Technische Universität München
Institut für Informatik

Diplomarbeit

Implementation of the 4stack Processor Using Verilog

Verfasser:	Bernd Paysan
Aufgabensteller:	Prof. Dr. A. Bode
Betreuer:	Dr. W. Karl, M. Leberecht, R. Lindhof
Abgabetermin:	15. August 1996

Ich versichere, daß ich diese Diplomarbeit selbständig verfaßt und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15.8.1996

Abstract

This paper reveals implementation details of the 4stack processor architecture, implemented in Verilog as a feasibility study as diploma thesis. Design flow, synthesis tools and verification methods are explained. Partition in functional units and space-time tradeoff considerations are discussed as well as techniques for efficient implementation of special arithmetic units.

Special interests have been laid onto the implementation of stack register files and corresponding spill buffers. Instruction and data caches have been implemented to satisfy the demands of a VLIW architecture. A fast, pipelined signal processing unit (integer multiply and accumulate with rounding) and an equally fast, pipelined floating point unit are described.

Contents

List of Figures

Chapter 1

Introduction

1.1 Goal of this Work

As conclusion of a long discussion in the usenet news group `comp.arch` in spring 1993 about the shortcomings of stack architectures compared to superscalar RISC processors, I proposed a very long instruction word (VLIW) stack architecture, the 4stack processor. Further discussions (especially with Kyle Hayes) improved the proposed machine and finally lead to a complete instruction set architecture (ISA) (see [14]).

Then a simulator for this ISA was implemented as part of a "Fortgeschrittenen-Praktikum (FoPra)" at Technische Universität München. This simulator allowed to run some benchmarks and showed that the fine grain parallelism in both digital signal processing (DSP) and graphic algorithms can be exploited with the proposed architecture, leading to both fast (due to the VLIW concept) and compact (due to the stack paradigm of single units) code.

The good results of simulation lead to optimistic prognoses of performance and usefullness of the 4stack processor, so as next step it was chosen to implement it as feasibility study using a hardware description language and synthesis tools to generate an ASIC (application specific integrated circuit) implementation of the processor architecture. This work now describes design flow, details, and pitfalls of this implementation.

The goal of this work thus is to implement the 4stack ISA in register transfer language (RTL) and structural Verilog and synthesize this for the design target ASIC process. Furthermore, economic aspects of design tradeoffs (e.g. space-time tradeoffs) have to be discussed.

1.2 Motivation

When the 4stack architecture basically was proposed three years ago, neither very long instruction word (VLIW) nor stack architectures seemed to be important instruction set architecture paradigms in the forseeable future.

Things have changed since then: according to recent announcements and rumors, Intel and Hewlett Packard are working on a processor with VLIW elements ("Merced") which is designed as successor of the most important personal computing processor (the "Intel Architecture"), and — in terms of system unit sales — the dominant workstation reduced instruction set computer (RISC) architecture. Sun tries to push a stack oriented architecture (the Java virtual machine, JavaVM [13]) in embedded control and a new class of terminals or home computing, the network computer (NC).

VLIW architectures are characterized by large instruction words combining direct and independent action control of every functional unit. Thus one control unit issues a wide instruction every cycle to utilize a large number of data paths and functional units in parallel.

Stack architectures organize register accesses as last-in first-out (LIFO) latches, thus accesses concentrate around a small window. Addressing of registers is implicit through the stack pointer, so register numbers are not encoded in the instruction word.

Both ISA paradigms have their own merits and shortfalls. VLIW exploits fine grain parallelism, but typically leads to bloated code. Stack architectures are supposed to give compact code, but this is depending on compiler quality and coding style. Single-stack architectures like the JavaVM suffer from lack of registers, especially to store more than one active (thus changing) values. Also it is hard to exploit parallelism using superscalar methods, because the top of stack (TOS) value is a bottleneck. Therefore stack machines either are concentrated to support stack languages (Forth processors like the RTX 2000 [5]) or have special stack frame support (JavaVM or AT&T's CRISP processor [3]).

Both the stack processor shortfall of too few active registers and difficulties to exploit parallelism and the VLIW's shortfall of uncompact code and thus very high recommended memory bandwidth and cache trashing can be overcome by applying both principles. The TOS bottleneck now serves to help write path separation and thus to keep register files small and fast.

"Idealized" models of VLIW architectures use uniform functional units connected to a central register file [2] [4] [9]. Using a central register file with a large number of ports, communication delays between the processing elements can be minimized. A central register file together with uniform functional units also simplifies code generation. However, central multiported register files increase chip complexity and slow down register access.

Therefore, already the first generation of VLIW architectures like the Multiflow TRACE machines [2] and the Cydra 5 [18] have either separate register files or use more sophisticated techniques like the Context Register Matrix.

In the past, the VLIW architecture paradigm has influenced the design of several microprocessor architectures, too. Examples are the LIFE-VLIW microprocessor, proposed by Philips Research, Palo Alto [12] [20], Intel's iWarp microprocessor [15] and Intel's i860 64-Bit microprocessor [10].

A more detailed overview of design issues of VLIW architectures and related architectures exploiting instruction-level parallelism can be found in [1] [17] [8].

1.3 Characterization of the 4stack Processor Architecture

The basic components of each processor's data path are ALU and registers to store values inside the processor. The most prominent feature of the 4stack architecture is the fact that these registers are organized as LIFO stacks, thus honoring the locality of register usage, particularly for intermediate values with short lifetimes. The implementation of stack operations can be characterized as follows:

- Arguments of operations are consumed, thus taken from the stack.
- Results of operations are pushed to the stack.
- All stack elements are of the same size (32 bit on the 4stack processor). Shorter values are extended, longer values are stored in more than one stack cell.
- Operations that operate on the stack, instead of the values (thus only exchange or duplicate values, not compute anything) are called "stack operations".
- All stack accesses concentrate around the top of stack (TOS). If deeper parts of the stack are not accessible through stack operations, they are not accessible at all.

The 4stack processor features four stacks, each connected to an ALU. As an extension of the basic stack operation model, stack operations may read the four topmost values of other stacks through a four way crossbar switch. Deeper values (up to 8th element of a stack) can be read only from each operation's stack. All stacks use an implicit spill and fill mechanism to and from memory.

Frequently used ALU operations can be merged with stack operations, which are then executed concurrently with the ALUoperations.

Signal processing is supported by two pipelined integer multiply-and-add units. Stack 0/1 and stack 2/3 respectively each share a DSP unit. The first pipeline stage of the DSP unit multiplies two 32 bit signed or unsigned 32 bit integers to a 64 bit product while the second stage contains a barrel shifter to shift the result of the multiplication, a 64 bit adder to accumulate sums of products, and a round unit that rounds the upper 32 bits of the resulting sum.

Two data units are responsable for accessing main memory which are shared by the evenly and oddly numbered stacks, respectively. Memory addresses are computed from a four-register set including a 64-bit base register (Ri), an offset register

(Ni, index or increment value), a limit (Li for range checks and to implement FIFOs (first in, first out buffers)) and usage flags (Fi).

The FPU comprises a floating point adder and a floating point multiplier. Evenly numbered stacks feed the left argument of floating point operations while oddly numbered stacks provide the right argument. The floating point input arguments are preserved.

The instruction prefetcher predecodes branch operations and tries to follow branches as soon as possible. Both the sequentially followed instruction and the branch target instruction are decoded.

1.4 Structure of this Work

Chapter 2 describes the design approach and tool usage. It generally explains how to get from an instruction set architecture to silicon. The following chapters elaborate functional units, partitioned into pipeline stages. Chapter 3 shows basic low level units and clock generation. The four pipeline stages are described in Chapter 4 (instruction fetch), Chapter 5 (instruction decode), Chapter 6 (execution phase), and Chapter 7 (memory interface).

Chapter 2

Design Approach

Starting with the data flow Figure 2.1 a top-down design approach was taken. The design is partitioned into stacks, arithmetic logic units (ALUs), data units, digital signal processing (DSP) units, floating point unit (FPU), instruction decoder, prefetch logic, data and instruction cache.

In the further design process these modules were divided into further parts to keep them manageable (see Figure 2.2 for the design flow).

A library of reusable basic functions like adders and multipliers was generated as part of this work. These components can be replaced and thus allow tradeoff decisions. This gives a three layer design: the first layer are the logical units as found on the data path diagram, the second layer are the parts of these units, and the finest grain are generic units like adders, multiplexers and so on.

The implementation then was bottom up, concentrating on single units to make them work. This design approach makes use of the specified interface and allows to verify simple parts, thus eases the debug process.

2.1 Hardware Description Language

Two hardware description languages today dominate hardware development: VHDL (**V**ery High Speed Integrated Circuits **H**ardware **D**escription **L**anguage), developed by the department of defense (USA) and the Verilog HDL (**Veri**fy **Log**ic **H**ardware **D**escription **L**anguage), a former proprietary language of Cadence, Inc.. Both HDLs now are standardized by IEEE, Verilog recently.

The differences between both HDLs are not big. VHDL's syntax resembles ADA, which is a consequence of it's origin, while Verilog resembles C. VHDL thus is more wordy and has a complex system of types, most of them useless for synthesis. Verilog, on the other hand, has a C-like preprocessor, which allows to emulate some of VHDL's features like structures (by giving vector subranches symbolic names). Verilog's main shortfall behind VHDL is that it does not provide conditional or looping instantiation of components, which can be partly overcome using an external macro

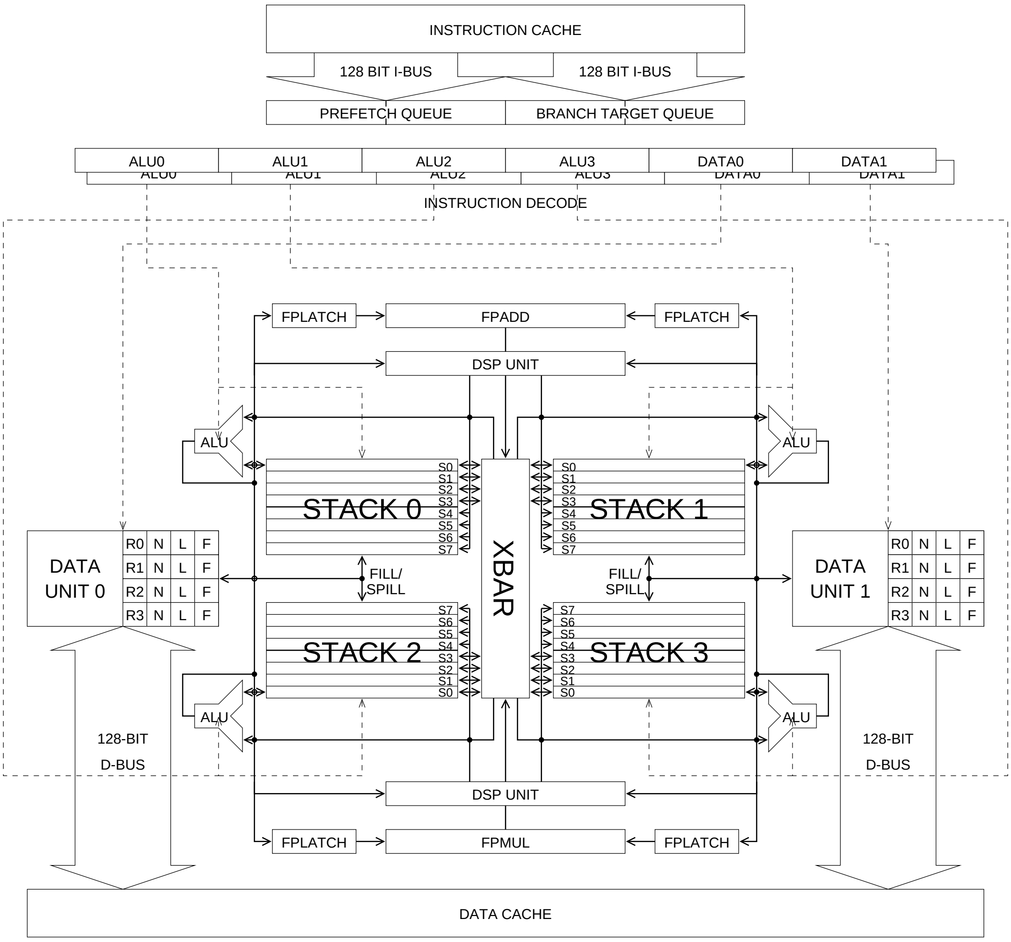

Figure 2.1: Data Path

preprocessor. In the actual design process, this missing feature is only partly a problem, since Verilog provides widely used repeated structures like memories or barrel shifters and vector operations.

2.1.1 Differences Between HDLs and "Classical" Computer Languages

The language target environment (the implementation in silicon) differs quite from the von Neumann's paradigm found by software. Logical functions are performed at any input change. To synchronize processing, clock signals and edge-triggered flip-flops are used. Verilog provides wires and continuous assignments, which is the equivalent of a logical function; registers and event-triggered assignment, the equivalent of edge-triggered flip-flops.

Instead of dividing software in procedures and functions, Verilog programs are

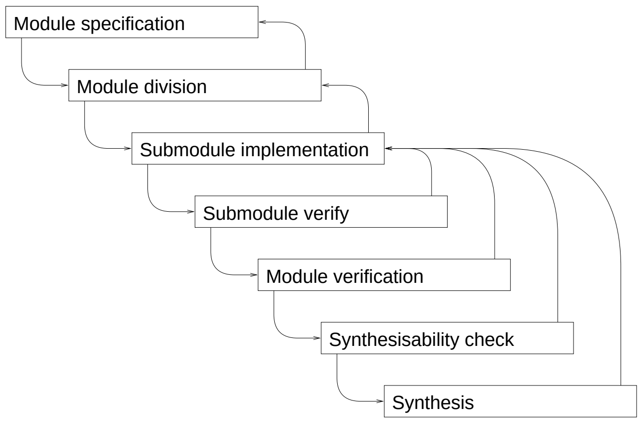

Figure 2.2: Design flow

divided into modules[1]. Modules have private data (per instantiation) and program parts. This is a partly object oriented approach, while the main principles of object oriented programming (inheritance and polymorphism) are missing. Among from providing reusability of parts, the module concept allows individual verification and synthesis of parts of a design.

Modules are divided into

- **module header** starting with the keyword `module`, declares module name and port list
- **port declaration** declares all entries of the port list as `input`, `output` or `inout`
- **gate declaration** declares used nets, registers, primitives, etc.
- **module instances** instantiates used modules
- **procedural block** specifies activities performed during simulation (RTL level or behavioral Verilog)

While module header, port and gate declarations can be compared with a function's head in C, procedural blocks are the functional equivalent of a function's body. However, procedural blocks are executed on events, not on calls:

- **initial:** the procedural block is executed at simulation start (at reset)
- **always:** the procedural block is executed always at the specified event. There are synthesizible constructs using events (clock edge triggered flip-flops, behavioral continuous assignments, level triggered latches), and non-synthesizible (multiple clocks for one edge triggered register, delays, incomplete inputs for behavioral continuous assignments).

[1]There are functions, too. However, they don't play an important role.

- **continuous assignments:** the assignment is executed always when an input variable changes. The disadvantage over continuous assignments using the `always` statement is that no control structures (like case, for or while) can be used, the advantage is that you don't have to specify the input list to trigger execution.

Inside procedural blocks, Verilog provides assignments, control structures such as `if–else`, `case` tables, `for` loops, `while` loops and `repeat` loops (fixed number of repetitions) and delays (for simulation only). Assignments are divided into blocking, non-blocking and continuous assignments (level triggered latches, released with deassign). Non-blocking assignments do not have an instantaneous effect, they are performed at the end of the simulation cycle, after all activated procedural blocks have been simulated (except if these assignments themselves trigger events).

Assignments provide logical and arithmetical operations (including addition, multiplication, division and shifts), bit-wise concatenation and function calls.

Verilog has two major synthesizible data types: wires and registers (single bit and multi-bit). There are integers and floats, too, to help verification.

2.1.2 Debugging/Verification

The main purpose of a simulation tool (the Verilog simulator) is to find bugs in the implementation and verify correctness. Both used Verilog simulators, Verilog-XL (Cadence) and Veriwell (Wellspring, shareware) support interactive debugging on the command line (after control-C-ing the simulator or invocation of the system function `$stop`) and program controlled debugging output (via system function `$display` and `$write`). Test vectors can be read using `$readmemh`. Cadence in addition has a waveform toolkit, cwaves. Waveforms are good to analyze basic data flow, but fail for arithmetic units, since in this case the bitwise equivalent of a value with the specification is of more importance. The waveform toolkit therefore hasn't been used.

Verification of modules without internal state and few input bits (e.g. the decoder parts) can be done exhaustive, thus a complete set of input vectors is generated in a verification module, and the output vectors are written to the log file using `$display`. The Unix commands `grep` (to select those output vectors that are of interest) and `diff` can now help to compare the output vectors with "correct" ones (e.g. generated using a computer language like C, perl, or Forth, or by the 4stack simulator), and to find faulty ones.

Logic with many input bits and internal state can't be tested exhaustive, so a mixture of random testing and edge conditions have to be found to verify these parts. Edge conditions concentrate on small parts of the logic and try to follow each path there.

Later test programs written in the 4stack assembly language were used to test more complex components. The `$display` sections were ranked into more and less important, using command line defined macros to skip them. The highest rank displays architectural status information in a format comparable to the 4stack simulator, so `diff` can be used to find differences in results.

2.2 Synthesis Tools

To close the gap between RTL level Verilog and gate level netlists synthesis tools are used. Netlists in a standard format (typically EDIF) can be sent to a ASIC manufacturer, who in turn produces gate arrays using this description. So the synthesis process finally leads to silicon hardware.

2.2.1 Gate Level Logic

Gates usually implement boolean functions or combinations of boolean functions using transistors as switches. Typically a library does not implement all 2:1 (two input, one output) boolean functions, but those that can be implemented with about the same amount of transistors. Furthermore, useful combinations of 2:1 boolean functions are provided, such as full-adders (3:2), edge and level triggered flip-flops, and multiplexers. 1:1 boolean functions are used as inverters and buffers. The latter are used to overcome signal strength restrictions. Last, but not least, three-state logic (with states 0, 1, and high impedance) is used to drive bidirectional busses.

2.2.2 Synthesis Tool Selection

Synthesis tools from two major synthesis companies have been used to synthesize parts of the design. Synopsis Design Analyzer does not support "`` `ifdef ``"s (conditional compile) and thus complained about the debugging output. It also was not able to synthesize reset states defined with `initial` and had problems with a mixture of blocking and non-blocking assignments in blocks. The preprocessing problems can be excused with the VHDL origin of the Design Analyzer, however, VHDL both supports mixing of blocking (signal) and non-blocking (wires) assignments, and reset initialization.

It has thus been decided to use Cadence's Synergy as synthesis tool, which does not have the problems above. Synergy provides better optimization support, e.g. it can collapse design hierarchy. To increase synthesis speed and to improve routing, higher level libraries can be created, supplying presynthesized user defined modules.

Synergy provides a number of different synthesis steps. The most important are:

- **Synthesizibility check:** check only for non-synthesizible constructs
- **Design optimization:** synthesize the design into a Verilog netlist. Optimization can go for speed or costs, optimization efforts can be specified by a number of different switches.

- **Tradeoff curve:** generate a curve of space-time tradeoff. This partly synthesizes the design with different optimization strategies (some more for space, some more for time). The tradeoff curve is especially important if both cost and timing limits are specified for a design and e.g. the smallest netlist for a given path length has to be found.

Design optimization also generates a list of paths, longest first. Since the library timing specifications are worst-case, this often provides more accurate estimation of timing problems than simulation of the synthesized netlists.

2.3 Synthesis Options

There is no "optimal" synthesis. Optimization can go for costs, for speed, or for minimal costs at required speed. Furthermore, synthesis takes lots of processor time and memory, especially for larger designs. Thus a synthesis tool must provide means to select efforts and synthesis time tradeoffs.

Synergy provides a number of different approaches to select costs and speed of the design and cost/speed of synthesis:

- **Synthesis effort:** synthesis effort ranges from 0 (lowest) to 4 (highest) and thus allows to select the time spent on optimizing netlist

- **Tradeoff:** cost or time optimization may be selected, either to minimal costs, to minimal time, or from a point on a tradeoff curve.

- **Module hierarchy:** modules, functions, and registers can be collapsed (highest expected optimization result), preserved in hierarchy (optimize modules independently, thus faster because repeated modules only get synthesized once, and netlists are smaller, so a close to optimal solution can be found easier) or not synthesized at all (to include synthesis from individual runs on this module, e.g. done before or concurrently on another machine/processor).

- **Standard modules:** Adders, multipliers, memory, and other basic multi-gates modules provided by the synthesis tools may be chosen. The broadest range of tuning optimization exists for adders, from ripple carry to carry lookahead tree adder.

- **Finite state machine generation:** finite state machines can be generated using "one hot" or "random" encoding.

- **Random access memory:** for memory blocks, the number of read and write ports can be specified. Memory blocks can be implemented decomposed (thus as building block from the higher level library) or behavioral, thus as individual gates arranged so that they fit the constraints (which is not optimal, though).

2.4 Routing

The final step of synthesis is routing. ASIC designs usually are routed by the ASIC manufacturer, since routing is very specific to the used cell geometry. Since routing affects space and timing conditions (through wire delays), this may be unsatisfying for the last edge of performance. However, a much greater edge of performance can be gained using a standard cell or a full custom implementation. In this case there are tools in Cadence's design framework II that allow routing and specification of cell geometries.

Chapter 3

Implementation of Functional Units

The 4stack processor is a pipelined processor. In this implementation, a basically three stage pipeline is used. The three pipeline stages thus are:

1. instruction fetch
2. instruction decode
3. execution

There are two main sources of stalls:

- instruction cache misses or mispredicted branches respectively skipped last entries of single instruction loops
- data cache misses or access conflicts.

Data cache misses stop the whole CPU except those parts that interface with external memory. Instruction cache misses are only noticed if they reach execution stage. They stall execution until valid instructions arrive.

3.1 Basic Components

Above the gate level, a number of basic components are typically used in hardware design:

- edge and level triggered multi-bit latches to form registers and permanent store of variables.
- multiplexers to select from a number of different input paths

- adders
- multipliers
- random access memory, thus latches combined into an array.
- barrel shifters

Verilog provides all these constructs on the behavioral level. For synthesis, however, a number of tradeoffs have to be taken into account. Synergy is able to synthesize all constructions above, however, with different results. There are a number of tuning switches, though, to select different approaches. I decided to implement adders and multipliers myself and to check whether these or Synergy's adders and multipliers are faster.

3.1.1 Adder

There are a number of different approaches to add two numbers using boolean logic (see for example [6]). A typical simple solution is ripple carry add, (see Figure 3.1).

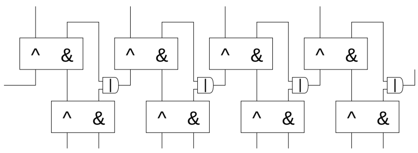

Figure 3.1: Ribble carry adder

Ripple carry add is a combination of full-adders (3:2 adders) with sequential carry propagation.

The classical enhancement is to form a carry lookahead tree network to overcome the restriction of the sequential carry propation. Basic principle is a divide and conquer strategy to analyze carry propagation. A carry is propagated, if the first half adder's sum result is 1. The carry propagation analysis thus is a AND tree. The real carry propagation is computed backward then. Each part outputs carry, if either both carry in and carry propagate is true, or the higher significant subpart outputs carry (see Figure 3.2).Carry lookahead adders are significantly faster than ripple carry adders, without much cost overhead. Additionally, they reduce the carry-in-carry-out path to about half of the longest path.

The other option is a recursive partitioned carry select adder. This adder also follows a recursive divide and conquer strategy. Each partition computes the sum with carry 0 and carry 1. Now based on the real carry propagation, the correct higher part of the sum is selected, using two n bit multiplexers (see Figure 3.3). This option depends on fast multiplexers, but even when using and, or, and inverter gates to emulate multiplexers, this approach is slightly faster than carry lookahead adding. It takes more space, though.

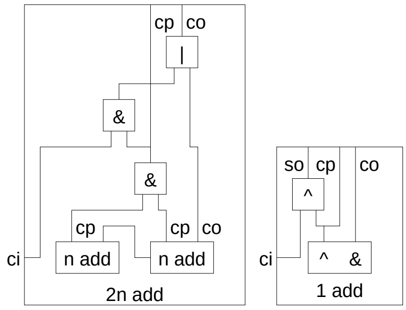

Figure 3.2: Carry lookahead

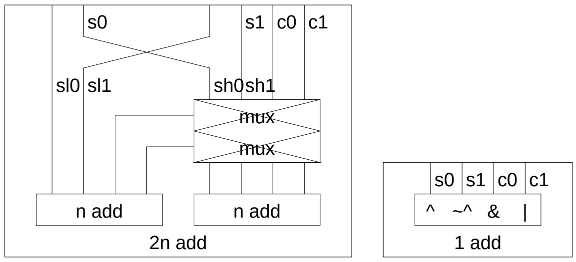

Figure 3.3: Carry select adder

The main advantage of the carry select adder is that other properties of the sum can be extracted while computing the sum. Two such properties are used:

- compute zero flag for ALU operations to get all flags at the same time
- compute number of leading zeros and ones for FP adder for following normalization.

In fact this approach can be used independently of the chosen adder implementation. This approach e.g. for zero flag computation is called "early out zero". Using this approach to implement addition itself seems not to be widely known. I did not even find a reference, [6] only shows non-recursive carry select adding (which takes less space, but is of $O(\sqrt{n})$ instead of $O(\log n)$).

To validate the assumption that a slow multiplexer doesn't affect the performance lead, a test synthesis using CellsLib, a library brought with Synergy, was performed under various optimization options (always optimizing for speed, not for costs). This library specifies the delay of a multiplexer almost twice as high as AND gate delay. The optimization target was a 64 bit adder with carry in/carry out. Note that the overhead for carry lookahead logic in this case almost vanishes compared to "ripple carry adder". The main reason is that CellsLib provides a 4 bit adder cell which is presumed to contain carry lookahead logic. The optimization with flattening hierarchy did not have a big effect on speed, but on size.

Synthesis Options	Longest Path	Area Units
Ripple Carry Adder	37.43 ns	160000.000
Carry Lookahead Adder Tree	13.95 ns	168622.578
Recursive Carry Select/Flatten Hierarchy	10.36 ns	303528.625
Recursive Carry Select/Preserve Hierarchy Tree	11.08 ns	365423.125

The same process has been done with a 1.0μ ASIC library (ES2)

Synthesis Options	Longest Path	Area Units
Ripple Carry Adder	26.58 ns	265620.000
Carry Lookahead Adder Tree	6.58 ns	641820.000
Recursive Carry Select/Flatten Hierarchy	5.06 ns	986485.938
Recursive Carry Select/Preserve Hierarchy Tree	7.83 ns	1892067.500

and a 0.7μ ASIC library (ES2)

Synthesis Options	Longest Path	Area Units
Ripple Carry Adder	23.93 ns	165668.266
Carry Lookahead Adder Tree	5.48 ns	299291.562
Recursive Carry Select/Flatten Hierarchy	3.85 ns	637828.500
Recursive Carry Select/Preserve Hierarchy Tree	6.24 ns	1168551.500

This case shows a larger speed improvement between ripple carry and carry lookahead adder (with area increasing more than two times), thus the assumption above obviously was valid. Flatten hierarchy this time had a huge effect on size and a noticeable effect on timing.

Conclusion: on process targets with comparable prohibitive multiplexer costs, a recursive carry select adder should be chosen only if the speed increase is worth it's price (e.g. for the FPU or the DSP unit). The costs of multiplexers thus have to be evaluated to back the decision whether to use RCS adders or CLA adders. Costs for additions without carry in are used by omitting the last large multiplexer (which also has fan-in problems), so this improves response time (to estimated 9.00 ns CellsLib/4.50 ns ES2), whereas CLA adders don't gain from omitting carry in.

3.1.2 Multiplier

As above, there are different approaches to implement a multiplier. The main approaches are:

- **Shift and add loop:** the multipliers are stored in two registers. There is a target register (of size $2n$). One source registers is a shift register, shifting to the right. Each step the other source register is added to the target register shifted to the left, if the lowest significant bit of the first source register is one, else the target register is shifted one bit to the left without adding.

Costs: the critical path is the adder. Compared with the edge-triggered latches, it is the significant part of the space costs, too. Timing is prohibitive; a n bit multiplication will take n cycles.

- **Shift and add loop using CSA:** the approach above can be greatly improved if a carry save adder (CSA) is used instead of a carry lookahead adder. The target register now stores both sum and carry bits from the CSA result. The basic loop is the same as above.

  Costs: the critical path is much shorter now (latch setup and carry save adder, a one bit full adder). However, execution is still $O(n)$, although n can be much shorter, e.g. if self-timed logic is used (which makes synthesis more difficult). The product (the two partial sums) has to be summed up using a carry lookahead adder or by turning the carry save adder network into a ripple carry adder (not recommended, because this takes another $2n$ cycles to finish the multiplication).

- **Multiplier array:** Instead of sequentially adding, an array of input bits is added together using cascaded carry save adders. Each array row is either 0 if the corresponding bit of the first input number is 0, or is the second input number. This array is reduced to two sums using carry save adder, and finally to the correct result using a 2:1 adder (carry lookahead or recursive carry select).

  Costs: a $n \times n$ bit multiplier uses n^2 AND gates for instantiation of the input array, and approximately n^2 full adders (carry save adder elements) to compute the result. Finally a 2:1 adder (size $2n$) is needed. The longest path is $O(\log n)$, though, so it's worth it's price.

All three multipliers can be converted into multiply and add units at almost no additional cost. Since the DSP units divide multiplication into two steps, the two partial sums form a natural pipeline point. Also, signed multiplication can be added at small costs. Unsigned multiplication can be converted into signed using the following equations (a and b are input values, hi is the more significant half of the result):

- if$(a < 0)$ $hi := hi - b$

- if$(b < 0)$ $hi := hi - a$

Since $a/2$ is already added to the result if b is negative (highest bit set), it is better to replace the topmost a instantiation by $-a$. Thus a 32×32 signed/unsigned multiplier instantiates a 32×33 array and uses additional XOR gates to compute the two topmost values. The 33rd array element is used for subtraction of b in signed multiplies.

3.2 Clock Generation

Pipelining introduces interlocks between different pipeline stages. Basically, there are two sources that produce stalls on the 4stack processor:

- instruction cache misses
- data cache misses

There are no interlocks between other instructions than those that access memory. Basically, data cache misses stall the whole CPU except the data cache itself, especially it has to stop fetching and decoding new instructions. Instruction cache misses only stall execution until the right instruction has been decoded.

To reach an even duty cycle, the input clock is divided by two. This also allows for generation of negated clocks which lag one half major cycle behind, to always trigger on positive clock edges[1] and to latch up in the second half of the cycle.

Stalls are indicated by two flags: cache miss (active low), and instruction valid (active high). If there is a cache miss at the rising major cycle clock occors, no other clock signal is activated. If there is no valid instruction, clock 2 does not raise. On falling major cycle clock, all positive edge clock values are copied to their second stage counterparts and set to zero. Figure 3.4 shows a sample clock flow (beginning at reset).

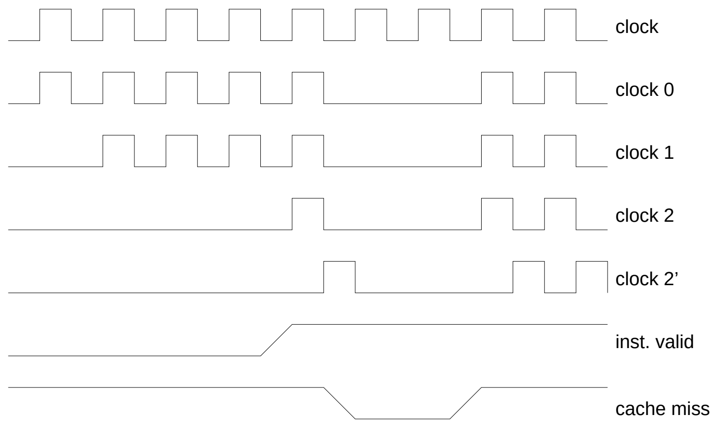

Figure 3.4: Clock flow

Some parts, especially the data cache, must be aware of the stall flags. Thus they set some latches (conditionally) on each raising or falling major cycle, but the conditions depend on the execution of instructions.

[1]Verilog simulation triggers the $x \to 0$ transition as negedge clock, so initially (when the clock goes to 0), all negedge blocks are executed

Chapter 4

First Stage: Instruction Fetch

The prefetcher supplies the CPU with instructions. As branches are resolved at the end of the execution stage, two following instructions have to be decoded at once, and up to four instructions have to be fetched (on a speculative basis). It is considered to be far too expensive to use a four ported cache to satisfy this need, so an alternative approach has to reduce the demanding cache ports.

First, branch patterns are not distributed equally; the typical case is either a branchless stream of instructions with rare branches (so no need for demanding cache accesses), or a single instruction loop. So a good heuristic is to follow branches, if there are any.

This approach could be improved using prefetch buffers (preferably cache line width), one supplying linear prefetch, the other prefetching at the branch target address. The Pentium™ processor uses this approach (see [7]). The main advantage of this approach is that a single ported instruction cache provides enough bandwidth to execute one branch per cycle, using branch prediction.

This implementation of the 4stack processor only buffers one additional instruction, because it reads aligned instruction pairs. Since there are two instructions in the decode queue, both buffers and ports have to be doubled. Larger prefetch buffers will help execution of a number of cascading branches, however, more branch target computing units will be needed, or a more intelligent approach when to compute branch target addresses has to be made.

The prefetcher thus fetches up to two pairs of instructions per cycle, based on a static selection of the up to four possible instruction locations. If the decision was wrong, no more than one stall cycle has to be inserted to finally fetch the remaining two candidates. In reality, this happens only on (rare) cascaded branches.

The prefetcher does not speculatively load on cache misses, it waits until the instruction really is executed. Since this delays cache fills only by one cycle, it is appropriate not to waste bandwidth and possibly trash the cache with speculative cache loads.

4.1 Details About Prefetching Order

The prefetcher needs to fetch up to four instruction destinations:

- linear two instructions ahead of the currently executed instruction
- the branch target of the next linear instruction to be executed
- the linear next instruction of the branch target of the current instruction
- the branch target of the branch target of the current instruction.

Only two of these four instructions are needed in the next step (decode). It is only clear at the end of the fetch instruction, which two this are.

Each of these addresses is stored into a 64 bit latch, called $ipll$, $iplb$, $ipbl$, and $ipbb$. "b" stands for executed branches, "l" stands for linear follower ("ipbb" thus is two consecutive branches executed, "iplb" is the branch target of the next instruction in linear order). Given that all instructions could be loaded successfully, there are two cases:

- if the executing branch instruction is taken, set $ipll := ipbl + 1$, $ipbl := ipbb + 1$
- if the executing instruction doesn't branch, set $ipll := ipll + 1$, $ipbl := iplb + 1$

The slots of $iplb$ and $ipbb$ are filled with the branch destinations of the two current instructions (if valid). Instruction pointers may only be incremented, if they did their job, thus the corresponding instruction was fetched. Also, if the instruction pointer is changed due to an ip! instruction or during exception handling, this new instruction pointer can only be used if the previous instruction was fetched successfully.

The selection of instruction pointers is done using cascaded multiplexer. These multiplexer also pass valid bits of the instructions. Instruction pointer latches are edge triggered with enable signal, so they take new values only when enable is set.

Because it is not possible to fetch four locations out of a dual ported cache, it is tried to save the lineare follower of one instruction if it is part of the cache request in two 128 bit latches (64 bit for instruction, 64 bit for branch target address) — each cache request outputs two instructions. Actually, this is only done for aligned instruction pairs.

The current priority scheme is to support first the branch target, then the linear follower. This could be changed to make at least use of the branch prediction hint stored in the instruction.

The opcode describes the following actions:

Position	Size	Purpose
0	1	set new IP at input address
1	1	force cache load at $ipll$
2	1	access mode (supervisor/user)
3	1	schedule exception (don't translate through MMU)
4	1	set loop start
5	1	set loop end
6	1	address size (for set loop start/loop end)

Exception handling slightly differs from normal prefetch executing: an exception stalls the normal instruction queue and fetches up to four instruction words from the interrupt table (not translated through MMU). If there is a branch in the exception code, normal instruction execution is resumed at this new instruction. Normal execution is resumed, too, if the four instructions have been exhausted without branching.

In fact, exception handling and resume is handled via "set new IP" and the last prefetched instructions are aborted.

The prefetch unit further handles hardware counted loops. This is done using the instruction pointer incrementer. If the incremented instruction pointer equals loop end address, and the loop counter is not zero, the loop start address is provided instead of the incremented address.

Single instruction loops pose one problem: the loop counter may be set in the loop start instruction. However, the first decision whether to loop or to continue has to be taken at the beginning of the decode stage of the loop start instruction. This problem is solved by always looping one instruction more than necessary for single instruction loops, and skipping the last instruction in the execution phase.

To follow branches, each instruction fetch has to compute the branch target address of all fetched instructions. The 4stack processor architecture eases this predecoding job by using one single bit to distinct between branching and other instruction. However, there are several (to be exact: four) different branch formats, two of them using relative addresses. The branch address computation is part of the instruction cache, so a more detailed description can be found in Section 7.2.

4.2 Instruction Pointer Incrementer

The instruction pointer incrementer consists of a 64 bit adder and a 64 bit compare logic. If the input instruction pointer is equal to loop end and the index is not zero, the new instruction pointer is loop start, otherwise the incremented instruction pointer. This selection is done using a 64 bit multiplexer.

Costs: a 64 bit adder, a 64 bit comparison unit, and a 64 bit multiplexer. The zero index flag is computed only once. There are four instruction pointer incrementer.

ISA changes: the original instruction set architecture described the loop end pointer to point to the first instruction after the loop. This would have required to move the comparator after the adder and thus lengthen the path (which is not really critical). However, explicit setting of loop end (with `loope!`) is supposed to use the value returned by `loope@`, so this architectural change is transparent.

4.3 Instruction Pointer and Output Selection

As stated above, the instruction pointers are updated using a bunch of multiplexers to select between different values, depending whether the currently executed instruction branches or not, and if the cache returned valid data ($valll$ to $valbb$). This section now shows detailed conditions.

4.3.1 Instruction Pointers

- $ipll$: If the current branch is taken, set it to $ipbl + 1$ or $newip$ (depending on $opcode[0]$) if $valbl[0]$ is true, else set it to $ipbl$. If the current branch is not taken, set it to $ipll + 1$ or $newip$ (depending on $opcode[0]$) if $valll[0]$ is true, else set it to $ipll$.

- $ipbl$: If the current branch is taken, set it to $ipbb + valbb[0]$, else $iplb + vallb[0]$

- $iplb$: Set it to $brip0$

- $ipbb$: Set it to $brip1$

4.3.2 Instruction Pointer Valid Tag

Since there are non-branching operations, not every instruction pointer is valid all the time. There exist four instruction pointer valid bits (vll to vbb).

- vll is always 1, since there is always a linear successor of the current instruction

- vbl is set to vbb if the current branch is taken, else to vlb.

- vlb is set to 1 if the currently fetched instruction (at $ipbl$ if branch is taken, else at $ipll$) is valid and a branching instruction.

- vbb is set to 1 if the currently fetched instruction (at $ipbb$ if branch is taken, else at $iplb$) is valid and a branching instruction.

4.3.3 Cache Access Address Selection

Since there are only two requests possible, two of the four instruction pointers have to be selected. Four single bit latches ($getll$ to $getbb$) hold the result of the selection decision, because further operation depends on it.

- $getll$ is true when the prefetch buffer for linear following address is empty ($nextllvalid = 0$)
- $getbl$ is true when vbl will be set to true and the prefetch buffer for branch target address is empty ($nextblvalid = 0$)
- $getlb$ is equal to vlb
- $getbb$ is equal to vbb

Since there are only two address ports, two out of the four addresses ($ip0$ and $ip1$) have to be selected using two 64 bit multiplexers.

- $ip0$ is $ipbb$ if $getbb$ is true, $ipll$ else. Access is valid, if $getll$ or $getbb$ is true.
- $ip1$ is $ipbl$ if $getbl$ is true, $iplb$ else. Access is valid, if $getbl$ or $getlb$ is true.

4.3.4 Opcode Result Selection

Not all four possible opcodes ($opll$ to $opbb$) could be fetched in one cycle. So opcodes and branch target addresses and the corresponding valid bits have to be selected (using 133 bit multiplexers) out of six locations: two output ports for the instruction cache with two instructions each and two prefetch buffers.

- $opll$ either comes form prefetch buffer 0 (if $nextllvalid = 1$) or from the first cache output port an is valid if $getbb = 0$
- $opbl$ either comes form prefetch buffer 1 (if $nextblvalid = 1$) or from the second cache output port, it's only valid if $getbl = 1$
- $oplb$ comes from the second cache output port and is valid if $getlb = 1 \wedge getbl = 0$
- $opbb$ comes form the first cache output port and is valid if $getbb = 1$.

Since both cache output ports output instruction pairs, another two 128 bit multiplexers have to select the correct instruction and branch target address selecting with the lowest significant instruction pointer bit (bit 60).

4.3.5 Branch Dependent Opcode Selection

Depending if the current instruction branches or not, the final two needed opcodes, their branch target addresses, and their valid bits are selected using two 133 bit multiplexers.

- $op0$ is $opbl$ if the branch was taken, $opll$ otherwise. The output instruction pointer is equal to next $ipll$.
- $op1$ is $opbb$ if the branch was taken, $oplb$ otherwise. The output instruction pointer is equal to next $ipbl$.

Furthermore, the input and the valid state for the prefetch buffers has to be selected using two 129 bit multiplexers. There is only one valid bit necessary, since prefetched instructions share all other properties with the original instruction.

- $nextllvalid$ is set, if the current $op0$ instruction wasn't the a loop and the instruction pointer wasn't set due to an ip! instruction. Further conditions depend on whether the current instruction branches: if so, $op1$ must be valid, $getbl = 1$, and the last bit of previous $ip1 = 0$. If not, $op0$ must be valid, $getbb = 0$, and the last bit of previous $ip0 = 0$.
- $nextblvalid$ is set, if the current $op1$ instruction wasn't the a loop and the instruction pointer wasn't set due to an ip! instruction. Further conditions depend on whether the current instruction branches: if so, $op0$ must be valid, $getbb = 1$, and the last bit of previous $ip0 = 0$. If not, $op1$ must be valid, $getbl = 0$, and the last bit of previous $ip1 = 0$.

Chapter 5

Second Stage: Decode

As there are up to four possible instructions at the beginning of the fetch stage, there are up to two remaining instructions at the beginning of decode stage. Therefore, there are two parallel decoders.

The basic instruction formats of the 4stack processor architecture can be found in Figure 5.1. Decoding is done in three basic steps:

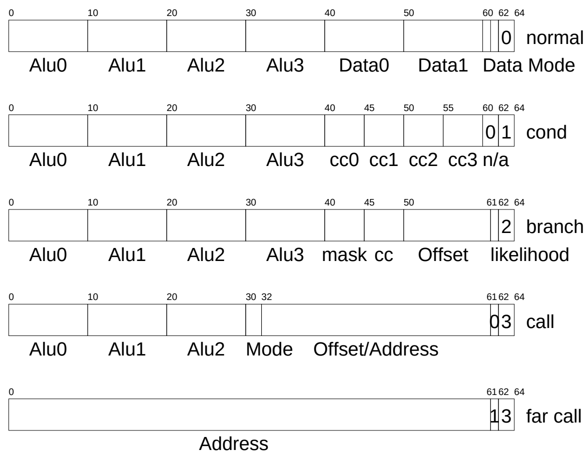

Figure 5.1: Instruction format

1. Fill in all those parts that are not covered by the actual instruction. E.g. branch instructions don't have a data operation field and a conditional setup field, while other instructions don't have a branch operation field. This is based on the last two bits of the instruction word using a 4:1 multiplexer.

2. Compute operation codes for the functional units and stack effects. Each operational unit is driven by an operation code, and for each stack, all the stack effects are computed.

3. Compute the actual stack operation code.

Decoding actually is data driven, so these three steps can be handled in one cycle. There are no more state information as the least two significant bits of the stack pointers and two stack effect queues for data loads (two bits per stack).

Basically, only the ALU and data operations need further decoding; the branch target address computation had been done in the instruction fetch unit and the code of conditional branches naturally fits the branch unit.

5.1 ALU Decode

The ALU operation decoder converts the 10 bit operaton format (see Figures 5.2, 5.3, and 5.4) into 6 bit ALU opcode (see Figure 6.8), 8 bit DSP unit opcode (see Figure 6.10),10 bit primar FPU opcode, 5 bit special register load/store opcode (see Figure), 10 bits of the primar stack opcode (numbers of pushs and pops), 3 bit pixel opcode, and one bit whether the flag is computed in the current operation or comes from deeper in the stack.

0	1	2 3 4 5	6 7 8 9
0	0	push 8 bit signed constant	
0	1	insert 8 bit constant	
1	group		extend

Figure 5.2: ALU instructions

or T1	add T1	addc T1	mul T1
and T1	sub T1	subc T1	umul T1
xor T1	subr T1	subcr T1	pass T1
T2	T3	T4	T5

Figure 5.3: Intruction groups

0	0	const <i>	
0	1	0	s<i>p
0	1	1	s<i+4>
1	stack		s<i>

0	0	pin s<i>	
0	1	0	s<i>p
0	1	1	s<i+4>
1	stack		s<i>

0	0	mul@
0	1	mul@+
1	compute flag	

0	0	shift
0	1	convert
1	0	<reg>@
1	1	<reg>!

0	floating point	
1	0	bit field
1	1	pixel

Figure 5.4: Group T1 to T5, based on extend bits

ALU operation decoding is done using a large case statement (as continuous assignment) which results in combinatorial logic and multiplexers after synthesis.

Costs: The primary stack operation is determined by a 6 entry case table (6:1 10 bit multiplexer). Each input computes the stack effects of one instruction group (and immediate number generation) using two or three bit comparators, OR, and AND gates. The ALU opcode is computed using a 15 entry case table (unoptimized a 15:1

6 bit multiplexer) and two 3 bit comparators. DSP opcode is computed using a five entry case table (5:1 8 bit multiplexer), FPU opcode a eight entry case table (8:1 10 bit multiplexer). The pixel opcode is computed using a 8 bit constant comparator. The zero flag origin bit uses an eight entry case table. Since the inputs of these case tables are not suited to drive a multiplexer, a number of comparators (and gates) have to be applied before.

5.2 Data Unit Decode

The data unit decoder decodes one 10 bit data operation code (see Figure 5.5) into a 37 bit data opcode (see Figure 6.14) using a 6 entry case table (mostly a 6:1 37 bit multiplexer). The stack effect of the various data operations is computed using an 18 entry case table (unoptimized a 18:1 7 bit multiplexer). Finally, 12 AND bits distribute stack effects to the correct stack half.

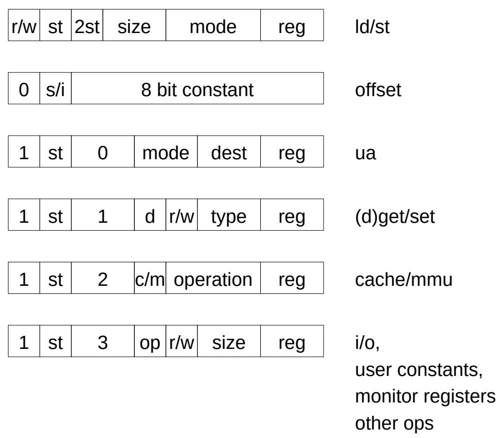

Figure 5.5: Data operation

5.3 Stack Decode

The stack operation turns the least two bits of top of stack position, the numbers of pushs and pops from ALU and data unit, and the stack extend opcode from other operations into a new top of stack position and multiplexer path selection opcode for stack register file, bypass unit and stack buffer (see Figures 6.4, 6.6, and 6.5). This unit does most of the work to turn physical registers and stack buffers into a real stack. See Section 6.1 for more details about how the stack works.

Stack decode is done in two steps. First, the stack relative positions are turned into absolute positions using two bit adders. The stack balance (thus the overall stack effect) and the TOS position at the end of the instruction are computed. Then, using these absolute positions and the number of individual pushs and pops of each unit are used to compute data paths.

The six absolute addresses into the stack are the final stack pointer, the destination stack pointer to write ALU values to, the stack pointer to get values to pass to the data unit (for storing into memory), and the four NOS accesses from own and other stacks.

The data paths then are computed using a stack position unit, since stack position numbers are encoded in four-bit vectors. One bit is set, if there is an access; if no bit is set, the output of this multiplexer is high impedance. So the stack position unit consists of four 3-input ands, comparing stack position number and ANDing it with the valid bit. Nine such units are used to

- compute TOS position (always)
- store output 0 position (always)
- pick from deeper in stack position
- the four NOS positions
- pin to elements 0-3
- pin to elements 4-7

The bypass opcode is computed using the stack position of the more significant ALU result and the number of pushs and pops, using four stack position generation units. Furthermore, it is computed which the two ALU outputs have to be stored to the stack, based on the ALU push and data unit pop numbers using a five entry case table (4:1 2 bit multiplexer, optimized some AND gates and inverters), and if NOS is bypassed (3 AND gates, two inverters).

The number and positions of stack entries that are rotated one element down is computed using four AND gates, one OR gate, and a four bit barrel rotator (an eight bit barrel shifter with two input bits, lower and higher half ANDed together).

5.4 Opcode and Instruction Pointer Tracking

The main decoder keeps track of opcodes and instruction pointers. At clock 1 rise, it saves both opcodes, instruction pointers and branch target pointers given from the prefetch unit in six 64 bit latches (when they are valid). At clock 2 rise, it saves the remaining instruction and branch target pointer in two 64 bit latches, using three 64 bit multiplexers (one selecting for exception handling just after branching).

If the prefetcher can't deliver the second instruction after a branch, the taken state falls back to normal state (since the instruction pointers in the prefetcher already are in their place as if the branch happened), and the current opcode0 is set to the previous valid opcode1.

The instruction pointer is needed for the `ip@` operation, and — when an exception occurs just after a taken branch — the branch target instruction pointer is used as new instruction pointer.

5.5 Exception Handling

Instruction pointer tracking also handles exceptions. Four categories of exceptions can be distinguished:

Number	Reason	Unit causing fault
0–F	General exception	Interrupts, instruction faults
0	Reset	Reset pin low, double fault
1	Trace	Trace bit set
2	Instruction Fault	Decoder
3	Privilege Violation	Decoder
4	FPU exception	Floating point unit
5	reserved	
6	instruction ATC Miss	Prefetcher
7	Instruction Memory Protection Fault	Prefetcher
8–F	Interrupts	Interrupt pin n

Number	Reason	Unit causing fault
10–1F	Address computation	Data unit
10	no exception	
11	Limit cross	"Odd" data unit
12	Misaligned Access	"Odd" data unit
13	reserved	
14	Limit cross	"Even" data unit
⋮		
1A	Misaligned Access	Both data units

Number	Reason	Unit causing fault
20–2F	Memory access	Data cache
20	no exception	
21	ATC miss	"Odd" access port
22	Memory Protection Violation	"Odd" access port
23	reserved	
24	ATC miss	"Even" access port
⋮		
2A	Misaligned Access	Both access ports

Number	Reason	Unit causing fault
30–3F	Stack ATC miss	Stack cache
31	Stack 3 ATC miss	
32	Stack 2 ATC miss	
⋮		
3F	All Stacks ATC miss	

The exceptions of the first group are indicated using a 16 bit vector, the other three groups are indicated using four bits each. A priority encoder generates an exception number out of these bit vectors. This exception number (shifted five bits to the left) is used as new instruction pointer. The prefetcher fetches this instruction without translating the address through the MMU. Instructions that are already in the decoding queue are aborted (thus not fed into decoding). Instead a “stack correction” instruction is generated to correct stack effects of aborted data loads[1].

Each time an exception occurs, a counter is set to four and decremented after each executed instruction until either a branch occurs or the counter is zero. Then exception handling resumes and other pending exceptions can be taken. Instruction fetching resumes at the interrupted instruction pointer.

[1]This hasn't been implemented yet.

Chapter 6

Third Stage: Execute

6.1 The Stack

Stack processors use a stack-like organization of the register file. This makes use of the locality of register accesses, thus only a small number of registers provide fast access time. Two special register names have an important role: the top of stack (TOS), which is source and destination of each operation, and the next of stack (NOS), which is second source of each operation.

The 4stack processor architecture allows to replace NOS with elements deeper in the stack (up to depth four) to access elder values, or with elements from other stacks (up to depth four), to access other recently computed values. This feature can be explained as parallel stack operation. Furthermore the 4stack processor architecture provides load and store concurrently to other computations on the stack.

The logical stack access flow in one cycle thus is (see also Figure 6.1):

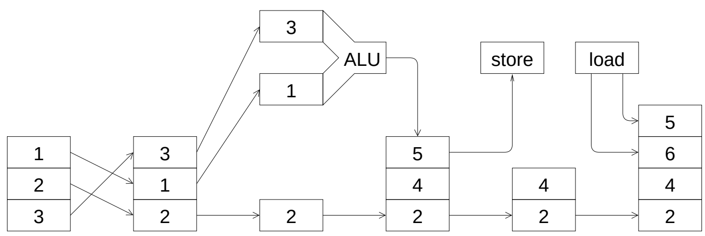

Figure 6.1: logical stack access flow

1. Compute stack operation, thus read value deeper in the stack or from other stack .

2. Rotate zero to three stack items deeper down the stack and push the value read in step one as new TOS.

3. Pop up to two values beginning with TOS, compute ALU, DSP or FP operation.

4. Push zero to two results back on the stack.

5. Pop up to two values beginning with TOS to store them to memory according to the data unit operation in the current instruction or pop TOS as part of a conditional branch or conditional setup operation.

6. Push up to two values loaded from memory according to the data unit operation of the last cycle.

Performing this sequentially it would take up to six cycles to perform this work. However, it can be shown that all these six actions can be collapsed in one single step.

All stack effects share one important property: they move TOS to NOS and replace TOS by a value deeper from the stack or from another stack. Instead of just pushing this value to the stack, it can be directly fed into the ALU, as input one (former TOS). TOS gives input two (former NOS). This reduces the amount of work to five steps, see Figure 6.2.

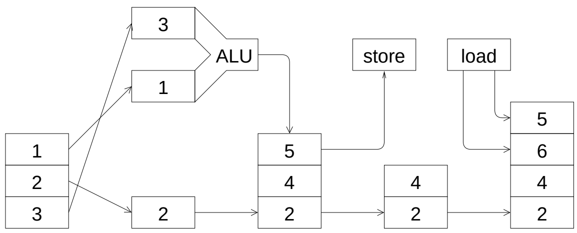

Figure 6.2: Merging stack operation and ALU input selection

Instead of pushing the result that is stored afterwards, a bypass can be used to feed the data unit. Furthermore the possible push operation from the ALU then can be merged with the data load pushes (see Figure 6.3).

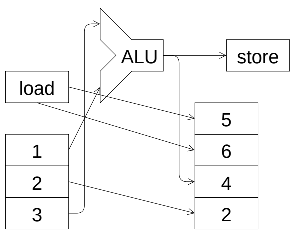

Figure 6.3: Store bypass, load and ALU push merged

This leads to the following requirements of the stack unit:

- deliver TOS to ALU
- rotate one to four elements

- provide up to two elements to store via data, either as bypass from ALU result or from stack
- read up to two elements from ALU
- read up to two elements from data unit.

As stacks spill to and fill from memory, this leads to a division of the stack unit into three parts:

- stack register file, providing the above requirements (fast access, high bandwidth)
- bypass, feeding storage unit with stack contents or ALU results
- stack fill/spill buffer to keep bandwith low on the cache side.

As four values can be pushed or popped in one cycle, the stack buffer has to supply or take up to four values per cycle. To reduce the number of ports to each buffer register, it is proposed to use four register rows, one for each of the four registers in the primary stack register files. These register rows are associated to one physical stack register. The actual mapping to the logical stack position is done by register "renaming". Instead of computing the translation of each logical stack position to the physical register, only the translation TOS to its physical representation (a two bit number) is computed and updated every cycle.

6.1.1 Stack Register File

The stack register file thus can do in one cycle:

- deliver TOS
- rotate one to four stack elements or store TOS to a deeper position to implement stack effects that change deeper positions of the stack
- output any four of four elements as second input (to other stacks, too)
- output up to two elements to data unit
- output up to four elements to stack buffer
- input up to four elements from stack buffer
- input up to two elements from ALU or other computing units
- input up to two elements from data unit.

Position	Size	Purpose
0	4	TOS position
4	4	Store output position
8	4	Valid input
12	4	Spill in values
16	4	Access deeper in stack
20	4	NOS own stack
24	4	NOS stack +1
28	4	NOS stack +2
32	4	NOS stack +3
36	4	Rotate down elements
40	4	Store TOS to deeper position

Figure 6.4: Stack register file opcode

The requirements thus are not only to hold the topmost four stack positions, but to provide a number of stack swapping operations, resulting in a network of multiplexers. The data paths are computed in the decode phase, see Figure 6.4.

The four 32 bit latches store the values in the order that is valid at the end of the cycle. Those parts of the stack that will be replaced by results form ALU or data unit contain the values to be spilled out.

Because the data unit takes either values from the stack (available at the beginning of a cycle) or from computation results (available at the end of a cycle), two additional 32 bit latches hold the value supplied from the stack.

Costs: the stack register file containts six 32 bit latches, one four bit latch to hold the input select opcode part, eight four to one multiplexers, four three to one multiplexers, and eight two to one multiplexers, each 32 bit wide.

6.1.2 Stack Bypass

The stack bypass passes up to two results from computation to the data unit and reorders both the computed results and the data load results to the physical order of the stack registers. The bypass opcode is described in Figure 6.5.

Costs: It contains a 18 bit latch to hold its opcode until the end of the cycle (where it is needed), two two to one multiplexers, and four four to one multiplexers, 32 bit each.

6.1.3 Stack Buffer

The stack buffer's main purpose is to reduce memory operations due to frequent stack pointer changes. It interfaces the fast and "narrow" interface to the stack register file with the wide and slower interface to the data cache. This assures that even in

Position	Size	Purpose
0	1	val0 to st0
1	1	val1 to st1
2	4	ld0 to in0-3
6	4	ld1 to in0-3
10	4	ld2 to in0-3
14	4	ld3 to in0-3

Figure 6.5: Stack bypass

the worst case (4 pushes or pops on each stack per cycle) there's only one data cache access per cycle.

This is done in two steps: the first stage consists of four rows holding four cells to be pushed or popped, used as a circular buffer. The second stage holds 2 16 byte lines, and is used as a hysteresis buffer[1]. This interfaces through a 16 byte bus to the data cache. There is a cache line size buffer which makes another level of decoupling. See Figure 6.6 for opcode parts.

Position	Size	Purpose
0	1	increment SP
1	1	decrement SP
2	4	select current/next buffer row
6	4	store TOS to deeper position

Figure 6.6: Stack buffer opcode

The stack pointer (SP) is incremented by 4 cells (16) in each incrementing or decrementing operation. The lower order stack pointer bits are replaced by the selection of current/next buffer row (four bits), see Figure 6.1.3.

The stack buffer's four rows are used as a ring buffer to interface between (slower) cache and (faster) stack register file. The current line (indexed by the lowest two bit of the stack pointer) and the next line take spilled values (in fact, they copy the whole contents of the stack register file, regardless if it's actually spilled or not). The next and the following line provide values to be filled into the stack register file (they are provided always, too). The last line fills or spills as whole to the hysteresis buffer, based on the stack pointer increment/decrement operation bits. If the hysteresis buffer is full and the stack pointer is decremented (thus the stack grows), one of the data cache's data buses is requested. This bus is shared between two stacks and their data unit. Until all requests are granted, the CPU stalls. The data cache's data bus is also requested when the hysteresis buffer is empty and a stack pointer increment

[1] A sort of bidirectional FIFO

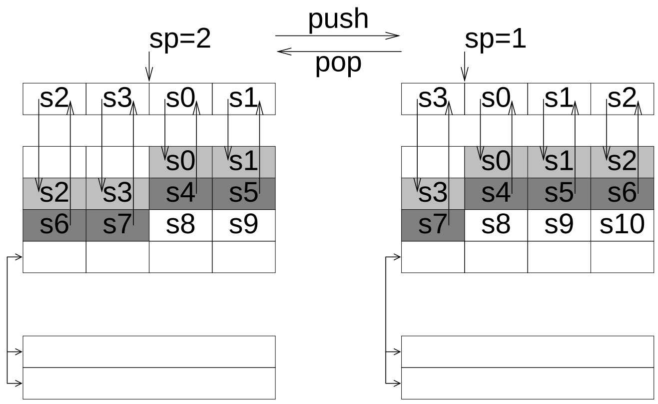

Figure 6.7: Stack buffer

occurs (the stack shrinks).

If a write request is granted, the last line is written out (on the data cache's data bus); on a granted read request the first line is written from the data cache's data bus. The data cache contains it's own stack pointer to keep track of stack spills and fills.

Tradeoff: the size of the hysteresis buffer is highly configurable. The bare minimum would be one element, buffering four stack cells[2]. In fact, the hysteresis buffer could be eliminated completely, at much higher costs, though. Some investigations, [11], lead to the conclusion, that each additional cached stack element halves the number of accesses to the next memory hierarchy. However, task switches scale linearly with the number of cached stack cells, so a good compromise has to be found. The actual implementation uses two buffer lines, because none of the sample programs reaches a stack depth greater than eight. The investigations cited above result in no measurable access of the next memory hierarchy for more than 16 cached values for a single data stack, so it is proposed that an overall buffer of 32 elements will cancel out any effects of non-uniform distribution of pushed values on the stack.

Costs: the stack buffer contains four 128 bit buffer rows, and two additional 128 bit hysteresis buffer rows. The hysteresis buffer is single ported, while each stack buffer row has three write ports for each element: one for the spilled out values, one for the TOS store up to depth eight in stack, and one to interface with the hysteresis buffer. In addition, there is a 128 bit tristate driver for the data cache's data bus, a number of one and two bit state registers, and the stack increment/decrement unit.

6.1.4 Stack Increment/Decrement Unit

The stack pointer basically either is incremented or decremented. Instead of providing an expensive and large carry lookahead adder (the stack pointer, as all other ad-

[2] More precisely, this buffers up to seven stack cells, as the three other buffer lines buffers up to three stack cells.

dresses, is 64 bit wide), a special two stage incrementer/decrementer could be used. Because this tradeoff decision has to be backed by measurements, both a stack pointer incrementer using a CLA adder and the ripple carry approach have been implemented.

The "cheap" approach uses a two bit wide ripple carry adder for the lower two bits of the stack pointer, and two registers for the upper part, which both can be updated by a slow ripple carry adder.

These two registers contain the actual and the either incremented or decremented higher 58 bits of the stack pointer. On increments, there are two possible transitions, depending which of the two registers is the actual stack pointer:

- If register 0 is the actual stack pointer, register 1 is the incremented stack pointer, so the actual selection bit changes from 0 to 1.
- If register 1 is the actual stack pointer, the ripple carry adder had at least four cycles time to compute the increment, so register 0 becomes register 1 and register 1 takes the computed increment. The actual selection bit stays 1.

The same thing, in reverse direction, holds for decrements.

Tradeoff: This part also can be implemented using a carry lookahead adder and a single 60 bit register. The advantage of this approach is that no special handling is required if the stack changes. Both approaches have been implemented and can be selected for synthesis.

Costs: one two bit latch and two 58 bit latches for lower and higher parts, 120 ANDs to compute the carry bits for decrements and increments, and 118 XOR gates (or jump and kill flipflops instead of normal edge triggered latches). There is one 58 bit multiplexer and one state bit.

6.1.5 Stack Component Glue

All three basic stack parts are combined into one module, which only contains wires to combine all these parts.

6.2 The Arithmetic Logic Unit

The 4stack ALU has the following capabilities:

- add and subtract with or without carry, $a + b$, $a - b$, or $b - a$
- logical operations: AND, OR, XOR
- shift/rotate left and right, signed and unsigned
- create flags

- convert chars and half words to words and vice versa
- immediate number generation
- population count, find first one
- bit field operations.

Thus there are eight parts, which all compute their specific operation, and a eight to one multiplexer selects which result to take. In fact, population count, find first one and the conversion units are one operation, and second operator as result is an additional ALU operation, used to implement stack effects. The ALU takes a 6 bit opcode (see Figure 6.8).

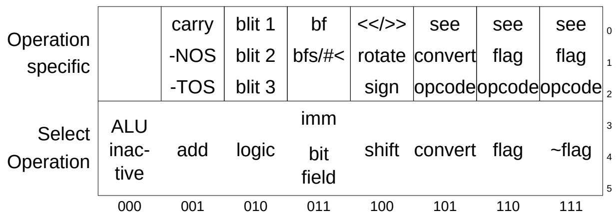

Figure 6.8: ALU operation code

Costs: the ALU consists of two 32 bit input latches to hold TOS and NOS values, a 6 bit latch to hold the ALU opcode, and a 32 bit 7:1 multiplexer to form the result.

6.2.1 Adder

In addition to adding two numbers and carry, the adder computes a zero flag. This requires an additional OR path for both results through the recursive carry select adder.

Costs: A 32 bit recursive carry select adder, 62 OR gates and 62 additional multiplexers for zero flag computation. A 34 bit multiplexer to finally select the result on carry. 64 XOR gates on the input negate either TOS or NOS. Three additional gates compute the actual value of carry.

6.2.2 Logic

Binary logic (OR, AND, and XOR) was implemented using a partly optimized bit operation ALU without inverted results ($0 \langle op \rangle 0$ is always 0). Thus only three bits of the bit block operation code are defined, the fourth is always zero. This logic also is used to pass NOS for pick operations. Generic logic works by selecting bits of the opcode using each input bit pair as selector. Thus per bit there are 3 2:1 multiplexers or the equivalent using AND, OR, and invert gates.

Costs: 96 single bit 2:1 multiplexers.

6.2.3 Shifts

ALU shifts either shift one bit to the left or one to the right. This is done using a 32 bit multiplexer. Some one bit multiplexer, AND, and inverter gates are used to select the left and right shift in bit (shift in carry, sign, or bit from other side to rotate).

Costs: A 33 bit multiplexer (value and carry) and eight other gates for shift in bits and overflow computation.

6.2.4 Converter

The ALU converts bytes and half words into words and vice versa, including byte and half word fractions (the byte or half word is the most significant part), signed and unsigned. This gives a total of six conversions (signed and unsigned conversion to byte or word fractions are equal). Additionally, find first one and population count are coded in the two unused conversions . A 33 bit 8:1 multiplexer (one additional zero indicator) is used.

Costs: one 33 bit 8:1 multiplexer, a 32 bit find first one unit, and a 32 bit population count unit. A total of 34 or/nor gates and three inverters compute zero flags.

6.2.5 Flag Computation

Flag computation uses carry and overflow state and TOS value to compute a flag. The zero flags from the conversion unit are used to determine if TOS is zero. The most significant bit of TOS is used as sign bit. The flag computation unit itself consists of a 8:1 multiplexer (one bit output, three bit control) and a few control gates to compute less than and less or equal.

6.2.6 Bit Field Unit

The bit field unit takes a bit field descriptor from TOS and the bit field value from NOS. Bit fields can both be (sign) extracted, created and cleared. Inserting values in bit fields is done using bit field creation on the inserted value and clearing on the bitfield, both parts are ORed together then.

The bit field descriptor consists of three byte-parts, describing the rotation count for NOS (least significant byte), the length of the mask (next byte), and the mask rotation count (next byte). Only the lower 5 significant bits are used. Mask length 0 is a special case, it indicates full mask length. This can be used for arbitrary rotations of NOS. Rotation is done using two barrel shifters ($32 : 5 \Longrightarrow 64$) and ORing the results together. Masking ANDs mask and result together. Finally, sign extension is obtained by replicating the highest bit of the output value (using the higher part of the mask) and the negated higher mask is ORed to the extraction result.

The bit field unit also contains immediate number generation. This either inserts 8 bits from the right into TOS or extends the 8 bit immediate number.

Costs: 3 32:5 barrel shifters (the one for mask generation can be simplified), 3 32 bit ORs, 3 32 bit ANDs, and one collapsing 32 bit OR tree to check for the highest valid bit. A 32 bit and a 24 bit multiplexer are used to select and generate immediate numbers.

6.3 DSP Unit

The digital signal processing unit (see Figure) is capable to multiply two 32 bit values to a 64 bit result (widening multiplication, signed or unsigned), to shift this result up to 64 bit to the left or right, to add another 64 bit number to this shifted product, and to round the upper 32 bit according to IEEE rounding conditions (to nearest even, to zero, to positive or negative infinity). The DSP unit is pipelined, thus it starts multiplication in one cycle, shifts and adds in the next cycle, while another multiplication can be started. There are only two DSP units, one for stack 0 and 1, one for stack 2 and 3.

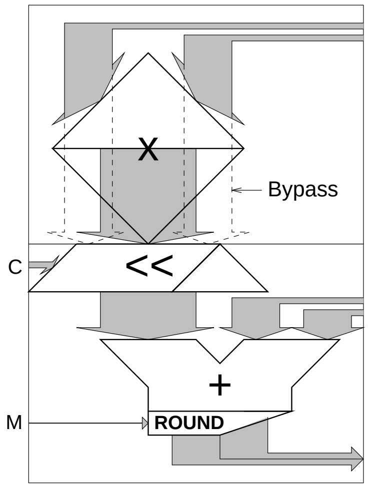

Figure 6.9: Digital signal processing unit

The DSP unit is subdivided in a multiplier array that gives an unfinished result (sum and carries), two barrel shifters that shift this result, a 128 bit carry save adder to add rounding offsets (e.g. $+0.5$), a 64 bit adder and a 64 bit carry logic (for the lower 64 bit of the shifted product) to compute the sum, and a multiplexer to finally select the correct rounded sum.

Four 64 bit latches are used to hold input multiplication values (two 32 bit numbers), intermediate multiplication results, and input sum value. These latches are set at positive clock edge if the corresponding conditions in the opcode (or previous opcode for intermediate values) are set. Multiplication input is set using a multiplexer, adding input is set using an alternative 64 bit multiplexer, ORing the two

Position	Purpose
0	multiply
1	signed/unsigned
2	bypass multiplier
3	add from result stack
4	negate multiplier result
5	round
6	shift multiplier result
7	result to low stack
8	result to high stack
9	multiply stack side

Figure 6.10: DSP unit opcode bits

possible inputs (even/odd stack) together, if they both are valid. The same is valid for shift count (8 bits) and rounding mode (3 bits). The ten opcode bits are saved, too.

6.3.1 Multiplier

Implementation details about the multiplier have been discussed in Section 3.1. In short, the multiplier takes two 32 bit input values, and a flag whether to multiply signed or unsigned. It returns two 64 bit values, sum and carries.

Costs: 1056 AND gates, 32 XOR gates, 32 inverters, and 1290 3:2 carry save adders.

Path length: 8 CSA elements, one AND, and one XOR gate.

6.3.2 Barrel Shifter

The barrel shifter can shift by up to 64 bit to the left or to the right, in the latter case both signed and unsigned. In fact, it shifts only to the left (by up to 128 bit positions), inverting the most significant bit of the shift count. The lower 64 bit are used as fraction parts and fed into the adder.

Costs: if implemented using a CMOS library, 8192 transistors and some buffers to work around fan-in and fan-out problems. If more routing resources than switching cells are available, it would be wise to divide the barrel shifter into two parts, greatly reducing the number of tristate buffers (transistors). There are two barrel shifter per DSP unit

6.3.3 Rounding Network

Results of addition are rounded using IEEE rounding modes (to nearest even, 0, positive and negative infinity). Since both rounding and addition of fraction parts may increment the result (thus by up to two), rounding offset has to be applied before addition using two carry save adders (one for each of the two possible conditions).

Rounding offsets are:

Rounding mode	round offset a	use offset a, when	round offset b
nearest even	$8000_0000	using b gives odd number	$7FFF_FFFF
zero	$0000_0000	using a gives positive number	$FFFF_FFFF
∞	$0000_0000	never	$FFFF_FFFF
$-\infty$	$0000_0000	always	$FFFF_FFFF

Since there are two sizes, rounding is either applied as 96 bit number (result will be rounded to 32 bit) or 64 bit number (result will be rounded to 64 bit). The lower bits always are extended from the lowest bit of the rounding offset above. Indeed, for each rounding offset only three boolean values have to be computed: bit 32 (for 32 bit result rounding), bits 33-63 (all the same) and bit 64.

The final result is selected from the two adders using a 64 bit multiplexer.

6.3.4 Adders

As mentioned above, there are two 3:1 adders, adding 128 bit inputs, giving a 64 bit output (more significant half), thus two 128 bit carry save adders, two 64 bit carry detection logic (a "crippled" version of the recursive carry select adder, which just selects carries and does not add), and two 64 bit recursive carry select adders.

The ability to negate the multiplier's output is implemented as follows: Instead of conditionally negating the two 64 bit multiplication results, the adder input sum is inverted conditionally (using 64 XOR gates). This affects the rounding inputs, too, so the the six boolean values have to be inverted, too. The final result then is inverted conditionally again. This makes use of the equation: $a - b = \overline{\overline{a} + b}$, valid in two's complement arithmetic. Since the DSP unit has to compute a zero flag, both zero (in the adding case) and $= -1$ (in the subtracting case) flag has to be computed while adding.

Costs: two 128 bit carry save adders, two 64 bit carry computation units, two 64 bit adders with zero and -1 detection for the higher 32 bits, and two 64 bit multiplexers.

6.4 Floating Point Unit

The floating point unit is capable to do the following things:

- multiply two input values
- add two input values (float and integers)
- forward the multiplier's and adder's result into the adder's input

- convert single to and from double floats (up to four times)
- scale by or extract exponent (up to four times)

The following tables describe opcode format:

Position	Size	Purpose
0	4	in0-3 is mullatch0/1 value
4	4	in0-3 is addlatch0/1 value
8	2	mulout is addlatch0/1 value
10	2	addout is addlatch0/1 value
12	2	negate mullatch0/1
14	2	negate addlatch0/1
16	2	convert addlatch0/1 from integer
18	3	out0 selection
21	3	out1 selection
24	3	out2 selection
27	3	out3 selection

Output selection is as follows:

Bits	Output is
000	high impedance
001	old input latch
010	adder output
011	multiplier output
100	single to double float
101	double to single float
110	exponent extraction
111	exponent scaling

The floating point unit handles input, output and functional subunit selection. Among containing converters and scaling/exponent instruction units, there is a multiplication unit and an addition unit. It contains eight 64 bit latches for adder and multiplier inputs (and saved input from previous operations) and four 12 bit adders for exponent exctraction and scaling. Four 64 bit 7:1 multiplexers select output for each stack.

6.4.1 Floating Point Multiplier

The floating point multiplier consists of two parts: — the first part is a 53x53 multiplier array (using a total number 3322 three (bits) to two carry save adders) — the second part is a recursive adder, includes rounding.

While the multiplier array is computing the partial product, the exponent computation is done, thus compute the new exponent, check for result 0, ∞ or not a number (NaN), and so on. The last step is a multiplexer that selects the right output (either one of the special conditions 0, infinity or NaN, or one of the two outputs, if carry is set or not).In detail, the following parts are used:

Input categories	Output category
$n * (n, 0, \infty, NaN)$	$(n, 0, \infty, NaN)$
$0 * (n, 0)$	0
$0 * (\infty, Nan)$	(∞, NaN)
$\infty * (n, \infty)$	∞
$\infty * (0, NaN)$	$(0, NaN)$
$NaN * (n, 0, \infty, NaN)$	NaN

Figure 6.11: Multiply category transition table

1. *A 53x53 bit multiplication array.*

2. *Rounding network:*

   Unlike the DSP unit, results are always positive (sign is part of the exponent), so carry in is sufficient to do the final increment by one in the round to nearest even case. Rounding to $\pm\infty$ depends on the sign of the result. The rounding vector is added to the two intermediate results using two 106 bit carry save adders, one for rounding if bit 0 of the result is 1, the other if bit 0 of the result is 0.

3. *Special number handling:*

   Floating point numbers fall into four categories: numbers, zero, infinity and not a number (NaN). This coding is obtained examining exponent and the first bit of mantissa. Fortunately, the used code allows to generate the output category with two simple AND gates. Figure 6.11 shows the category transition table more detailed.

4. *Exponent computation adder:*

   A 12 bit adder adds the two input exponents. Since floating point exponents are biased, first a 12 bit 3:2 carry save adder subtracts the bias.

5. *Final adder:*

   Two 106 bit adders (52 bit full addition, 54 bit carry computation) add the two outputs from the multiplier network together. This is done in the second cycle, the two multiplier network outputs are latched in two 106 bit latches.

6. *Output selection:*

   A 63 bit multiplexer selects the correct normalized output. Another 64 bit multiplexer selects normal path or special category depending on the category code of the result.

6.4.2 Floating Point Adder

The floating point adder has different input and output paths, depending on what is needed to compute: it can take two stack item pairs as input, and replace each input with either multiplier bypass or previous adder result. In fact, all these inputs go into one quite large shift and adder pair.

There are a number of cases which have to be considered when adding:

- is it a subtraction or an addition
- are the exponents equal or not
- if they are and it is a subtraction, is the result negative or not.

This affects rounding conditions, among other things.

So this is the basic algorithm:

- Compare signs — if equal, it is an addition, else it is a subtraction. Compare exponents — if equal, go straight on and add, if not, you have to shift the smaller input right by the exponent's difference. If this difference is > 52, you can completely omit the addition.

  The postprocessing depends on these conditions, too. If it was an addition, this is simple: just check for carry, and increment the (larger) exponent, if carry was set. The rounding mask can just be appended to the larger input value, things get rounded so without problem.

- If it was a subtraction, there are two cases: equal and different exponents (thus one input was shifted to the right). If one input was shifted, it is straight forward: just check the most significant bit, and shift to the left (decrementing the exponent), and go on. If both exponents were the same, this is less difficult — the result may have become negative (then invert it) and there are an unpredictable number of leading zeros. These leading zeros must be counted, and the result must be shifted to the left (decrementing the exponent appropriately). This may even be the case if the smaller input is shifted to the right by one.

  Yet, this is even more complicated with the forwarded inputs from multiplier and adder. The multiplier's output isn't finished, just the exponent is known with an accuracy $-0/1$, so the subtraction with leading zeros (no initial shifting) can occur even with a exponent difference of two.

- The worst problem is the forwarded adder. It isn't finished, too, and especially if a subtraction occurred, the exponent isn't computed, only the exponent correction is known (either the single bit from normal addition or the number of leading zeros). So the exponent correction is fed into the exponent difference computation, and the mantissa is normalized, before it is fed again into the adder.

Figure 6.12 shows the data flow in the floating point adder.

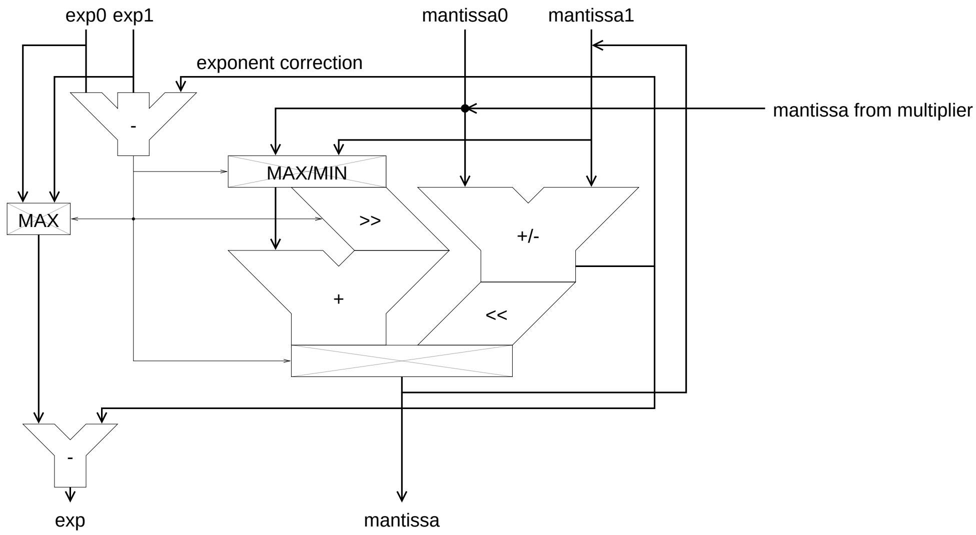

Figure 6.12: Floating point adder

Operation on input categories	Resulting category
$n \pm (n, 0)$	n
0 ± 0	± 0
$\infty \pm (n, 0)$	∞
$\infty + \infty$	∞
$\infty - \infty$	NaN
$NaN \pm any$	NaN

Figure 6.13: Category transition table for addition

In details, the following parts are used:

1. *Exponent differences:*

   - Two 16 bit 3:1 multiplexers select input operands
   - Three 16 bit wide AND gates select exponent corrections (either 0 or from counting leading zeros)

- A 16 bit 3:1 adder (CSA and carry lookahead or recursive carry select adder) computes the exponent difference. A 12 bit adder has been found to be insufficient. 13 bit would be enough.
- Two 16 bit adders compute exponent correction for both exponents
- Three cascading 11 bit multiplexers select exponent differences, depending on signs of the differences and a final correction by one bit (the mantissa of the bypassed input operands are known now)

2. *Input selection multiplexer:*

   - Two 54 bit multiplexers conditionally swap input operands depending on which was larger.

3. *Shift lower part left:*

   - A 53 bit shifter shifts the larger input conditionally one bit to the left
   - A 128 bit barrel shifter shifts the other input up to 64 bit to the right

4. *Special number handling:*

   - Numbers like 0, ∞ or NaN (not a number) have to be treated separately. A category code thus is computed for both inputs. The transition table found in Figure 6.13 computes the category of the result. Only category n is computed using the adder path. The categories of the input numbers are computed either using bypassed category codes or by examining the exponent and the first mantissa bit according to IEEE floating point. Exponent all one is either infinity (first mantissa bit 0) or NaN (first mantissa bit 1). A zero exponent is zero or denormal. This floating point unit doesn't handle denormals, so they are assumed to be zero, too.

5. *Addition featuring find first one/ zero of unshifted data:*

   - A 64 bit find first one/zero adder adds the (almost) unshifted inputs. "Almost" means, that they may be shifted to the left, if their origin is a bypass path and the most significant bit was zero. The find first one/zero adder basically is a recursive carry select adder which additionally computes the number of leading zeros/ones based on the values returned from both the higher and the lower part.

6. *Pipeline latch setup for second pipeline step, latches setup time on positive edge of clock 2:*

   - The results of barrel shifting are stored into a 54 bit and a 128 bit latch. The second is conditionally inverted, if the operation is a subtraction.

- The (uncorrected) exponent is selected using a 11 bit multiplexer and saved in a 11 bit latch
- Signs, exponent correction bits, the operation type, and estimation, which path will be used are stored in five single bit latches
- The results of the addition applying find first one/zero are stored in two 64 bit latches (for mantissa, result and negated result) and two7 bit latches (find first one/zero results).

7. *Rounding input:*

   - A four entry case table computes three bits which are used to form the rounding input. Unlike in the DSP unit adder, all rounding occurs on positive values and there are only two input values (not three), so rounding may only increment results only by one (compared with rounding to zero). In this case the carry logic can be used to add the additional one for round to nearest even. Rounding to $\pm\infty$ depends on the sign of the sum. For the path where rounding is needed, this sign is known at this stage.

8. *Carry save adders for rounding:*

   - Two 128 bit 3:2 carry save adders compute intermediate sums for both final cases (carry set or not) of the left adder path (one argument initialy shifted).

9. *Recursive carry select adders for rounded sum:*

   - Two 64 bit recursive carry select adders compute the sum of the two 128 bit latches that hold the shifted input path, aided by two 64 bit carry generators. Depending on the last valid bit and the current rounding mode (round to nearest even), carry in is chosen. Carry in chooses the correct carry out bit of the carry generators, and this is used to select the proper sum (thus is carry in for the recursive carry select adders). This case shows the importance of the short path for carry in-carry out given by the recursive carry select adder.

10. *Generate output:*

    - The amount of positions to shift the right path (non-shifted input) is selected using the carry out of the corresponding adder (carry out signals that the result has to be negated) to drive a 7 bit multiplexer.
    - The use of the right path is backed, if both the initial assumption is true, and the leftmost bits of either sums (normal or negated) on this path are equal.

- The exponent correction is either the shift amount or $\{0, 1, 2\}$ based on initial exponent corrections (necessary for bypass paths).
- The exponent is corrected using a 16 bit recursive carry select adder. This exponent correction can be corrected by another offset of 1 by computing carry in later.
- A 64 bit multiplexer and a 64 bit barrel shifter (shifts by up to 64 bits to the left) normalize the result on the right path
- A 64 bit multiplexer cascade finally selects the correct path, depending on operation type (subtract or add), left or right path (if it was a subtraction) and normalizes the results for the right path (shift by at most one bit to the left). This multiplexer cascade is duplicated to provide the bypass back to the adder input. This second path doesn't need normalizing.
- A final 64 bit 2:1 multiplexer selects output based on the category of the output value
- Based on the category codes and the output sign, the result flag is computed. Overflow and Carry are set if the result is NaN or ∞ respectively.

6.5 Data Unit

The data units provide the interface between stacks and data cache. There is one unit for stack 0 and 2, the other unit for stack 1 and 3. Each data unit has four register sets for normal execution, and another four register sets for interrupt and exception execution. A register set can be used (together with IP and TOS of one of the stacks) to form an address.

The data unit is composed of register file, managing logic, data ALU and data multiplexer.

6.5.1 Data Register File

The data register file hold 8 register sets (R, N, M, F), four for normal execution, four for interrupt processing. It is capable to

- compute one memory address per cycle, and eventually update one register
- load or store up to 64/128 bits per cycle, using one recently computed address
- pop/push up to two values on up to two stacks
- generate 8 bit immediate constant per cycle
- check for boundary crossing

Positon	Size	Purpose
0	1	32/64 bit mode
1	1	mode: 0 normal, 1 interrupt
2	2	register number
4	2	target register number
6	2	register type (R, N, L, F)
8	3	add0 mode (+offset+immediate, subtract) (to reg)
11	3	add1 mode (+offset+immediate, subtract) (to cache)
14	2	stack side, both stacks
16	2	operand size (byte, half word, word, double word)
18	1	reg mode: reg/(offset/ip)
19	1	offset mode; s0/next
20	1	do register load/store
21	1	read/write
22	1	store to register
23	1	0: new immediate, 1: insert lowest 8 bits into immediate
24	8	immediate number
34	3	delayed operation type (see Figure 6.15)

Figure 6.14: Data unit opcodes

The opcode bits are described in Figures 6.14 and 6.15.

The data unit consists of three eight entry 64 bit memories for R, N, and L registers, and one eight entry 16 bit memory for F registers. These memories' read ports are addressed using the register and mode field of the opcode. The write port is addressed using the target register and mode field of the opcode.

A number of multiplexers compute the input values for the data ALU, base and offset. Base can either be the R register contents, the current instruction pointer, or the TOS value from the selected stack side (a 64 bit 3:1 multiplexer thus is needed). Offset can either be the N register contents or the TOS value from the selected stack side (a 64 bit 2:1 and a 32 bit 2:1 multiplexer).

Two 64 bit 2:1 multiplexer select the source for register set operations (32 or 64 bit mode, stack side).

The real access size is computed by adding the "both stack side" flag to the "size" opcode field. The scaling operator is either the operation size or 0, depending on F register bit 15. The subtraction opcode field for the data ALU has to be one, if both F register bit 11 and the store opcode bit are one (stack style access).

Since data access occurs a cycle later, a number of results have to be stored in latches at rising clock 2 edge. The lowest four bits of the address are used to select the correct value through the data multiplexer. Access size, stack side and endianess are required, too. A two bit FIFO is used to keep track if either the data cache's result

Value	Purpose
0	no operation
1	load
2	cache check
3	MMU check
4	no operation
5	store
6	cache alloc
7	MMU change

Figure 6.15: Delayed operations

or values from register get instructions have to be pushed on the stack.

Register updates (computed by the data ALU or from stack) are stored to their destinations on rising edge of clock 2, too.

A 128 bit 2:1 multiplexer is used to select between output from cache and from the register file. Another 64 bit 4:1 multiplexer selects the register type to push to stack. The store mask computation unit is used to compute the valid bits for the store operation.

6.5.2 Data ALU

The data ALU is used to compute addresses. Addresses are formed by adding a register value, an offset value (either from the N part of the register set, or from one of the stacks) and an immediate value. Therefore a three-to-one adder (a carry save adder followed by a carry lookahead adder) has been used to do the calculation. The data ALU also is responsible for register updates. In load and store instructions, register update just add register and offset, in address update (ua) instructions, the immediate value is added, too. So a second three-to-one adder is needed for register update computation.

Address computation includes bound checks. Thus the result of an address computation is compared with the limit (L part of the register), and if it is greater or equal, it was a boundary crossing. Note that "0" represents the value "∞" for limits. The reaction for boundary crossings are up to the main data unit. Both address computation and register update independently have to be bound checked.

For FFT, the data ALU provides reverse-carry addition. Since there are published self-sorting in-place FFTs (see for example [16]), and unpublished self-sorting, in-place and unit stride radix 2 FFTs (according to a posting in comp.arch [19]), it can be argued that this part is superfluous. It therefore has been made a configurable option(via preprocessor defines). Bit reversing does not take many gates (just two multiplexer in and one out), but these multiplexers take a lot of area (and there are a total of six per data ALU). A second, mirrored adder could be used instead, which only helps routing, if it interleaves with the forward carry adder.

It is therefore proposed to synthesize with reverse carry add option, to mark this architectural feature as "obsolete", and to give a sample implementation of a self sorting radix 2 FFT in the user manual instead, to help users that are not aware of this option. Later, the carry reverse option can be omitted completely, if the self sorting FFT shows up to be as fast (or faster, due to unit stride access) as the bit reverse addressing FFT.

The data ALU takes the following opcode:

Position	Purpose
0	use offset for register update
1	use immediate value (for r.a.)
2	negate offset (for r.a.)
3	use offset for address calculation
4	use immediate value (for a.c.)
5	negate offset (for a.c.)
6	reverse carry addition (obsolete)

Costs: Four times 64 AND gates to select usage of offset and immediate values, two times 64 XOR gates to invert offsets, two 64 bit carry save adders, two 64 bit adders and two 64 bit greater-equal comparators, 63 OR gates to check if limit is zero ("infinity"). 6 64 bit bit-reversing multiplexers or another two 64 bit adders, if the bit reverse add option was enabled.

6.5.3 Data Multiplexer

The data unit loads and stores bytes, half words, words and double words (single and dual, little and big endian) over the same 128 bit bus. Therefore a multiplexer must convert raw memory data into the stack format and vice versa. Furthermore, conversion between little and big endian has to be done.

It turns out that all these things can be done using two fancy 64 bit to 64 bit multiplexers. Figure 6.16 shows how this multiplexer words. Each number stands for one byte, 0 is the leftmost (lowest address), 7 the rightmost (highest address). The multiplexer is divided into eight parts, each selecting one byte.This multiplexer can shift a number of any power of two size to any aligned position (bold face), big and little endian (italic). This is exactly what is needed, so that any aligned memory data can be transformed into a 64 bit word going to the stack, and any 64 bit word (or part extractions) can be shifted into the propper memory bus position.

This approach gives true bi-endian support, too. Unlike using the XOR trick[3], patented by MIPS/SGI and used in other bi-endian CPUs like DEC Alpha and Pow-

[3]This trick simply reverts the address bits below word addressing (not below alignment). It is cheap and instantly allows to make both big and little endian systems, but makes mixed-endian systems (those that switch at run-time and between different tasks) inherently difficult.

address	0	1	2	3	4	5	6	7
0	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1	*1*	***0***	3	2	5	4	7	6
2	2	3	**0**	**1**	6	7	4	5
3	*3*	*2*	*1*	***0***	7	6	5	4
4	4	5	6	7	**0**	**1**	**2**	**3**
5	5	4	7	6	*1*	***0***	3	2
6	6	7	4	5	2	3	**0**	**1**
7	*7*	*6*	*5*	*4*	*3*	*2*	*1*	***0***

Figure 6.16: Data multiplexer

erPC, byte storing order is not changed when switching into another endianess. So byte oriented binary datas, typical format for data exchange, can be shared between big and little endian application. This is of especial importance if programs of different byte order share the same address space and run under the same operating system.

Another interesting property is that both read and write conversions can use the same address computation logic (computation of th 7 bit address) because of the symmetry on aligned addresses. The lowest valid addressing bits have to be inverted (not including those bits below the alignment restriction). For little endian accesses, the bits below alignment have to be set, for big endian accesses, they have to be cleared. Even strange endianesses could be supported, too, like the PDP11 floating point endianess (big endian two byte words, little endian words, see address 2 and 6).

Finally, the unneeded parts have to be masked out. This is a different job for load and store. Load just masks out the higher bits. The following table contains 1 for the bytes passed, 0 for the bytes masked out:

Byte	0	0	0	0	0	0	0	1
Half Word	0	0	0	0	0	0	1	1
Word	0	0	0	0	1	1	1	1
Double	1	1	1	1	1	1	1	1

Store does not need to mask unused parts out, since the valid bits computed by the store mask unit do this job. However, for double stores, two values have to be merged using the store merge unit. Since double stores go to aligned value pairs, the two masks are each other's complement. The bits in the first mask can be computed using all 16 two input, one output boolean functions on the size argument, or this table:

Byte	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
Half Word	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
Word	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0
Double	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0

Costs: there are two entry level 64 bit multiplexers to select the correct 64 bit words out of the 128 bit data bus, and one 128 bit multiplexer to select read or write operation. 128 tristate drivers are needed to drive data bus output on writes. Two of the fancy 64 to 64 bit multiplexers are needed (basically eight 64 to 8 multiplexers, using three input bits). Gates and space donated for address and mask computation can be almost neglected. 128 AND gates for masking and 16 8 bit multiplexers from the store merge unit are needed, too.

Caveat: this multiplexer can't be used for non-aligned accesses. A different variant using a barrel rotator (rotating multiples of eight bits) could do this; however, the bi-endian feature would be lost then.

6.5.4 Store Mask

The store mask computing unit computes the 16 valid bits using the four least significant addressing bits, operation size, stack side and both stack flag. It makes use of the alignment restrictions. The store mask for both stacks (low and high stack half) are computed individually and ORed together. The individual store masks are computed generating the following mask parts, which are finally ANDed together:

For each access size, a repetition of the pattern 10 (address "even" in terms of alignment units) or 01 (address "odd") is generated, where 1 and 0 are repeated $size$ times. The overall pattern is repeated to fill all 16 bits of the mask. If the access size is greater than $size$, the mask is set all one. In other words: a byte access mask is true only if a half word, word, and double word mask using the same address is true, too. A word address however, is true where word address and all subword accesses using this address and offsets below the alignment conditions, are true, too.

Chapter 7

Memory Interface

7.1 Bus Interface

The 4stack processor can keep a high memory bandwidth busy. A fast and wide bus therefore is recommended. However, there is a limit of pin numbers, so a compromise has to be found. For the actual ASIC implementation, a multiplexed 64 bit bus with byte enable is proposed. The tradeoff of multiplexing is small, when bursting is allowed. As one cache line is 64 bytes large, each burst reads 8 words, so the overhead is neglectable. The following control wires are required:

- 64 data lines
- 8 byte enable lines
- bus request (active low)
- bus grant (active low)
- read (active low)
- write (active low)
- acknowledge (from memory, active low)

Read and write line and acknowledge are used as handshake lines. The bus direction is handled through the protocol. The first transaction is always in memory direction, the address. Bus snooping is possible using two wires for read and write. If the bus is free, the memory controller has to keep read, write and first data high. If read and first data is low, too, a read address bus cycle is issued — the standard case for snooping. To signal snoop hit answer, write is set to low by the snooping processor. Then it has to supply data as in a normal memory write cycle. This both transports data from one CPU to the other and writes it into memory. The transition shared to modified in the cache is signalled by a zero data write cycle (thus after the first acknowledge signal comes, the bus is released).

Currently bus snooping is not supported. See Figure 7.1 for a sample read burst access, and Figure 7.2 for a write burst access.

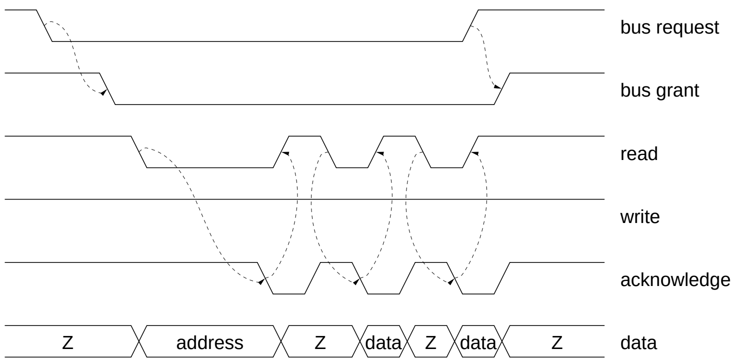

Figure 7.1: Read burst access

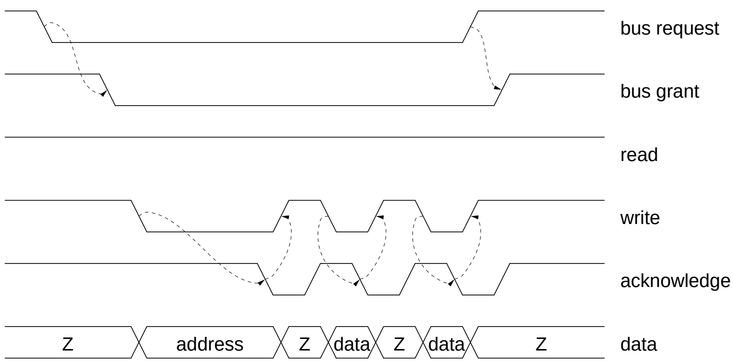

Figure 7.2: Write burst access

7.2 Instruction Cache

As has been explained in section 4, the instruction cache has two 128 bit ports, providing up to four instructions per cycle. For simplification, a direct mapped cache is used. The cache is physically indexed. Each cache line is 64 bytes, eight instructions wide.

The instruction cache's opcode has the following fields:

Position	Size	Purpose
0	2	read from address 0/1
2	1	fill cache from memory using address 0
3	1	access mode (supervisor=1, user=0)
4	1	translate instruction through MMU (active low)

The instruction cache returns opcode, branch target address and a five bit access valid code. The valid codes say the reason why an access was invalid (1 signals true, 0 false):

Position	Purpose
0	access valid
1	cache miss
2	ATC miss
3	protection fault
4	access mode

The access mode is equal to the initial access mode, except if the page was a "gate page" (see Section 7.4), then it reflects the new access mode. The instruction cache module also contains the instruction address translation cache (ATC). Address translation is done in parallel to cache access, and tag comparing is done in parallel with branch target address computation (see Figure 7.3).The lowest address bits are used

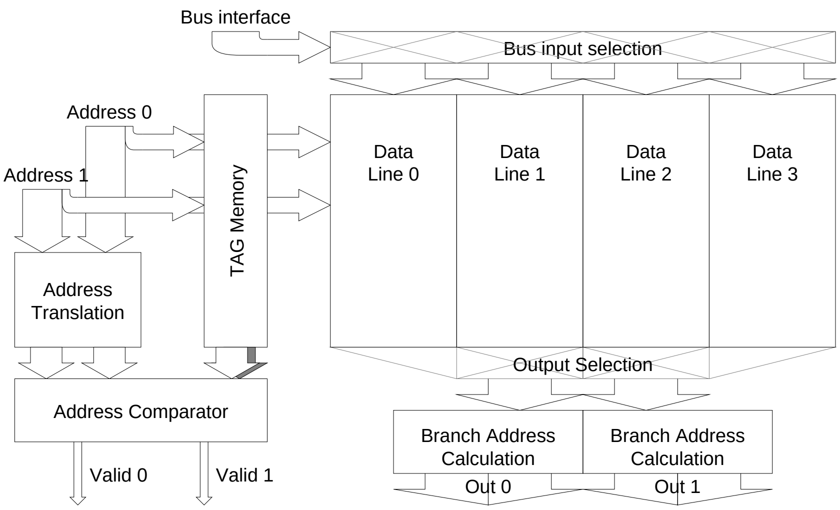

Figure 7.3: Instruction cache access flow

to select the appropriate cache line, the higher address bit associatively access the address translation buffer (see below). The results form address translation are compared with the address tags, the protection bits are checked, and together with the cache valid bits the valid bit vectors are formed.

Concurrently the four instructions fetched are predecoded. Bit 62 indicates if the instruction is branching. In this case, the branch target instruction is computed.

On cache misses, as indicated by opcode bit 2, the cache goes into read state. It issues a bus request and waits until bus grant is received. Then it sets 'r' and transmits the address according to the bus protocol (see section 7.1). A three bit counter is used to keep track until the bus transaction is finished. A one-bit state flag distinguishes between address transmission and data transmission. Because the cache is partitioned in 128 bit subblocks, a 64 bit buffer is used to hold the first, third, fifth and seventh data transaction.

The cache itself is non-blocking. While accessing memory, requests can continue. In fact, no request will be made until the requested instruction pair is stored into the cache. Then execution can continue. If they cause another cache miss, the current memory access is finished before starting the next request.

Costs: Cache costs are highly configurable, since the cache size isn't fixed in the design. The current design allows cache sizes from page size down to one cache line (general case: 2^n cache lines) without major modifications. Each cache line consists of 512 static RAM cells, and, since it is dual ported, a second access gate per bit (at least a second CMOS transistor) for data, $58 - n$ SRAM cells and additional access gates for tag memory (2^n cache lines, so 58 bits is the tag size for a one-line cache) and four SRAM cells plus access gates for the four valid bits, one for each cache line partition. A 4K cache as used in simulation thus uses about 36K dual ported SRAM cells (dual output port only).

7.2.1 Branch Address Calculation

Since branch address calculation is in the critical path (second step of cache access), it is important to make it fast. The instruction set therefore is optimized for quick decoding of branches. The last three instruction bits indicate everything that is needed for branches.

Bit 63 distinguishs between short (conditional) and long branches.

Bit 62 distinguishs between branching and non-branching operations.

Bit 61 distinguishs between far (cross-4GB segment) and local (inter-4GB segment) branches.

Bit 30 distinguishs between relative and absolute local branches.

Relative branches require an adder. The two different cases for adder input (short and long) can be done with bit 63, using a multiplexer for the most significant 18 bits (either sign extended from bit 50 or copied from bits 32 to 49 from the opcode). The lower half of the branch target address is either the adder output or the lower half of the opcode. The higher half of the branch target address is either the instruction address or the higher half of the opcode (far branches). This requires two additional 32 bit multiplexers.

The three least significant bits are used as status of the branch instruction. The first two bits are interpreted as hint, the last bit states wether the instruction branches really (it is cleared for "do" instructions). Currently, hints are not used, except, that a hint 0 on "do" instruction would misinterpreted as a normal instruction. As this hint is never generated, it doesn't hurt.

Costs: one 32 bit (fast) adder to form the branch target address ($ip+offset+!isdo$), one 18 bit and two 32 bit multiplexer, and a total number of about 13 or 14 gates for decoding and control.

Alternatives: It can be argued that the adder isn't necessary. Long relative branches can be converted (at cache miss time) to absolute branches in the cache, as well as short relative branches (with the addition of one cache state bit, which selects current or next/previous 8K "page", depending wether the instruction is in the first or the last half of the current page; one of the currently unused hint bits could be used for this purpose). The problem with this approach is that it makes position independent shared libraries hard to do (would require a cache flush each task switch).

7.3 Data Cache

Basically, the data cache seems to have about the same characteristics as the instruction cache: two 128 bit ports, thus dual ported, and page size direct mapped in this implementation. However, the data cache has more unit to serve: the two data units and the four fill/spill buffers for the stacks. Each of these units can be active, and all they either read or write data. It would be a waste of resources if 6 read-write access ports were implemented, since cache fills and spills are rare, and the dominant data access usually is load, not store. Figure 7.4 shows the main difference between instruction cache and data cache.

The implementation uses a cache with two read ports and one write port, with the same costs as the instruction cache. However, the access conflicts have to be resolved. This was by ways the most difficult part of this feasibility implementation.

Another problem is that data accesses are not as regular as instruction accesses. So a direct mapped cache hurts much more. Since a two-way cache would have introduced delays (the demultiplexer could not be used concurrently to tag address comparison) a eight element victim buffer was implemented. This reduces the worst case to the equivalent of an eight way set associative cache (with one line per set), and introduces one cycle delay per access. It also reduces the amount of time for cache writebacks, because they do not block further cache accesses.

Since each stack itself has a very predictable access pattern, a special stack cache interface has been designed, too. Each cache has one cache line to fill in or spill out. This reduces access conflicts, since there is no guarantee, that stack pointers point into the same part of different pages (in fact, after reset, this is exactly the case).

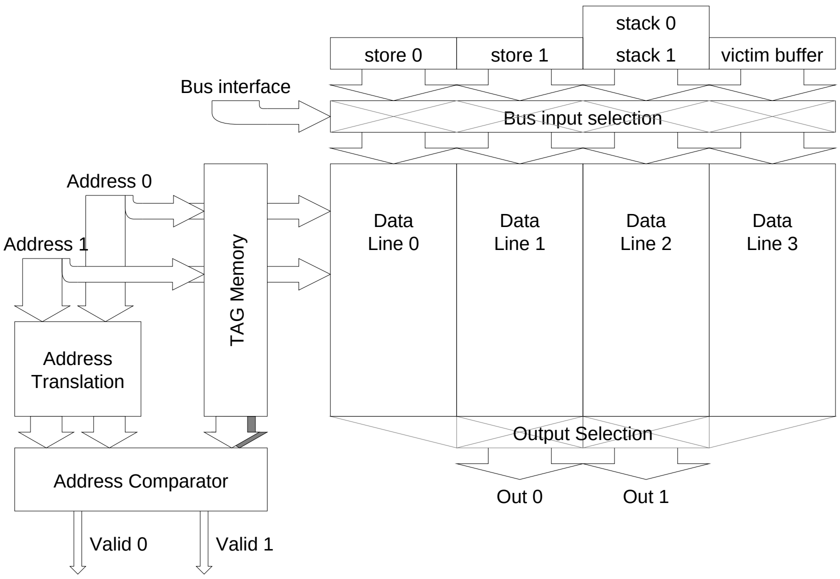

Figure 7.4: Data cache

Furthermore, write accesses have to be serialized. There are eight different sources for cache writes: the two data units, the four stacks, memory transaction and victim buffer restoring.

Since the stacks and data units share two busses (for "even" and "odd" data unit respectively stacks), these busses have to be shared, too. Stacks have higher priority (and among stacks, lower numbers have higher priority). As a side effect this reduces the amount of write conflicts to be resolved.

The data cache can't use the stack pointer register, since this points to the top of stack, using virtual addresses. The data cache however needs the physical address at the bottom of the stack buffer, so it keeps track of this address. This also keeps stack pointer translation off the data address translation unit. In fact, stack address translation is completely done in software.

7.3.1 Stack Cache

The stack cache is an interface between the "narrow" 128 bit port and the wide 512 bit cache line size. It thus reduces the amount of reads and writes to the cache due to stack changes. The stack cache holds a like of 512 bits per stack, organized as four 128 bit latches. It also keeps track of the stack changes and thus has a second stack pointer (which now is a physical address, pointing to the bottom of the stack buffer).

The stack pointer is updated on transaction requests — incremented on reads, decremented on writes. Only the lesser significant eight bits really are incremented or decremented using a cheap ripple carry adder, the higher part is a page address, thus it has be handled in software. Two pages, the current and the one above or below are kept in eight 52 bit latches (two per stack). One bit holds the state (below/above). If the stack pointer crosses both pages, the entries are swapped, and the below/above state is inverted. One fourth of a page before an invalid swap (thus swap a page below as above) is done, an exception is raised. This is the case if the two topmost counter bits and the state bit are all 1 or all 0. If the counter crosses a modulo 4 boundary, a cache request must be issued (either read or write, depends on the direction).

Since there is only one bus for two stacks, accesses have to be serialized. This is done by priorizing lower stacks (thus stack 0 can transfer before stack 2, stack 1 before stack 3). To handle this, a priority encoder forms the four bus grant bits and a clock generator forms another four clock signals using the generation clock. Each of this clock is true if clock 2 would raise at the same time (and no collision is detected), or, if a collision was detected, in later cycles, when the other access is handled.

Push transactions are stored to the corresponding stack buffers, pop transactions are read from there. Exhausted buffers, as said above, lead to a cache access. If this cache access is finished, the whole cache line is stored in the stack buffer. Full buffers lead to cache accesses, too. Since two buffers may overflow at once (due to two push transactions), two 512 bit multiplexers select the output buffers to be stored in the cache.

Conflicts (thus two overflows at once) are resolved by writing one cache line to the cache, the other to the victim buffer. Another possible strategy could be to block the second push transaction for one cycle.

Costs: The stack buffer consists mostly of 2048 dual ported SRAM cells and two 512 bit multiplexers. Eight 52 bit latches hold stack page addresses, and four eight bit counters keep in-page stack addresses.

7.3.2 Victim Buffer

The victim buffer is an essential part of the data cache to drastically increase performance at some access patterns that can't be handled by direct mapped caches. It is organized as an eight cache line deep FIFO. Two input lines, each 512 bit wide, and one output line (another 512 bit) connect it to data and stack cache. Each input and output line is accompanied by a 60 bit address line and two state bits (the MESI bits, for the four states modified, exclusive, shared and invalid).

A 128 bit 4:1 multiplexer converts cache lines to smaller pieces for the bus interface.

The victim buffer is full associative, thus there are sixteen address comparators (60 bit each) to compare the two current addresses with the tags in the FIFO. Cache misses that hits the victim buffer thus can be detected early, so only one wait state has to be inserted. The victim buffer returns only one cache line to the cache per cycle, since the cache has only one write port.

Two three bit counters are used to keep track of the low and high water mark of the FIFO. The victim buffer tries to get rid of the entries at the low water mark, if the distance between low and high water mark is 6 or more. Then new write requests would overflow the buffer. Dirty lines at the low water mark lead to bus requests and further bus cycles (while blocking the CPU), all other states are just changed to "invalid", since the data is valid in memory. To prevent blocking of the CPU, dirty lines try to get the bus (at low priority) once the distance between low and high water mark is 2. Then the bus cycles can be done concurrently to further execution without stalls.

Costs: The 4096 dual SRAM cells for the cache lines, and another 512 dual ported SRAM cells for address tag and MESI state bits. There is a 512 bit latch for lines returned to the cache.

7.3.3 Store Merge Unit

Concurrent stores that go to different cache parts can be handled concurrently. However, some important applications access the same cache part at the same address. In this case, the conflict is resolved using the store merge unit. Two accesses that go to the same address are merged together using 16 eight bit multiplexer. The valid bits of both accesses are ORed together, thus forming one access. Furthermore, the store merge unit buffers store data until it can be handled by the data cache. If no further operation accesses the cache, conflicting operations thus can be serialized without stalling the CPU.

Costs: one 60 bit address comparator, two 128 bit latches to hold store data, two 16 bit latches to hold valid bits, 16 eight bit multiplexers, and one 16 bit multiplexer to select valid flags. A few gates compute the number of cache write accesses finally needed.

7.3.4 Store Conflict Handling Details

Two conflict sources groups are handled differently. The first group are all those accesses that are not related to current accesses or store whole cache lines, thus victim buffer hits, stack cache spills and writes form the bus interface (to fill the rest of the cache line after the initial request has been satisfied). The second group are accesses that result either from data stores or from that one of the first group that was successful.

Using priority encoders, the following priority is established:

- Writes from the bus interface precede stack cache spills and victim buffer hits. Since there is at most one write every second cycle, the delay only lasts one cycle
- Stack cache spills precede victim buffer hits.

All these operations buffer their data in one 512 bit latch, thus there is a 512 bit 3:1 multiplexer to select this data. 58 bit address tag (and cache line address) are stored in another latch. Valid bits (both which parts of the cache line are already valid and which parts have to be updated) are stored in two four bit latches. Two bits MESI state and a signal to request the final store must be kept, too.

Four priority encoders detect conflicts for each 128 bit wide cache part. The priority order is:

- stores from the tree groups above have highest priority,
- stores form the "even" data unit precede finally
- stores from the "odd" data unit.

Since victim buffer hits forward their data to the store merge unit, in this case the precedence order is reverted. The store form the victim buffer on this cache part is omitted, if a store operation goes to the same cache line and the same cache part.

7.3.5 Access Address Selection

Stack accesses is stated above, precede data accesses. The cache has two read ports, thus two read addresses are hold in two 60 bit latches. Each rising basic clock edge it is checked if a stack access or a new[1] or pending data access is requested. The "pending" bit is set if both a stack access and a new data access are requested at the same time, and reset if the data access finally is accepted.

7.3.6 Victim Buffer Input

Each write access may overwrite a valid cache line. Thus a write is preceded with a read to that cache line. If the write would replace this line (thus is a new write from the bus interface, a write from the stack cache, or a victim buffer hit) and the line is valid, it is a valid input for the victim buffer. A 512 bit 2:1 multiplexer selects the cache line (there is only one at a time) that is target of the next write.

Furthermore, data stores may be cancelled, if the line they store to is moved to the victim buffer. Therefore another eight 128 bit 2:1 multiplexers select either cache output or data from the store merge unit.

7.3.7 Outlook

The data cache was one of the most complex parts and very difficult to debug. It is presumed that not all conflict conditions have been found and detected. Larger separate stack caches would be a big win in safety, however, not in performance, since except artificial cache testing programs, many real applications don't stress the cache

[1]Thus clock 2 is raising, and the cache opcode isn't a nop.

in this way. In other words: many of the bugs found in the cache have not been produced with the benchmarks and applications written before, but with special cache test programs.

7.4 Memory Management Units

The MMUs (both for instruction and data accesses) consist mainly of a dual ported ATC. Each ATC entry contains one virtual address and four physical addresses, thus four pages are covered within one ATC entry.

On updating the ATC, there are two cases:

1. The virtual address tag already matches, but there is no translated physical address. Update is easy: Just store it into the appropriate physical address slot. Check with m[cd]get if the update was successful.

2. The virtual address tag doesn't match. The OS software notices this because m[cd]get returns 0. Then it has to replace one of the previous occupied slots, and store the virtual tag into it using m[cd]set with least significant bit set, and from bit 11 down the position in the ATC buffer. Bit 1 is reserved for setting larger pages.

The lower part of a physical page address consists of two parts: access rights and page properties. Access rights are read, write, execute (rwx) for supervisor and user (thus 6 bits). Properties consist of 3 bits for the system (cache coherence protocol, setmode, "ccs") and 3 bits free for application purposes, which could be typically used for access stamps (read, modified, executed), but these bits are not processed by the MMU.

The format thus is in short

	ac	sys	usr
ccs	rwx	rwx	rwx

The setmode bit changes the execution mode when executing an instruction in this page, and the processor is in a mode not allowed to execute instructions.

The cache coherence protocol defines four different types of pages: totally uncached, consistently updated ("write through"), coherent cached and stale cached (not using the cache coherent protocol for shared to exclusive transitions).

Pages can be considered as empty, if none of the rwxrwx bits for supervisor and user are set. However, it is up to the ATC fault handler to convert (and validate!) page table entries.

The MMUs have only partly been implemented, since correct MMU support relies on exception handling, a weak point of this feasibility study. Especially the logic to abort load and store instructions in process is missing. Therefore a MMU supplement can be used for synthesis, which "translates" all addresses one by one and returns an “all valid, full cacheable” access right descriptor. This has the further effect, that the distinction between supervisor and user state is (almost) useless.

Chapter 8

Conclusion and Outlook

This implementation study shows that the 4stack processor architecture does not pose too hard problems when implemented for an ASIC process. The Verilog code developed as part of this work is synthesizible and partly has been synthesized with expected results. A final synthesis run will be made and the resulting netlist will be given to an ASIC manufacturer. This piece of “first silicon” then can be used as demonstration object to attract commercial partners for further development. Other high end processors required hundreds of man-years, so it is estimated that a lot of work is still to be done.

Implementation in hardware is only part of the success of a processor. A compiler backend for popular languages (C, Fortran) has to be written. The 4stack processor architecture claims to be general purpose, but this requires to be able to run a modern operating system like Unix. To test first silicon, an evaluation board has to be developed.

To attract interrest, a paper has been submitted to the Micro 29 conference for publication.

Bibliography

[1] G. Böckle. Exploitation of fine-grain parallelism. In *Lecture Notes in Computer Science*, volume 942. Springer-Verlag, 1995.

[2] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman. A VLIW architecture for a trace scheduling compiler. *IEEE Transactions on Computers*, C–37(8):697–979, Aug. 1988.

[3] D. Ditzel, H. McLellan, and A. Berenbaum. The hardware architecture of the crisp microprocessor. *The 14th Annual Int. Symp. on Computer Architecture; Conf. Proc.*, pages 309–319, 2-5 June 1987, Pittsburgh.

[4] K. Ebcioğlu. Some design ideas for a vliw architecture for sequential natured software. In *Proceedings of IFIP WG 10.3 Working Conference on Parallel Processing*, pages 3–21, Pisa, 1988. Elsevier Science Publishers B. V.

[5] Harris Semiconductor. *HS-RTX2010RH Data Sheet*. Harris Corporation, 1996.

[6] J. L. Hennessy and D. A. Patterson. *Computer Architecture—A Quantitative Approach*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1990.

[7] Intel. *Pentium™ Processor User's Manual*. Intel Corporation, Santa Clara, CA, USA, 1993.

[8] W. Karl. *Parallele Prozessorarchitekturen – Codegenerierung für superskalare, superpipelined und VLIW-Architekturen*, volume 93 of *Reihe Informatik*. BI-Wissenschaftsverlag, Mannheim, 1993.

[9] W. Karl. Some design aspects for vliw architectures exploiting fine–grained parallelism. In A. Bode, M. Reeve, and G. Wolf, editors, *Proceedings PARLE '93, Parallel Architectures and Languages Europe*, number 694 in Lecture Notes in Computes Science, pages 582–599. Springer-Verlag, Berlin, 1993.

[10] L. Kohn and N. Margulis. Introducing the intel i860 64-bit microprocessor. *IEEE Micro*, 9(4):15–30, Aug. 1989.

[11] P. Koopman. *Stack Computer—the New Wave.* Ellis Horwood, New York, NY, USA, 1989.
URL `http://www.cs.cmu.edu/~koopman/stack_computers/index.html`.

[12] J. Labrousse and G. A. Slavenburg. A 50 MHz microprocessor with a very long instruction word. In *IEEE International Solid–State Circuit Conference*, 1990.

[13] T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification.* Addison-Wesley, Reading, MA, USA, Sept. 1996.
URL `http://java.sun.com/docs/language_vm_specification.html`.

[14] B. Paysan. *A Four Stack Processor.* Fortgeschrittenenpraktikum, Technische Universität München, 1994.
URL `http://www.informatik.tu-muenchen.de/~paysan/4stack.ps.gz`.

[15] C. Peterson, J. Sutton, and P. Wiley. iWarp: a 100 MOPS, liw microprocessor for multicomputers. *IEEE Micro*, 11(3), June 1991.

[16] Z. Qian, C. Lu, M. An, and R. Tolimieri. Self-sorting in-place FFT algorithm with minimum working space. *IEEE Trans. on Signal Processing*, 42(10):2835–2836, Oct. 1994.

[17] B. Rau and J. Fisher. Instruction-level parallel processing: History, overview and perspective. *The Journal of Supercomputing*, 7(1), 1993.

[18] B. R. Rau, R. P. L. Yen, W. Yen, and R. A. Towle. The cydra 5 departmental supercomputer: design philosophies, decisions and trade-offs. *IEEE Computer*, 22(1), 1989.

[19] D. A. Schwarz. Re: In-place in-order FFT. Usenet posting to `comp.arch,comp.arch.arithmetic,comp.dsp`, 1996.
URL `news:schw-3107960949020001@news.hrl.hac.com`.

[20] G. A. Slavenburg, A. S. Huang, and Y. C. Lee. The LIFE family of high performance single chip VLIWs. In *HOT CHIPS Symposium*, 1991.