltage swing was fixed at ±250mV. This drives the current switch well beyond the points with maximum noise margin (gain=1) and results in a voltage gain of 2.6 at 360K. The voltage swing is determined by a trade off between the delay sensitivity to capacitive loading and the voltage gain & noise margin of the logic elements. Further, the current should be switched almost 100% left or right at nominal input voltage levels. Otherwise logic level degradation will occur if current switches are cascaded or stacked to build current trees.
The voltage swing is an important characteristic of a logic family. A simple gate model can reveal the relationship between the voltage swing and the time it takes to charge or discharge the wiring capacitance at the output of a gate. The output of a gate can be approximated by two current sources that are switched on and off to apply charge or remove charge from the output signal wire. If the wire capacitance is $C_w$ and the switching current is $I_s$ the following relation ship holds for the switching time $\Delta t_s$ if the logic swing is $V$: 

$$ I_s = 400\mu A, R_L=625\text{ohm} \quad \text{gain} = 2.3 \text{ at } 300K / \text{gain} = 2.6 \text{ at } 360K $$

**Figure 7. Characteristic of Current Switch** ($R_L=625\Omega, I_s=400\mu A$)
\[ \Delta t_s = \frac{V_i \cdot C_w}{I_s} = R_s \cdot C_w \]

To obtain a low sensitivity \( R_s \) towards capacitive loading of a gate and hence, fast loaded switching times either the logic swing \( V_i \) must be low or the switching current \( I_s \) must be high. A high switching current increases, however, the power dissipation of the gate. Hence, a low logic swing is the only option to obtain a low power, high speed gate family. However, the devices must exhibit high gain and the switching noise of the gates must be low to support low logic swings! Bipolar logic with a logic swing of only 250mV has a big advantage over CMOS with a logic swing of 3-5V in this respect.

C. Differential Current Tree Logic

The high speed and low switching noise of differential logic make it very attractive for Bipolar [MPD89,MS88] or GaAs logic [SK85]. Differential GaAs logic is called Source Coupled FET Logic (SCFL). The high performance and efficient logic implementation of cascaded differential logic trees has led to the development of a similar CMOS logic family at IBM [LGH84], called Cascode Voltage Switch Logic (CVSL).

Figure 8 shows a differential AND/OR gate with three levels of series gating and Figure 9 shows a conventional single ended OR gate. The single ended OR gate needs twice the voltage swing of the differential gate to obtain the same noise
Current Tree Emitter Followers

Figure 8. Differential Three Input AND Gate

margin. The reference voltage for single ended gates is nominally in the middle of the logic high and logic low levels. Twice the voltage swing is sufficient, despite the fact that the generation of the reference voltages is sensitive to supply voltage drops on power rails, because doubling the voltage swing also doubles the maximum gain of the current switch. To obtain twice the voltage swing either the load resistance \( R_i \) or the switching current \( I_s \) must be increased by a factor of two.
The three input signals of the differential AND gate have different offset voltage levels. These levels are offset by one base emitter junction voltage $V_{BE0}=0.85V$ to avoid saturating the bipolar devices at the different levels in the tree. The logic swing at each level is nominally $\pm 250mV$ at $T=300K$. Stuck at faults or leakage currents that reduce the voltage swing of a signal can cause testability problems. In such cases the circuit might show full functionality at DC tests but fail to operate at full speed or over the full temperature range.

Any boolean function of three variables can be implemented with a single current tree by using collector dotting at the top level since a full current tree with
Figure 10. Carry Propagate Gate

three levels of current switches forms a three to eight decoder. An efficient logic implementation is obtained by eliminating current switches with both collectors connected together and by using collector dotting at level two for intermediate decoding states. For example in the carry propagate gate shown in Figure 10 only four of seven current switches are needed. A four input multiplexer gate can also be implemented with a single current tree as shown in Figure 11. By using feedback from the outputs of the current tree data latches with any dual input gate can be implemented as shown in Figure 12. The feedback signals are taken from the top of the tree rather than from the output because of layout considerations.

Differential signals can be inverted with zero delay and power by exchanging the true and inverted signal pair connections at any input or output
Figure 11. Differential Four Input Multiplexer

This reduces the number of cells in the standard cell library since dual gates like AND/OR are physically identical. However, both gates are available in the schematic library for the designer. Dual gates get mapped into the same cell during netlist generation.

Emitter followers are used to increase the drive capability of the gates and to shift output levels. A standard cell has only one output at a fixed offset level
because emitter followers tend to ring if they have to drive outputs at multiple levels. Since most of the power is dissipated in the emitter followers, each logic gate is available with three different drivers to give the designer flexibility in power allocation. Low power gates (6mW) have emitter followers with $I_e=400\mu A$, medium power gates (10mW) have followers with $I_e=800\mu A$, and high power gates (14mW) have followers with $I_e=1.2mA$. Within gates that require multiple current trees for implementation, like master/slave latches, current mode logic is used for connecting the trees to reduce power. The power of master/slave latches is, therefore, only 2mW higher than the power of simple data latches.
The propagation delays of differential logic depend upon the path the current takes through the tree. The delay from inputs at a given level to the top of the tree can, therefore, depend upon input signals at higher levels. This makes it difficult to model a current tree AND with a simple digital AND primitive as provided by digital simulators. For example in the differential AND gate shown in Figure 8 the delay from the lowest level input depends upon whether the current flows through current switch S2 to q or through S2 and S3 to q or qb. The maximum propagation delays for a medium power AND gate with a level one output are 90ps from level one (input c), 135ps from level two (input b), and 180ps from level three (input a). These delays are valid for a positive input signal transition and a fanout of one. The single ended OR gate has a propagation delay of 95ps for the OR output. The delay sensitivity $R_s$ of the single ended OR gate is 131Ω for the rising edge and 257Ω for the falling edge with a power dissipation of 10.5mW. The medium power differential AND/OR gate has a power dissipation of 10mW and a delay sensitivity $R_s$ of only 116Ω. The differential gate has no decisive speed advantage over the single ended gate if only the propagation delays with a fanout of one are considered. However, the delay sensitivity of the differential gate is considerably lower and does not depend upon the signal transition. The delay sensitivity values $R_s$ have been matched to SPICE simulation results by a six point linear regression analysis in the interval 0-500fF. The delays were measured between zero crossings of differential input and output signal pairs.
While differential logic is faster than single ended logic due to its low logic swing and can be efficiently implemented with current trees there are also disadvantages. Twice as many signal interconnections must be routed. This increases the average interconnect length since the density of digital bipolar circuits is generally interconnect limited. Further, two emitter followers are needed for every gate which increases power dissipation. However, differential logic requires no power for inverters or reference voltage generators, its sensitivity towards voltage drops on power rails is low, and its switching noise is very small. High speed switching transients cause a voltage drop on the parasitic power supply inductance. This can lead to a saturation of the bipolar devices if the collector-emitter voltage drops below \( \approx 400\text{mV} \).

Existing CAD tools can easily be modified to support three different offset levels, differential signal inversion, and checking of input level violations that cause saturation in standard cells. However, the designer has to assign output levels for each gate avoiding level violations and keeping the propagation delays on the critical path minimal. The standard cell router should support differential wiring. Differential wires should be routed right next to each other to have equal loading on differential nets. Parallel routing of differential signals further reduces crosstalk since the electric fields will be concentrated between the differential wire pairs and crosstalk signals will couple almost equally to both wires and thereby produce
mainly a common mode crosstalk noise. The current switches will largely reject common mode noise.

D. Emitter Followers and Buffers

Emitter followers have a tendency to ring which increases the output settling time. The input impedance of an emitter follower with a capacitive load has an inductive component at high frequencies. Since there are always parasitic capacitances and inductances at the base node LC type oscillations can be exited. Propagation delays are quite difficult to predict for input signals that arrive while the outputs have not yet settled. Therefore, emitter follower and level shifter configurations were developed to obtain faster settling times and lower interconnect delays.

The improved emitter followers have a damping resistor between the differential outputs to reduce ringing as shown in Figure 13. For level two and three level shifters a f doubler circuit is used which reduces ringing but also increases driving capability. The damping resistors cause a maximum loss of 20mV in voltage swing since the current flowing through the emitter base junction is higher for the transistor with a logic high output signal. Figure 14 shows the delays of buffers with the emitter followers and level shifters shown in Figure 13. The basic buffers for level two and three show longer propagation delays than the improved buffers. Only the basic level 1 buffer shows an 8ps lower propagation
Figure 13. Basic and Improved Emitter Followers
delay without load. However, it has an underdamped step response with a long settling time at low loads. Current sources are used for the emitter followers and level shifters rather than long resistor tails since resistors would require more space due to the low sheet resistance of the polysilicon layer.

Emitter followers under high capacitive loads have largely different rise and fall delays. The rise time delay is quite small due to the high transconductance $g_m$ of the bipolar devices. The fall time is dominated by the available pull down current. This leads to highly asymmetrical transitions in current starved ECL.
Figure 14. Delays of ECL Buffers
delay without load. However, it has an underdamped step response with a long settling time at low loads. Current sources are used for the emitter followers and level shifters rather than long resistor tails since resistors would require more space due to the low sheet resistance of the polysilicon layer.

Emitter followers under high capacitive loads have largely different rise and fall delays. The rise time delay is quite small due to the high transconductance $g_m$ of the bipolar devices. The fall time is dominated by the available pull down current. This leads to highly asymmetrical transitions in current starved ECL.
A special buffer is available for driving long interconnect lines as encountered in clock distribution trees. This super buffer (SBUF1H) has a delay of only 68ps and a sensitivity $R_s$ of only 60Ω at a power dissipation of 12mW. The SBUF1H circuit consists of a current switch buffer with a switched current source for the emitter followers as shown in Figure 15. This results in a push pull output stage with a high pull down current of 2mA. Resistor R3 provides damping and keeps a minimal current of 800μA flowing through Q5 or Q6 if the corresponding output signal is high. It also prevents the high output from slowly charging up to the Vcc power level through the base emitter junctions of the output transistors. The SBUF1H is a better driver than a standard ECL buffer with high power output drivers ($T_o=77ps$, $P=14mW$, $R_s=97Ω$) because of its lower power and lower
interconnect sensitivity. However, the equivalent input capacitance of the super buffer is three times higher.

E. I/O Circuits

High speed I/O drivers are especially important in Advanced Bipolar logic since large circuits need to be partitioned because of power dissipation limits and fabrication yields. Two different types of drivers & receivers are provided as shown in Figure 16. Single ended 10K ECL compatible drivers/receivers with a driver plus receiver delay of 300ps for a rising edge and 312ps for a falling edge with an I/O pad capacitance of 1pF. The driver has standard ECL level outputs (high -860mV, low -1.725mV at 360K). The transmission line must be terminated with a 50Ω resistor to Vtt (-2V). The driver has the typical unbalanced power dissipation of single ended I/O drivers. For a low output the power dissipation is 45mW and for a high output the power dissipation is 80mW. These unbalanced drivers cause considerable switching noise because of voltage drops on bondwires and power rails. Therefore, a dedicated power rail Vpp (0V) is used to keep the noise away from the standard cell core. Bypass capacitors for the Vpp power supply must be connected to Vtt.

The high current (16mA) that is switched on and off by single ended drivers causes a significant voltage drop on the bondwires which have an inductance of about (20pH/mil) and are typically 10-15mils long. Simulations predict 30mV of
Figure 17. Layout of I/O Test Circuits
switching noise for a single ended I/O driver with a 15mil bondwire on the Vpp
power supply. Therefore, only 2-3 drivers can be supplied with one Vpp power pad
else the voltage drop on the bondwire and power rail can cause saturation of the
output devices. By using tab bonding or a flip chip die mount the power supply
inductance could be substantially reduced.

The second driver is a differential open collector driver with a voltage swing
of only ±250mV. The two transmission lines are terminated with 50Ω resistors to
Vcc. The differential driver plus receiver delay is only 240ps with a pad
capacitance of 1pF. Differential drivers have the disadvantage of using up two I/O
pads, however, since they have balanced power dissipation fewer power pads per
I/O driver are required than for single ended drivers. The receivers use the same
circuit configuration as the super buffer to be able to drive the typically long
interconnect lines from the chip periphery to the core with the standard cells.
Figure 17 shows a test chip with single ended and differential I/O drivers and
receivers, an interconnect delay monitor, and a bias voltage generator.

F. ALU Circuit

The 32bit ALU is on a critical path of FRISC since the datapath had to be
partitioned into four 8bit slices. The ALU has a 3ns time slot to produce a 32bit
result from the arrival of the second level operand inputs. A carry select adder is
used to speed up carry propagation. The carry for each slice is calculated in two
parallel carry chains one for an assumed carry in of one and the other for a carry in of zero. The actual carry in of the slice selects only the result of the appropriate carry chain. This reduces the fall through time for the carry to a receiver, multiplexer, and driver delay if the carry chains have had time to settle. Further, the carry in signal of the first slice must only be available on chip when the carry chains have settled. This effectively removes the carry in signal from the critical path. An additional advantage of the carry select scheme is that it can be implemented with only five current trees per bit as shown in Figure 18.
The carry propagate gate CARRP1M and the multiplexer with clear MUXCLR1M are medium power (10mW) gates since they are on the critical path but drive only short interconnect. The programmable function gate ALUMAC2L generates the boolean XOR, OR, or AND function of the two input operands. A low power gate (6mW) is used since it is not on critical path. A high power gate (14mW) is used for the data latch with XOR inputs DLXOR1H since it has to drive long interconnect and is on a critical path.

Differential I/O drivers and receivers are used to minimize the carry fall through time. The ALU can perform ADD, AND, OR, and XOR functions. Subtraction is implemented by inverting the carry in and input operand B. The output latch DLXOR1H not only latches the result but also generates the sum by performing an XOR of the carry and the XOR of the two input operands generated by the ALUMAC2L gate. The table below shows worst case propagation delays for a 32bit add based upon SPICE simulations.

The simulation results include an average on chip interconnect length of 600μm between the clusters of cells that form a bit slice. The two carry propagation cells and the multiplexer are placed right next to each other in every bit slice to keep the interconnect length on the carry chain minimal. The carry in receiver and carry out driver are placed right next to each other to avoid routing the carry in signal all the way across the chip. Long on chip interconnect shows RC type signal propagation with very long rise and fall times and must, therefore,
Figure 19. Layout of ALU Test Circuit
Table III. Worst Case Silicon Delays for 32bit Add

be avoided for critical signals. Figure 20 shows the signal waveforms for an 8bit slice. The four data path slices are mounted right next to each other on a Wafer
Scale Hybrid. The off chip interconnect between slices is at most 8mm long. The micro-transmission lines on the hybrid have a polyimide dielectric with an \( \varepsilon_r \) of 3.2 resulting in a low interconnect delay of 6ps/mm. The 32bit ADD delay is the silicon delay plus three chip to chip interconnect delays (3x48ps) resulting in a worst case delay of 2.792ns. Assuming the clock skew can be controlled within \( \pm 100\)ps the ALU can perform the 32bit ADD within the allocated 3ns time slot. By optimizing the first slice using carry select over a group of three and then five bits the delay of the first slice could be reduced to 850ps resulting in a worst case delay of only 2446ps. Figure 19 shows the layout of an 8bit ALU test circuit. The circuit is configured as a counter to reduce the number of I/O pads.

G. 32x8bit Register File

The development of Advanced Bipolar processes has led to a tremendous reduction in access time for 1Kbit ECL memories. Nippon Telegraph and Telephone Corporation (NTT) has developed several 256x4bit ECL memories over the last decade. In 1978 NTT reported a 7.5ns 1Kbit memory with an access time of 7.5ns and a power dissipation of 784mW using bipolar devices with an \( f_t \) of 2GHz [KK78]. In 1984 NTT reported a 1Kbit memory with a 1.5ns access time and a power dissipation of 714mW using Super Self-Aligned Technology (SST) Advanced Bipolar devices with an \( f_t \) of 7GHz [HM84]. In 1986 a subnanosecond 1Kbit memory was reported with an access time of 0.85ns and a power dissipation
of 950mW using SST Advanced Bipolar devices with an $f_t$ of 12.4GHz and 1μm design rules [HM86].

For many applications the access time to external memory is too large, therefore, small on chip memories are needed that can be implemented without special processing steps. A 32x8bit memory cell with an access time of 600ps and
Figure 22. Memory Signals for 1ns Read and Write Cycles

A power dissipation of 800mW was developed particularly for the register file of FRISC.

Figure 21 shows the circuitry of the 32x8bit memory. A fast and power efficient address decoder with multi-emitter AND gates is used. A feedback resistor (R6) between collector and base is used to obtain a small voltage gain. The voltage swing on the address lines is 750mV and 850mV on the word lines. A novel address line driver is used which provides a slightly larger pull down current than needed for the multi-emitter decoder to speed up wordline deselection. The additional current is important because it reduces noise on the word lines during
Figure 23. Layout of Register File Circuit
address transitions. The memory cell is a Shottky clamped cell with a hold current of 200μA and a hold voltage of ±250mV. The read/write current on the bit lines is 800μA. Bit line voltage sensing is used rather than bit line current sensing because it is slightly faster and simplifies the layout. A word line clamp with 1.6mA of current is used to provide additional current for the deselection of a word line. Figure 22 shows the memory signals for a 1ns read cycle followed by a 1ns write cycle.

<table>
<thead>
<tr>
<th>Key Memory Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Organization</td>
</tr>
<tr>
<td>Access Time</td>
</tr>
<tr>
<td>Power</td>
</tr>
<tr>
<td>Read/Write Current</td>
</tr>
<tr>
<td>Cell Hold Current</td>
</tr>
<tr>
<td>Cell Hold Voltage</td>
</tr>
<tr>
<td>Cell Size</td>
</tr>
</tbody>
</table>

**Table IV. Key Memory Parameters**

Reducing the cell hold current together with a more power efficient read/write logic design resulted in a 20% higher access time but reduced power dissipation by 27% compared to a previous design [HG88]. The address line drivers for the word line decoder dissipate 34%, the memory cells dissipate 32%, the sense amplifiers and bit line current sources dissipate 20%, and the read/write
logic including the threshold voltage generator dissipate 8% of the total power.

Figure 23 shows the layout of a register file test chip.

H. Digital Delay Line

![Figure 24. Digital Delay Element](image)

The clock distribution system of a partitioned system is critical since unequal delays on the clock lines cause clock skew. The clock skew can quickly become a significant part of the minimal clock cycle of GaAs or Advanced Bipolar processors. Hence, clock skew compensation is desirable to increase performance. The unequal delays are due to variations in the clock signal drivers & receivers, clock fanout trees on the individual chips, and variations in the clock signal propagation delays. The length of the clock distribution lines from the central clock...
generation and distribution chip to the different units can be made equal by hand routing these signals to keep the propagation delay differences minimal.

Figure 25. Differential Schmitt Trigger

The skew can be eliminated if each of the clock signals from the central distribution chip can be adjusted by the appropriate amount. The clock signal for a particular chip would be delayed such that its synchronous output signals show minimal skew with respect to an arbitrarily chosen reference signal. The digital delay lines needed for this fine tuning could be implemented directly on the clock generation chip with digital delay line elements as shown in Figure 24. One delay line is needed per clock signal fanout. A fully differential ramp generator with a variable ramp step is connected to a Schmitt trigger circuit with a variable
threshold. This provides a relatively large variable delay compared to the insertion, or minimal delay. Figure 25 shows the differential Schmitt trigger circuit.

![delay line circuit](image)

**Figure 26. Characteristic of Delay Element**

Both the ramp generator and the buffer are differential to obtain equal delays for the rising and falling edge of the clock signal and to reduce switching noise. The delays of the delay lines can be held constant over temperature variations with a phase locked loop (PLL) [PHG88]. The PLL circuit monitors the delay of a matched delay line on the same chip and generates a control signal to keep the insertion delay and the delay of the delay elements independent of temperature.

The delays of each delay line would be programmed by shifting in a digital control code for each delay line. Figure 26 shows the delay characteristic of a
delay line element. The necessary variable delay range for an application is achieved by cascading a sufficient number of delay elements. Figure 27 shows a test circuit with two digital delay lines, two phase detectors and an integrator for the PLL circuit. The delays lines consist of four cascaded delay elements with a differential receiver at the input and a differential open collector driver at the output.
Figure 27. Layout of Digital Delay Line Test Circuit
I. Standard Cell Library

The following list shows the differential standard cells used for the FRISC project. Many cells map into dual logic gates like AND and OR. These dual cells are made available in the schematic library but are mapped onto the same cell during netlist expansion. Further, every input and output port on differential cells can be inverted without inverters. Most cells are available at three different power levels \(<p>\) (low power=6mW, medium power=10mW, high power=14mW) and with three different output levels \(<l>\) (level 1, level 2, level 3). Master/Slave latches are implemented with two current trees and dissipate 2mW more than latches.

Combinational Cells

AND2<p,1> dual input AND gate
XOR2<p,1> dual input XOR gate
AND3<p,1> three input AND gate
XOR3<p,1> three input XOR/Full Adder
COMPl<p,1> comparator with enable
ANDOR<p,1> AND/OR gate
ALUMAC<p,l> programmable AND/XOR/OR gate
CARRYp<l> carry propagate gate

Multiplexer Cells

MUX2<p,1> dual input multiplexer
MUXCLR<p,1> dual input multiplexer with clear
MUX4<p,1> four input multiplexer

Buffers & Level Shifters

BUF<p,l> buffer
SBUFH<l> super buffer
LS<l> level shifter

Storage Cells

SRF<l> set reset flip flop
DL<p,l> simple data latch
DLC<p,l> data latch with synchronous clear
DLAND<p,1> data latch with AND gate inputs
DLXOR<p,1> data latch with XOR gate inputs
DLMUX<p,1> data latch with MUX gate inputs
MSL<p,1> master/slave latch
MSAND<sub>p,l</sub> master/slave latch with AND gate inputs
MSMUX<sub>p,l</sub> master/slave latch with MUX gate inputs

I/O Cells
SEDs    single ended driver ECL 10K
SER    single ended receiver ECL 10K
DD     differential driver
DR     differential receiver

Special Cells
RF32x8  32x8bit memory cell
SYNC  four phase clock generator

A simple delay model is given to the designer which allows quick evaluation of different configurations. The table below gives approximate delay figures for the current switches and the emitter followers.

<table>
<thead>
<tr>
<th>Typical Logic Delays</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Current Switch Delay any input to q, qb outputs</td>
<td>45ps</td>
</tr>
<tr>
<td>Level 1 Emitter Follower + tree interconnect</td>
<td>40ps</td>
</tr>
<tr>
<td>Level 2 Emitter Follower + tree interconnect</td>
<td>45ps</td>
</tr>
<tr>
<td>Level 3 Emitter Follower + tree interconnect</td>
<td>55ps</td>
</tr>
<tr>
<td>Fanout Penalty per Current Switch</td>
<td>5ps</td>
</tr>
</tbody>
</table>

**Table V. Typical Logic Delays**

The fanout penalty for a medium power gate is only 5ps per active current switch. However, gates like the four input multiplexer shown in Figure 11 have two current switches connected to the same cell input port. Only one of the current
switches can, however, be active. A detailed delay model will be described in the following chapter. The following table shows typical interconnect delays and their dependence upon gate power.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Low Power Gate 6mW $I_{dr}=400\mu A$</th>
<th>Medium Power Gate 10mW $I_{dr}=800\mu A$</th>
<th>High Power Gate 14mW $I_{dr}=1.2mA$</th>
</tr>
</thead>
<tbody>
<tr>
<td>level 1</td>
<td>48ps/mm</td>
<td>29ps/mm</td>
<td>24ps/mm</td>
</tr>
<tr>
<td>level 2</td>
<td>62ps/mm</td>
<td>35ps/mm</td>
<td>26ps/mm</td>
</tr>
<tr>
<td>level 3</td>
<td>74ps/mm</td>
<td>42ps/mm</td>
<td>31ps/mm</td>
</tr>
</tbody>
</table>

Table VI. Interconnect Delays
Chapter III.

Modeling of Differential Current Tree Logic

Initial problems with the modeling of differential current tree logic led to the modelling technique presented here. The current tree logic could not be properly modelled with standard logic primitives as provided by most simulators. A special current switch primitive and other supporting primitives had to be added to the simulator FASTSIM, a Tektronix event driven simulator.

Current tree logic has several properties that needed to be modelled. The signal path from an input through a current tree to the output can depend upon input signals at higher levels in the tree. Thus the propagation delays from a certain level input to the output can depend upon other input signals. This requires a structural tree model that simulates the actual current tree based on current switch primitives. The output of a current tree can be independent of a signal at a lower level. For example, if the lowest input signal of an AND current tree is undefined (X) the output should still be low if any of the other input signals is low. This is very important because the simulator sets all nodes initially to the undefined state. Further, the treatment of glitches is important for latches. Clock signals generated by a gate with an unsymmetrical tree (AND) have short glitches at each differential signal transition. The two signals of a differential pair are both low or both high during the glitch. Latches must be able to capture valid data if the
necessary setup and hold times have been observed despite the glitches on the clock signal.

The design of high speed digital circuits relies heavily on accurate circuit simulation to detect problems and predict performance before fabrication. For simulation at the circuit level SPICE provides excellent results, however, its simulation speed is prohibitively slow for large digital circuits. Digital simulators use simple digital models and event driven timing control [EGU69] which allows fast simulation of very large circuits. However, most simulators are geared towards CMOS because of its dominance in the market place. As described in [PK83], single ended Bipolar transistor subcircuits can be mapped into equivalent logic gates that can be simulated on a conventional digital simulator. Logic states represent in the transformed circuit either voltages or currents. Another modelling technique transforms the transistor level circuits into labeled weighed graphs for simulation [INH87] requiring a highly specialized simulation tool.

The model presented here uses a current switch, two or more transistors connected at a common emitter node, as a simulation primitive, and allows the simulation of either differential or single ended circuits. It deals explicitly with differential inputs and inputs that are connected to a threshold voltage. Only the mapping of transistors with a common emitter node to a current switch is required for deriving a simulation model from a transistor level (SPICE) circuit description.
The structure of the models is the physical structure. Further, the model can easily be added to digital simulators that support user primitives.

A. Digital Current Switch Model

Extended Truth Table
X=undefined, T=threshold, ?=(0,1, X, T)

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>COM IN INB</td>
<td>Q Qb</td>
</tr>
<tr>
<td>1 ? ?</td>
<td>1 1</td>
</tr>
<tr>
<td>X ? ?</td>
<td>X X</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0 1</td>
</tr>
<tr>
<td>0 0 1</td>
<td>1 0</td>
</tr>
<tr>
<td>0 1 1</td>
<td>0 0</td>
</tr>
<tr>
<td>0 0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>0 X X</td>
<td>0 0</td>
</tr>
<tr>
<td>0 1 T</td>
<td>0 1</td>
</tr>
<tr>
<td>0 0 T</td>
<td>1 0</td>
</tr>
<tr>
<td>0 T T</td>
<td>0 0</td>
</tr>
<tr>
<td>0 X T</td>
<td>0 0</td>
</tr>
<tr>
<td>T ? ?</td>
<td>0 0</td>
</tr>
</tbody>
</table>

COM, Q, Qb represent current levels (low=on)
Symmetry: [Q, Qb]=f(IN, INb, COM)*[Qb, Q]=f(INb, IN, COM)
Signal Strength : low > high
Delay Model
Delay(IN -> q) = TDi + Ri = C(q)
Delay(INb -> qb) = TDi + Ri = C(qb)
Delay(com -> qx) = TDc + Ri = C(qx)

Figure 28. Digital Current Switch Model

The delay of a signal propagating through a current tree depends upon the signal path and can, therefore, depend upon input signals higher up in the tree. This makes the accurate modelling of current tree delays difficult. Most digital
simulators allow only modelling of static delays. All delays are calculated before the simulation starts and are constant during the simulation. Current tree delays can, therefore, not be simulated accurately with a behavioral model based upon a simple truth table specification. Only worst case delays through the tree can be captured. A structural model based upon the basic building block, the current switch, can capture path dependencies easily because it actually simulates the signal propagation through the tree. The current switch itself can be described with a simple behavioral model that is easy to implement on most digital simulators. Asymmetrical current trees have non simultaneous output signal transitions and transient glitches. The output signals of a tree can be equal for short transients even though no output change should occur according to the truth table specification. If such glitches occur on clock lines latched data can be disturbed. Therefore, the transient glitches inherent in current tree logic should be modelled. The current switch model must, therefore, deal with differential and non differential input conditions as shown in Figure 28.

The simulation of differential logic on the current switch level increases the number of nodes and elements in the netlist and will slow down simulations. A full tree has seven switches and seven internal nodes. However, the slower simulation time must be traded off against increased accuracy and the ability to capture transients which might affect circuit performance.
Simulation efficiency could be improved by representing each differential signal pair with a single digital node. The two differential current tree output signals (q, qb) can be converted into a single ended signal with a differential to single ended converter. This converter marks non differential outputs of the current tree with an unknown logic state (X) signal. The current switch can be reduced for single ended simulations to a four terminal device. The four terminal current switch has the same truth table specification as the five terminal current switch with one base input fixed at threshold. While the single ended modelling of differential signals requires only half as many nodes inverter primitives are required for differential signal inversion. If each signal needs to be inverted the number of nodes will actually be larger due to the additional convertors in the circuit. This is, however, a highly unlikely case. The single ended modelling of differential signals makes probing and saving of simulation results more efficient and allows the use of standard fault simulation techniques for finding test vectors.

Stuck at fault simulation of differential circuits without the conversion of differential nets into single ended nets will fail. If only one signal of a differential pair is simulated as stuck high or low the fault can only be detected at the output of a tree if the other signal of the differential pair is at the same logic level. This represents, however, a non differential input signal and will show up at the outputs of all current switches connect to the net as a non differential signal! All gates connected to the faulty net will, therefore, generate invalid outputs (X) as soon as
they sample the non differential net. The non differential signal will propagate quickly throughout the simulation and finally show up at the primary outputs as a non differential signals or unknown logic states (X). However, in the actual circuit one of the differential outputs is most likely slightly lower or higher than the other one, even for a short between differential signals, because of asymmetries in the layouts and small device differences. Since the standard cells have voltage gain even small differences in differential signal will be amplified by the gates connected to the faulty net. By the time the signal has propagated to the primary outputs the signal can be restored to full logic levels and the circuit can not be diagnosed as faulty just by checking for non differential output conditions as indicated by the fault simulator.

Figure 28 shows the digital current switch model and its extended truth table. Negative logic is used to represent a current flowing in or out of the common emitter node or the q,qb output nodes to avoid signal inversion at the top of the tree. The current switch model allows, thereby, to use the physical structure of the tree for gate modelling. Both outputs are active if the two inputs are equal and current is flowing into the common emitter node. This is important for modelling latches like the simple data latch shown in Figure 29. The latch would lose data just copied if the current switch connected to the clock signal would output no current for non differential inputs. However, if it outputs current on both sides no data is lost as long as the input current switch and the feedback current switch
Figure 29. Simple Data Latch

outputs agree. This will be the case as long as the data input is stable. The model will therefore correctly indicate a longer hold time and not loss of data whenever the clock signal has non simultaneous signal transitions at the end of a write pulse. Sending current on both outputs for a non differential input condition reduces modelling pessimism in general since the cell output might not depend upon which way the current flows for a given set of input signal states.

The current switch model uses only a simple inertial delay model and can, therefore, easily be implemented on a digital simulator. It includes capacitors to model input and output loading. This model assumes differential input signals since the base capacitors are physically between base and emitter and can only be
modeled as shown for differential input signals. However, most digital simulators support only capacitors connected to ground. The model parameters depend on the operating conditions of the current switch like the switching current $I_s$, the voltage swing at the output, and $V_{cb0}$ of the transistors. The capacitor value $C_b$ is about 20% larger in an active current switch. This represents a dynamic load change which is hard to simulate. However, it will be shown that a simple current switch model with operating point independent parameters can give excellent results since the dependencies are intrinsically small. In order to see the small differences the digital simulator would have to be run with a time step $\Delta t$ below 5ps. The biggest simulation error is introduced by always using the $C_b$ value for $I_s$=on. The following table shows the parameters for the current switch with $I_s$=400$\mu$A, $V_i$=250mV, $V_{cb0}$=0.

<table>
<thead>
<tr>
<th>Parameters of Current Switch Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>$C_c$</td>
</tr>
<tr>
<td>$C_b$</td>
</tr>
<tr>
<td>$C_t$</td>
</tr>
<tr>
<td>$T_a$</td>
</tr>
</tbody>
</table>

Table VII. Parameters of Current Switch Model

The current switch model can be generalized for circuits with more than two transistors connected to a common emitter node. However, simulation times can
be noticeably improved if a dedicated and, therefore, more efficient two transistor current switch primitive is used for differential trees. Both single ended and differential current tree logic can be modelled with the same primitives if the simulator provides a fourth logic state to indicate threshold voltages.

B. Modelling of Current Trees

![Spice and Digital Model of XOR3T](image-url)

**Figure 30. SPICE and Digital XOR3T Model**

Figure 31 and Figure 30 show a comparison of SPICE simulation results with a digital simulation of a three input AND/OR tree and an XOR tree. The match for the symmetrical XOR (full adder) gate is excellent. The asymmetry of the AND tree results in non simultaneous output transitions that show up as rise time degradation in the SPICE output as shown in Figure 31. The match for the
Figure 31. SPICE and Digital AND3T Model

Asymmetrical AND gate is clearly not as good as for the XOR gate since the $V_{CB0}$ of the current switches in the AND gate is different for all three switches and the difficulty of matching level crossings of signals with rise time variations.

Figure 32 shows a characteristic glitch of the AND/OR current tree if the lowest level signal goes high with level two input high and level one input low. One might not expect an output transient for an AND gate with one input kept at a static low. However, the current has to propagate through two current switches after the level three input transition before it pulls the q output low again. The SPICE output shows only a signal level degradation which can, however, lead to erroneous switching in a noisy environment. The digital model marks the glitch with non differential outputs allowing the detection of circuit that are sensitive to these
transients. All current trees with unequal path delays from a particular current switch to the output show similar glitches. The three input AND gate represents the worst case and should be used with caution on clock lines.

C. Modelling Accuracy

For verification of the accuracy of the standard cell models the 8bit ALU slice shown in figure Figure 18 was modelled with SPICE and FASTSIM. The available current switch model was incorporated with C language user primitives. Signal strength values were used to mark offset levels. This made it possible to detect level violations in the current switch primitive. The output signal strength of a current switch low signal is set equal to the base input strength. Each current
switch instance issues a level violation warning if the current switch below has base input signals that are at the same or a higher level. The output strength of a current switch high was set to the lowest of the four available strength levels to support collector dotting by implicit wired OR connections. The table below shows that excellent agreement is possible even with a simple current switch model. However, the digital simulator must be run with a sufficiently small time step 5-10ps to avoid large rounding errors.

<table>
<thead>
<tr>
<th>PATH</th>
<th>SPICE</th>
<th>FASTSIM Δt= 1ps</th>
<th>FASTSIM Δt= 5ps</th>
<th>FASTSIM Δt=25ps</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Op-&gt;Cout</td>
<td>1042ps</td>
<td>1024ps</td>
<td>1015ps</td>
<td>1075ps</td>
</tr>
<tr>
<td>Cin-&gt;Cout</td>
<td>425ps</td>
<td>419ps</td>
<td>425ps</td>
<td>450ps</td>
</tr>
<tr>
<td>Cin-&gt;Sum</td>
<td>444ps</td>
<td>446ps</td>
<td>450ps</td>
<td>475ps</td>
</tr>
</tbody>
</table>

**Table VIII.** SPICE and FASTSIM Results without Interconnect Delays

The next table shows a comparison of the simulation results including wiring delays. The results show still good agreement if the time step of the digital simulator is small. The delay sensitivities towards interconnect capacitance of the emitter followers and I/O drivers & receivers in the digital models was matched to SPICE data by a six point linear regression analysis in the range 0-500fF.

The worst case delay of the 32bit ALU is only 13% longer due to interconnect delays. However, this required manual placement and medium power
<table>
<thead>
<tr>
<th>PATH</th>
<th>SPICE</th>
<th>FASTSIM $\Delta t = 1\text{ps}$</th>
<th>FASTSIM $\Delta t = 5\text{ps}$</th>
<th>FASTSIM $\Delta t = 25\text{ps}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Op-&gt;Cout</td>
<td>1196ps</td>
<td>1185ps</td>
<td>1185ps</td>
<td>1300ps</td>
</tr>
<tr>
<td>Cin-&gt;Cout</td>
<td>451ps</td>
<td>448ps</td>
<td>450ps</td>
<td>475ps</td>
</tr>
<tr>
<td>Cin-&gt;Sum</td>
<td>550ps</td>
<td>567ps</td>
<td>570ps</td>
<td>550ps</td>
</tr>
</tbody>
</table>

**Table IX.** SPICE and FASTSIM Results with Interconnect Delays

gates (10mW) for the carry propagate chain.
Figure 33. FRISC Wafer Scale System

Figure 33 shows the high speed components of a FRISC system which would have to be implemented densely on a multichip package. This high speed node includes the FRISC processor, high speed instruction and data cache...
memories, an optional math coprocessor, and control logic. The processor has a Harvard architecture with separate instruction and data buses to allow parallel access to instructions and data. This is important since RISC processors can fetch an instruction in every cycle. The data and instruction cache memories must have a sufficiently large storage capacity to isolate the high speed node effectively from main memory. The main memory is typically much slower than the cache memories because it must provide a much larger storage capacity which can only be implemented with slow but high density integrated memory circuits. If the instruction or data word accessed by the processor is stored in the high speed cache memories, a so called cache hit will occur and no main memory access is necessary. The processor can, therefore, continue processing at full speed after a cache hit. Only after a cache miss the processor is stalled until the data or instruction has been fetched from main memory. Hit ratios of about 99% can be obtained with 32kByte cache memories [AJS87].

The data bus is split into a data input and a data output bus to obtain a simpler and faster interface to the data cache. Branch target instruction addresses computed by branch or jump instructions and data addresses computed by load or store instructions are transferred over the shared address bus to the cache memories. Only branch target addresses need to be sent to the instruction cache. Consecutive instruction addresses are generated on the instruction cache by a remote program counter (RPC). The sharing of the address bus for data and
branch target address transfers never causes bus contention since LOAD/STORE and BRANCH/JUMP instructions use the address bus in the same pipeline stage. Hence, only a single address bus is needed to avoid the slower time multiplexing of address and data on the instruction and data bus. The valid instruction address (v_iad) and valid data address (v_dad) signals indicate whether a data or instruction address is transferred over the shared address bus. Data is byte addressed and instructions are word addressed. All buses are 32bits wide.

The directly mapped cache memories [HSS87] indicate a cache hit with the hit_i, hit_d signals. On a cache miss the processor is stalled and will assert the stall signal until the cache line has been fetched from main memory (refresh operation). If the processor is writing into the data cache the cache line must first be read to check for a hit or miss before the data in the cache line can be overwritten. If a miss is detected and the cache line dirty flag indicates that the line has been modified since it was copied from the main memory the whole line must be copied back to main memory (copyback operation) before the cache line can be refreshed. The processor will resume instruction execution as soon as the cache memory indicates that the copyback/refresh is completed by setting the hit_i/hit_d signal high. The processor is not interrupted on cache misses to avoid a time consuming pipeline flush/restart. Hence, cache misses are transparent to the processor. External cache support logic must be provided to refresh the cache
after a miss. Only page faults (trap_d, trap_i) trap signals like reset, error, or user interrupt will interrupt the processor.

The stall signal is used to synchronize the cache memories with the processor. If the processor stalls both caches must repeat the last cache access or perform a cache line refresh from main memory if the stall was caused by a cache miss. The cache line address tags are virtual to avoid an address translation for every access. The physical address can be looked up in a translation lookaside buffer (TLB) [HSS87] in parallel to the cache memory access. This way the physical memory address is immediately available if a cache miss occurs. The physical address is then transferred over the system bus to main memory to fetch the appropriate cache line with a Direct Memory Access (DMA). The system bus should be at least 64 bits wide and the main memory should be interleaved to reduce the time for cache line transfers of 32 data words or instructions.

B. Processor Architecture

FRISC-F is designed to provide high performance with a small amount of hardware and a partitioned implementation. Features like large register files divided into multiple windows or banks as found in CMOS RISC’s [RWS84] can not be implemented with Advanced Bipolar circuit technology. Other, features like barrel shifters can not be supported because a partitioning would require too many I/O pins or the lack of pass transistors would increase the hardware cost considerably.
Figure 34. FRISC Architecture
FRISC must, therefore, provide a high speed advantage over CMOS processors to compensate for the lack of VLSI hardware features. This can only be achieved through the use a very high speed circuit technology combined with a highly pipelined design that makes very efficient use of the available hardware. FRISC uses a seven stage instruction pipeline. The seven pipeline stages consist of two instruction fetch states (I1,I2), a decode stage (DE), an execution stage (EX), two data input/output stages (D1,D2), and a data write stage (DW).

Figure 34 shows the architecture of FRISC-F. The processor has a uniform 32x32bit register file. Two source operands are read from the register file and one result can be written into the register file during each 4ns cycle. A single 32bit ALU performs all data modifications and address calculations. The ALU has four functions ADD, AND, XOR, and OR. The operands read from the register file can be inverted or replaced before they reach the ALU input latches (OPA,OPB). ALU operand A can be either a register, a register shifted {left 2,left 1,right 1,right 2}, a special purpose register (PC,PSW), or the constant zero. ALU operand B can be replaced with a literal constant from the instruction and it can be inverted. Subtraction is implemented by an inversion of the input carry whenever operand B is inverted. A bitwise inversion of an operand together with an inversion of the input carry is equivalent to adding the two's complement of the operand (operation = A+(-B)). The ALU result can be output to the address bus through the address register (ADDR) as a branch target or data address, or it can be copied into the
processor status word (PSW), the program counter (PC_I1), or the pipeline register RES_D1.

The processor status word contains the Negative, Carry, Overflow, and Zero flags used by conditional branch instructions. It further contains the Interrupt Enable flag and the Protected Mode flag that shows the operating mode of the processor. The processor goes into protected mode after an interrupt. The shifter input/output bits C1,C0 needed for extended shift are also stored in the processor status word. The program counter PC_I1 is incremented after each instruction cycle. The PC_I1 is overwritten with the ALU result if a branch instruction was taken. The instruction decoder flushes the instruction pipeline and inserts a ghost 'JUMP trap_vector' instruction after an exception.

Only LOAD and STORE instructions can access memory in FRISC a typical RISC characteristic [DAP82]. Data loaded from the data cache memory is copied from the 32bit data input bus (D_BUSI) into a data holding register (D_IN). This holding register is tagged. Whenever, an instruction source register address is equal to the register's tag and the tag is valid the register file is bypassed. This feature is also called feed forward, or short stopping. The holding register is only written into the register file when the next LOAD instruction is executed because LOAD instructions have an empty write register file pipeline stage slot just before the data input register is overwritten. If a following instruction overwrites the register in the register file that corresponds to the tagged D_IN register the tag is
invalidated to prevent register file bypassing and writing of the D_IN register into the register file.

Data is transferred from the data output registers (OUT_EX, OUT_D1) through the data output bus (D_BUSO) to the data cache. The OUT_EX register just delays the data by one cycle. This is necessary because the register specified in the source B field of a STORE instruction is used as the data output register. The effective address of a STORE instructions is, therefore, restricted to operand A ± 3bit constant.

The result of ALU instructions, register to register type instructions, is only written into the register file after a delay of two cycles. This is necessary to obtain a precise interrupt mechanism [JES88] that allows to restart the instruction stream after an exception. To support a precise interrupt with the seven stage FRISC pipeline it must be possible to flush (to abort or reverse) the instructions that were in the first six pipeline stages at the time of the interrupt.

The result of an ALU instruction is immediately available after the ALU operation. The overwrite of a register in the register file by an ALU instruction must be delayed by at least two cycles otherwise an ALU instruction can overwrite a source register for a preceding LOAD or STORE instruction before it has completed. This would make a restart of LOAD or STORE instructions impossible. The pipeline registers RES_D1 and RES_D2 delay the ALU result write by two cycles. These two registers and the RES_EX register are tagged like the data input
register. If an instruction source operand address is equal to a valid register tag
the data is fed forward from the tagged register instead of using the register file
data that has not yet been updated. Feed forward conflicts are solved by giving
priority to the most recently generated data (RES_EX->RES_D1->RES_D2->D_IN->Register File). After an interrupt all ALU instructions that have not yet completed
can be flushed by invalidating the tags of the pipeline registers. Without the feed
forward mechanism an instruction could only use the result of an ALU instruction
after the result has been written into the register file.

A write to the process status word changes the process state like a register
file write. Therefore, a copy of the status word is kept in PSW_D2 to restore the
register to its previous value after an exception.

Seven program counter registers are necessary for the FRISC pipeline.
Each program counter is associated with one pipeline stage and points during
normal operation to the instruction that is currently in that stage. If no instruction
a bubble, is in one of the last three execution stages the corresponding program
counter is set to zero. Pipeline bubbles can be created by branch instructions or
pipeline flushes.

Figure 35 shows the user visible or process state. The state consists of the
32x32bit register file, the status word, and the PC registers. Only the PC_DE and
the PC_dw can be directly accessed. During normal operation the PC registers
are shifted whenever the instruction pipeline advances. After an exception the
processor goes into protected mode. In protected mode the last four PC registers are frozen and can only be read out sequentially with the GETLPC instruction.

Further, asynchronous interrupts are disabled in protected mode to prevent an another interrupt before the current state has been saved. The last four instruction
only be inserted in certain locations without increasing the hardware cost considerably. Further, the insertion of pipeline registers adds a small delay to each stage. Pipeline registers are usually inserted at the input or output of functional blocks. Thus the block with the longest delay determines in practice the cycle time and limits the throughput. Hence, the minimal cycle time of a pipeline is determined by the pipeline stage with the longest delay path.

The sustained throughput of a pipelined processor is lower than the peak throughput due to pipeline interlocks and timing hazards. For example, if an
instruction uses a resource in stage n and another instruction uses the same resource in stage n-m a resource conflict will occur if the instructions enter the pipeline m cycles apart. Similarly, if the result of particular instruction is generated in stage n and an other instruction needs the result in stage n-m a timing hazard will occur if the instructions enter the pipeline less than m cycles apart. Such timing hazards or conflicts must be prevented in RISC processors by the compiler or a code reorganizer through the insertion of NOOP instructions. These NOOP instructions increase the code size and reduce throughput. Therefore, the pipeline should be kept as short as possible for a given cycle time and the pipeline should exhibit as few pipeline hazards as possible.

A seven stage instruction pipeline is necessary for FRISC to achieve a 250MIPS throughput with the given technology and implementation strategy. Figure 36 shows the seven stage instruction pipeline of FRISC. A four phase clock is used so that each stage can be subdivided into four segments. The pipeline starts with two instruction fetch stages (I1,I2) to allow a pipelined cache memory access. The address transfer, cache read, and instruction transfer are pipelined. This gives the instruction and data cache memories a full 4ns cycle for read/write operations. Two pipeline stages for memory access are used in most high speed GaAs processors [VM90] since cache memories with an effective size can not be implemented if address transfer, cache read, and instruction transfer must all be performed in a single cycle. The two stages for instruction fetch optimize the time
available for cache read/write operation considering that one instruction must be provided every cycle.

After the instruction fetch stages follows the decode stage (DE). In this stage the instruction is decoded and, in parallel, the two source operands are read from the register file. The RISC instruction format supports this by fixed instruction field locations for the two source register addresses. Thus the address fields of the potential source operands are known without any decoding. In the last phase of the decode stage an operand feed forward and shift is performed if necessary. At the end of the decode stage the ALU and its operands are setup for the execution of the instruction. Hence, the first three pipeline stages are called the setup stages and the last four pipeline stages are called the execution stages.

During the execution stage the ALU has 3ns (p1-p3) to generate a result. No full cycle is available since the destination register of an ALU instruction could be a source register for the instruction in the DE stage. The ALU result must then be feed forward and it might have to pass through the shifter in the worst case.

The next two stages (D1,D2) are for pipelined data I/O. Two data I/O stages are provided for the same reason as for the instruction fetch. Results of ALU instructions just pass from the RES_EX register to the RES_D1 and then from the RES_D1 to the RES_D2 pipeline registers during data I/O stages. At the end of the D2 stage the register file write is initiated. Hence, ALU instructions have completed once they passed the D2 stage!
During the last pipeline stage, the data write stage (DW), the register file write occurs for ALU instructions. If a LOAD instruction is in the DW stage the data input bus (D_BUSI) is latched into the data input register (D_IN) at the end of phase two.

![Diagram of Instruction Cache and Data Cache](image)

**Figure 37. Cache and Instruction Pipelines**

STORE instructions first read the data cache line during D1.p2-D2.p2 just like LOAD instructions to check the cache line tag before they overwrite the data. The instruction pipeline is stalled until LOAD/STORE instructions can complete if a cache miss occurs.

Instruction cache misses also cause a pipeline stall until the cache line has been refreshed from main memory. The instruction pipeline can, further, be stalled by asserting the processor halt signal. The instruction schedule has been designed
such that the processor never needs to stall the pipeline because of internal pipeline hazards.

<table>
<thead>
<tr>
<th>Stage</th>
<th>ALU Instructions</th>
<th>(JUMP, BRANCH)</th>
</tr>
</thead>
</table>
| I1    | P1-2: transfer address if BT  
P3-4: read ICACHE | P1-2: transfer address if BT  
P3-4: read ICACHE |
|       | Fetch 1          | Fetch 1        |
| I2    | P1-2: read ICACHE  
P3-4: transfer instr. to ID  
P4: latch instr. in IR | P1-2: read ICACHE  
P3-4: transfer instr. to ID  
P4: latch instr. in IR |
|       | Fetch 2          | Fetch 2        |
| DE    | P1-2: decode instr.  
P1: transfer SRCB to DF  
P2: transfer SRCA to DF  
P2: read SRCB from RF  
P3: read SRCA from RF  
P3: transfer IMM to DF  
P1: transfer SRCB to DF  
P2: transfer SRCA to DF  
P2: read SRCB from RF  
P3: read Current PC  
P3: transfer IMM to DF |
|       | Decode           | Decode         |
| EX    | P1-3: ALU operation  
P4: copy result to RES EX  
P2: generate branch sig.  
P3: transfer branch to ID  
P4: latch BT in RES EX  
P4: transfer BT to TCACHE|
|       | Execute          | Execute        |
| D1    | P1: update Condition Code\(^1\)  
P2: RES EX to RES D1 | P1-2: transfer BT to ICACHE\(^2\)  
P2-4: read BT instruction\(^2\) |
| D. I/O | P1: update Condition Code\(^1\)  
P2: RES EX to RES D1 | D. I/O 2 |
| D2    | P2: RES D1 to RES D2  
P3: transfer DEST to DF  
P4: address DEST reg. | P1-2: read BT instruction\(^2\)  
P3-4: transfer BT i. to ID\(^2\) |
| D. I/O | P2: RES D1 to RES D2  
P3: transfer DEST to DF  
P4: address DEST reg. | D. I/O 2 |
| DW    | P1: write RES D2 into DEST | P1: latch BT in IR\(^3\) |
| D. Write | P1: write RES D2 into DEST | D. Write |

\(^1\) only if SCC bit is set in instruction  
\(^2\) only if branch condition is true (bza_1)  
DF = Data Path  
ID = Instruction Decoder  
ICACHE = Instruction Cache  
BT = Branch Target  
IMM = Immediate Constant Instruction Field  
SRCA = Source A Instruction Field  
SRCB = Source B Instruction Field  
DEST = Destination Instruction Field  
CC = Condition Code  
IR = Instruction Register  
RES_EX = ALU result register for pipeline stage EX  
RES_D1 = ALU result register for pipeline stage D1  
RES_D2 = ALU result register for pipeline stage D2  
PC.DE = Program Counter for DE stage  
Z,N,C,V = Zer, Negative, Carry, Overflow flag

Table X. Scheduling of ALU and Branch Instructions

The processor and the cache memories must stay synchronized with each other. If the processor stalls, the cache memories must stall and vice versa.
Figure 37 shows the instruction pipelines and the cache pipelines. The signals for synchronization are hit instruction cache (hit_i), hit data cache (hit_d), and the processor output signal stall. Note that the instruction is latched before the cache hit signal. Thus the latched instruction could be invalid! This provides additional time for the generation of the hit signal. The cache hit signal can only be generated after the cache line has been read by comparing the cache line tag with the instruction address.

A detailed execution schedule for the most important instruction classes is shown in the two tables above. In the design of the instruction execution schedule the partitioning of the processor as well as the relative delays of the functional blocks had to be considered. Further, pipeline interlocks between different instructions had to be avoided. This simplifies code generation and increases throughput. With the above scheduling FRISC makes very efficient use of the hardware. At peak throughput all major hardware components are used all the time! Figure 38 shows when the different pipeline stages use the hardware resources or buses. Since a different instruction is in each of the seven pipeline stages the instruction decoder must generate the appropriate control signals for the instruction in each pipeline stage and is, therefore, more complex than a decoder for a sequential processor. However, most of the control signals are generated in the decode stage to set up the ALU operation as shown in Figure 38.
<table>
<thead>
<tr>
<th>Stage</th>
<th>LOAD</th>
<th>STORE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>P1-2: transfer address if BT</td>
<td>P1-2: transfer address if BT</td>
</tr>
<tr>
<td>Instr.</td>
<td>P3-4: read ICACHE</td>
<td>P3-4: read XCACHE</td>
</tr>
<tr>
<td>Fetch 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I2</td>
<td>P1-2: read ICACHE</td>
<td>P1-2: read IC cache</td>
</tr>
<tr>
<td>Instr.</td>
<td>P3-4: transfer instr. to ID</td>
<td>P3-4: transfer instr. to ID</td>
</tr>
<tr>
<td>Fetch 2</td>
<td>P4: latch instr. in IR</td>
<td>P4: latch instr. in IR</td>
</tr>
<tr>
<td>Decode</td>
<td>P1: transfer SRCB to DP</td>
<td>P1: transfer SRCB to DP</td>
</tr>
<tr>
<td></td>
<td>P2: read SRCB from RF</td>
<td>P2: read SRCB from RF</td>
</tr>
<tr>
<td></td>
<td>P2: transfer SRCA to DP</td>
<td>P2: transfer SRCA to DP</td>
</tr>
<tr>
<td></td>
<td>P3: read SRCA from RF</td>
<td>P3: read SRCA from RF</td>
</tr>
<tr>
<td></td>
<td>P4: feed forward &amp; shift</td>
<td>P4: feed forward &amp; shift</td>
</tr>
<tr>
<td>EX</td>
<td>P1-3: ALU:ADD operand A + IMM</td>
<td>P1: copy SRCB to OUT EX</td>
</tr>
<tr>
<td>Execute</td>
<td>P4: copy addr to RES_EX</td>
<td>P1-3: ALU:ADD operand A + IMM</td>
</tr>
<tr>
<td></td>
<td>P4: transfer addr. to DCACHE</td>
<td>P4: copy address to RES_EX</td>
</tr>
<tr>
<td></td>
<td>P4: transfer addr. to DCACHE</td>
<td>P4: transfer addr. to DCACHE</td>
</tr>
<tr>
<td>D1</td>
<td>P1-2: read addr. to DCACHE</td>
<td>P1-2: copy OUT_EX to OUT D1</td>
</tr>
<tr>
<td>D. I/O 1</td>
<td>P2-4: read DCACHE</td>
<td>P1-2: transfer data to DCACHE</td>
</tr>
<tr>
<td></td>
<td>P2-4: transfer data to DP</td>
<td>P2-4: transfer data to DCACHE</td>
</tr>
<tr>
<td></td>
<td>P2-4: write DCACHE if hit_d (^1)</td>
<td>P2-4: write DCACHE if hit_d (^1)</td>
</tr>
<tr>
<td>D2</td>
<td>P1-2: read DCACHE</td>
<td>P1-2: read DCACHE</td>
</tr>
<tr>
<td>D. I/O 2</td>
<td>P2-4: transfer data to DP</td>
<td>P2-4: write DCACHE</td>
</tr>
<tr>
<td></td>
<td>P2-4: write DCACHE if hit_d (^1)</td>
<td></td>
</tr>
<tr>
<td>DW</td>
<td>P2: latch hit_d on ID</td>
<td>P2: latch hit_d on ID</td>
</tr>
<tr>
<td>D. Write</td>
<td>P1-2: transfer Data &amp; align</td>
<td>P1-2: transfer DCACHE</td>
</tr>
<tr>
<td></td>
<td>P3: latch Data in DR on DP</td>
<td></td>
</tr>
</tbody>
</table>

\(^1\) if no hit_d wait till cache line is written back to memory and hit_d is set high

DP = Data Path  
ICACHE = Instruction Cache  
DCACHE = Data Cache  
ID = Instruction Decoder  
RF = Register File  
BT = Branch Target Address  
IMM = Immediate Constant Instruction Field  
SRCA = Source A Instruction Field  
SRCB = Source B Instruction Field  
IR = Instruction Register  
OUT_EX = Data Output Register  
OUT_D1 = Data Out Out Register, drives D_BUS

### Table XI. Scheduling of Load/Store Instructions

### D. Timing Constraints / Critical Paths

The instruction execution scheduling imposes timing constraints on the hardware. A potential critical path exists between any two pipeline registers. Since the instruction set is small and very regular the decoder is not on a critical path.
<table>
<thead>
<tr>
<th>RESOURCE</th>
<th>CYCLE N</th>
<th>CYCLE N+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Decoder</td>
<td>decode instruction <strong>DE</strong></td>
<td>transfer instruction <strong>DE</strong></td>
</tr>
<tr>
<td>Register File Address</td>
<td>transfer <strong>DE</strong></td>
<td>transfer <strong>DE</strong></td>
</tr>
<tr>
<td>Register File</td>
<td>read <strong>DE</strong></td>
<td>write <strong>DE</strong></td>
</tr>
<tr>
<td>ALU &amp; Shifter Feed Forward L.</td>
<td>ALU operation <strong>EX</strong></td>
<td>FF&amp;SH <strong>DE</strong></td>
</tr>
<tr>
<td>Processor Status Word</td>
<td>eval <strong>CC EX</strong></td>
<td>write <strong>PSV DE</strong></td>
</tr>
<tr>
<td>Program Counter</td>
<td>increment PC_Il</td>
<td></td>
</tr>
<tr>
<td>PC Registers</td>
<td>shift PC's</td>
<td>read <strong>PC DE</strong></td>
</tr>
<tr>
<td>Address Bus</td>
<td>transfer branch target</td>
<td></td>
</tr>
<tr>
<td>Instruction Cache</td>
<td>read instruction cache <strong>I1</strong></td>
<td></td>
</tr>
<tr>
<td>Instruction Bus</td>
<td>transfer instruction <strong>I1</strong></td>
<td></td>
</tr>
<tr>
<td>Data Cache</td>
<td>read data cache or <strong>D1</strong></td>
<td>write data cache <strong>D2</strong></td>
</tr>
<tr>
<td>Data Bus</td>
<td>transfer data from cache <strong>D1</strong></td>
<td>transfer data from cache <strong>D2</strong></td>
</tr>
<tr>
<td>Clock Phase</td>
<td>Phases 1-4</td>
<td>Phases 1-2</td>
</tr>
</tbody>
</table>

Figure 38. Usage of Resources
Figure 39. Dataflow during I1, I2 Pipeline Stages
Thus all important timing constraints are associated with the data flow or the communication between the datapath and the instruction decoder chip. Most of the control signals are generated on the instruction decoder chip and are sent to all four datapath slices.

Figure 39 shows the dataflow during the first two instruction fetch stages (I1, I2). The instruction address can either be generated by a branch instruction or by the remote program counter on the instruction cache. The instruction is latched in the instruction register on the instruction decoder chip at the beginning of the decode stage. This provides 3ns for the instruction address transfer, 4ns for the pipelined instruction cache read, and 2ns for the transfer of the instruction to the instruction decoder chip.

Figure 40 shows the data flow during the decode stage. During the decode stage the ALU operands OPA and OPB are setup. Thus most data paths ultimately lead to the ALU input latches. The dataflow is rather complex because several different operand sources can be specified in the instruction. Further, the source register(s) specified in the instruction might actually be in one of the tagged registers (RES_EX, RES_D1, RES_D2, D_IN) and require a feed forward from a tagged register.

The two source operands are sequentially read from the single port register file. The register file access is pipelined such that 1ns is available for each register file read. Operand B is copied into the data output register (OUT_EX) for STORE
Figure 40. Dataflow during DE Pipeline Stage
instructions. All potential sources for operand A like the PSW and the PC registers are accessed during the decode stage. Since these registers are clocked during phase two of a cycle the timing is not critical on these paths. However, the feed forward path is critical since the ALU result of the previous instruction is only available at the beginning of phase four. In the worst case the ALU result must be feed forward to operand A and pass through the shifter in 1ns. All of the ALU control signals are generated on the instruction decoder and are transferred to the four datapath slices.

Figure 41 shows the dataflow during the EX stage in which the instruction is executed on the ALU. In the worst case a full 32bit add must be performed with the carry propagating from the least significant to the most significant bit passing through all four datapath slices. Thus the ALU delay path is definitely a critical path for FRISC. The ALU result can be transferred over the address bus to the data cache for load/store instructions or the instruction cache for branch instructions. An address transfer on the shared address bus can take 3ns. The ALU result is copied into the program counter (PC_I1) if a branch is taken or it is copied into the result pipeline register RES_D1 for ALU type instructions. The result is copied into the PSW for SETPSW instructions.

Figure 42 shows the dataflow during the data input/output pipeline stages. The data or instruction address calculated in the previous stage is transferred to the data or instruction cache. The ALU result is passed from the RES_EX to
Figure 41. Dataflow during EX Pipeline Stage
Figure 42. Dataflow during D1,D2 Pipeline Stages
RES_D1 and then from RES_D1 to the RES_D2. The data output register is sent over the data output bus to the data cache at the beginning of the d1 stage. Since the data cache write is only starting at phase two of the D2 stage 6ns are available for data transfer and optional data alignment hardware. At the end of the D2 stage the register file write is initiated for ALU instructions.

Figure 43 shows the dataflow during the data write pipeline stage. The ALU result that is by now in the RES_D2 is written into the register file during the first phase of the cycle. The destination address was already setup during phase four of the previous stage to guarantee that the address is stable when the write signal is applied. Hence, 2ns are available for a register file write. If a load instruction is in the DW stage the data from the data cache is latched into the data input holding register at the end of phase two. A full cycle of 4ns is available for this data transfer and optional data alignment hardware.

The most important timing constraints for a cycle time of 4ns are summarized in Table XII. The most critical timing constraints for an Advanced Bipolar implementation of FRISC are the ALU path, the feed forward and shift path, and the control signal path between the instruction decoder and the four datapath slices. All of these critical paths involve chip to chip I/O. Dense packaging is essential to meet the timing constraints.
Figure 43. Dataflow during DW Pipeline Stage
### Timing Constraints

<table>
<thead>
<tr>
<th>Resource</th>
<th>Operation</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register File</td>
<td>Read</td>
<td>1ns</td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td>2ns</td>
</tr>
<tr>
<td>ALU</td>
<td>32bit ADD</td>
<td>3ns</td>
</tr>
<tr>
<td>Control Signal I/O</td>
<td>Transfer</td>
<td>1ns</td>
</tr>
<tr>
<td>Address Bus</td>
<td>Transfer</td>
<td>3ns</td>
</tr>
<tr>
<td>Instruction Bus</td>
<td>Transfer</td>
<td>2ns</td>
</tr>
<tr>
<td>Data Bus</td>
<td>Transfer</td>
<td>4ns</td>
</tr>
<tr>
<td>Instruction Cache &amp; Data</td>
<td>Read</td>
<td>4ns</td>
</tr>
<tr>
<td>Cache</td>
<td>Write</td>
<td>8ns</td>
</tr>
</tbody>
</table>

**Table XII.** Timing Constraints

**E. Load and Branch Latencies**

While pipelining increases the peak throughput it also introduces load and branch latencies. These latencies will reduce the average throughput of the processor. The performance impact depends on the instruction and data dependencies in the code. The cause for branch and load latencies is shown in Figure 43. The destination register of a LOAD instruction is only available in the last pipeline stage (DW) pipeline stage. The three instructions following the LOAD can not depend on the destination register (R1) since the ALU operands are set up in the DE stage. Hence, feeding data from the DW to the DE pipeline stage takes three cycles. If a code reorganizer or instruction scheduler can not fill the
three slots in the load latency region with instructions that do not depend on the destination register, NOOP's must be scheduled. Thus load instructions have a latency of three cycles and can take one to four cycles to execute.

![Figure 44. Branch and Load Latencies](image)

The branch target address is calculated during the EX pipeline stage. If the branch condition is true the branch target address becomes the address for the next instruction entering the I1 stage. This branch target instruction will only be executed after a delay of four cycles. Hence, a branch instruction has a latency of three cycles and can take one to four cycles to execute. The high branch penalty in pipelined computers has lead to the development of several branch optimization schemes involving both hardware and compiler techniques [DJL88]. The three instructions immediately following the BRANCH are already partially executed by the time the branch target instruction is entering the instruction pipeline. These partially executed instructions are flushed if the branch is taken in the conventional
branch with flush. This causes a considerable performance loss because of the high frequency (3/4) of branch instructions that are taken. Most of today’s RISC processors use, therefore, the delayed branch scheme [AM88, MN88, LK89]. In the delayed branch scheme the partially executed instructions in the branch latency region are always executed. NOOP’s must be inserted if the branch latency region can not be filled with useful instructions by a code reorganizer/optimizer. The code reorganizer is trying to fill the branch latency region with instructions from the code segment just before the BRANCH instruction or in the code segment starting at the branch target address if the branch is unconditional [VM87]. For the deep FRISC pipeline the code reorganizer would have to find three instructions to fill the branch latency region! The following static branch fill in probabilities have been reported [VM87] for an equivalent six stage GaAs RISC pipeline.

<table>
<thead>
<tr>
<th>Branch Fill in Probabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-fills</td>
</tr>
<tr>
<td>26.5%</td>
</tr>
</tbody>
</table>

Table XIII. Branch Fill in Probabilities

While almost half of the delayed branch slots could be filled with useful instructions the other half had to be filled with NOOP instructions. These NOOP instructions only delay the following instructions wasting one instruction cycle. They
also increase the code size and lower the throughput. Hence, a more flexible branch optimization scheme, called branch with execute, is used for FRISC. It includes the branch with flush and the delayed branch as special cases.

<table>
<thead>
<tr>
<th>CC</th>
<th>condition code</th>
<th>/SCC</th>
<th>set condition code</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>branch condition is true</td>
<td>/EXE</td>
<td>branch with execute option</td>
</tr>
<tr>
<td>T,F</td>
<td>address labels</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I1</th>
<th>/SCC</th>
<th>I1</th>
<th>/SCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2</td>
<td>BRANCH CC T</td>
<td>I2</td>
<td>;always exec.</td>
</tr>
<tr>
<td>I3</td>
<td></td>
<td>I3</td>
<td>;always exec.</td>
</tr>
<tr>
<td></td>
<td>BRANCH CC T</td>
<td>I3</td>
<td>;always exec.</td>
</tr>
<tr>
<td>F:</td>
<td></td>
<td>F:</td>
<td>;always exec.</td>
</tr>
<tr>
<td>I2,</td>
<td>;flush if true</td>
<td>I2,</td>
<td>;flush if true</td>
</tr>
<tr>
<td>I3,</td>
<td>;flush if true</td>
<td>I3,</td>
<td>;flush if true</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T:</td>
<td></td>
<td>T:</td>
<td></td>
</tr>
<tr>
<td>I1,</td>
<td></td>
<td>I1,</td>
<td></td>
</tr>
<tr>
<td>I2,</td>
<td></td>
<td>I2,</td>
<td></td>
</tr>
<tr>
<td>I3,</td>
<td></td>
<td>I3,</td>
<td></td>
</tr>
</tbody>
</table>

Branch with Flush  Delayed Branch  Branch with Execute

1. Branch Execution Schemes

In the branch with execute scheme the number of branch fill in instructions can be specified in a two bit instruction field (EXE). The other instructions in the branch latency region are flushed if the branch is taken. Thus, no NOOP instructions have to be inserted if not enough branch fill in instructions can be found. The code reorganizer enters in the EXE field how many branch fill in instructions it has found. Listing 1 shows the different branch execution schemes.

The branch with execute scheme has the same code density as the branch with flush and the branch penalty is even lower than for the delayed branch. The table below shows the average number of NOOP instructions scheduled per branch instruction and the average branch penalty in cycles. The fill in probabilities
are taken from the table shown above and a branch probability of 75% is assumed.

<table>
<thead>
<tr>
<th></th>
<th>NOOPs</th>
<th>Branch Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch with Flush</td>
<td>0.00</td>
<td>3.25</td>
</tr>
<tr>
<td>Delayed Branch</td>
<td>1.67</td>
<td>2.67</td>
</tr>
<tr>
<td>Branch with Execute</td>
<td>0.00</td>
<td>2.26</td>
</tr>
</tbody>
</table>

**Table XIV.** Branch Performance

Of the three branch execution schemes the delayed branch is most simple to implement since it does not create bubbles in the pipeline. The instruction decoder does not need to know whether a branch was taken or not. The branch with execute scheme requires only a modest amount of hardware support. However, since it creates bubbles it increases the design complexity.

Finding the optimal instruction scheduling for a pipelined processor is a non polynomial (NP) complete problem [DB89]. However, fast algorithms that perform only local optimization can provide good results in a reasonable time [VM87]. In general a trade off between code size and execution speed is required for loop optimization. Advanced compiler optimization techniques as shown in [DAP86] should also be used to increase the execution speed.
F. Sustained Performance

Load and branch penalties reduce the sustained performance of a pipelined processor. Due to the limited cache size that can be implemented with today's high speed memories on a Wafer Scale Hybrid, cache misses will also have a significant performance impact. In the following an estimate of the sustained performance based on statistical data will be derived.

An important figure of merit for a processor pipeline is the expected number of execution cycles per instruction (CPI). The ideal CPI figure for a RISC processor is one. The original SPARC processor from SUN had a CPI figure of 1.3 with an infinite cache and a CPI figure of 1.42 with a 128kByte unified cache, a single cache for data and instructions [MN89]. Some of today's RISC processors are so called superscalar processors [LK89,GFG90] that can execute several instructions in a single cycle and can have CPI figures below one. The sustained performance can be calculated from the number of cycles per instruction and the cycle time by:

$$\text{Average Throughput} = \frac{1}{\text{CPI} \times T_{\text{cycle}}}$$

A dynamic instruction mix as shown in the table below is assumed [MN88]. Note the high frequency of ALU (60%) and branch (20%) instructions. Most of the branches (75%) are taken. This figure includes unconditional branches.

The expected cache miss ratios are taken from reference [AJS87]. The miss ratios shown below are for 32kByte instruction and data caches with 128byte
Dynamic Instruction Mix

<table>
<thead>
<tr>
<th>Instruction Class</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>15</td>
</tr>
<tr>
<td>STORE</td>
<td>5</td>
</tr>
<tr>
<td>BRANCH (taken)</td>
<td>15</td>
</tr>
<tr>
<td>BRANCH (not taken)</td>
<td>5</td>
</tr>
<tr>
<td>ALU (Reg. to Reg.)</td>
<td>60</td>
</tr>
</tbody>
</table>

Table XV. Dynamic Instruction Mix

cache lines. Since the cache lines must isolate the processor from slow main memory a sufficiently large cache size is necessary. The cache memories are directly mapped to get a low implementation cost.

Cache Miss Ratios

<table>
<thead>
<tr>
<th>Event</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Cache Miss</td>
<td>0.007</td>
</tr>
<tr>
<td>Data Cache Miss</td>
<td>0.012</td>
</tr>
<tr>
<td>Data Cache Miss &amp; Cache Line is Dirty</td>
<td>1/2</td>
</tr>
</tbody>
</table>

Table XVI. Cache Miss Ratio

To estimate the impact of cache misses certain assumptions about the main memory and the system bus must be made. In the following a four way interleaved
main memory is assumed. Further, it is assumed that transfers between caches
and main memory are performed with Direct Memory Access (DMA) and that the
main memory is eight times slower than the cache memories.

<table>
<thead>
<tr>
<th>System Memory Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus Transfer Time</td>
</tr>
<tr>
<td>Bus Width</td>
</tr>
<tr>
<td>Main Memory Access Time $^2$</td>
</tr>
<tr>
<td>Cache Write Cycle Time $^3$</td>
</tr>
<tr>
<td>Cache Line Size</td>
</tr>
</tbody>
</table>

$^2$ Four way interleaved, 64bits Wide
$^3$ Pipelined Cache Memory 32bits Wide

Table XVII. System Memory Parameters

Figure 45 shows the activities during a cache line replacement in a system
with a four way interleaved main memory. It takes 45 cycles to replace the cache
line. The virtual to physical memory address translation can be performed in
parallel to the cache access so that the physical address is immediately available
when the cache miss is detected.

Figure 46 shows the activities during a copyback of a cache line to main
memory after a data cache miss. If a cache miss occurs during a cache write
(store) and the cache line is dirty the line must be first copied to main memory
before the line can be replaced and the write can finally complete. A copyback of
Figure 45. Cache Line Replacement Time

A 128-byte cache line takes 36 cycles.

Figure 46. Cache Copyback Time

The table below shows the contributions of branch and load latencies as well as cache misses to the expected number of execution cycles per instruction without any code optimization. Three NOOP's are scheduled after each load instruction and branch instructions flush all three instructions in the branch latency region if the branch is taken.
<table>
<thead>
<tr>
<th>Instruction / Event</th>
<th>Probability</th>
<th>Cost</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>0.15</td>
<td>4</td>
<td>0.6</td>
</tr>
<tr>
<td>STORE</td>
<td>0.05</td>
<td>1</td>
<td>0.05</td>
</tr>
<tr>
<td>BRANCH (taken)</td>
<td>0.15</td>
<td>4</td>
<td>0.6</td>
</tr>
<tr>
<td>BRANCH (not taken)</td>
<td>0.05</td>
<td>1</td>
<td>0.05</td>
</tr>
<tr>
<td>ALU</td>
<td>0.60</td>
<td>1</td>
<td>0.6</td>
</tr>
<tr>
<td>Instruction Cache Miss</td>
<td>0.007x1</td>
<td>45</td>
<td>0.315</td>
</tr>
<tr>
<td>Load with Cache Miss</td>
<td>0.012x0.15</td>
<td>45</td>
<td>0.081</td>
</tr>
<tr>
<td>-&gt; cache line replace</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STORE with Cache Miss</td>
<td>0.012x0.05</td>
<td>45</td>
<td>0.027</td>
</tr>
<tr>
<td>-&gt; cache line replace</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STORE with Cache Miss</td>
<td>0.012x0.05</td>
<td>36</td>
<td>0.011</td>
</tr>
<tr>
<td>line dirty -&gt; copy back</td>
<td>x0.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Average CPI</strong></td>
<td></td>
<td></td>
<td><strong>2.334</strong></td>
</tr>
</tbody>
</table>

* Assuming Branch with Flush

Table XVIII. Number of Cycles per Instruction

The NOOP’s scheduled after load instructions and the flushed instructions after a taken branch contribute 0.9 to the total CPI of 2.333! Instruction cache misses contribute 0.315 cycles. The contributions from data cache misses (0.119) are small in comparison. This shows the importance of a large instruction cache. Instruction caches with only 16kByte, 8kByte and 4kByte would increase the cache
miss penalty from 0.315 to 0.54, 0.72, and 0.44 cycles. A CPI figure of 2.334 would give a sustained throughput of 107MIPS with a 4ns cycle time. A code optimizer/reorganizer for an equivalent pipeline could replace about half of the NOOP’s or flushed instructions with useful instructions [VM87]. Thus the contributions from load and branch penalties could be reduced 0.45 cycles. Hence, the CPI figure could be reduced from 2.334 to 1.884 and the sustained throughput could be increased by 24% to 133MIPS. Thus code optimization is highly desirable for a deeply pipelined processor like FRISC.

<table>
<thead>
<tr>
<th>Sustained Performance of FRISC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
</tr>
<tr>
<td>without code optimization</td>
</tr>
<tr>
<td>with code optimization</td>
</tr>
</tbody>
</table>

**Table XIX.** Sustained Performance

G. Partitioning

Figure 47 shows the partitioning of the FRISC-F architecture. The FRISC-F processor is divided into five chips, an instruction decoder chip and four 8bit datapath slices. Other processor partitionings have been investigated [HJG87, HG87, HJG88]. The partitioning shown in Figure 47 is a more conventional bit slice architecture. It has the advantage that none of the high speed buses between the
ALU and the register file are cut. The partitioning has a major impact upon the cycle time. For example, in the ALU the carry in can propagate all the way through to the carry out. The number of I/O delays encountered on this critical path is a substantial part of the worst case ALU delay. Thus the ALU delay would increase considerably if only 4bit wide datapath slices can be fabricated.

Figure 47. FRISC-F Partitioning

The partitioning has further a strong influence on the architecture itself. Features like barrel shifters or parallel multipliers can not be implemented because of I/O pin limitations. For example, a particular bit can be shifted to any other bit
position in a barrel shifter. Thus, each 8bit slice would have to read in all 32bits of the operands! Therefore, only a simple shifter can be supported in a partitioned RISC. The data alignment hardware necessary for supporting byte or half word LOAD/STORE operations is for similar reasons not supported. External data alignment hardware must be provided with 32bit chips.

With the above partitioning and the 2bit shifter a two bit multiply step and one bit divide step could be implemented. The highest implementation cost for the multiply instruction would be an additional special purpose register to hold the multiplier and shift it two positions in each cycle. This register would be part of the user visible state, just like the PSW. Thus the processor would have to be able to save and restore this register. Even if multiply and divide step instructions are provided a 32bit multiply with a 64bit result would take 34 cycles and a 32bit division would take 66 cycles. For most scientific applications this would still be too slow and a math coprocessor would be required anyway. Hence, no support of multiplication or division is provided.
H. Instruction Set

RF = Register File, PC = Program Counter, PSW = Processor Status Word, IMM = literal constant

<table>
<thead>
<tr>
<th>Instruction Fields</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODE&lt;X&gt;</td>
<td>&lt;X&gt; code bits</td>
</tr>
<tr>
<td>SRCA</td>
<td>0..31 register file address of operand A</td>
</tr>
<tr>
<td>SRCB</td>
<td>0..31 register file address of operand B</td>
</tr>
<tr>
<td>DEST</td>
<td>0..31 register file address for ALU result</td>
</tr>
<tr>
<td>AOF</td>
<td>ADD, AND, XOR, OR ALU function</td>
</tr>
<tr>
<td>CC</td>
<td>GT, GE, LT, LE, HI, LO, C, ¬C, PL, MI, EQ, NE, V, ¬V, 0, 1 Branch Cond. Code</td>
</tr>
<tr>
<td>OPA</td>
<td>RF, 0, PSW, PC operand A select</td>
</tr>
<tr>
<td>OPB</td>
<td>RF, IMM operand B select</td>
</tr>
<tr>
<td>INV</td>
<td>0, 1 invert operand B</td>
</tr>
<tr>
<td>C</td>
<td>0, 1 carry enable</td>
</tr>
<tr>
<td>SHE</td>
<td>0, 1 shift extended</td>
</tr>
<tr>
<td>SRC</td>
<td>left 2 bits, left 1 bit, right 1 bit, right 2 bits</td>
</tr>
<tr>
<td>IOB</td>
<td>byte, half word, word, cache control 3 access control bits</td>
</tr>
<tr>
<td>EXE</td>
<td>0, 1, 2, 3 number of instruction to be always executed</td>
</tr>
<tr>
<td>RTN</td>
<td>0, 1 return from protected mode</td>
</tr>
<tr>
<td>SIZ</td>
<td>byte, half word, word size of ALU operands</td>
</tr>
<tr>
<td>XXX</td>
<td>unused instruction field</td>
</tr>
</tbody>
</table>

a) Short Immediate Instructions

-if OPB = IMM then source operand B is an 8 bit literal constant.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>CODE5, OPA, INV, DEST, SIZ, OPB, SCC, XXX, C, SRCA, SRCB/IMM</td>
</tr>
<tr>
<td>ADDR</td>
<td>CODE5, SHC, INV, DEST, SIZ, OPB, SCC, SHE, C, SRCA, SRCB/IMM</td>
</tr>
<tr>
<td>AND</td>
<td>CODE5, OPA, INV, DEST, SIZ, OPB, SCC, XXX, C, SRCA, SRCB/IMM</td>
</tr>
<tr>
<td>ANDR</td>
<td>CODE5, SHC, INV, DEST, SIZ, OPB, SCC, SHE, C, SRCA, SRCB/IMM</td>
</tr>
<tr>
<td>XOR</td>
<td>CODE5, OPA, INV, DEST, SIZ, OPB, SCC, XXX, C, SRCA, SRCB/IMM</td>
</tr>
<tr>
<td>XORR</td>
<td>CODE5, SHC, INV, DEST, SIZ, OPB, SCC, SHE, C, SRCA, SRCB/IMM</td>
</tr>
<tr>
<td>OR</td>
<td>CODE5, OPA, INV, DEST, SIZ, OPB, SCC, XXX, C, SRCA, SRCB/IMM</td>
</tr>
<tr>
<td>ORR</td>
<td>CODE5, SHC, INV, DEST, SIZ, OPB, SCC, SHE, C, SRCA, SRCB/IMM</td>
</tr>
<tr>
<td>LOAD</td>
<td>CODE5, OPA, INV, DEST, SRC, IOB, SIZ, SCC, SRCB/IMM</td>
</tr>
<tr>
<td>LOADR</td>
<td>CODE5, SHC, INV, DEST, SRC, IOB, SIZ, SCC, SRCB/IMM</td>
</tr>
<tr>
<td>STORE</td>
<td>CODE5, OPA, INV, DEST, SRC, IOB, SIZ, SCC, SRCB/IMM</td>
</tr>
<tr>
<td>STORR</td>
<td>CODE5, SHC, INV, DEST, SRC, IOB, SIZ, SCC, SRCB/IMM</td>
</tr>
<tr>
<td>SETPSW</td>
<td>CODE5, OPA, INV, XXX, SRC, SRCB/IMM</td>
</tr>
<tr>
<td>JUMP</td>
<td>CODE5, OPA, INV, CC, RTN, EXE, OPB, SRC, SRCB/IMM</td>
</tr>
</tbody>
</table>

2. Instruction Set Overview

FRISC-E has a small and very regular instruction set with all the typical RISC features [DAP82, JH83, GR83, MGV85]. An ALU instruction can specify one destination register (DEST), and two source operand addresses (SRCA, SRCB). These register file address fields are always at the same location. This important RISC feature allows to overlap the operand fetch with the instruction decoding.
b) Long Immediate Instructions

- source operand B is a 16 bit literal,  
  source operand A is either R[DEST], PSW or PC

<table>
<thead>
<tr>
<th>BRANCH</th>
<th>CODE5,OPA,INV,CC,RTN,EXE,X,IMM</th>
<th>IMM</th>
<th>L=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDPSW</td>
<td>CODE5,OPA,INV,XXX,</td>
<td>IMM</td>
<td>L=1</td>
</tr>
<tr>
<td>ANDPSW</td>
<td>CODE5,OPA,INV,XXX,</td>
<td>IMM</td>
<td>L=1</td>
</tr>
<tr>
<td>XORPSW</td>
<td>CODE5,OPA,INV,XXX,</td>
<td>IMM</td>
<td>L=1</td>
</tr>
<tr>
<td>ORPSW</td>
<td>CODE5,OPA,INV,XXX,</td>
<td>IMM</td>
<td>L=1</td>
</tr>
<tr>
<td>GET</td>
<td>CODE5,OPA,INV,DEST,XXX,</td>
<td>IMM</td>
<td>L=0</td>
</tr>
<tr>
<td>GETLPC</td>
<td>CODE5,OPA,INV,DEST,XXX,</td>
<td>IMM</td>
<td>L=0</td>
</tr>
<tr>
<td>NOOP</td>
<td>CODE5,XXX</td>
<td></td>
<td>L=0</td>
</tr>
</tbody>
</table>

L = latency of instruction

3. Long Immediate Instructions

As in any RISC processor only LOAD and STORE instructions can access memory. The following memory addressing modes are available for load instructions, Register, Register + Index, Register ± 8bit displacement, and Program Counter Relative. An absolute addressing mode with a positive or negative 8bit address is also available. The addressing mode of STORE instruction is restricted to Register ± 3bit displacement. A three bit field (IOC) in load/store instructions is provided to control external data alignment hardware for byte, half word, and word data references. The field is large enough to allow a Kernel, User, or I/O space specifications for memory protection purposes. LOAD instructions have a latency of three cycles. The instructions in the latency region can reference the load destination register but will get the old register value. NOOP instructions must be
scheduled if a data dependency on the load destination register exists in the code. Consecutive LOAD instructions can be scheduled, however, no load/store instruction should be scheduled immediately following a STORE instructions if the cache has only a single memory bank.

The instruction set contains four basic integer manipulation instructions, ADD, OR, XOR, AND. Operand B can be bitwise inverted or negated for ADD instructions. Operand A can be shifted one or two bit positions to the left or right. A two's complement data format is assumed for all operands. Add or shift overflows are trapped.

Two different control transfer instruction, BRANCH and JUMP, are available. These instruction can branch on signed or unsigned integer condition codes. The branch target address is computed by adding or subtracting a 16bit immediate constant from operand A. Operand A can be either the PC_DE register or the constant 0. The JUMP instruction supports the same condition codes as the BRANCH instruction. However, the branch target address of a JUMP instruction can be a register ± 8bit displacement, or even the sum of two registers. Therefore, the JUMP instruction can branch to any instruction in the address space. Both control transfer instructions have a two bit execute (EXE) instruction field. This field specifies how many of the three instructions in the branch latency region should always be executed. By setting the EXE field to 00 a branch with flush is executed. By setting the EXE field to 11 a delayed branch is executed. The BRANCH, JUMP
instructions can also reset the CPU mode from Protected Mode to Normal mode. The CPU goes into protected mode after an interrupt or trap. In Protected Mode asynchronous user interrupts are disabled to prevent a second interrupt before the interrupt service routine had time to save the CPU state. A conditional software trap can be executed by setting the ST field of the BRANCH instruction to one. This can be used for the support of supervisor or kernel calls.

The GET, GETLPC instructions can save the program counters PC_DE, PC_DW, or the program status word into the register file. They can then be saved on the system stack. A sequence of GETLPC will read out all the program counters even in protected mode! Positive or negative 16bit literals can be loaded into registers with GET or GETLPC. A 32bit constant can be assembled with two GET instructions by using the LDH bit field. If the LDH bit is set the 16bit literal from the instruction is loaded into the upper 16bits of the immediate constant and the lower 16bits are set to zero.

The ADDPSW, ANDPSW, XORPSW, and ORPSW instructions are provided for bit manipulations on the program status word. The SETPSW instruction is provided for restoring the PSW after exceptions.
Some frequently used instructions can be synthesized from core instructions as shown below.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Synthesis</th>
</tr>
</thead>
<tbody>
<tr>
<td>INC RX</td>
<td>ADD RN &lt;- RN+#1</td>
</tr>
<tr>
<td>DEC RX</td>
<td>ADD RN &lt;- RN-#1</td>
</tr>
<tr>
<td>MOV RX&lt;-RY</td>
<td>ADD RX &lt;- RX+#0</td>
</tr>
<tr>
<td>CLR RX</td>
<td>ADD RX &lt;- 0+#0</td>
</tr>
<tr>
<td>NEG RX</td>
<td>ADD RX &lt;- 0-RX  // two’s complement</td>
</tr>
<tr>
<td>INV RX</td>
<td>XOR RX &lt;- RX^#-1 // one’s complement</td>
</tr>
<tr>
<td>CMP RX,#li</td>
<td>ADD RT &lt;- RX-#val // compare Reg to literal</td>
</tr>
<tr>
<td>CMP RX,RY</td>
<td>ADD RT &lt;- RX-RY // compare Reg to Reg</td>
</tr>
<tr>
<td>SHL1 RX</td>
<td>ADD RX &lt;- RX*2+#0 // arithmetic shift</td>
</tr>
<tr>
<td>SHL2 RX</td>
<td>ADD RX &lt;- RX*4+#0</td>
</tr>
<tr>
<td>SHR1 RX</td>
<td>ADD RX &lt;- RX/2+#0</td>
</tr>
<tr>
<td>SHR2 RX</td>
<td>ADD RX &lt;- RX/4+#0</td>
</tr>
<tr>
<td>PUSH RX</td>
<td>INC SP; STORE [SP+#0] &lt;- RX // SP = Stack Pointer</td>
</tr>
<tr>
<td>POP RX</td>
<td>LOAD RX &lt;- [SP+#0]; DEC SP // SP = Stack Pointer</td>
</tr>
</tbody>
</table>

4. Synthesized Instructions
Instruction Class: ALU \{ADD, SUB, AND, OR, XOR\}

<table>
<thead>
<tr>
<th>I</th>
<th>0</th>
<th>ALU</th>
<th>OPA</th>
<th>I`</th>
<th>DEST</th>
<th>SIZ</th>
<th>O</th>
<th>B</th>
<th>SRCA</th>
<th>SRCB/8BIT</th>
<th>LIT.</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>28</td>
<td>24</td>
<td>20</td>
<td>16</td>
<td>12</td>
<td>8</td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

Operation: 
Register[DEST] <- Operand_A <ALUOP> Operand_B

Operands:
- OPA=00: Operand_A = Register[SRCA]
- OPA=01: Operand_A = 0
- OPA=10: Operand_A = Processor Status Word
- OPA=11: Operand_A = Program Counter (PC, DE)
- OPB=0, INV=0: Operand_B = Register[SRCB]
- OPB=0, INV=1: Operand_B = ~Register[SRCB]
- OPB=1, INV=0: Operand_B = 8bit literal
- OPB=1, INV=1: Operand_B = ~8bit literal

Status:
SCC=1: Negative, Carry, Overflow, Zero
SCC=0: no flags are changed

Traps:
add overflow causes an arithmetic trap if SIZ!=11

Function:
- ALUOP==00: ADD
  C=0, INV=0: carry_in= 0
  C=1, INV=0: carry_in= Carry
  C=0, INV=1: carry_in= 1
  C=1, INV=1: carry_in=~Carry

  ALUOP==01: AND

  ALUOP==10: XOR

  ALUOP==11: OR

  SIZ=00: Add Byte (8bit)

  SIZ=01: Add Half Word (16bit)

  SIZ=10: Add Word (32bit)

  SIZ=11: Add Word, no trap (32bit)

Description:
The ALU function <ALUOP> is performed with Operand_A and Operand_B. The result is saved the DEST register. Operand_B can be bitwise inverted by setting the INV bit. During ADD operations the carry in is inverted if the INV bit is set. This is equivalent to adding the two's complement of Operand_B (SUB). The SIZ field only affects the Negative, Carry, Overflow, and Zero flags. ALU instructions have no latency.

1 negative for ADD function
I. Class: ALUSH \{ADD, SUB, AND, OR, XOR\}

<table>
<thead>
<tr>
<th>31</th>
<th>28'</th>
<th>24'</th>
<th>18'</th>
<th>12'</th>
<th>8'</th>
<th>4'</th>
<th>0'</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>I</td>
<td>ALU</td>
<td>SHC</td>
<td>DEST</td>
<td>SIZ</td>
<td>SRC</td>
<td>SRCB/BBIT</td>
</tr>
</tbody>
</table>

Operation:
Register[DEST] <- Operand_A <ALUOP> Operand_B

Operands:
- SHC=00: Operand_A = Register[SRCA]*4
- SHC=01: Operand_A = Register[SRCA]*2
- SHC=10: Operand_A = Register[SRCA]/2
- SHC=11: Operand_A = Register[SRCA]/4
- OPB=0, INV=0: Operand_B = Register[SRCB]
- OPB=0, INV=1: Operand_B = ~Register[SRCB]
- OPB=1, INV=0: Operand_B = 8bit literal
- OPB=1, INV=1: Operand_B = ~8bit literal

Status:
SCC=1: Negative, Carry, Overflow, Zero
SHE=1: Cl, C0

Traps:
add or shifter overflow causes an arithmetic trap if SIZ!=11

Function:
ALUOP==00: ADD C=0, INV=0: carry_in= 0
- C=1, INV=0: carry_in= Carry
- C=0, INV=1: carry_in= 1
- C=1, INV=1: carry_in= ~Carry

ALUOP==01: AND
ALUOP==10: XOR
ALUOP==11: OR

SIZ=00: Add Byte (8bit)
SIZ=01: Add Half Word (16bit)
SIZ=10: Add Word (32bit)
SIZ=11: Add Word, no trap (32bit)

Description:
The ALU function <ALUOP> is performed with Operand A and Operand_B. The result is saved the DEST register. Operand_B can be bitwise inverted by setting the INV bit. During ADD operations the carry in is inverted if the INV bit is set. This is equivalent to adding the two's complement of Operand_B (SUB). The SIZ field only affect the Negative, Carry, Overflow, and Zero flags. ALUSH instructions have no latency. No extended shifts for byte or half word operands are supported.

\footnote{negative for ADD function}
Instruction: LOAD

<table>
<thead>
<tr>
<th>1 1 0 0 0</th>
<th>OPA</th>
<th>DEST</th>
<th>0</th>
<th>IOCNT</th>
<th>SRCA</th>
<th>SRCB/8BIT LIT.</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 28 24 16 12 8 4 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation: \( \text{Register}[\text{DEST}] \leftarrow [\text{Operand}_A + \text{Operand}_B] \)

Operands:
- OPA=00: \( \text{Operand}_A = \text{Register}[\text{SRCA}] \)
- OPA=01: \( \text{Operand}_A = 0 \)
- OPA=10: \( \text{Operand}_A = \text{Processor Status Word} \)
- OPA=11: \( \text{Operand}_A = \text{Program Counter (PC.DE)} \)
- OPB=0, INV=0: \( \text{Operand}_B = \text{Register}[\text{SRCB}] \)
- OPB=0, INV=1: \( \text{Operand}_B = -\text{Register}[\text{SRCB}] \)
- OPB=1, INV=0: \( \text{Operand}_B = \text{8bit literal} \)
- OPB=1, INV=1: \( \text{Operand}_B = -\text{8bit literal} \)

Access Control:
- IOC=000: Byte Memory Access
- IOC=001: Half Word Memory Access
- IOC=010: Word Memory Access

Status: no flags are changed

Traps: overflow causes an arithmetic trap

Description: The DEST register is loaded with data from memory location \([\text{Operand}_A + \text{Operand}_B]\). The IOC field determines the size of the data loaded. LOAD instructions have a latency of three cycles. If an instruction in the load latency region references the DEST register, the old value will be obtained.
Instruction: LOADSH

```
1 1 1 0 0  SHC | DEST | D B | IOCNT | SRCA | SRCB/8BIT LIT.
```

Operation: \[ \text{Register}[\text{DEST}] \leftarrow [\text{Operand}_A + \text{Operand}_B] \]

Operands:
- \( \text{SHC}=00 \): \( \text{Operand}_A = \text{Register}[\text{SRCA}] \times 4 \)
- \( \text{SHC}=01 \): \( \text{Operand}_A = \text{Register}[\text{SRCA}] \times 2 \)
- \( \text{SHC}=10 \): \( \text{Operand}_A = \text{Register}[\text{SRCA}] / 2 \)
- \( \text{SHC}=11 \): \( \text{Operand}_A = \text{Register}[\text{SRCA}] / 4 \)
- \( \text{OPB}=0, \text{INV}=0 \): \( \text{Operand}_B = \text{Register}[\text{SRCB}] \)
- \( \text{OPB}=0, \text{INV}=1 \): \( \text{Operand}_B = -\text{Register}[\text{SRCB}] \)
- \( \text{OPB}=1, \text{INV}=0 \): \( \text{Operand}_B = 8\text{bit literal} \)
- \( \text{OPB}=1, \text{INV}=1 \): \( \text{Operand}_B = -8\text{bit literal} \)

Access Control: IOC=000: Byte Memory Access
- IOC=001: Half Word Memory Access
- IOC=010: Word Memory Access

Status: no flags are changed

Traps: add or shifter overflow causes an arithmetic trap

Description: The DEST register is loaded with data from memory location \([\text{Operand}_A + \text{Operand}_B]\). The IOC field determines the size of the data loaded. LOADSH instructions have a latency of three cycles. If an instruction in the load latency region references the load DEST register the old value will be obtained.
Instruction: STORE

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>OPA</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>28</td>
<td>24</td>
<td>28</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>IRCNT</td>
<td>SRCA</td>
</tr>
</tbody>
</table>

Operation: Register[SRCB] -> [Operand_A + #constant]

Operands:
- OPA=00: Operand_A = Register[SRCA]
- OPA=01: Operand_A = 0
- OPA=10: Operand_A = Processor Status Word
- OPA=11: Operand_A = Program Counter (PC_DE)

INV=0: #constant = 3bit literal
INV=1: #constant = -3bit literal

Access Control
- IOC=000: Byte Memory Access
- IOC=001: Half Word Memory Access
- IOC=010: Word Memory Access

Status: no flags change

Traps: overflow causes an arithmetic trap

Description: The SRCB register is copied into memory location [Operand_A + #constant]. The IOC field determines the size of the data stored. STORE instructions keep the cache busy for two cycles. The write cycle is preceded by a read cycle. Therefore, no load/store instruction can immediately follow a STORE if the cache has only a single memory bank!
**Instruction: STORESH**

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>24</th>
<th>20</th>
<th>16</th>
<th>12</th>
<th>8</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>11110</td>
<td>SHC</td>
<td>IOCNT</td>
<td>SRCA</td>
<td>SRCB</td>
<td>LIT</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Operation:**  
Register[SRCB] -> [Operand_A + #constant]

**Operands:**  
- **SHC=00:**  
  
  Operand_A = Register[SRCA]*4  
- **SHC=01:**  
  
  Operand_A = Register[SRCA]*2  
- **SHC=10:**  
  
  Operand_A = Register[SRCA]/2  
- **SHC=11:**  
  
  Operand_A = Register[SRCA]/4

INV=0:  
#constant = 3bit literal

INV=1:  
#constant = -3bit literal

**Access Control**  
- **IOC=000:** Byte Memory Access
- **IOC=001:** Half Word Memory Access
- **IOC=010:** Word Memory Access

**Status:**  
no flags are changed

**Traps:**  
overflow causes an arithmetic trap

**Description:**  
The SRCB register is copied into memory location [Operand_A + #constant]. The IOC field determines the size of the data stored. STORESH instructions keep the cache busy for two cycles. The write cycle is preceded by a read cycle. Therefore, no load/store instruction can immediately follow a STORESH if the cache has only a single memory bank!
Instruction: SETPSW

Operation: \[ \text{PSW} = \text{Operand}_A + \text{Operand}_B \]

Operands:
- OPA=00: \( \text{Operand}_A = \text{Register}[\text{SRCA}] \)
- OPA=01: \( \text{Operand}_A = 0 \)
- OPA=10: \( \text{Operand}_A = \text{Processor Status Word} \)
- OPA=11: \( \text{Operand}_A = \text{Program Counter (PC DE)} \)

- OPB=0, INV=0: \( \text{Operand}_B = \text{Register}[\text{SRCB}] \)
- OPB=0, INV=1: \( \text{Operand}_B = -\text{Register}[\text{SRCB}] \)
- OPB=1, INV=0: \( \text{Operand}_B = 8\text{bit literal} \)
- OPB=1, INV=1: \( \text{Operand}_B = -8\text{bit literal} \)

Status: all flags are overwritten

Traps: none

Description: The sum of \( \text{Operand}_A \) and \( \text{Operand}_B \) is copied into the status word (PSW). The updated status word (PSW) is only available after a latency of one cycle. However, \text{JUMP} or \text{BRANCH} instructions immediately use the updated status flags for branch condition evaluation.
**Instruction: JUMP**

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>OPA</th>
<th>1</th>
<th>CC</th>
<th>B</th>
<th>N</th>
<th>EXE</th>
<th>DB</th>
<th>SRCA</th>
<th>SBRC/8BIT</th>
<th>LIT.</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>28</td>
<td>24</td>
<td>20</td>
<td>16</td>
<td>12</td>
<td>8</td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Operation:** if (branch==1)  
Program Counter (PC_I1)=Operand_A+Operand_B  

**Operands:**  
OFA=00: Operand_A = Register[SRCA]  
OFA=01: Operand_A = 0  
OFA=10: Operand_A = Processor Status Word  
OFA=11: Operand_A = Program Counter (PC_DE)  
OPB=0,INV=0: Operand_B = Register[SRCB]  
OPB=0,INV=1: Operand_B = -Register[SRCB]  
OPB=1,INV=0: Operand_B = 8bit literal  
OPB=1,INV=1: Operand_B = -8bit literal  

**Condition:**  
CC=0000:branch= ((N ^ V) + Z) ;le (signed)  
CC=0001:branch=~((N ^ V) + Z) ;gt (signed)  
CC=0010:branch= (N ^ V) ;lt (signed)  
CC=0011:branch=~(N ^ V) ;ge (signed)  
CC=0100:branch= (~C + Z) ;ls (unsigned)  
CC=0101:branch= (~C + Z) ;ht (unsigned)  
CC=0110:branch= C ;_c  
CC=0111:branch=~C ;_nc  
CC=1000:branch= N ;mi  
CC=1001:branch=~N ;pl  
CC=1010:branch= Z ;eq  
CC=1011:branch=~Z ;ne  
CC=1100:branch= V ;_v  
CC=1101:branch=~V ;_nv  
CC=1110:branch= 1 ;_1  
CC=1111:branch= 0 ;_0  

**Flush:**  
EXE==00:if (branch) flush i1,i2,de stages  
EXE==01:if (branch) flush i1,i2 pipeline stages  
EXE==10:if (branch) flush il pipeline stage  
EXE==11:always execute next three instructions  

**Status:** if RTN=1 & branch=1 the protected mode (P flag) will be cleared after 3 cycles.  

**Traps:** none  

**Description:** If the branch condition is true the branch target address (operand_A + operand_B) is copied into the program counter (PC_I1). The JUMP instruction has a latency of three cycles. The EXE field determines the number of instructions in the latency region that are always executed.
Instruction: NOOP

Operation: no operation
Status: no flags are changed
Traps: none
Description: The NOOP instruction does not change any register or status flag. Its only function is to delay the execution of the next instruction by one cycle. NOOP instructions might be necessary in the load latency region of a LOAD instruction or between STORE instructions.
Instruction: BRANCH

![Instruction Format](image)

Operation: if (branch==1)

- Program Counter (PC_Il) = Operand_A + #constant

Operands:
- OPA=00: Operand_A = illegal (Register[DEST])
- OPA=01: Operand_A = 0
- OPA=10: Operand_A = Processor Status Word
- OPA=11: Operand_A = Program Counter (PC_DE)
- INV=0: #constant = 16bit literal
- INV=1: #constant = -16bit literal

Condition:
- CC=0000: branch= ((N ∧ V) + Z) ; le (signed)
- CC=0001: branch= ~((N ∧ V) + Z) ; gt (signed)
- CC=0010: branch= (N ∧ V) ; lt (signed)
- CC=0011: branch=~(N ∧ V) ; ge (signed)
- CC=0100: branch= (~C + Z) ; ls (unsigned)
- CC=0101: branch=~(~C + Z) ; ht (unsigned)
- CC=0110: branch= C ; c
- CC=0111: branch=~C ; nc
- CC=1000: branch= N ; mi
- CC=1001: branch=~N ; pl
- CC=1010: branch= Z ; eq
- CC=1011: branch=~Z ; ne
- CC=1100: branch= V ; _v
- CC=1101: branch=~V ; _nv
- CC=1110: branch= 1 ; _1
- CC=1111: branch= 0 ; _0

Flush:
- EXE==00: if (branch) flush i1, i2, de stages
- EXE==01: if (branch) flush i1, i2 pipeline stages
- EXE==10: if (branch) flush i1 pipeline stage
- EXE==11: always execute next three instructions

Status:
- if RTN=1 and branch=1 the protected mode (P flag) will be cleared after 3 cycles.

Traps:
- software trap if ST=1 and branch=1 (latency=2)

Description:
If the branch condition is true the branch target address (Operand_A + #constant) is copied into the program counter (PC_Il). The BRANCH instruction has a latency of three cycles. The EXE field determines the number of instructions in the latency region that are always executed.
I. Class: MPSW \{ADDPSW, ANDPSW, XORPSW, ORPSW\}

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ALUOP</td>
</tr>
<tr>
<td>01</td>
<td>OPA</td>
</tr>
<tr>
<td>1</td>
<td>IN</td>
</tr>
<tr>
<td>DEST</td>
<td>LSR</td>
</tr>
<tr>
<td>16</td>
<td>L</td>
</tr>
<tr>
<td>16BIT LITERAL</td>
<td></td>
</tr>
</tbody>
</table>

**Operation:**
Processor Status Word = Operand_A<ALUOP>#const.

**Operation:**
- ALUOP==00: ADD
  - INV=0: carry_in=0 (32bit)
  - INV=1: carry_in=1 (32bit)
- ALUOP==01: AND
- ALUOP==10: XOR
- ALUOP==11: OR

**Operands:**
- OPA=00: Operand_A = Register[DEST]
- OPA=01: Operand_A = 0
- OPA=10: Operand_A = Processor Status Word
- OPA=11: Operand_A = Program Counter (PC_DE)

**INV=0, LDH=0:** #constant= 16bit lit.; lower 16bits
**INV=0, LDH=1:** #constant= 16bit lit.; upper 16bits
**INV=1, LDH=0:** #constant= ~16bit'lit.; lower 16bits
**INV=1, LDH=1:** #constant= ~16bit'lit.; upper 16bits

**Status:**
All flags are overwritten, if the SPM bit is set the protected mode (P flag) is set after 3 cycles.

**Traps:**
None

**Description:**
The result of the operation ALUOP is copied into the processor status word (PSW). If the INV bit is set the 16bit literal is bitwise inverted. The updated status word is only available after two cycles. However, JUMP or BRANCH instructions will immediately reference the updated status flags during branch condition evaluation.

\(^1\) negative for ADD function
Instruction Class: GET \{LD\#, GETPSW, GETPC\}

```
0 1 1 0 0 OPA 5 16BIT LITERAL
 31  28  24  20  16  12  8  4  0
```

Operation: \[ \text{Register}[\text{DEST}] = \text{Operand}_A + \#\text{constant} \]

Operands:
- OPA=00: \( \text{Operand}_A = \text{Register}[\text{DEST}] \)
- OPA=01: \( \text{Operand}_A = 0 \)
- OPA=10: \( \text{Operand}_A = \text{Processor Status Word} \)
- OPA=11: \( \text{Operand}_A = \text{Program Counter (PC\_DE)} \)

INV=0, LDH=0: \#\text{constant}=16\text{bit lit.}; \text{lower 16bits}
INV=1, LDH=0: \#\text{constant}=-16\text{bit lit.}; \text{lower 16bits}
INV=0, LDH=1: \#\text{constant}=16\text{bit lit.}; \text{upper 16bits}
INV=1, LDH=1: \#\text{constant}=-16\text{bit lit.}; \text{upper 16bits}

Status: no flags are changed

Traps: none

Description: The GET instructions loads the DEST register with the sum of the Operand A and a 16bit literal from the instruction. The literal can be loaded into the lower or higher 16bits of the immediate constant, thus providing a means for assembling 32bit constants from the instruction stream.
Instruction: GETLPC

```
0 1 0 0 0 0 0 0 0 OPA 1 0 DEST 0 0 0 0 0 0 0 0 0 0 0 0 16BIT LITERAL
```

Operation: \( \text{Register[DEST]} = \text{Operand}_A + \text{#constant} \)

Operands:
- OPA=00: \( \text{Operand}_A = \text{Register[DEST]} \)
- OPA=01: \( \text{Operand}_A = 0 \)
- OPA=10: \( \text{Operand}_A = \text{Processor Status Word} \)
- OPA=11: \( \text{Operand}_A = \text{Program Counter (PC\_DW)} \)

\( \text{INV}=0, \text{LDH}=0: \#\text{constant}=16\text{bit lit.}; \text{lower 16bits} \)
\( \text{INV}=1, \text{LDH}=0: \#\text{constant}=-16\text{bit lit.}; \text{lower 16bits} \)
\( \text{INV}=0, \text{LDH}=1: \#\text{constant}=16\text{bit lit.}; \text{upper 16bits} \)
\( \text{INV}=1, \text{LDH}=1: \#\text{constant}=-16\text{bit lit.}; \text{upper 16bits} \)

Status: no flags are changed

Traps: none

Description: The GETLPC instruction loads the DEST register with the sum of Operand\_A and a 16bit literal. The GETLPC is the only instruction that allows to access the program counter for the DW pipeline stage. A sequence of GETLPC can read out the program counters, pc\_ex, pc\_d1, pc\_d2, and pc\_dw, in protected mode.
I. FRISC Macro Assembler

A two pass assembler has been written for FRISC to facilitate the writing of test programs. The assembler uses recursive descent for parsing source files. It supports symbolic references, labels, and strings. The assembler source is listed in the appendix. The assembler can be used together with the UNIX preprocessor m4. The preprocessor m4 allows the user to include source files and to define aliases and macros. If the ’-s’ flag is used on the macro preprocessor m4 command line, error messages from the FRISC assembler print the source file and line number where the error was detected. Thus the problems can be fixed directly in the source file without looking at the preprocessor output.

The assembler generates an output listing for the user and two data files that can be used to initialize the data and instruction memories in the FRISC system implemented on the VERILOG simulator. The assembler source and the command line options are case insensitive.

1. FRISC Assembler Syntax

\[
\begin{align*}
<\text{reg}> &= R<31..0> \\
&= SP \quad // \text{Stack Pointer} \\
<\text{dest}> &= <\text{reg}> \\
&= PSW \quad // \text{Processor Status Word} \\
&= CFC \quad // \text{Current PC (PC\_DE)}
\end{align*}
\]
<opa> =  <reg>  // <reg>=<dest> for PSW and GET instr.
  =  <reg>*2  // only for ADD, AND, XOR, OR and LD, ST
  =  <reg>*4  // only for ADD, AND, XOR, OR and LD, ST
  =  <reg>/2  // only for ADD, AND, XOR, OR and LD, ST
  =  <reg>/4  // only for ADD, AND, XOR, OR and LD, ST
  =  0
  =  PSW
  =  CPC
  =  LPC  // Last PC (PC_DW) only for GET

<oph> =  <reg>  // illegal for ST, BRA, GET, and PSW...
  =  $<num>  // 3bit for ST, 8bit for ALU, SETPSW, JMP
  =  @<name>  // Symbol Reference
  =  @<label>  // label reference

<dir> =  .[word|long] <name> <list>
  =  .short <name> <name> <list>
  =  .byte<char> <name> <list>
  =  .string <name> "<string>"
  =  .ip <num>  // set instruction pointer/address
  =  .ip [+|-] <num>  // set instruction pointer relative
  =  .dp <num>  // set data pointer/address
  =  .dp [+|+] <num>  // set data pointer relative

<line> =  [<label>:]<instr>; <comment-to-end-of-line>
  ;<comment-to-end-of-line>
  /* comment */
  =  <dir>

<label> =  <string>

<name> =  <string>

<string> =  [A-Z|a-z ]<rem>

<rem> =  [A-Z|a-z |0-9]<rem>

<num> =  [1-9]<digits>  // decimal number
  =  0<octal_digits>  // octal number!
  =  0x<hex_digits>  // hex number

<list> =  <num>,<list>
  =  <num>

<size> =  [/BYTE]/[CHAR]  // 8bit operands
  =  /SHORT  // 16bit operands
  =  [/WORD]/[LONG]  // 32bit operands, default value
  =  /NOAT  // 32bit operands, no overflow trap
```assembly
<cc>  =  _le  //  (N'V)+Z  ;less equal (signed)
    =  _gt  //  -(N'V)+Z  ;greater than (signed)
    =  _lt  //  (N'V)  ;less than (signed)
    =  _ge  //  -(N'V)  ;greater equal (signed)
    =  _ls  //  (-C+Z)  ;less same (unsigned)
    =  _ht  //  (-C+Z)  ;higher than (unsigned)
    =  _c  //  C  ;carry flag
    =  _nc  //  -C  ;no carry flag
    =  _n  //  N  ;negative flag
    =  _mi  //  N  ;minus (signed)
    =  _nn  //  -N  ;no negative flag
    =  _pl  //  -N  ;plus (signed)
    =  _z  //  Z  ;zero flag
    =  _eq  //  Z  ;equal
    =  _nz  //  -Z  ;no zero flag
    =  _ne  //  -Z  ;not equal
    =  _v  //  V  ;overflow flag
    =  _nv  //  -V  ;no overflow flag
    =  _0  //  0  ;never
    =  _1  //  1  ;always
    =  _  //  1  ;default

  =AND <dest>=<opa>[$&<opb>[<size>]] [/[SHEX] [/[SCC]]
  =XOR <dest>=<opa>[^<opb>[<size>]] [/[SHEX] [/[SCC]]
  =OR <dest>=<opa>[\|\|<opb>[<size>]] [/[SHEX] [/[SCC]]
  =SETPSW FSW=<opa>[+|-]<opb>
  =[LOAD][LD] <dest>=\[<opb>[+|-]<opb>\] [/IO=<num>]
  =[STORE][ST] <dest>=\[<opb>[+|-]<opb>\] [/IO=<num>]
  =[JUMP][JMP] <cc>CPC=<opb>[+|-]<opb> [/EX=<num>]/[CLRPM]
  =[JUMP][JMP] <cc>[<label> [/[EX=<num>]/[CLRPM]
  =[BRANCH][BRA] <cc>[<label> [<EX=<num>]/[CLRPM]/[TRAP]
  =[BRANCH][BRA] <cc>[<label> [/[EX=<num>]/[CLRPM]/[TRAP]
  =PSWADD FSW=<opa>[+|-]<opb> [/LDHI] [/[SETPM]
  =PSWAND FSW=<opa>[$&<opb> [/[LDHI] [/[SETPM]
  =PSWXOR FSW=<opa>[^<opb> [/[LDHI] [/[SETPM]
  =PSWOR FSW=<opa>[\|\|<opb> [/[LDHI] [/[SETPM]
  =GET <dest>=<opa>[+|-]<opb> [/LDHI]
  =NOP
  =MOV <dest>=<opa>[/[SCC] [/<size>]
  =SRL2 <reg> [/[SCC] [/[SHEX] [/<size>]
  =SRL1 <reg> [/[SCC] [/[SHEX] [/<size>]
  =SRLR <reg> [/[SCC] [/[SHEX] [/<size>]
  =SRL2 <reg> [/[SCC] [/[SHEX] [/<size>]
  =INC <reg> [/[SCC] [/<size>]
  =DEC <reg> [/[SCC] [/<size>]
  =CLR <reg> [/[SCC] [/<size>]
  =CMP <opb>=<opb> [/<size>]

<option>  =/[SCC]  ;set condition code, by default off
  =/C  ;enable carry, by default disabled
  =/[SHEX]  ;enable extended shift, by default off
  =/LDHI  ;load immediate high, by default off
  =/TRAP  ;enable software trap, by default off
  =/CLRPM  ;clear protected mode, by default off
  =/SETPM  ;set protected mode, by default off
  =/IO=<num> ;set I/O control field, by default 2
  =/EX=<num> ;set EXE control field, by default 0
```
2. Assembler Command Line Options

Invocation: fas <file_name> -<option>

<option>
=SP=<num> ; expand SP to register <num>, default r31
=RTMP=<num> ; use register <num> as temporary register
            for CMP instructions, default 30
=debug ; set debug option, run disassembler
=symbol ; show symbol table
=o=<string> ; change name of output files
            default "frisc"
=ic=<num> ; set size of instruction memory in words
            default 256
=dc=<num> ; set size of data memory in words
            default 256

3. M4 Macro Preprocessor

To use the M4 macro preprocessor the following line at the beginning
of the assembler source code file must be included.

    include(std.h)

The -s option should be used when invoking the preprocessor:

    m4 -s <file_name>

include file:

    include(<file_name>)

define alias:

    define(<alias>, {<string>})

define macro:

    define(<macro_name>, {<macro_body>})
    in <macro_body> $1,$2,$3...$n symbols are place holders
    for macro arguments

invoke macro:

    <macro_name>({arg1,arg2,...arg<n>})

Predefined macros in std.h:

    define(push,{
        add sp=sp+4;
        st $1=[sp+0];})
    define(pop,{
        ld $1=[sp+0];
        add sp=sp-4;})

The m4 man page provides a detailed description of the preprocessor.
2. Assembler Command Line Options

**invocation:** fas <file_name> -<option>

- `<option>`
  - `-sp=<num>`: expand SP to register <num>, default r31
  - `-rtmp=<num>`: use register <num> as temporary register for CMF instructions, default 30
  - `-debug`: set debug option, run disassembler
  - `-symbol`: show symbol table
  - `-o=<string>`: change name of output files
  - `-frisc`:
  - `-ic=<num>`: set size of instruction memory in words
    - default 256
  - `-dc=<num>`: set size of data memory in words
    - default 256

3. M4 Macro Preprocessor

To use the M4 macro preprocessor the following line at the beginning of the assembler source code file must be included.

```plaintext
#include (std.h)
```

The `-s` option should be used when invoking the preprocessor:

```plaintext
m4 -s <file_name>
```

**include file:**

```plaintext
include (<file_name>)
```

**define alias:**

```plaintext
define (<alias>, {<string>})
```

**define macro:**

```plaintext
define (<macro_name>, {<macro_body>})
```

- in `<macro_body>`: `$1$, $2$, $3$, .. $n$ symbols are place holders for macro arguments

**invoke macro:**

```plaintext
<macro_name>(arg1, arg2, ..., arg<n>)
```

**predefined macros in std.h:**

```plaintext
define (push, {
    add sp=sp+4;
    st $1=[sp+0];
})
define (pop, {
    ld $1=[sp+0];
    add sp=sp-4;
})
```

The `m4` man page provides a detailed description of the preprocessor.
4. Sample Program

Recursive Calculation of Fibonacci's Numbers

C Source Code Description

```c
int fibonacci(N)
int N;
{
    int V;
    if (N<=1) V=1;
    else V=fibonacci(N-1)+fibonacci(N-2);
    return V;
}
```

generates sequence: 1,1,2,3,5,8,13,21,34,55,89,133,233,377,610,....

Stack Frame

```
sp+0 : N
sp+4 : V
sp+8 : return address
```

Register Assignment

```
r0 : return value
r1 : N
r2 : V
r3 : temp return address
r4 : temp stack address
```

```
start: MOV sp=1020 ; initialize stack pointer
       MOV r1=5 ; set N = 5
       ADD sp=sp-16 ; allocate stack frame
       ST r1=[sp] ; save N on stack
       MOV r3=@rtn0 ; r3 = return address
       ADD r4=sp+8 ; r4 = temporary address pointer
       ST r3=[r4] ; save return address on stack
       JMP @fib ; call fibonacci(5)

rtn0: ADD r0=r0 ; dummy instruction to show r0 on trace
      ADD r1=r1 ; dummy instruction to show r1 on trace
```
JMP @rtn0 ; infinite loop at end of program
NOOP

stop: NOOP ; set break point at this location

fib: CMP r1-1 ; compare N to 1
BRA _gt @else ; if (N>1) goto else
JMP @rtn /ex=3 ; goto return (latency=3)
MOV r0=1 ; set return value
LD r3=[sp+8] ; r3 = return address
LD r2=[sp+4] ; restore r2

else: DEC r1 ; r1 = N-1
ADD sp=sp-16 ; allocate stack frame
ST r1=[sp] ; save N on stack
JMP @fib /ex=3 ; call fibonacci(N-1)
MOV r3=@rtn1 ; r3 = return address
ADD r4=sp+8 ; r4 = temporary address pointer
ST r3=[r4] ; save return address on stack
MOV r2=r0 ; V=fibonacci(N-1)
DEC r1 ; r1 = N-2
ADD sp=sp-16 ; allocate stack frame
ST r1=[sp] ; save N on stack
MOV r3=@rtn2 ; r3 = return address
JMP @fib /ex=3 ; call fibonacci(N-2)
ST r2=[sp+4] ; save V on stack
ADD r4=sp+8 ; r4 = temporary address pointer
ST r3=[r4] ; save return address on stack
ADD r0=r2+r0 ; r2 = fibonacci(N-1)+fibonacci(N-2)
LD r3=[sp+8] ; r3 = return address (latency=3)
LD r2=[sp+4] ; restore r2

rtn: LD r1=[sp] ; restore r1
ADD sp=sp+16 ; deallocate stack frame
JMP cpc=r3 ; return from subroutine (latency=3)
Figure 61. I/O Interface

Figure 61 shows the I/O interface of FRISC-F. The timing information for the signals is given with the following syntax <Stage>.<Phase>. An output signal that is going through a transition at the beginning of p3 in the execution stage of the instruction would have the timing tag ex.p3. By convention any signal name with
an "_" character is a single ended ECL signal. The other signals are differential. All input signal must be synchronized with the four phase clock with the exception of the reset_, and int_ signals. These signal are conditioned by sending them first through three latches. The latches are given one cycle to restore the input signal to zero or one in case the asynchronous input signal was going through a transition as the input latch was clocked. Without this signal conditioning different parts of the processor can interpret the marginal input signal differently resulting in a hardware failure [HJV80,HHC86].

The following figures show the timing for an instruction cache read, a data cache read, and a data cache write.

Figure 62. Instruction Cache Read

1) processor outputs branch target address (branch taken)
2) instruction cache latches branch target address or increments address
3) instruction cache outputs instruction
4) instruction decoder latched instruction
5) instruction decoder latches instruction cache hit signal
Figure 63. Data Cache Read Timing

1) processor outputs data address
2) data cache latches data address
3) data cache outputs data word onto dbus_o
4) instruction decoder latches data cache hit signal
5) processor latches data

Figure 64. Data Cache Write Timing

1) processor outputs data address
2) data cache latches data address
3) data cache outputs data and generates cache hit signal
4) data cache write start if cache hit
5) instruction decoder latches data cache hit signal
<table>
<thead>
<tr>
<th>Signal/Bus</th>
<th>Timing</th>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_bus[31:0]</td>
<td>out:ex.p4</td>
<td>ICACHE DCACHE</td>
<td>Shared Address Bus for (v_dad=1) Branch Target Address (v_iad=1) Data Addresses</td>
</tr>
<tr>
<td>v_iad</td>
<td>out:ex.p4</td>
<td>ICACHE</td>
<td>Valid Instruction Address</td>
</tr>
<tr>
<td>v_dad</td>
<td>out:ex.p4</td>
<td>DCACHE</td>
<td>Valid Data Address</td>
</tr>
<tr>
<td>w</td>
<td>out:ex.p4</td>
<td>DCACHE</td>
<td>Write Data</td>
</tr>
<tr>
<td>_m_stall</td>
<td>out:p4</td>
<td>ICACHE DCACHE CONTROL</td>
<td>Pipeline Stall -&gt; repeat last instruction or update cache after miss</td>
</tr>
<tr>
<td>exc_</td>
<td>out:p4</td>
<td>ICACHE DCACHE CONTROL</td>
<td>Exception Condition -&gt; Pipeline Flush</td>
</tr>
<tr>
<td>iocmt_[2:0]</td>
<td>out:ex.p4</td>
<td>ICACHE DCACHE</td>
<td>Input Output Control Field for LOAD/STORE Instructions</td>
</tr>
<tr>
<td>halt_</td>
<td>in:p2</td>
<td>CONTROL</td>
<td>Halt Processor -&gt; causes a pipeline stall and forces bus signals low.</td>
</tr>
<tr>
<td>int_</td>
<td>in:p2</td>
<td>CONTROL</td>
<td>User Interrupt</td>
</tr>
<tr>
<td>trap d</td>
<td>in:p2</td>
<td>CONTROL</td>
<td>Data Trap / Page Fault</td>
</tr>
<tr>
<td>trap i</td>
<td>in:p2</td>
<td>CONTROL</td>
<td>Instruction Trap / Page Fault</td>
</tr>
<tr>
<td>error</td>
<td>in:p2</td>
<td>CONTROL</td>
<td>Hardware Failure -&gt; abort</td>
</tr>
<tr>
<td>reset</td>
<td>in:p2</td>
<td>CONTROL</td>
<td>Processor Reset</td>
</tr>
<tr>
<td>d_buso[31:0]</td>
<td>out:d1.p1</td>
<td>DCACHE</td>
<td>Data Output to DCACHE</td>
</tr>
<tr>
<td>d_busi[31:0]</td>
<td>in:dw.p3</td>
<td>DCACHE</td>
<td>Data Input from DCACHE</td>
</tr>
<tr>
<td>hit d</td>
<td>in:dw.p2</td>
<td>DCACHE</td>
<td>Data Cache Hit / No stall</td>
</tr>
<tr>
<td>hit i</td>
<td>in:de.p2</td>
<td>ICACHE</td>
<td>Instruction Cache Hit / No stall</td>
</tr>
<tr>
<td>i_bus[31:0]</td>
<td>in:de.p1</td>
<td>ICACHE</td>
<td>Instruction Bus</td>
</tr>
<tr>
<td>clk</td>
<td>in:</td>
<td>CLOCK</td>
<td>Differential 500MHz Clock</td>
</tr>
<tr>
<td>run_stop</td>
<td>in:</td>
<td>CONTROL</td>
<td>Startup Signal for Four Phase Clock Generator</td>
</tr>
</tbody>
</table>

\(^1\) delayed by three cycles

Table XX. FRISC-F Input/Output Signals
K. Interrupt Handling

Figure 65. Pipeline Flush after Trap

Most of the complexity in the design of a highly pipelined processor is due to interrupts and exceptions. Any process can be interrupted on a processor with a virtual memory system because of page faults. FRISC must, therefore, be able to restart an interrupted process. To reduce the overhead associated with interrupts the restart of the processor should not involve a time consuming analysis of the type of instructions in the pipeline at time of the interrupt. The interrupt response should be fast for real time or embedded controller applications. FRISC supports, therefore, vectored interrupts.

To be able to restart the instruction stream after an interrupt instructions must not complete out of sequence! If this rule is followed the processor state is consistent with a sequential model of instruction execution and the interrupt is
precise [JES88]. The restart point after a precise interrupt is the instruction immediately following the last instruction to complete. An instruction has completed once it has modified the process (user visible) state irreversibly. For example, overwriting a register will change the state irreversibly unless a copy of the original value is kept to restore the register. In FRISC a copy of the previous processor status word is kept to restore the PSW after an exception. An other way of preventing an irreversible process state change is the copying of the new register value into a temporary register that is tagged. This solution is used for the ALU result in FRISC.

In RISC processors load instructions can only complete in the last pipeline stage because the data is only available on the processor at this point. Further, page faults are only detected when load and store instruction have reached the last pipeline stage. Thus load or store instructions can be aborted in the DW stage. ALU instructions could, however, complete in the D1 stage since the result and the condition code flags have been generated in the execution stage. This would, however, violate the no out of order completion rule. An ALU instruction could follow a load/store instruction which could be aborted in the DW stage. Thus the completion of the ALU instruction must be delayed by at least two cycles! The ALU result is, therefore, passed through the ALU result pipeline registers (RES_EX, RES_D1, RES_D2) before it is written into the register file. These result registers must be tagged otherwise ALU instructions would have a latency of three cycles.
ALU instructions complete once they passed the D2 stage while load/store instruction complete once they passed the DW stage. Hence, ALU instructions can not be aborted in the DW stage! The instruction stream must, therefore, be restarted at the instruction in the D2 stage if the instruction in the DW stage was an ALU instruction when the interrupt occurred. This particular scheme was chosen for FRISC since it requires no result register for the DW stage with all the associated tagging and feed forward logic! It also provides an empty write register file slot for load instructions just before the data input register (D_IN) is overwritten. Therefore, no pipeline interlocks are necessary for consecutive load instructions!

After an interrupt has been detected the processor flushes all pipeline stages except the DW stage if an ALU instruction is in that stage as shown in Figure 65. In the middle of every cycle the decision is made whether the cycle can complete or not. A cycle can not complete if a stall condition has occurred or an interrupt is pending. Critical signals like write register file or write status word can, therefore, only be generated in the second half of an instruction cycle!

In the cycle following the interrupt the processor goes into the exception state and inserts a ghost "jump trap_vector" into the DE pipeline stage. From the exception state the processor goes into protected mode as shown in Figure 66. In protected mode asynchronous user interrupts are disabled to avoid a second interrupt before the machine state has been saved. Further, the last four PC register values are frozen. They can be read out destructively by four GETLPC
Figure 66. CPU States

Instructions.

If a load/store instruction was in the DW stage the PC registers are immediately frozen when the trap occurred. Otherwise one additional shift of the PC registers is performed such that the address in the PC_DW points actually to the instruction in the D2 stage. This avoids a time consuming analysis of the instruction type in the DW stage by the interrupt handler to find out if the instruction in the DW stage is an ALU instruction and the instruction stream must
be restarted at the instruction in the D2 stage. The processor stays in protected mode until a jump with return from protected mode (RTN) is executed. This should be done as soon as the state has been saved and the processor is ready to accept another interrupt. The processor can be forced into protected mode by executing a MPSW instruction with the SPM bit field set.

![Diagram](image)

**Figure 67. Restart of Instruction Stream**

Four instruction addresses are necessary to restart an interrupted instruction stream because of the branch latency of three instructions. An instruction address can be either the previous instruction address plus one or be a branch target address of a branch instruction executed four cycles ago. The potential instruction address links are shown in Figure 67. Branch target addresses can in general not be derived from any previous address. Thus potential branch target addresses
generated by instructions that have already completed and are, therefore, not
restarted must be saved as state information! If the instruction at address IA0 is
a branch instruction that was taken, IA4 is a branch target address and can not be
derived from previous addresses. See Figure 67. Similarly IA3 and IA2 could be
branch target addresses of branch instructions that have already completed.
IA1, IA2, IA3, and IA4 must all be saved to reconstruct the instruction stream. The
instruction addresses IA5 and so forth can be reconstructed from the first four
addresses since they are either linked to or are branch target addresses of
restarted instructions.

A complication arises from bubbles generated by branch instructions. Branch instructions can flush instructions in the branch latency region. These
bubbles in the instruction stream must be regenerated or replaced by NOOP’s
after a pipeline restart. Again the worst case is a BRANCH instruction at address
IA0. If the branch is taken the instructions at IA1, IA2, and IA3 could be flushed.
Bubbles at or after IA4 can only be generated by branch instructions that will be
restarted and thus these bubble(s) will automatically be regenerated. Therefore,
only the last three instruction addresses must reflect whether the corresponding
instruction was flushed or not. In FRISC the last three program counter registers
are cleared to indicate that the corresponding instruction was flushed. If the
instruction at address zero is always a NOOP the instruction stream can be
restarted quickly. A pipeline restart sequence for the interrupt handler is shown in
5. Restart of Instruction Pipeline

The instruction stream is restarted by four consecutive JUMP register instructions. This will create the instruction stream IA1-IA4 because of the branch latency of three cycles. By overlapping these JUMP instruction with load instructions three of the registers that are necessary to hold the branch target addresses IA1-IA4 can be restored. Only one register (R30) can not be restored by the interrupt handler. This register should, therefore, not be used by user programs other than as a write only register. For example, the compare instruction
(CMP) could always designate this register as its destination. The result of a CMP instruction is the setting of the condition flags. However, a destination register must be specified but even though it is never used.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
<th>Level</th>
<th>Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>reset_</td>
<td>processor reset</td>
<td>8</td>
<td>00000</td>
</tr>
<tr>
<td>restart</td>
<td>trap in protected mode</td>
<td>7</td>
<td>00100</td>
</tr>
<tr>
<td>error_</td>
<td>system error</td>
<td>6</td>
<td>01000</td>
</tr>
<tr>
<td>trap_d</td>
<td>trap on data, page fault</td>
<td>5</td>
<td>01100</td>
</tr>
<tr>
<td>a_t</td>
<td>arithmetic overflow trap</td>
<td>4</td>
<td>10000</td>
</tr>
<tr>
<td>s_t</td>
<td>software trap</td>
<td>3</td>
<td>10100</td>
</tr>
<tr>
<td>trap_i</td>
<td>trap on instr., page fault</td>
<td>2</td>
<td>11000</td>
</tr>
<tr>
<td>int_</td>
<td>user interrupt</td>
<td>1</td>
<td>11100</td>
</tr>
</tbody>
</table>

| \(^2\) | internally generated signal if any trap other than reset is detected in protected mode or by user (I = 0) |

Table XXI. Trap Vectors

Table XXI shows the trap signals with the associated trap vectors. The reset signal has the highest priority to insure that the processor can always be reset to a known state. To detect spurious interrupts a restart trap is generated if the processor detects an interrupt in protected mode. Trap vectors are only five bits wide. The vectors point to addresses in the first memory page. The trap vectors
are offset by only four instructions. This is sufficient to transfer control with a delayed branch instruction to the interrupt service routine while losing only a single cycle for the control transfer.

L. Perspective

![Diagram of FRISC-E and FRISC-F architectures]

Figure 68. Control Signals

The FRISC-F architecture introduced in this chapter is the latest version of FRISC. The architecture as presented has been modified to incorporate the experience gained from the previous versions [JFM86, HG87, HJG90]. The major modification is the partitioning. The previous partitioning divided the datapath into
four register file slices, four ALU slices, and one status word chip. This partitioning was used because of the foundry’s yield expectations. Meanwhile the Tektronix foundry has fabricated dual 4bit A/D converts [VEG88] and other test circuits that have shown that circuits of larger complexity can be built. Further, the register file chip and status register chip turned out to be strictly I/O bound. The partitioning also cut the high speed bus between the register file and the ALU. Two operands and one ALU result must be transferred on this L-Bus during a 4ns cycle. A differential bus had to be used in order to increase the bus bandwidth and to reduce the switching noise. The number of I/O pins on the three chips for this bus, therefore, doubled. Since most of the integrated circuits turned out to be I/O bound the control signals between the instruction decoder and the datapath had to be time multiplexed. This, however, caused many routing problems since control signals that can be time multiplexed are not necessarily generated or used on nearby chip locations. Therefore, the simpler and faster direct control signal transfer is used for FRISC-F as shown in Figure 68 and the register file chip and ALU chip are fused together to form a single 8bit datapath bitslice.

The L-Bus or any other high speed net with multiple drivers should further be avoided because of a specific transmission line effect. The transmission line interconnect must be routed as a chained tree to avoid reflections at branch points. If a driver which is not at the beginning or end of the transmission line drives the net the signal is split into two waves with only half the voltage swing propagating
up and down the line [DBJ63] as shown in Figure 69. This occurs because the driver sees an interconnect impedance of only 25Ω. No first incident switching can be assumed since the noise in the system might be too large to guarantee proper switching when the first half wave arrives. Thus the receiver at the terminator end will only switch reliably after the reflection from the unterminated end superimposes with the direct wave. Thus the electrical length of the net can be longer than its physical length!

The worst case on the L-Bus would be a signal transfer from the ALU to the shifter. The signal transfer would take 540ps on a 5cm (300ps) long transmission line if the reflected wave must arrive at the terminator end before the receiver switches. Fortunately, this not a transfer that will occur. The worst case transfer that can occur, is a transfer from the Status Word chip to the Shifter (420ps). Only 1ns is available for a transfer from the Status Word to the shifter output. driver, receiver and shifter delay is 450ps. Thus the L-BUS will be on a critical path.
if the clock skew is more than ±50ps. The reflected wave will pass the driver that is currently driving the net since the open collector drivers used for differential output signals have a high output impedance. They are equivalent to switched current sources.

With single ended drivers the situation would be worse since single ended Emitter Coupled Logic (ECL) drivers have a low output impedance causing a reflection at the driver. This would lead to multiple reflections and cause significant noise.

The instruction set of FRISC-F has been extended to include Byte, and 16bit ALU instructions. Further, the ability to load a 32bit constant from the instruction stream with two instructions has been added. These capabilities would be beneficial for workstation type application. Other instructions have been slightly modified to reduce the number of gates required for decoding on the instruction decoder. The instruction decoder turned out to be the chip with the highest power dissipation.

A register transfer level model has been written for FRISC-F using the behavioral simulator VERILOG from Cadence. VERILOG is a compiler for a simulation language and provides fast behavioral and gate level simulations. Thus a FRISC-F processor with a small memory subsystem can be simulated. This model system together with the macro assembler can be used for initial code development and as a teaching tool for FRISC. The behavioral description can
further serve as a technology independent specification for the ongoing FRISC research. The model timing is phase accurate.

While additional features have been added to the simulation model a prototype GaAs FRISC processor should not necessarily include all features since these features increase the design complexity, design time, power, and number of I/O pins. The inclusion of the following features in a prototype should therefore be carefully examined under these aspects:

- byte and 16bit operand support for ALU instructions.
- branch with execute; delayed branch is simpler to implement.
- shift by two bit positions.
- interrupts; the number of interrupt signals could be reduced to:
  reset, restart, non maskable interrupt, maskable interrupt.
  tristating of bus lines and bus signals if halt signal is applied.
Chapter V.

FRISC-F Simulator

A. Model Structure and Basic Features

Figure 70 shows the module structure of the register transfer model of FRISC-F written in the VERILOG simulation language. The processor core, frisc, consists of the Instruction Decoder module and a 32bit Datapath module. The datapath is modelled with a single 32bit module instead of four 8bit slices to increase the simulation speed. Even with this modification the throughput is only four instructions per second on a Sun3/60. The FRISC-F model includes two identical memory modules (icache, dcache) with a capacity of 1kByte and two data alignment modules to support byte and half word load/store instructions. The memory capacity can easily be increased by changing a module parameter in the source file, cache.v. The instruction and data memory modules can emulate cache misses, page faults, and breakpoints. The VERILOG sources for FRISC-F are listed in the appendix.

The user can single step instructions or run a fixed number of instruction cycles. By turning off the stepping mode the simulator can be run until a breakpoint in the data or instruction memory is reached. The program execution can be observed through the instruction trace on the screen. The trace shows the instruction address and the instruction in the execution stage, and the input and
Figure 70. Modules of the FRISC-F VERILOG Model

output of the ALU. A sample trace shown below. Address and data are displayed in hexadecimal format.
6. Sample Instruction Trace

The snapshot for trace data is taken at the beginning of phase 4. The instruction is displayed as an mnemonic string generated by the built in FRISC-F code disassembler. Events like interrupts, pipeline stalls and flushes, and CPU
state transitions are also displayed on the screen. Whatever is displayed on the trace screen is also captured in the log file, verilog.log. A graphics window that displays the user visible state of the processor can be invoked. It shows the registers, status word, ALU input and output latches, data I/O and address latches, and the program counters as shown in below. The symbol %h represents a hexadecimal value, %d represents a decimal value, and %b represents a binary value.

The simulator is invoked by calling the VERILOG compiler with the top level source module, frisc.v, as input file. A shell script, vv, is provided that calls the VERILOG compiler with the necessary password and library options. The compiler will look up all the modules referenced by the top level module in the library directory and then prompt the user for input. The simulator will print first a HELP message that lists the most important user interface routines, called tasks in VERILOG. A listing of the help message is shown below. This message can be recalled at any time by typing HELP:.

At the prompt the user can load the data and instruction memory with a LDGO("<filename>") statement. The LDGO task loads the file <filename>.ins into the instruction memory and the file <filename>.dat into the data memory. These two files are generated by the FRISC-F assembler. After loading the instructions and data into memory the LDGO task sets the processor reset signal high for two cycles and returns. The reset signal will cause a reset trap, forcing the processor
FRISC-F - USER VISIBLE STATE: cycle = %d    phase = %d

<table>
<thead>
<tr>
<th>DOUT</th>
<th>=h</th>
<th>DIN</th>
<th>=h</th>
<th>TAG</th>
<th>=h</th>
</tr>
</thead>
<tbody>
<tr>
<td>R30</td>
<td>=h</td>
<td>R31</td>
<td>=h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R28</td>
<td>=h</td>
<td>R29</td>
<td>=h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R26</td>
<td>=h</td>
<td>R27</td>
<td>=h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R24</td>
<td>=h</td>
<td>R25</td>
<td>=h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R22</td>
<td>=h</td>
<td>R23</td>
<td>=h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R20</td>
<td>=h</td>
<td>R21</td>
<td>=h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R18</td>
<td>=h</td>
<td>R19</td>
<td>=h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R16</td>
<td>=h</td>
<td>R17</td>
<td>=h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R14</td>
<td>=h</td>
<td>R15</td>
<td>=h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R12</td>
<td>=h</td>
<td>R13</td>
<td>=h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R10</td>
<td>=h</td>
<td>R11</td>
<td>=h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R8</td>
<td>=h</td>
<td>R9</td>
<td>=h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R6</td>
<td>=h</td>
<td>R7</td>
<td>=h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R4</td>
<td>=h</td>
<td>R5</td>
<td>=h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td>=h</td>
<td>R3</td>
<td>=h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R0</td>
<td>=h</td>
<td>R1</td>
<td>=h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rfa</td>
<td>=h</td>
<td>rfb</td>
<td>=h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>aopa</td>
<td>=h</td>
<td>aopb</td>
<td>=h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>res_ex</td>
<td>=h</td>
<td>TAG</td>
<td>=h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>res_d1</td>
<td>=h</td>
<td>TAG</td>
<td>=h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>res_d2</td>
<td>=h</td>
<td>TAG</td>
<td>=h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC_I1</td>
<td>=h</td>
<td>FSW:</td>
<td>NCVZ=%b, (P,I,C1,C0)=%b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC_I2</td>
<td>=h</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC_DE</td>
<td>=h</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC_EX</td>
<td>=h</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC_D1</td>
<td>=h</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC_D2</td>
<td>=h</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC_DW</td>
<td>=h</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

7. Format of User Visible State Display

to flush the instruction pipeline and to jump to interrupt vector 0. This startup sequence is shown in the sample trace above.
FRISC-F SIMULATOR V1.0 COMMAND LEVEL TASKS

HELP          ; shows this list; ?<CR> prints VERILOG help menu
TRACE([1|0])  ; turn instruction trace mode on/off
DEBUG([1|0])  ; turn debug mode on/off
STEP([1|0])   ; turn single cycle step mode on/off
PSTEP([1|0])  ; turn phase step mode on/off
RUN(<num>)    ; run <num> cycles and then prompt user
RESET         ; generate reset trap
ERROR         ; generate error trap
INT([1|0])     ; set or reset user interrupt signal
TRAPD         ; generate data trap
TRAPI         ; generate instruction trap
DUMPIC        ; dump instruction memory to file:dump.ic
DUMPPDC       ; dump data memory to file:dump.dc
BREAKD(<num>) ; break at data memory address <num>
STALLD(<num>) ; stall at data memory address <num>
FAULTD(<num>) ; stall and page fault at data memory address <num>
CLEARD(<num>) ; clear break, stall, or page fault at address <num>
BREAKI(<num>) ; break at instruction address <num>
STALLI(<num>) ; stall at instruction address <num>
FAULTI(<num>) ; stall and page fault at instruction address <num>
CLEARI(<num>) ; clear break, stall, or page fault at instr. address <num>
WINDOW        ; turn register window on
LDGO("<str>") ; load file: <str> and generate reset
QUIT          ; quit and print statistics

$ input("<str>"); read commands from file: <str>
-> TO RUN A TASK ENTER: <TASK_NAME>,<CR>
->control-c stops the simulation and prompts user for input
->control-d or QUIT:.,<CR> at the prompt will terminate the program

8. Simulator Help Message

B. User Interface

The FRISC-F simulator is invoked with the shell script vv.

vv frisc.v [-i <file_name>.key] +<option>

If the -i option is set the user input is read from the specified key file instead of the console. The options control the initial startup mode and the user visible state display. All options can also be invoked from the interactive prompt. The following startup options are currently defined.
window      : turn on the graphics display of the user visible state
stepoff     : turn off single step mode initially
traceoff    : turn off trace mode initially
debug       : turn on debug mode

The default startup mode is trace on, single step on, and state display off.
The graphic state display slows down the simulation and should only be used to
assist debugging. Note, it can not be turned off once it has been invoked! In debug
mode any change in a signal between instruction decoder and datapath, as well
as the most important signals between the memories and the CPU are displayed.
Each module has its own dummy debug register to allow the debugging of
individual modules. Hierarchical module path names must be used to reference
signals within submodules. For example, the debug mode for the instruction
decoder with the hierarchical path name, frisc.id, can be turned on with the
statement:

frisc.id.debug=1;

Single step and trace modes can be set or reset with the STEP and TRACE
tasks. Stepping can be turned off for a fixed number of cycles with the RUN task.
The PSTEP task allows the user to turn on single clock phase step mode which
provides maximum timing resolution. The user visible state display can be turned
on at the user prompt by calling the task WINDOW. Input statements can be read
from a key file with system task, $input("<file_name>.key");. The input source is
switched back to the console when the end of the key file is reached. Signals or
registers can be monitored with the $monitor system task. For example, the bra_
signal generated on the datapath and sent to the instruction decoder can be monitored with the following statement.

\$monitor("-\text{bra}_\_=%b",\text{frisc.bra}\_);

The current state of any signal or register in the system can be displayed with the \$showvars() system task. Entering a question mark at the prompt followed by a carriage return (<CR>) will display the VERILOG help message. Most of the VERILOG statements can be entered at the prompt. The simulation language and the interactive commands are described in the VERILOG-XL manual from Cadence.

All commands must be terminated with a semicolon followed by a <CR>. Multiple line statements can be entered. The statements will only be executed after entering a point followed by a carriage return. A previous statement can be repeated by entering its line number followed by a carriage return. For further information refer to the VERILOG help message (?<CR>). Program execution can be stopped at any time by entering a Control-C and continued by entering a point followed by a carriage return. The program can be terminated by typing a Control-D at the user prompt, by entering $\text{finish};<$CR>, or by entering QUIT::<CR>. The user task QUIT will display some statistical information before terminating the program as shown in the sample trace.

The user can load an assembly program and execute it with the LDGO("<file_name>") user task. LDGO will load the instruction and data memory
with data from the memory initialization files <file_name>.ins and <file_name>.dat generated by the FRISC-F assembler. It then resets the processor to cause a jump to trap vector 0. The user can reset the processor a any time by with the task RESET. Error traps and user interrupts can be generated with the ERROR and INT tasks.

The user can set breakpoints in the instruction or data memory with the BREAKI(<word_address>) or BREAKD(<byte_address>) tasks. Note, addresses are byte oriented but instruction addresses are word oriented. The user can enter data or addresses in decimal, hexadecimal, octal, or binary formats.

<table>
<thead>
<tr>
<th>decimal</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>hex</td>
<td>'h17</td>
</tr>
<tr>
<td>octal</td>
<td>'o27</td>
</tr>
<tr>
<td>binary</td>
<td>'b10111</td>
</tr>
</tbody>
</table>

Breakpoints cause a user prompt whenever a byte of the memory word location marked is accessed. Besides breakpoints the user can trigger cache misses or page faults at a specified memory location with STALLI(<word_address>) STALLD(<byte_address>) and FAULTI(<word_address>), FAULTD(<byte_address>) tasks. Only one event can be triggered at a particular data or instruction memory word. The two least significant bits of the data memory address specified in a STALLD, FAULTD, or BREAKD are, therefore, irrelevant. A cache miss, page fault, or breakpoint can be