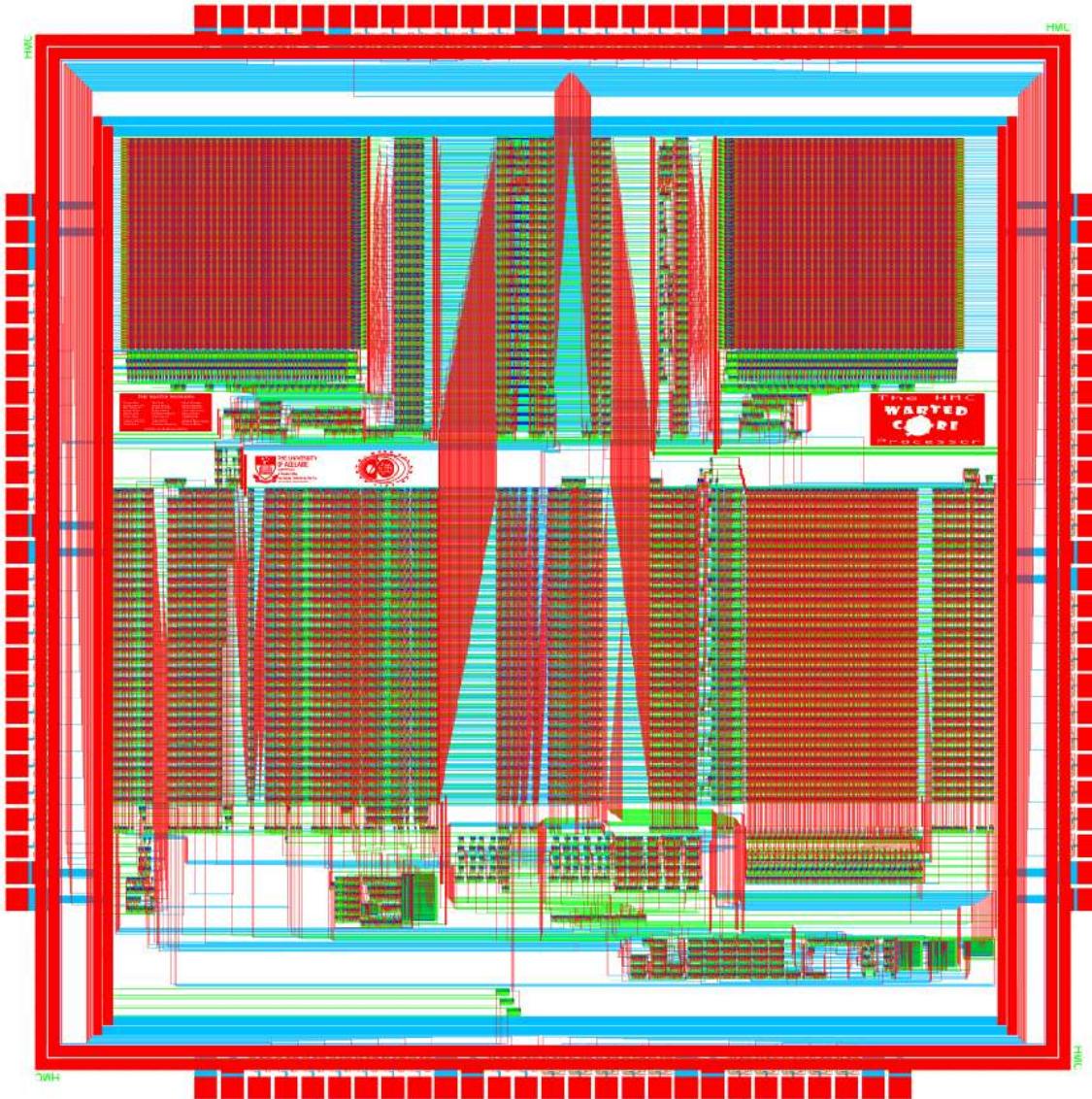


Spring 2007 Class Project

32-Bit MIPS Microprocessor



Engineering 158: CMOS VLSI Design
Harvey Mudd College

Contents

Microarchitecture Cluster.....	10
1. INTRODUCTION.....	10
2. RESPONSIBILITIES.....	10
2.1. CARL NYGAARD - CHIEF MICROARCHITECT	10
2.2. NATE PINCKNEY - MEMORY MICROARCHITECT	10
2.3. THOMAS BARR - EXCEPTION MICROARCHITECT	10
2.4. MATT TOTINO - VALIDATION MICROARCHITECT	10
3. DESIGN SPECIFICATION	11
3.1. INSTRUCTION SET	11
3.2. BASIC R-TYPE OPERATIONS	11
3.2.1. <i>Shifts</i>	11
3.2.2. <i>Moves</i>	11
3.2.3. <i>Arithmetic</i>	12
3.3. ARITHMETIC IMMEDIATE	12
3.4. JUMP/BRANCH	13
3.4.1. <i>R-type Jumps</i>	13
3.4.2. <i>Branch and Link</i>	13
3.5. COPROCESSOR (COPROCESSOR 0 ONLY).....	13
3.6. LOAD OPERATIONS	14
3.7. STORE OPERATIONS	14
3.8. TRAPS AND EXCEPTIONS	14
3.9. OTHER.....	14
4. FEATURE SET	15
4.1. BRANCH DELAY SLOT AND HAZARD DETECTION	15
4.2. MEMORY AND CACHE SYSTEM	15
4.2.1. <i>Overview</i>	15
4.2.2. <i>Memory Interface Conventions</i>	15
4.2.3. <i>Memsys</i>	16
4.2.4. <i>Data and Instruction Caches</i>	17
4.2.5. <i>Write Buffer</i>	18
4.2.6. <i>External Memory</i>	19
4.3. EXCEPTIONS AND INTERRUPTS	20
5. SYSTEM OVERVIEW WITH BLOCK DIAGRAMS.....	21
5.1. PROCESSOR OVERVIEW BLOCK DIAGRAM	21
5.2. FETCH STAGE.....	22
5.3. DECODE STAGE.....	23
5.4. EXECUTE STAGE	24
5.5. MEMORY STAGE	25
5.6. COPROCESSOR 0 (SYNTHESIZED)	25
5.7. MEMORY AND CACHE SYSTEM OVERVIEW	26
6. TEST PLAN	27
6.1. MICROARCHITECTURE VALIDATION RESULTS	28

<i>6.1.1. Branch offsets</i>	28
<i>6.1.2. Cache Size</i>	28
<i>6.1.3. Hazards and Exceptions</i>	28
<i>6.1.4. Program Counter</i>	28
<i>6.1.5. Status Register</i>	28
<i>6.1.6. Endianness</i>	28
6.2. RANDOM TESTING.....	28
<i>6.2.1. Multdiv Tester</i>	29
<i>6.2.2. Code Generator</i>	29
7. SCHEDULE & MILESTONES.....	30
7.1. WINTER 2006.....	30
7.2. SPRING 2007	30
8. LESSONS OF THE DESIGN	31
8.1. CHIEF MICROARCHITECT - CARL NYGAARD	31
8.2. MEMORY MICROARCHITECT - NATE PINCKNEY	31
8.3. EXCEPTION MICROARCHITECT - THOMAS BARR	32
8.4. VALIDATION MICROARCHITECT - MATT TOTINO	32
Chip Cluster.....	34
9. PRELIMINARY FLOORPLANNING.....	34
10. DETAILED FLOORPLANNING.....	34
11. CHIP.....	35
11.1. CHIP I/O	35
11.2. SPECIAL UNIT.....	36
11.3. CHIP SCHEMATIC	36
11.4. CHIP LAYOUT	37
11.5. CHIP TESTING.....	37
11.6. CHIP SUBCOMPONENTS.....	38
<i>11.6.1. Core</i>	38
<i>11.6.2. TopMIPS</i>	39
<i>11.6.3. Pad frame</i>	41
<i>11.6.4. MIPS</i>	44
12. DATAPATH.....	45
12.1. FUNCTION	45
12.2. I/O.....	46
12.3. PLA GENERATION	46
12.4. SCHEMATIC	47
12.5. LAYOUT	48
12.6. TESTING	48
12.7. SUBCOMPONENTS.....	49
<i>12.7.1. Fetch Stage</i>	49
<i>12.7.2. Decode Stage</i>	54
<i>12.7.3. Register File SRAM Array (in Decode Stage)</i>	60
<i>12.7.4. Register Array Decoder</i>	64
<i>12.7.5. ALU (Subunit in Execute Stage)</i>	66

12.7.6. <i>Shifter</i>	69
12.7.7. <i>Execute Stage</i>	73
12.7.8. <i>Multiplier/Divider Controller</i>	82
12.7.9. <i>Memory - Writeback Stage</i>	87
12.7.10. <i>Hazard</i>	94
12.7.11. <i>Five Bit Datapath</i>	97
13. CONTROLLER	99
13.1. FUNCTION	99
13.2. UNIT I/O.....	99
13.3. PLA GENERATION	100
13.4. SPECIAL UNIT.....	100
13.5. SCHEMATIC	100
13.6. LAYOUT	101
13.7. TESTING	102
13.8. SUBCOMPONENTS.....	102
13.8.1. <i>Branch Decoder</i>	102
13.8.2. <i>Main Decoder</i>	104
13.8.3. <i>Main Decoder PLA</i>	107
13.8.4. <i>ALU and Shifter Decoder</i>	109
13.8.5. <i>Coprocessor Decoder</i>	112
13.8.6. <i>aluoutsrsrcD</i>	113
13.8.7. <i>specialregsrcD</i>	115
13.8.8. <i>Control Top Level Logic</i>	116
13.8.9. <i>Branch Controller</i>	117
13.8.10. <i>Control Registers</i>	119
13.8.11. <i>hilocontrol</i>	125
14. MEMORY UNIT	126
14.1. UNIT FUNCTION	126
14.2. UNIT I/O.....	126
14.3. SPECIAL UNITS.....	127
14.4. UNIT SCHEMATIC	127
14.5. UNIT LAYOUT	128
14.6. SUBCOMPONENTS.....	129
14.6.1. <i>Memsys Module</i>	129
14.6.2. <i>Memsyscontroller Module</i>	134
14.6.3. <i>Cacheram Module</i>	137
14.6.4. <i>Cachecontroller Module</i>	142
14.6.5. <i>Write Buffer Module</i>	148
15. COPROCESSOR 0.....	161
15.1. UNIT I/O.....	161
15.2. SPECIAL UNITS.....	162
15.3. UNIT SCHEMATIC	162
15.4. UNIT LAYOUT	163
15.5. UNIT SUB-COMPONENTS.....	163
15.5.1. <i>Staturegunit</i>	163

<i>15.5.2. Causeregunit</i>	167
<i>15.5.3. Epcunit</i>	170
<i>15.5.4. Exceptionunit</i>	171
16. LESSONS LEARNED AND EXPERIENCE GAINED	175
16.1. DATAPATH TEAM	175
16.2. CONTROLLER TEAM	178
16.3. MEMORY TEAM.....	179
16.4. COPROCESSOR 0 TEAM.....	181
17. CHIP CLUSTER APPENDIX.....	183
17.1. PRELIMINARY FLOORPLANNING.....	183
17.2. PAD GENERATION FILE.....	187
17.3. PLA SIMULATION REPORT.....	191
Systems Cluster	193
18. PACKAGE DEFINITION	193
18.1. PACKAGING FUNCTION	193
18.2. I/O TABLE	195
18.3. SPECIAL UNITS.....	196
18.4. PAD FRAME SCHEMATIC	196
18.5. PAD FRAME LAYOUT	197
18.6. LESSONS LEARNED AND EXPERIENCE GAINED	198
19. EXTERNAL MEMORY & I/O SYSTEM.....	199
19.1. HARDWARE.....	199
19.2. DESIGN OVERVIEW (RTL)	199
19.3. DESIGN.....	200
19.4. SIGNALS	200
19.4.1. <i>Inputs</i>	200
19.4.2. <i>Outputs</i>	201
19.4.3. <i>InOuts</i>	202
19.5. TIMING	202
19.5.1. <i>ROM Read</i>	202
19.5.2. <i>RAM Read</i>	202
19.5.3. <i>RAM Write</i>	203
19.5.4. <i>I/O Write</i>	203
19.6. I/O DEVICES.....	203
19.7. MEMORY MAP	204
19.8. TEST PLAN	204
19.9. VERILOG	208
19.9.1. <i>External Memory Top (extmem.v)</i>	208
19.9.2. <i>Memory Controller (memcon.v)</i>	211
19.9.3. <i>ROM.v Template</i>	215
19.9.4. <i>RAM (RAM.v)</i>	216
19.10. TEST FILES	217
19.11. LESSONS LEARNED AND EXPERIENCE GAINED.....	218

20. COMPILER, BENCHMARKS, AND DEMO SOFTWARE.....	219
20.1. COMPILER USER INSTRUCTIONS.....	219
20.1.1. <i>Introduction</i>	219
20.1.2. <i>Toolchain Setup Instructions</i>	219
20.1.3. <i>Writing Programs for the HMC-MIPS</i>	220
20.1.4. <i>Compiling Programs for the HMC-MIPS</i>	222
20.1.5. <i>The Boot Loader</i>	223
20.1.6. <i>Modifying the Memory Map of Generated ROMs</i>	224
20.1.7. <i>Modifying Program Entry Location</i>	225
20.1.8. <i>Simulating and Synthesizing Compiled Programs</i>	225
20.2. USING SUPPLIED PROGRAMS	225
20.2.1. <i>Test Programs</i>	225
20.2.2. <i>Demo Program</i>	226
20.2.3. <i>Dhrystone Benchmark</i>	226
20.3. USING SUPPLIED LIBRARIES.....	230
20.3.1. <i>muddCLib</i>	230
20.3.2. <i>mtRand</i>	231
20.4. LISTING OF COMPILER FILES	232
20.4.1. <i>Source Files</i>	232
20.4.2. <i>Makefiles</i>	233
20.4.3. <i>Text Files</i>	233
20.4.4. <i>Toolchain Scripts</i>	233
20.5. PRINTOUTS OF SOURCE FILES.....	235
20.5.1. <i>Boot Loader</i>	235
20.5.2. <i>Bootstrapper</i>	238
20.5.3. <i>Dhrystone Header</i>	239
20.5.4. <i>Dhrystone</i>	241
20.5.5. <i>Dhrystone Functions</i>	249
20.5.6. <i>Lights Out! Header</i>	255
20.5.7. <i>Lights Out! Source</i>	256
20.5.8. <i>Corner-Case Test</i>	263
20.5.9. <i>Button Test</i>	265
20.5.10. <i>LCD Test</i>	268
20.5.11. <i>LED Test</i>	270
20.5.12. <i>Simple Test</i>	271
20.5.13. <i>LED Assembly Test</i>	272
20.5.14. <i>Instruction Checker Script</i>	272
20.5.15. <i>Verilog Generator Script</i>	274
20.5.16. <i>ROM Generator Script</i>	278
20.5.17. <i>String to Charater Array Script</i>	282
20.5.18. <i>Toolchain Memory Specifications</i>	283
20.5.19. <i>Verilog Template</i>	283
20.5.20. <i>Makefile</i>	283
20.5.21. <i>Template Makefile</i>	285
20.5.22. <i>Boot Makefile</i>	286
20.5.23. <i>Dhrystone Makefile</i>	286

20.5.24. <i>Lights Out! Makefile</i>	287
20.5.25. <i>Corner-Case Test Makefile</i>	287
20.5.26. <i>Button Test Makefile</i>	288
20.5.27. <i>LCD Test Makefile</i>	288
20.5.28. <i>LED Test Makefile</i>	289
20.5.29. <i>Simple Test Makefile</i>	289
20.6. LESSONS LEARNED AND EXPERIENCE GAINED.....	290
21. FPGA EMULATION.....	291
21.1. INTRODUCTION.....	291
21.2. EXTERNAL CLOCK MODIFICATION.....	291
21.3. SINGLE FPGA EMULATION	292
21.4. DUAL FPGA EMULATION	294
21.5. TEST RESULTS.....	298
21.6. PCB TESTING.....	299
21.7. LESSONS LEARNED	299
22. PRINTED CIRCUIT BOARD.....	300
22.1. PCB FEATURES	300
22.1.1. <i>PGA Socket</i>	300
22.1.2. <i>Power</i>	300
22.1.3. <i>Miscellaneous Features</i>	301
22.1.4. <i>Liquid Crystal Display</i>	301
22.1.5. <i>Connections</i>	301
22.2. PCB POST-FAB TEST PLAN	301
22.3. PCB SETUP AND CORRECTIONS	302
22.4. CONCLUDING COMMENTS	303
23. POST-SILICON TEST PLAN.....	304
23.1. INTRODUCTION.....	304
23.2. CONTENTS.....	304
23.3. QUICK START GUIDE.....	304
23.3.1. <i>Unpack the box</i>	304
23.3.2. <i>Set up the single-FPGA emulation</i>	304
23.3.3. <i>Set up the FPGA-PCB package</i>	304
23.3.4. <i>Set up the Compiler and Toolchain</i>	305
23.4. TEST PLAN	305
23.4.1. <i>Clock Frequency Verification</i>	305
23.4.2. <i>Device-Specific Test Programs</i>	306
23.4.3. <i>Demo Program</i>	306
23.4.4. <i>Random Tests</i>	306
23.4.5. <i>Benchmark</i>	306
24. SYSTEMS CLUSTER APPENDIX.....	308
24.1. PIN SPECIFICATIONS	308
24.2. V1.0 SCHEMATIC	310
24.3. V1.0 LAYOUT.....	311
24.4. V1.1 SCHEMATIC	312

24.5. V1.1 LAYOUT.....	313
24.6. BILL OF MATERIALS.....	313
Library Cluster	314
25. LIBRARY (MUDLIB07.JELIB) CREATION.	314
25.1. PERSONNEL:.....	314
25.2. LESSONS LEARNED:.....	315
26. THE PLA GENERATOR	316
26.1. PERSONNEL:.....	316
26.2. LESSONS LEARNED	318
27. CHIP LOGO	318
27.1. PERSONNEL:.....	318
27.2. LESSONS LEARNED	319
28. CHIP REPORT	319
28.1. PERSONNEL:.....	319
28.2. LESSONS LEARNED	319
29. LIBRARY CLUSTER, APPENDIX A: LIBRARY CELLS.....	320

Microarchitecture Cluster

1. Introduction

Harvey Mudd College's Spring 2007 VLSI class decided to implement the MIPS ISA. This project required a microarchitecture specification written in Verilog. This report summarizes the implementation process and results.

2. Responsibilities

The Microarchitecture Team was broken up into the following roles.

2.1. Carl Nygaard - Chief Microarchitect

The responsibility of the Chief Microarchitect was to develop the overall design of the microprocessor and see to the implementation of the microprocessor in HDL. This included specifying and implementing the processor's instruction set as well as choosing and implementing various features.

2.2. Nate Pinckney - Memory Microarchitect

The Memory Microarchitect was responsible for designing the on-chip memory and cache system. This included designing and implementing the caches, a write buffer, and a module to arbitrate requests between all of the memory system modules, the processor, and the external memory. Multiple cycle latency test memory was implemented to aid in testing. Lastly, specifications for the memory and cache system were written.

2.3. Thomas Barr - Exception Microarchitect

The Exception Microarchitect was responsible for determining the behavior of the CPU on invalid operation and interrupts, and implementing the necessary hardware to deal with exceptions and interrupts. Several exceptions were required for debugging, system calls and allowing the system to recover from invalid programs and data. The nature of the CPU required that the hardware to do this be extremely efficient in order to fit in the given area.

In addition, the exception microarchitect assisted the chief microarchitect in adapting the CPU HDL to be in structural form, and developed a synthesizable automatic test bench that allowed the creation of an automated tester alongside the CPU in and FPGA to rapidly verify the proper functionality of the chip.

2.4. Matt Totino - Validation Microarchitect

The responsibility of the Validation Microarchitect was to design and implement tests in addition to those developed by the other module designers for use in their regression testing. The additional tests checked implementation of required operations more extensively and gave specific attention to cases in which many different functions of the chip (for example, hazards, exceptions, and branching) were used together.

3. Design Specification

To keep the specification process simple and agile, the overall system has been defined primarily by the instructions supported by the system. Several features of the system are also provided in this section.

3.1. Instruction Set

The following list of instructions is supported by the processor. This includes all the instructions from the MIPS-I ISA **except** the patented misaligned memory access instructions (lwl, lwr, swl, and swr), all floating point operations, and instructions used for TLB management (tlbr, tlbw, tlbwr, and tlbp).

For more details on the actual expected behavior or each instruction, see *See MIPS Run* by Dominic Sweetman.

3.2. Basic R-type Operations

3.2.1. Shifts

sll	Shift Left Logical
srl	Shift Right Logical
sra	Shift Right Associative
sllv	Shift Left Logical Variable
srlv	Shift Right Logical Variable
sraw	Shift Right Associative Variable

3.2.2. Moves

mfhi	Move from HI (HI and LO are special purpose registers used by mult, etc)
mthi	Move to HI
mflo	Move from LO
mtlo	Move to LO

3.2.3. Arithmetic

mult	Multiply (signed)
multu	Multiply Unsigned
div	Divide (Does NOT throw exceptions, puts quotient in LO and remainder in HI)
divu	Divide Unsigned
add	Add with overflow trap
addu	Add with no traps
sub	Subtract with overflow trap
subu	Subtract with no traps
and	And
or	Or
xor	Xor
nor	Nor
slt	Set on Less Than ($rd \leq rs < rt$)
sltu	Set on Less Than Unsigned

3.3. Arithmetic Immediate

addi	Add Immediate with traps
addiu	Add Immediate with no traps
slti	Set Less Than Immediate
sltiu	Set Less Than Immediate Unsigned
andi	And Immediate
ori	Or Immediate
xori	Xor Immediate

lui	Load Upper Immediate (loads a constant into upper half of a word)
-----	---

3.4. Jump/Branch

j	Jump
jal	Jump and Link
beq	Branch if equal
bne	Branch if not equal
blez	Branch if less than or equal to zero
bgtz	Branch if greater than zero
bltz	(opcode 1) Branch if less than zero
bgez	(opcode 1) Branch if greater than or equal to zero

3.4.1. R-type Jumps

jr	Jump Register (Jump to an address given in a register)
jalr	Jump and Link Register

3.4.2. Branch and Link

bltzal	(opcode 1) Branch if less than zero and link (unconditionally stores the return address)
bgezal	(opcode 1) Branch if greater or equal to zero and link (unconditionally stores the return address)

3.5. Coprocessor (Coprocessor 0 only)

mfc0	Move from coprocessor 0 (Copies coprocessor register into CPU register)
mtc0	Move to coprocessor 0

3.6. Load Operations

lb	Load Byte (Sign extends a 1 byte value retrieved from memory, puts in register. This does not have to be word aligned)
lh	Load Halfword (Sign extends 2 byte value, must be naturally aligned, so the LSB of the address must be zero)
lw	Load Word (Must be word aligned)
lbu	Load Byte Unsigned
lhu	Load Halfword Unsigned

3.7. Store Operations

sb	Store Byte (Does not need to be word aligned)
sh	Store Halfword (LSB of address must be zero)
sw	Store Word (Must be word aligned)

3.8. Traps and Exceptions

MIPS uses polled interrupts with a hardwired exception handler address (We will use 0x00000004).

syscall	System call (Immediately transfer control to the exception handler – R-type)
break	Breakpoint (Immediately transfer control to the exception handler – R-type)
rfe	Return from Exception

3.9. Other

nop	Do nothing
-----	------------

4. Feature Set

4.1. Branch Delay Slot and Hazard Detection

The branch delay slot behavior of the MIPS I instruction set was duplicated. Hazard detection was also implemented to support various stalls and pipeline dependencies.

4.2. Memory and Cache System

4.2.1. Overview

The memory and cache system is composed of the following modules:

- Two identical 512 byte caches: one for data and one for instructions organized as SRAM.
- A write buffer to reduce number of stalls when writing to main memory.
- A memsys controller which multiplexes data between the above modules, processor, and external memory.

4.2.2. Memory Interface Conventions

All of the primary memory and cache system modules follow a common interface convention for control signals and address/data buses. The interface convention includes:

- *addr*: word address
- *data*: bi-direction data
- *byteen*: byte mask for writing data less than a word in length
- *rwb*: read/write bar, 0 = write, 1 = read
- *en*: enable read/write
- *done*: asserted when memory operation is complete

Most of the modules include the interface convention twice, one for requests from "lower" modules to the module and one for requests from the module to a "higher" module. "Lower" and "higher" modules refer to the position of the module within the memory hierarchy shown in Figure 1. For example, from the write buffer's perspective, the "lower" module is the caches and the "higher" module is external memory.

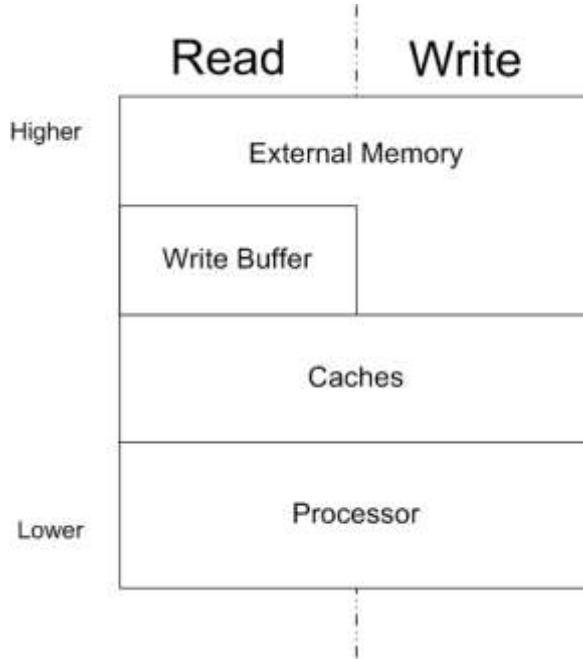


Figure 1. Memory Hierarchy

We'll call the "lower" module requesting a memory operation the *master* and the current module executing the operation the *slave*. When the slave's input pins (*adr*, *data*, *rwb*, etc...) from the master are valid, the master asserts *en*. *En* must be asserted for the entire duration of the operation. The slave will execute the operation until completion, where it will drive necessary pins to the master and assert *done*. There will always be at least a one cycle latency between the master asserting *en* and the slave asserting *done*, except between the MIPS processor/cache interface (for reading) and the external memory interface.

All of the interfaces for primary modules are detailed below and are from the slave's point of view.

4.2.3. Memsys

The MIPS memory system (*memsys*) module is the entire memory/cache system and interfaces the processor, caches, write buffer, and external memory. Additionally, it handles operations such as cache swapping. Memsys also includes the memsys controller (*memsyscontroller*) containing logic to control many multiplexers and tri-state buffers, to prioritize and arbitrate external memory operations. Memsys's interface is:

```

input ph1, ph2, reset,
input [31:2] pcF,
output [31:0] instrF,
input reF,
output instrackF,

input [29:0] adrM,
input [31:0] writedataM,
input [3:0] byteenM,
output [31:0] readdataM,
input memwriteM, reM,
```

```

    output dataackM,
    input swc,
    output [26:0] memadr,
    inout [31:0] memdata,
    output [3:0] membyteen,
    output memrwb,
    output memen,
    input memdone

```

Note that an uncached address read or write operation will cause a cache miss (and hence processor stall) whenever *en* is asserted. Hence, it is necessary for the processor to control the data and instruction cache *en* inputs, so no cycles will be wasted on unneeded memory stalls (e.g. if the current instruction does not use memory).

Since the external memory only has one data and address bus, the memsys is responsible for multiplexing read/writes to the external memory from the two caches and write buffer. The cache controller will wait until the current memory operation is complete and start new operations with the following priority (top indicates highest priority):

1. Write buffer write
2. Data cache read
3. Instruction cache read

The data cache (or instruction cache if swapped) is connected directly to the write buffer if it requests a memory write. When a module is using the external memory, the memory and module are multiplexed together directly, so once the memory asserts *done* the component's *memdone* will be HIGH. After one cycle, memsys disconnects the module and the memory, and connects a LOW to the *memdone* input of the module.

4.2.4. Data and Instruction Caches

Both caches are identical, one for instructions and one for data. Each cache will be 512 bytes in size and synchronous. Cache features include:

- *Write-through*: When a data write is requested from the processor, it is written immediately to memory (or write buffer) as well as the cached. The cache never holds a newer copy of data than main memory (or write buffer).
- *Write buffer*: To improve performance, so the CPU does not stall for the entire external memory write time, a FIFO (first-in, first-out) write buffer is used. Once the write buffer is full, the CPU stalls until a space is available in the write buffer.
- *Direct-mapped*: The lower seven bits of the memory address are used as the tag in the cache memory.
- *Physically addressed*: The address the cache uses in the tag data is based upon the physical address of the data in external RAM.
- *Bypassing*: The cache can be bypassed via the upper bits (explained in the memory map section) so that certain data (e.g. memory mapped I/O) is never cached.

- *Swapping*: The caches can be swapped. This is mainly useful for cache invalidation during boot loading of the processor.

Since each cache is 512 bytes and a word is 4 bytes, each cache can hold 128 words. A *tag* must be associated with each word, representing the lower bits of the address. $128 = 2^7$, so each tag is 7 bits long. The *tag data* must hold the upper bits of the address, and all addresses should be word aligned, hence $32 - 7 - 2 = 23$ bits of tag data. Additionally, a *valid* bit must be associated with each tag. The total width of each cache slot is then $23 + 1 + 32 = 56$ bits. The total size of the cache's memory is then $128 * 56 = 7168$ bits or 896 bytes per cache.

The *cache slots* (containing the tag data, data, and valid bit) are stored in the *cacheram* module, and the controls are generated from a *cachecontroller* module. The cache interface is:

```

input ph1, ph2, reset,
input [29:0] adr,
inout [31:0] data,
input [3:0] byteen,
input rwb, en,
output done,

output [26:0] memadr,
inout [31:0] memdata,
output [3:0] membyteen,
output memrwb,
output memen,
input memdone

```

The cache is synchronous with the processor completes a cache hit within one clock cycle. So, the cache will respond exactly as the HMC ENGR85 (Digital Electronics and Computer Engineering) MIPS processor's instruction and data memory did for cached memory locations that are valid. In this case, a *done* signal will be asserted within the same clock signal as the request. When the cache misses, the *done* signal is not asserted until the memory fetch or write is complete. During the time *done* is not asserted, the hazard unit causes the CPU to stall.

When a word write is requested to an address that is cachable, but not currently in the cache, data is written to both the cache and to the write buffer. When a half-word or byte address write is requested, it is sent directly to the write buffer and invalidates its cache line. Hence, a store other than word length will never write to the cache: just invalidate. This is to retain compatibility with R2000 MIPS processors and to allow invalidation of cache lines. Note that the caches do not support cache isolation because a read-modify-write cycle was not implemented (read-modify-write cycles appear to only have been implemented on R3000's and above). If a cache is given an uncached address (to bypass the cache), it will not read/write from/to the cache line (or invalidate it) but otherwise completes the operation as above.

4.2.5. Write Buffer

The write buffer is a FIFO (first-in first-out) buffer four entries deep, as suggested as an optimal depth by *See MIPS Run*. The write buffer (module *writebuffer*) contains the module *wbram*,

which represents the buffer's memory. Dual-port SRAM is needed for *wbram* because the write buffer must be able to send data to the external memory while reading data from the cache to write, and so two independent bit-line buses are needed with corresponding word-line buses. Data cannot be read from external memory until after the write buffer is empty, so a cache read miss directly after a memory write will stall the CPU until all writes are complete. This prevents the processor from reading outdated data. In addition to storing the address and data for a write, the write buffer must also store the *byteen* byte mask to support storing data smaller than a word.

The interface is:

```

input ph1, ph2, reset,
input [26:0] adr,
input [31:0] data,
input [3:0] byteen,
input en,
output done,

output [26:0] memadr,
output [31:0] memdata,
output [3:0] membyteen,
output memen,
input memdone

```

4.2.6. External Memory

The memory system expects a single external memory interface. The exact timings of the main memory are very flexible. The main memory should expect an *en* (enable) input when the cache system requests a read/write. Once the main memory has performed the operation, and driven any necessary output ports with the correct value, it asserts *done*.

The interface from the main memory's view is:

```

input ph1, ph2,
input [31:0] adr,
inout [31:0] data,
input [3:0] byteen,
input rwb, en,
output done

```

If the memory does not have these specifications, then a memory controller will need to be implemented to interface the cache system with the RAM. On the test implementation used for simulation, a reset pin was added to reset the state machine used to generate memory cycle latency. *done* can be asserted within the same cycle, so the memory should not drive *done* HIGH until the memory operation is complete. Note that there are only 27 address bits from *memsys*, so 3 MSB's (because of physical mapping) and 2 LSB's (because it is word addressed) of the external memory's address are driven LOW.

4.3. Exceptions and Interrupts

We implemented the following exceptions:

- 0 - Interrupt
- 4 - AdEL - Attempt to load or fetch from a misaligned address (word aligned for lw, halfword aligned for lhw. impossible to throw for lb)
- 5 - AdES - Attempt to store to a misaligned address (same rules as for AdEL)
- 8 - Syscall
- 9 - Breakpoint
- 10 - RI (invalid opcode)
- 11 - CpU - thrown on attempt to use CP1-3 (FPU)
- 12 - Ov - Arithmetic overflow

This set of exceptions should allow the CPU to recover from any invalid code or data passed to it, as long as the exception handler is functionally correct, and all hardware is properly functioning. In addition, this restricted subset of the MIPS exceptions allows all exceptions to be caught at or before the execute stage, simplifying the layout dramatically, as only the execute stage needs to interface with the unit.

The general functionality of the exception handling of the MIPS CPU is as follows: Every exception has an associated instruction that is responsible for triggering that exception. In the case of a syscall, break, invalid opcode or fpu, the exception is detected by the decode stage, and a flag is raised and passed down the pipeline to the execute stage, where the exception handler detects the flag, and exception handling begins. In the case of the AdEL or AdES exceptions, the output of the ALU is checked to ensure that proper alignment is being calculated, and if not, the exception handler detects this, and begins exception handling. The behavior of the overflow exception is similar. Finally, when an interrupt pin is driven high, the exception handler takes the instruction in the execute stage as the "excepting instruction".

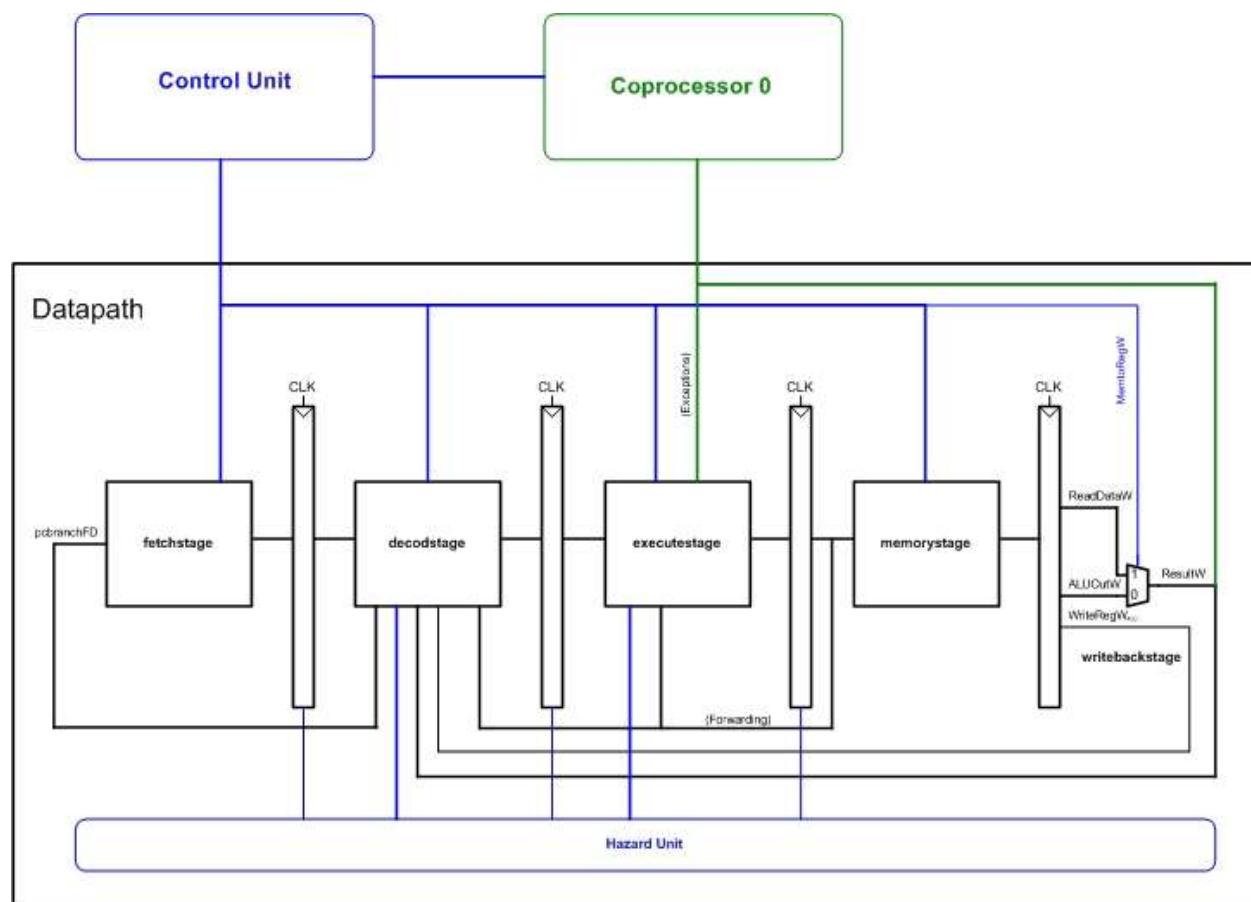
When an exception is raised, the CPU saves the location of offending instruction in the cause register, and jumps to the hard-coded location of the exception handler. SR[IEc] is set low by the exception handler, which ensures that interrupts are not served during exception handling (thereby preventing an infinite loop if an interrupt line is driven high, and held there). The exact mechanism of dealing with the excepting behavior is up to the implementation of the system software, but it is important to run the rfe instruction (typically in the branch delay slot of the jump back to user code) before re-executing any user code. In our CPU, this resets SR[IEc], and allows the CPU to handle exceptions. The implementation of the rfe instruction is designed such that if an interrupt line is high when the rfe instruction is called, the interrupt occurs during the first user instruction, not in the exception handler.

5. System Overview with Block Diagrams

The following block diagrams are meant to provide a high-level understanding of the chip and its various components.

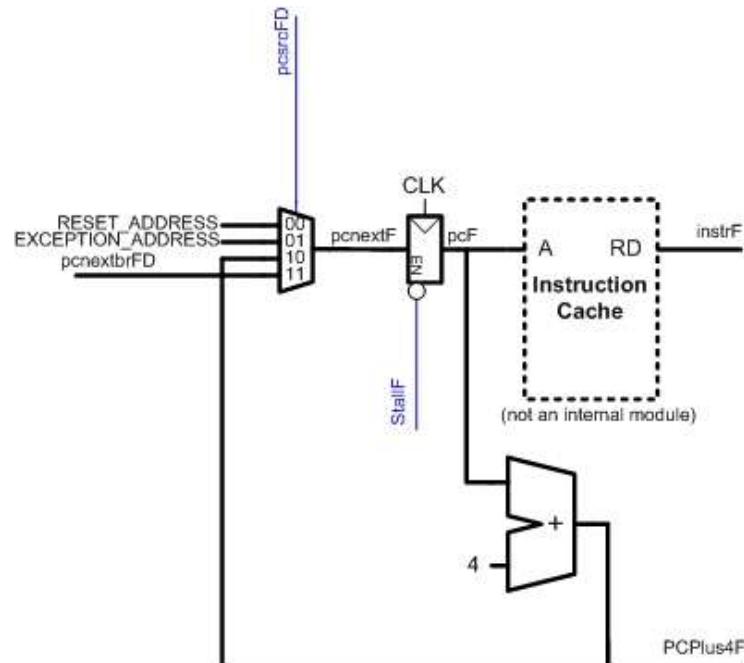
5.1. Processor Overview Block Diagram

The MIPS processor is pipelined. It contains five pipeline stages: Fetch, Decode, Execute, Memory, and Writeback. A separate control unit interprets the instruction and dictates the behavior of the processor through an array of control wires. Coprocessor 0 handles meta operation such as the Status and Cause Registers. The coprocessor also interacts with the execute stage to carry out exceptions. The Hazard Unit protects against hazards by managing register forwarding, branch stalling, memory stalls, etc.



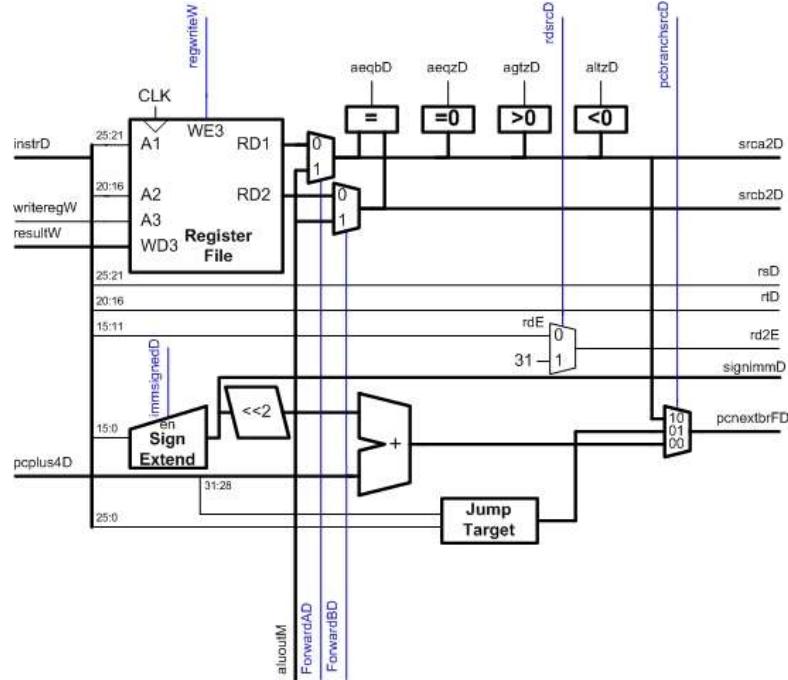
5.2. Fetch Stage

The Fetch stage retrieves instructions from memory. Which instruction is chosen is determined by the Program Counter (PC).



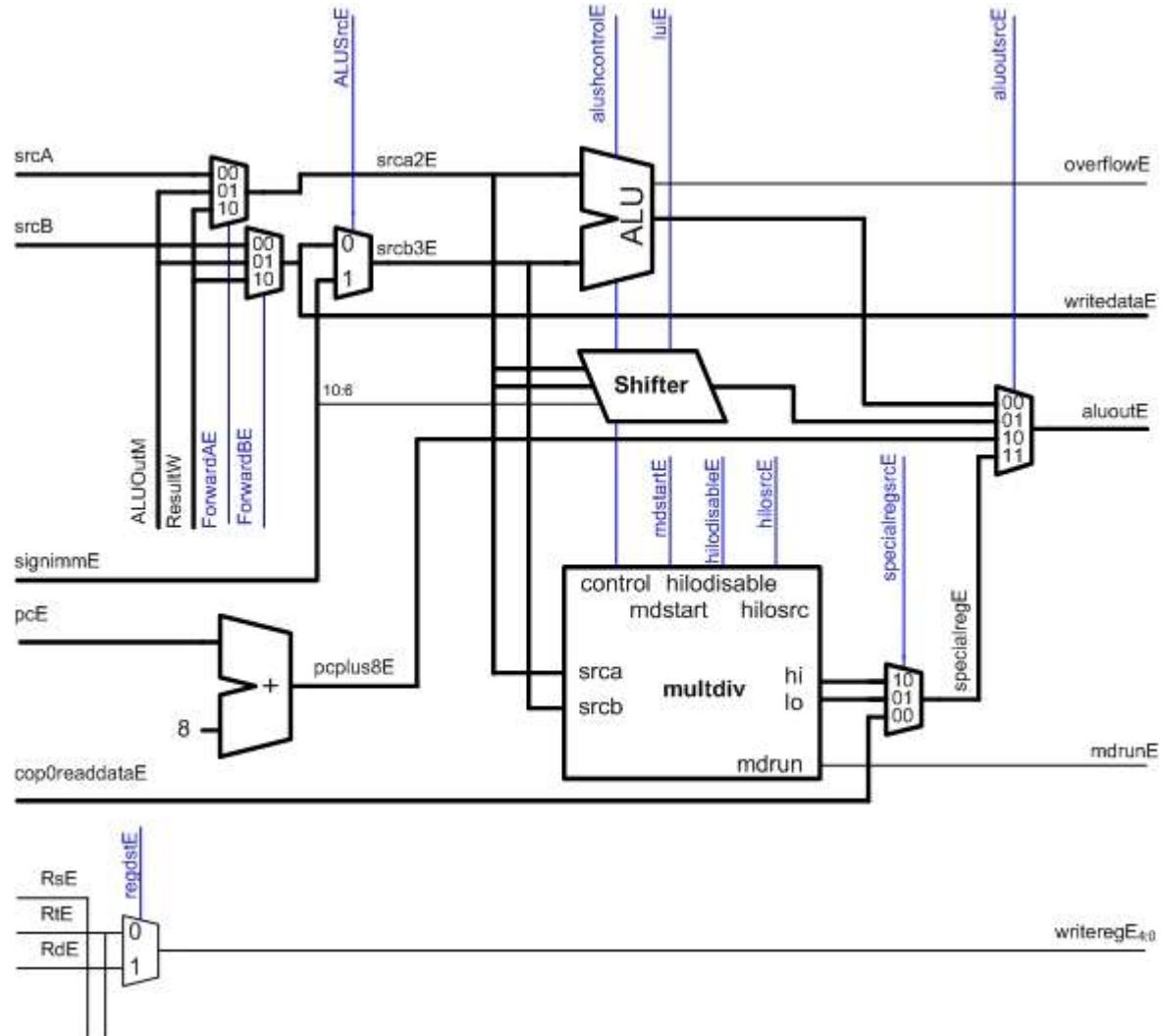
5.3. Decode Stage

The Decode stage has several responsibilities. The instruction taken from the Fetch stage is broken up into subcomponents. The Controller decodes the instruction while the decode stage pulls out appropriate register values from the Register File. Additionally, jumps and branches are carried out during the Decode stage. By branching in this stage, the instruction following a branch is introduced to the pipeline. This addition instruction is known as the branch delay slot, and it is always executed following a jump or branch.



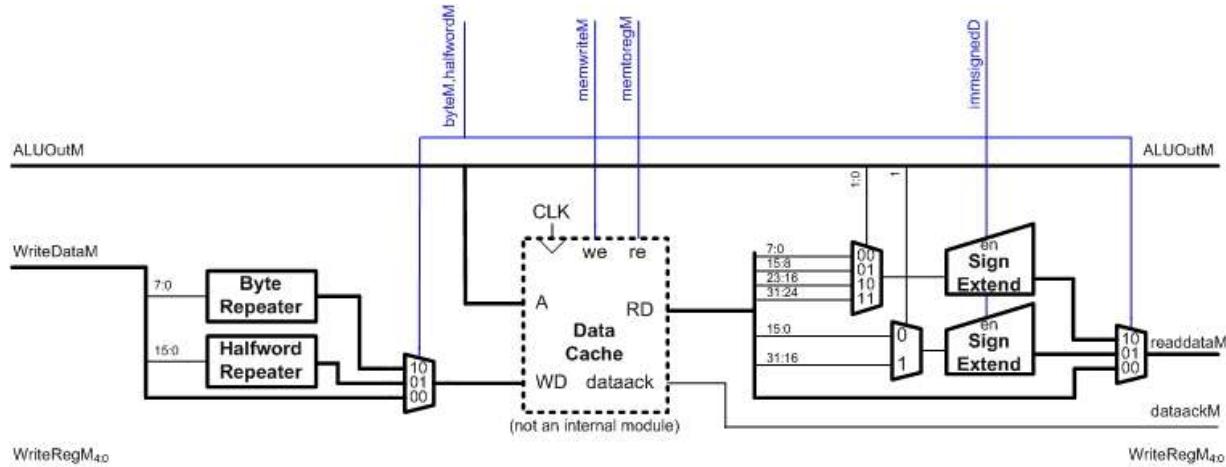
5.4. Execute Stage

The Execute stage carries out the majority of the computation of the processor. The ALU performs basic addition, subtraction, etc. The Shifter performs logical and arithmetic shifts. The Multiply/Divide unit performs multiplication and division. Because multiply and divide take multiple clock cycles, the Multiply Divide unit has the ability to operate independent of the rest of the pipeline, storing its results in the HI and LO register, which are accessed via special instructions.



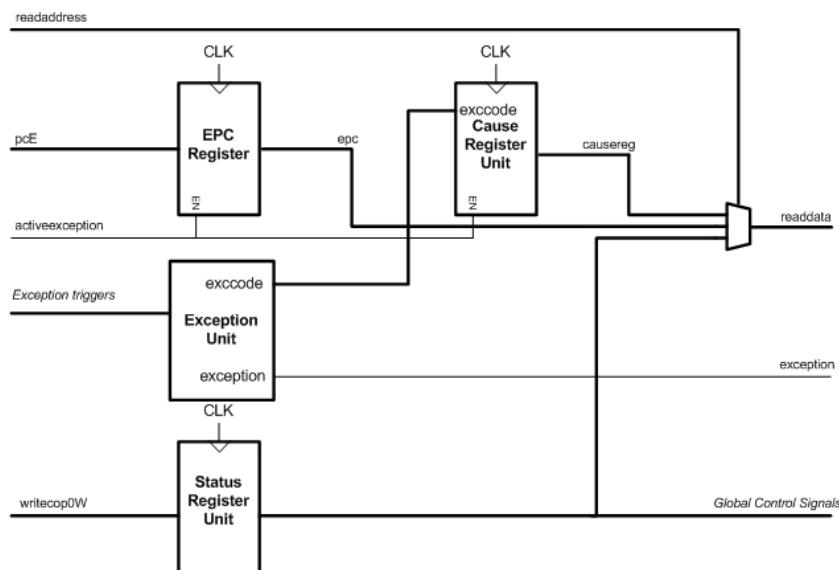
5.5. Memory Stage

The Memory stage is responsible for reading and writing data. Most notably, this stage supports byte, halfword, and word sized reads and writes.

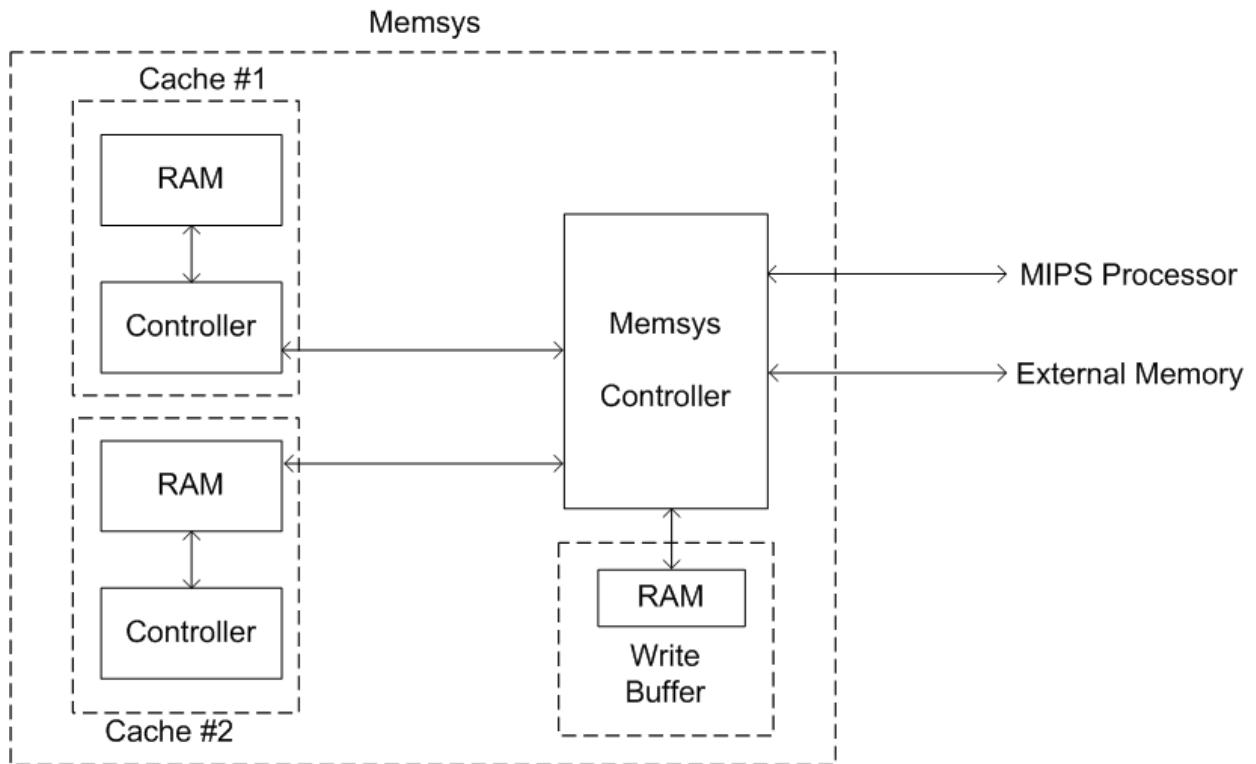


5.6. Coprocessor 0 (Synthesized)

Coprocessor 0 holds meta information and handles exceptions. The Status register allows for software-defined options such as the endianness of the system. The exception unit determines when an exception should occur. The Cause register holds information about an exception when it occurs, and the Exception PC (EPC) register retains the location of a previous exception.



5.7. Memory and Cache System Overview



6. Test plan

We created a wide array of tests to assess both the functionality of all the operations available to the MIPS processor as well a number of significant corner cases. The basic tests implement every operation at least once to ensure successful completion. The more complex tests check important operations rigorously and cover critical situations related to overall hardware design, including branch delay slot and exception testing.

The final test battery compiled by the Microarchitecture team includes 26 separate tests. All can be simulated sequentially under a single test bench. Each test is implemented in the test bench by conducting a series of operations and eventually writing a specific value to a specific address in memory. The value/address can depend on the operations (with a known result) so that they act as a check that the operations ran properly, or simply reaching the write operation can indicate success. In either case, the test bench looks for the correct memwrite operation and considers the test a success if it is found.

The table on the next page outlines the purpose of each test.

Test Number	Operations/Functions Tested
000	General operation (calculates Fibonacci sequence)
001	General operation
002	Immediate instructions, hazards
003	All R-type arithmetic instructions (excepting mult and div)
004	All shift operations
005	Branching and branch delay slots, all branch and jump commands
006	Coprocessor 0 status register
007	Store commands
008	Load commands
009	Overflow exceptions
010	General exception test (syscall, break, bad opcode, and floating-point access)
011	Misaligned read/write exceptions
012	Basic multiply test
013	Interrupt test
014	Arithmetic overflow limits
015	Extensive branch delay slot testing
016	Multiply/divide unit
017	Cache initialization and basic read/write
018	Misaligned PC exception
019	Conditional branch corner cases
020	Basic Read/write to data cache and write buffer
021	Basic Read/write to instruction cache and write buffer, cache swapping
022	Byte enable functionality and cached writes for both caches (little endian)
023	Cache bypassing
024	Byte enable and cached writes for both caches with reverse endian (big endian)
025	Reading/writing to addresses with identical tags (same lower 8 bits)

6.1. Microarchitecture Validation Results

The overall validation process used by the Microarchitecture team relied principally on regression testing. In other words, each incremental hardware change was tested using the full test bench that existed at that point, generally with the addition of a simple test addressing the functionality of the new hardware additions specifically. As a result, we did not generate many formal bug reports; most problems were detected and resolved before being committed to the official source code. The following were significant bugs that we dealt with during the design of the processor.

6.1.1. Branch offsets

The base address used in branching (which uses a relative PC jump) was initially just the program counter PC. This made testing using the PCSPIM simulator as a reference convenient since it employs this convention, but the target gcc compiler uses PC+4 as a reference. Therefore, we changed the branch offset to be relative to PC+4 (which eliminated the ability to use PCSPIM as a reference platform.)

6.1.2. Cache Size

Initial designs presumed on-chip caches of multiple kilobytes (for each of the instruction and data caches). The sizes were reduced multiple times down to the current specification of 512 bytes for each cache. The driving force behind the changes was the issue of physical space requirements on the chip.

6.1.3. Hazards and Exceptions

The design was changed to allow the hazard unit control over when an exception takes effect. This is somewhat contrary to the idea that exceptions (particularly interrupts) should generally have top priority, but was needed to ensure that the processor would not get caught in an infinite stall cycle.

6.1.4. Program Counter

The fetch stage program counter was not holding the correct value. The issue was fixed, but the processor now requires reset to be held high for at least two cycles in order to deal with possible stalls.

6.1.5. Status Register

The initial implementation of the status register had misaligned elements that were fixed. Additionally, the SwC bit that controls cache swapping was undefined after a reset of the processor, so it was changed to default to normal mode on reset.

6.1.6. Endianness

Default endianness of the cache was changed from big to little to reflect architecture standards.

6.2. Random Testing

To test the functionality of a complicated device, simply testing known and expected configurations is rarely sufficient. It is imperative to test configurations the developers did not

think of, which implies some sort of random test. Entropic input is fed into the device, expected output computed from some reference implementation, and the output from the simulated chip compared against. For a large project where the microarchitecture is designed in an HDL, and then schematics generated to match that microarchitecture, a large random testbench can demonstrate consistency between the two implementations. Additionally, when the reference implementation is written as a clean-room reimplementation from the specification in some higher-level language, algorithm errors and mistakes made in designing the microarchitecture can be uncovered. For the hmc_mips project, two such testbenches were developed.

6.2.1. Multdiv Tester

A directed random test vector generator was developed for the multiply/divide unit of the chip. The code generated a pair of 32-bit input values, either from a list of possible corner cases or randomly, and multiplied or divided them. Testing against the chip revealed that while the multiplication unit functioned properly, the divide unit, when run against signed input, provided results inconsistent with any other integer division and modulo definition. Higher-level languages (such as Python, which the tester was written in) use so-called “True Division” (in this division, the remainder is always positive), and low-level languages calculate the magnitude of the quotient and remainder from the absolute values of the input values, and the signs reintroduced after division. The sign for the remainder was improperly calculated in our initial design, (the sign of the remainder should be the sign of the divisor, not dividend, as recommended by at least one web site). Fixing this allowed our chip to generate results for multiply and divide operations consistent with PowerPC C.

6.2.2. Code Generator

To test the entire chip, a random MIPS assembly generator, codegen, was developed. This system randomly produced MIPS assembly, including all R-type instructions, word-length memory operations and branch operations. It simulates the state of the CPU as it generates code (to prevent cycles, branches are only emitted for the forward direction), and at the end of the test, all registers are XORed together, and the result stored in a known memory location. By comparing the final result from the code generator with the result stored by the simulated CPU, the CPU (both as HDL Microarchitecture and a netlist generated from the schematic), can be shown to properly execute code. The CPU ran over 4,000 random instructions in schematic form and 100,000 instructions in HDL form with zero errors. As with the rest of the chip, hmc_mips is distributed under the MIT open-source license, and should be usable for 32-bit and 64-bit IPS projects. It can be acquired from the project’s SVN repository under testing/codegen. If you use codegen in your project, please let the author (tbarr [at] cs [dot] hmc.edu) know about your experiences.

7. Schedule & Milestones

7.1. Winter 2006

This schedule shows the dates that milestones were completed prior to the start of the Spring 2007 semester. These tasks were completed by Carl Nygaard.

12/23/06	Development platform setup with simple test
12/26/06	Arithmetic Immediate
12/31/06	R-Type Instructions
1/2/07	Jump/Branch and Branch & Link
1/4/07	Load and Store Operations (with basic cache model and stalling)
1/6/07	Exceptions and Coprocessor 0 Infrastructure

7.2. Spring 2007

This schedule shows the dates that milestones were completed during the Spring 2007 semester. These tasks were completed by the entire Microarchitecture team.

1/20/07	Microarchitecture Team Organization
1/28/07	Test Plan/Implementation
1/28/07	Multiply/Division Instructions
1/28/07	Exceptions
1/28/07	Memory System Specification/Implementation
1/28/07	CPU Function Completed
2/1/07	RTL Function Completed
2/6/07	CPU Block Diagram
2/8/07	RTL Tuning Completed
2/10/07	CPU Validated
2/15/07	RTL Validated - Code freeze handed off to class for hardware implementation
2/27/07	Chip Report

8. Lessons of the Design

8.1. Chief Microarchitect - Carl Nygaard

First, I would like to recognize Professor David Harris's help throughout this project. I began with a functioning (albeit incomplete) MIPS processor originally developed by Professor Harris. Although that original code has gone through a huge transformation over the past two months, the base code deserves the majority of the credit for the underlying design.

The biggest challenge in this project for me was implementing an existing standard. Coming up with a sufficient list of instructions as well as implementing those instructions as a single functioning system seemed impossible at times. The main benefit of implementing an existing instruction set, especially a RISC instruction set like MIPS, is that many challenging aspects of the architecture are already decided by the instructions themselves. For example, to read and write from memory, MIPS require an explicit load or store operation. Unfortunately the required instructions are at the same time constraining. Developing support for each instruction while needing to minimize hardware and speed was surprisingly time-consuming and challenging. In the typical introductory class on digital design, the processor is presented to the student. Understanding the hardware is one thing, but developing it is quite another.

As a computer scientist, I have a reasonable amount of experience in software development, but developing a large hardware system has drastically improved my project skills. I consider teamwork, iterative development, and testing to be critical for a successful project. These three elements flowed together in this project as easily as any computer science project. Our success illustrates that hardware developers can also use tools more typically used for software. The team successfully used version control software for concurrent development. In addition, the regression test suite's instant feedback facilitated rapid development. Finally, the most critical aspect of this project was the team's dedication and perseverance.

8.2. Memory Microarchitect - Nate Pinckney

The MIPS processor has been the largest technical project I have helped with and I've learned much during the process. In retrospect, I learned that I should have spent more time on the higher level functionality, splitting the functionality into discrete modules, and developing working specifications instead of trying to implement immediately.

I have learned the value of good workflow from this project, such as developing test benches on a Linux machine, updating modules in Modelsim's editor, and quickly using the waveform view in Modelsim to debug. Towards the beginning, my debugging technique was poor, but later I discovered how to quickly utilize waveforms and to restart simulations without quitting the simulator to speed up debugging. I learned the value of creating test cases for quick and thorough testing of modules.

I also gained a deeper appreciation for structuring large projects into small teams and the importance of documentation to effectively communicate ideas to other team members. I enjoyed working on the project and am grateful for the experience I gained.

8.3. Exception Microarchitect - Thomas Barr

The design of the exception handler and other responsibilities of the exception microarchitect took many long nights in the lab to complete. The process, however, was greatly eased by careful examination of possible implementation options, and ensuring that the course of action taken was the best one. An hour of thinking could easily save many days of coding. In addition, the use of a regression test bench and design iterations was key in the success of the project. A designer could implement a small, atomic part of any given piece of functionality and ensure that all existing functionality remained unbroken. The process repeats until a feature is fully implemented, and the regression test bench augmented to test the functionality. The designer's code is synchronized with the version control repository, and a new feature is implemented using the same process until the design is complete. This design model provides several advantages to the designer. When a bug is introduced, it is immediately obvious, as the test bench will fail. Since the design iterations can usually be made very small, bugs are easy to spot and fix in the minimally changed code. This also precludes the need for bug tracking in the design phase, as code that does not run the test bench without failure is never committed.

The most difficult part of this design model is the difficulty in developing the test bench. "Correct behavior" often changes greatly with the development of the chip. Without a correct model of the CPU, code is developed for the test bench without regard for strict correctness, and when an exception is added, earlier code will fail, because it is now invalid. Implicit trust in the test bench, however, leads the designer to feel that the flaw lies in the code added, and that she or he somehow "broke the chip".

In addition, the tool chain used in the development process, from code editors to simulators, is often quite poor. The simulator is unnecessarily slow (while the simulation process is understandably computationally intensive, the GUI, responsible for a relatively small subset of data to display at any given moment should be much faster). Advanced features found in modern software development tools such as auto complete features in IDEs such as Microsoft Visual Studio, and the open-source Eclipse, would be welcome in HDL development. Additionally, the state of the debugger (the waves drawn and position in time) is unnecessarily reset with the testing of new code. In general, while experienced designers likely have gotten used to their tools, tools with usability and performance in mind from the very start could probably improve productivity greatly in the highly competitive realm of hardware development. A more detailed study of this could make a very interesting research project at HMC.

8.4. Validation Microarchitect - Matt Totino

Coming into the project I had not worked extensively with processor design of any sort, or really on any HDL coding project of this scale. It took me a while to get up to speed, but in the process I learned a lot about the MIPS architecture overall and, more specifically, a lot about some of the trickier issues in processor design like hazards and exceptions. This project was also my first time working in a team environment where everyone had to track changes made in the source code using the versioning tools in order to contribute their pieces. The importance of keeping everyone on the same page of the development process was very apparent.

Also, in the area of testing, I learned how different modes of testing can contribute to a validation scheme that gives good confidence about the performance of the device. In particular, our method of using a test bench seemed quite useful, adding new elements to test new hardware and running tests on each iteration of the design. We caught a lot of issues early on that could have become serious bugs that would have affected not only the module being worked on, but other modules as well.

Chip Cluster

This document explains the layout and schematic of the 32-bit MIPS processor built by the Harvey Mudd College E158-VLSI class during the spring semester of 2007. The entire chip is separated into four units, Datapath, Controller, Memory and Coprocessor 0. Each of the units is further divided into blocks. This report provides documentations for creation of every unit and block in the processor

9. Preliminary FloorPlanning

Once the microarchitecture team has created the RTL Verilog describing the function and operation of the chip, a preliminary floorplanning for the chip was performed. During this step, a rough estimate of the organization, size and connectivity between the different units were planned. The information and documentation produced during this step was used as a foundation for the detailed floorplanning performed on each block of the chip by individually block owners.

The fundamental limitation of this Chip is the layout area. In order for the chip to be manufactured, the chip has to fit on a 4 mm x 4 mm area in a 0.3 μm process. An additional 0.5mm space was allowed on the length and width of the chip, however the team performed the overall planning of the chip with 4mm per side in mind.

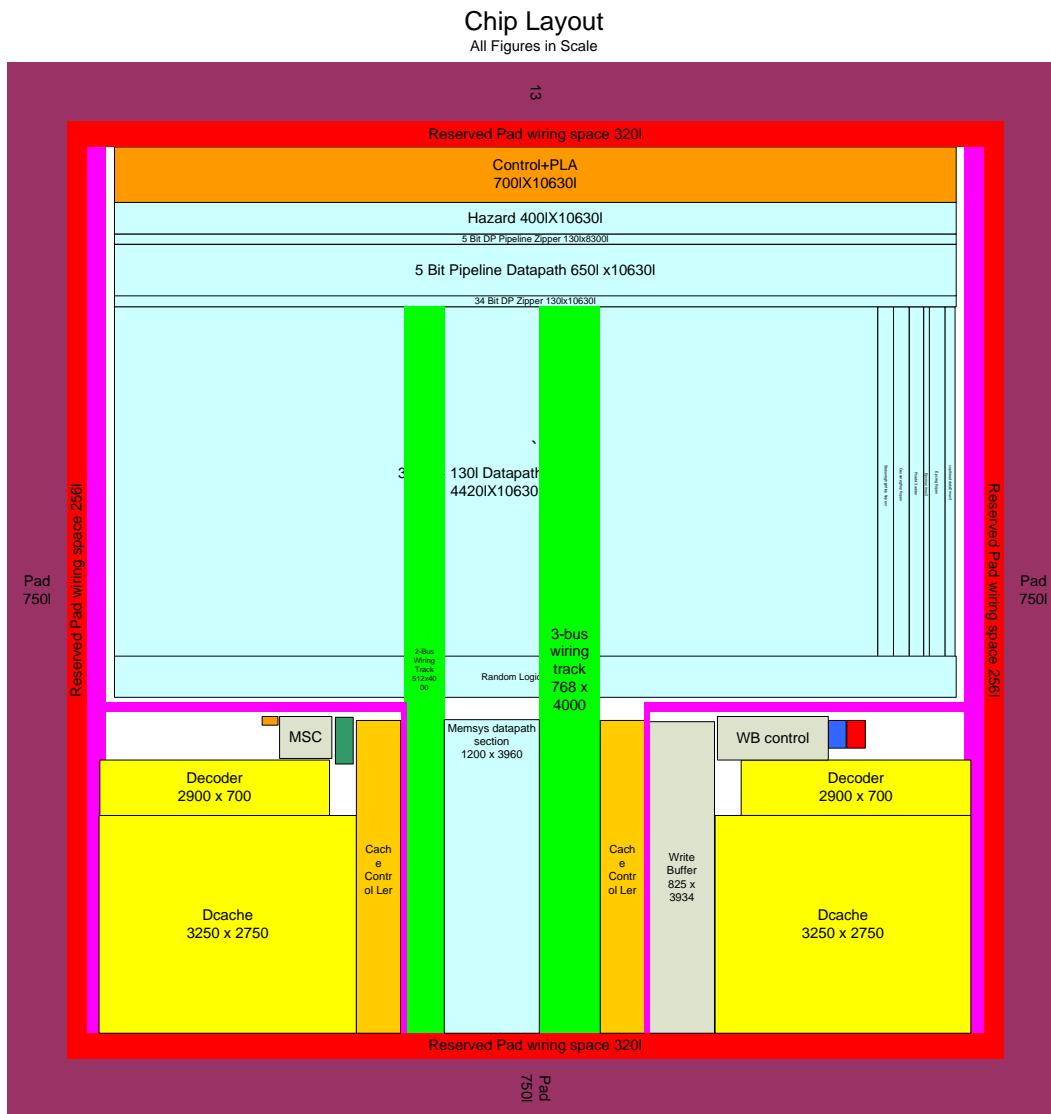
The exact procedure and result of the preliminary floorplanning phase is included in Section 17.1 Preliminary Floorplanning. As the project progressed, much of the predictions made by preliminary floorplanning became obsolete. However, this process did provide the team with valuable information on the feasibility of the chip as well as other fundamental constraints on the chip such as the size of the cache.

10. Detailed Floorplanning

Once preliminary Floorplanning was complete, each unit managers assigned blocks in their unit to their block owners who are responsible for completing those blocks from floorplanning to the final layout.

Each block owner first created a detailed floorplan for their unit, that including an estimate of area, the type of gate used in their block, and the structure of the blocks. These individual block plans were then merged by the unit managers into unit floorplan. Finally the chip cluster manager combined the units into the detailed floorplan for the entire chip.

Through out of the project, numerous minor changes were added to the floorplan. As block owners dove deeper into their blocks, the size estimates they have chosen before often no longer represent their need. All these changes combined have significantly altered the overall look of the chip from the first draft to the actual floorplan implemented on the chip. The figure below shows the final detailed floorplan that matches almost identically to the final version of the chip:



11. Chip

This module contains the entire MIPS processor as well as the pad frames to provide for off chip communication. This is the topmost module for this processor.

11.1. Chip I/O

Input	Origin	Output	Destination
Interrupts[7:0]	Pads	Memadr[31:0]	Pads
Memdone	Pads	Memdata[31:0]	Pads
Ph1	Pads	Membyteen[3:0]	Pads
Ph2	Pads	Memrw	Pads
Reset	Pads	Memen	Pads

Table 1. Input/Outputs of Chip

11.2. Special Unit

The pads for the chip were generated using the pad generator in Electric; it is further explained in later sections

11.3. Chip schematic

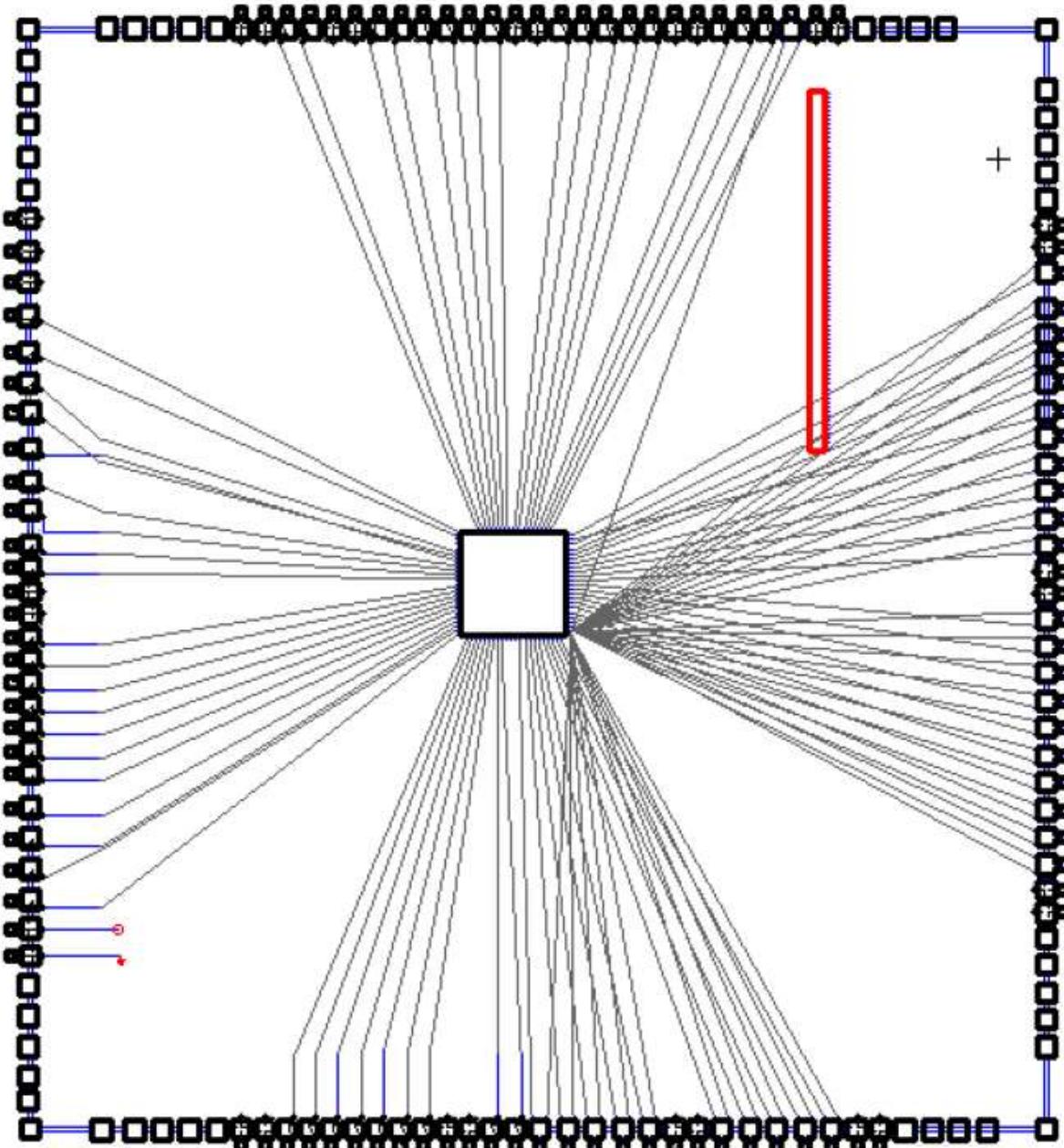


Figure 2. Chip Schematic

11.4. Chip layout

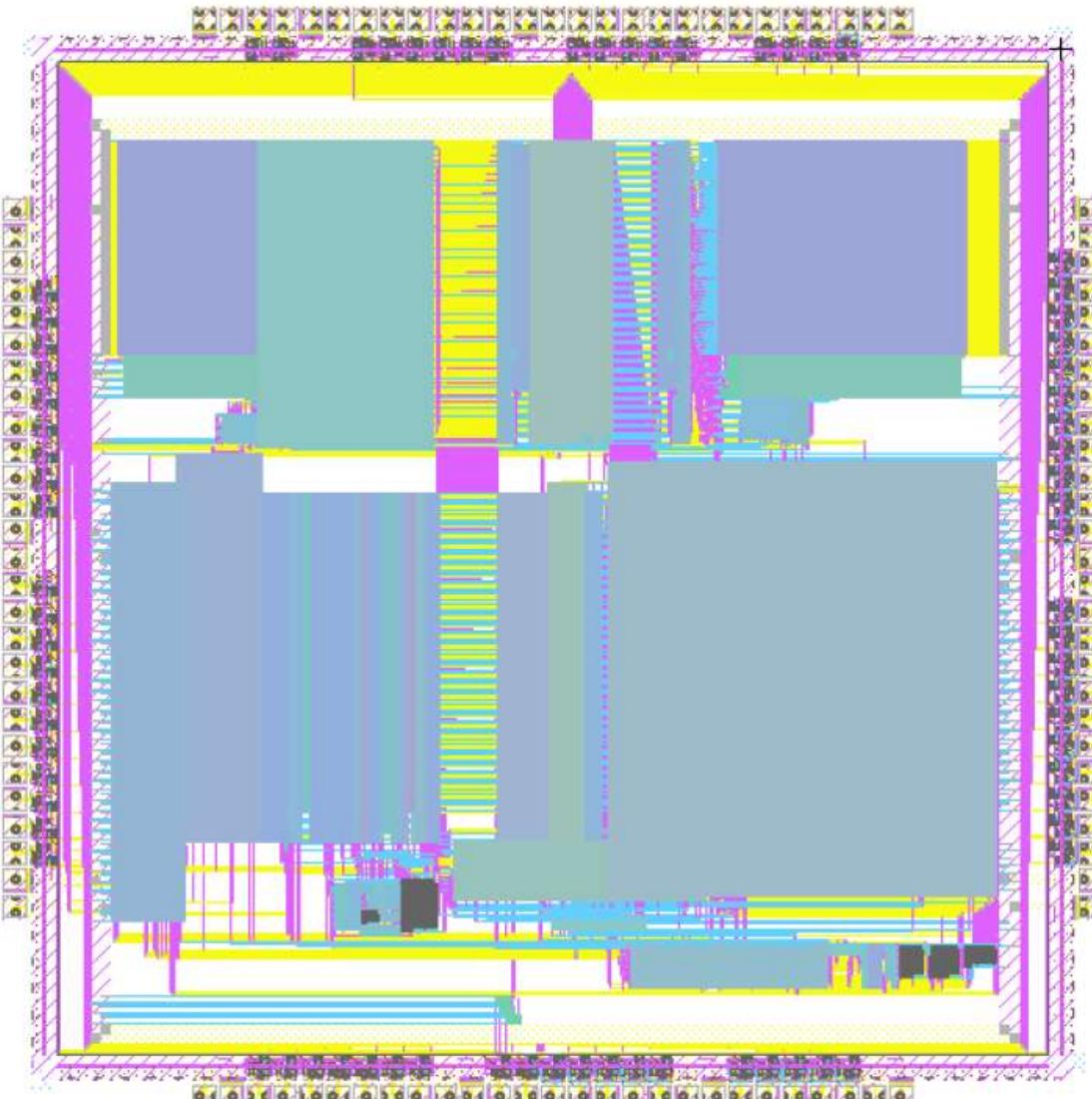


Figure 3. Chip Layout

11.5. Chip Testing

After the schematic and layout of the chip had been finalized, assortments of tests were run on the chip before it was ready for tape out.

The schematic of the chip first passed the DRC test. It was then converted into Verilog using Electric's built-in Verilog generator. This file was then inserted into the top.v files from the RTL Verilog by the microarchitecture team. Then the entire test suite created by the microarchitecture team was performed on the generated Verilog using Modelsim. The schematic Verilog passed all 27 tests.

The layout of the chip passed DRC, NCC, and ERC check prior to the insertion of the Logos. With the Logos there are 27000+ DRC errors. However, these errors are only internal to the logos and did not create any conflict with the chip layout. When the chip is converted to the CIF file format 24000+ CIF error were detected. These errors are created by invisible traces that falls on quarter λ grids while the CIF resolution is permanently set on half λ . These errors were not corrected because there are so many and should not significantly impact the chip and of them. Every standard cell in muddlib contains over 100 of them. In the future, it is recommended that a “CIF” test be performed alongside DRC, ERC, and NCC tests for the layout.

11.6. Chip subcomponents

11.6.1. Core

Function

The module was created to provide another level of the chip for the clock/reset buffer to be place. Thick VDD and GND wires that consolidate all of the power required of the chip are also added to the layout of this module and ready to be connected to the pad of the chip.

I/O Table

All I/O for this module is identical to that of topmips

Special Units

No additional special circuit were used in this module

Schematic

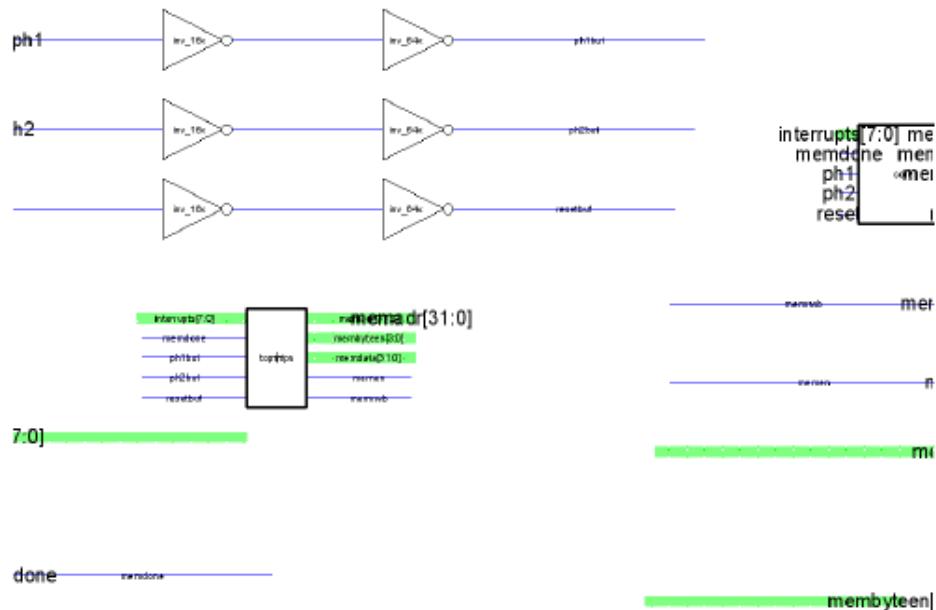


Figure 4. Schematic

Layout

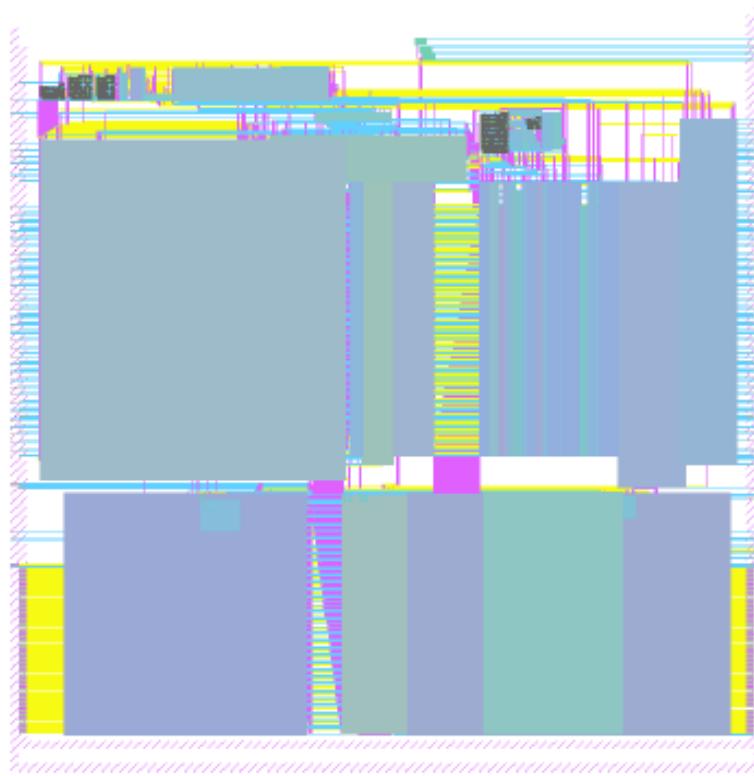


Figure 5. Layout

Testing

Since no significant changes were made between this module and the topmips module, block testing was not performed on this module.

11.6.2. TopMIPS

Function

This module contains all function unit of the MIPS processor. It has all the functionality of a MIP processor but lacks any connection to external devices.

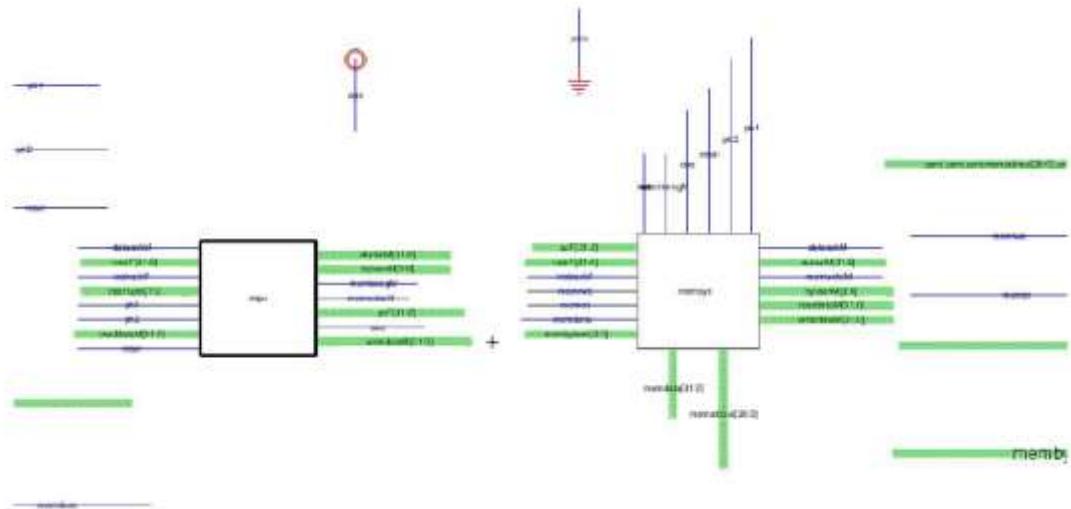
I/O Table

Input	Origin	Output	Destination
Interrupts[7:0]	Pads	Memadr[31:0]	Pads
Memdone	Pads	Memdata[31:0]	Pads
Ph1	Pads	Membyteen[3:0]	Pads
Ph2	Pads	Memrwb	Pads
Reset	Pads	Memen	Pads

Special Units

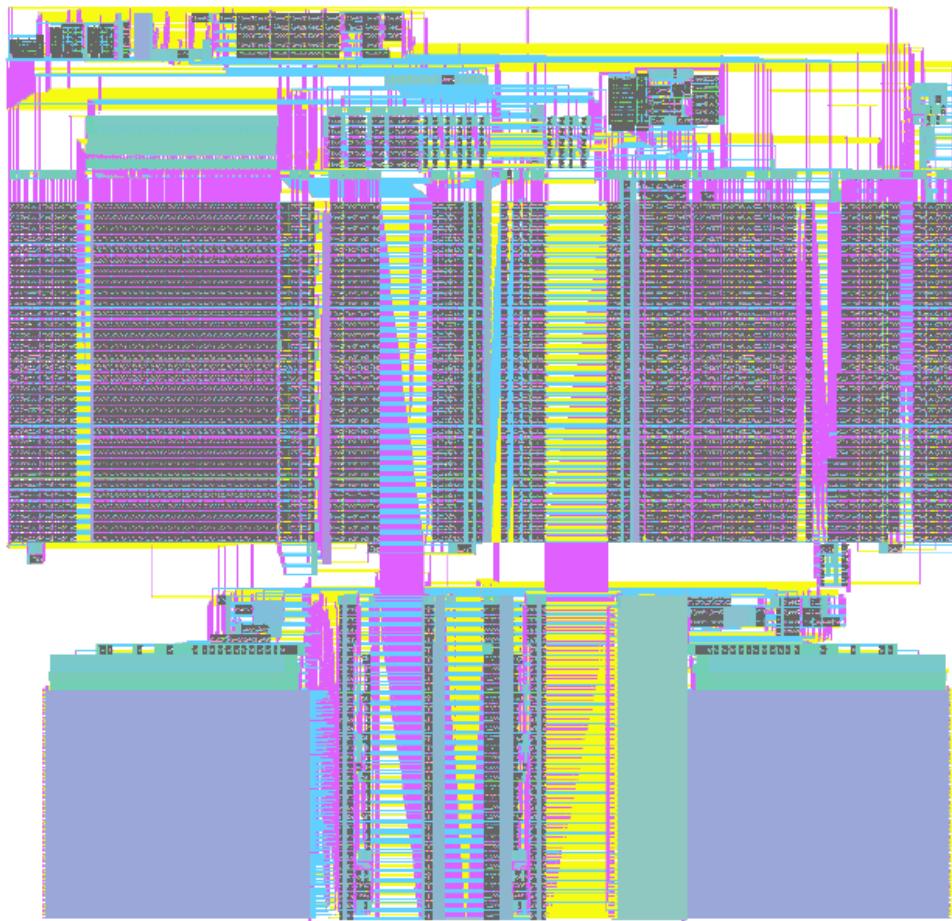
No additional special circuit were used in this module

Schematic



The module is created by linking the MIPS processing module with the memory module. This was done to how the RTL for this chip was written and facilitates the simulation of the chip.

Layout



Testing

This module was converted into Verilog netlist through Electric. The generated Verilog code then replaced the topmips module in the top.v file. The set of 27 test vectors from the microarchitecture team were ran the processing. All 27 tests passed.

11.6.3. Pad frame

Function

The pad frame for the chip provides a link between the small chip exports and large external connection wires.

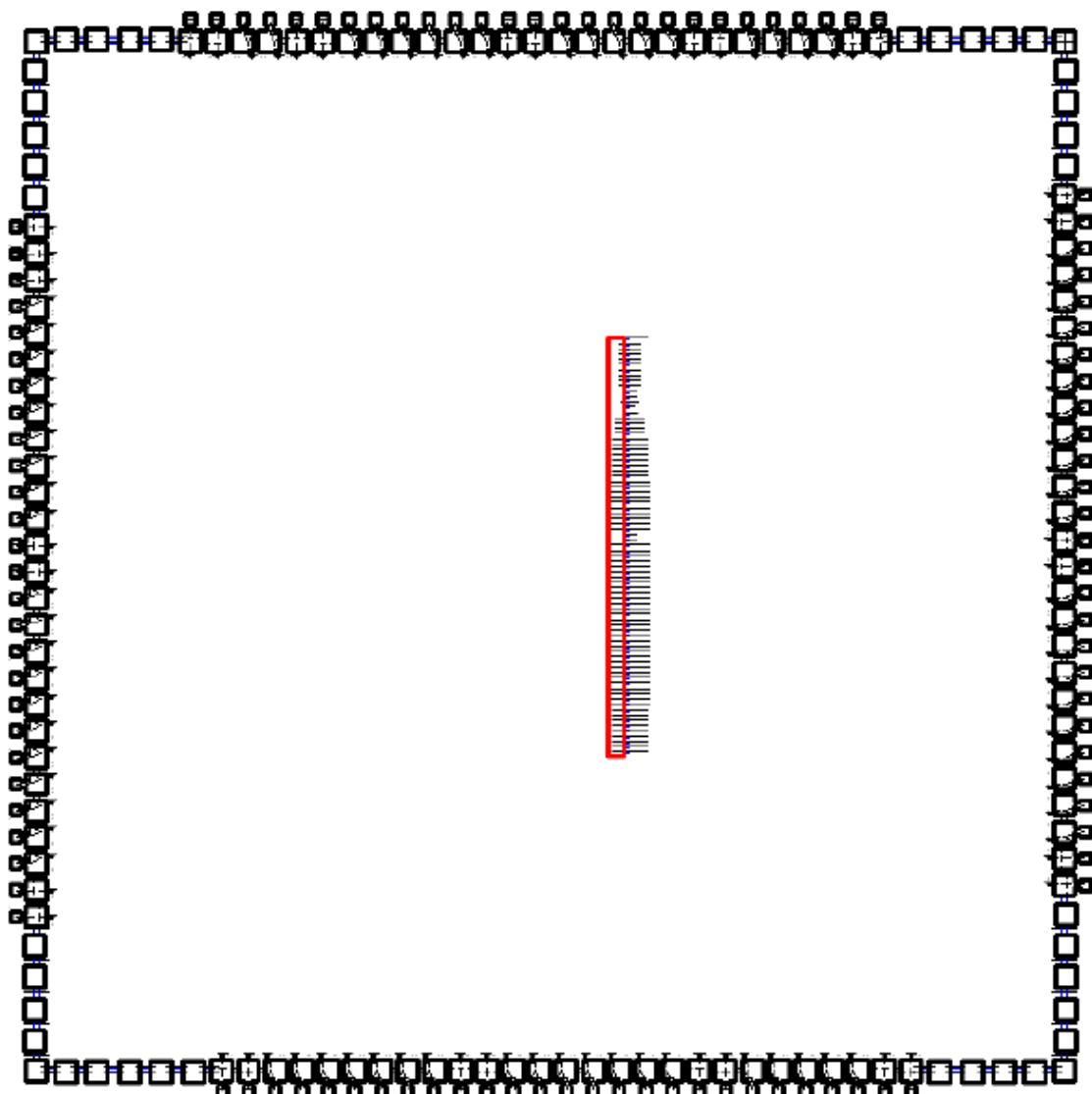
I/O Table

All IO for the pads are identical to the I/O of the chip.

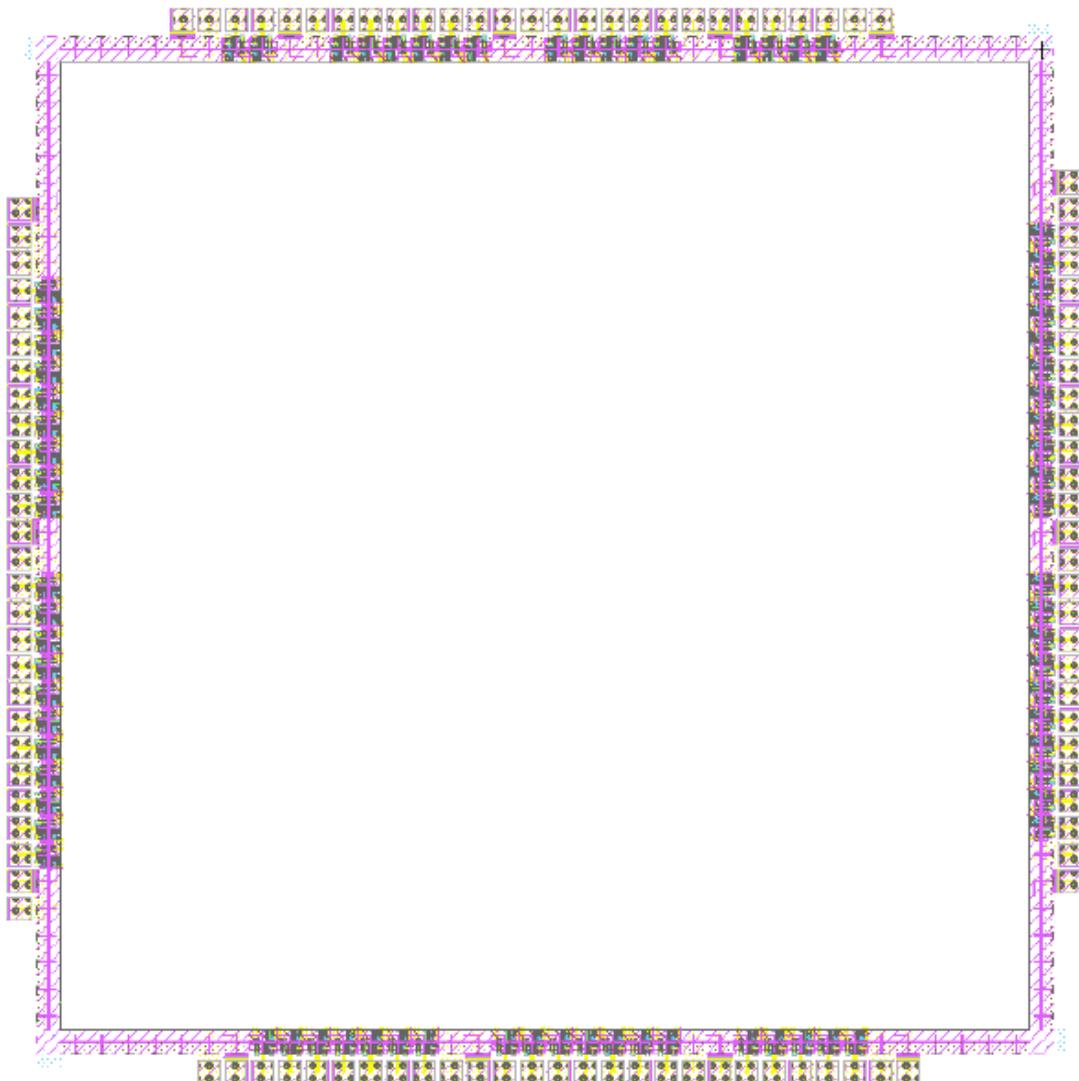
Special Units

This unit was generated with the pad generator in Electric. The content of the file is shown in Section 17.2 Pad generation File.

Schematic



Layout



Testing

No other testing beyond layout tests were performed on this module

11.6.4. MIPS

Function

This module combines the Control, Datapath, and Coprocessor 0 of the chip. This module was created to facilitate chip testing as the microarchitecture team put these 3 units in a block and the Memory module on a separate block.

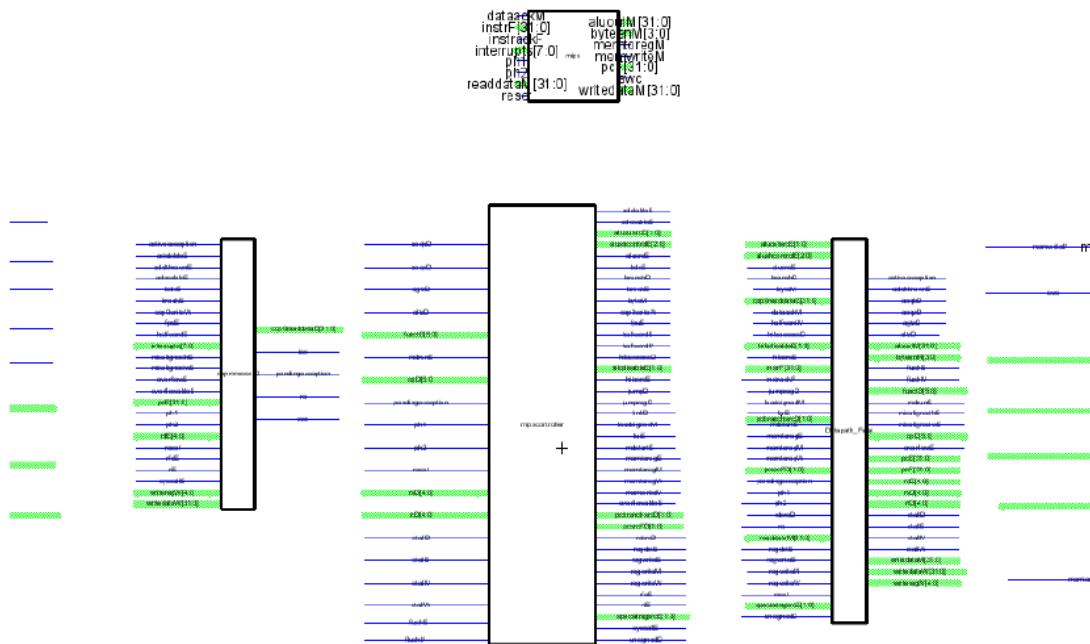
I/O Table

Input	Origin	Output	Destination
readdataM[31:0]	Memory	pcF[31:0]	Memory
InstrF[31:0]	Memory	AlutoutM[31:0]	Memory
Interrupt[7:0]	Chip	WritedataM[31:0]	Memory
dataackM	Memory	byteenM[3:0]	Memory
InstrackF	Memory	Swc	Memory
Ph1	Chip	memwriteM	Memory
Ph2	Chip	memtoregM	Memory
Reset	Chip		

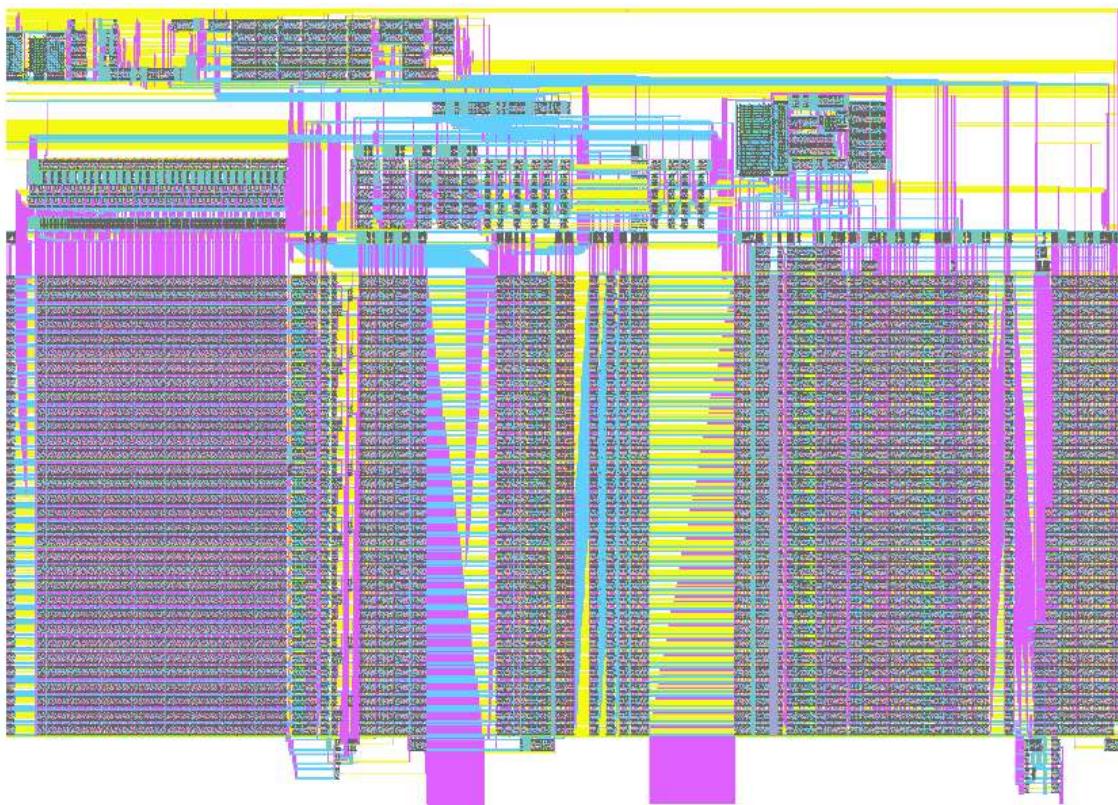
Special Units

No additional special circuit were used in this module

Schematic



Layout



Testing

The MIPS schematic was netlisted into Verilog using Electric. The generated Verilog was copied into the `mipspipelined.v` file and replaced all of the original RTL code. The file was then subjected to the microarchitecture test suit passing all 27 tests.

12. Datapath

12.1. Function

The MIPS datapath handles all of the arithmetic operations on the chip. These operations include addition, subtraction, multiplication, division, and their corollaries such as comparisons and equalities. The datapath operates in a pipeline to increase the rate of instruction execution. The datapath is divided into six subcomponents: Fetch, Decode, Execute, Memory-Writeback, Hazard, and Fivebit-datapath. Together these subcomponents accept instructions from the memory, hold and access the thirty-two MIPS registers, compute arithmetic operations, and stall registers when necessary.

12.2. I/O

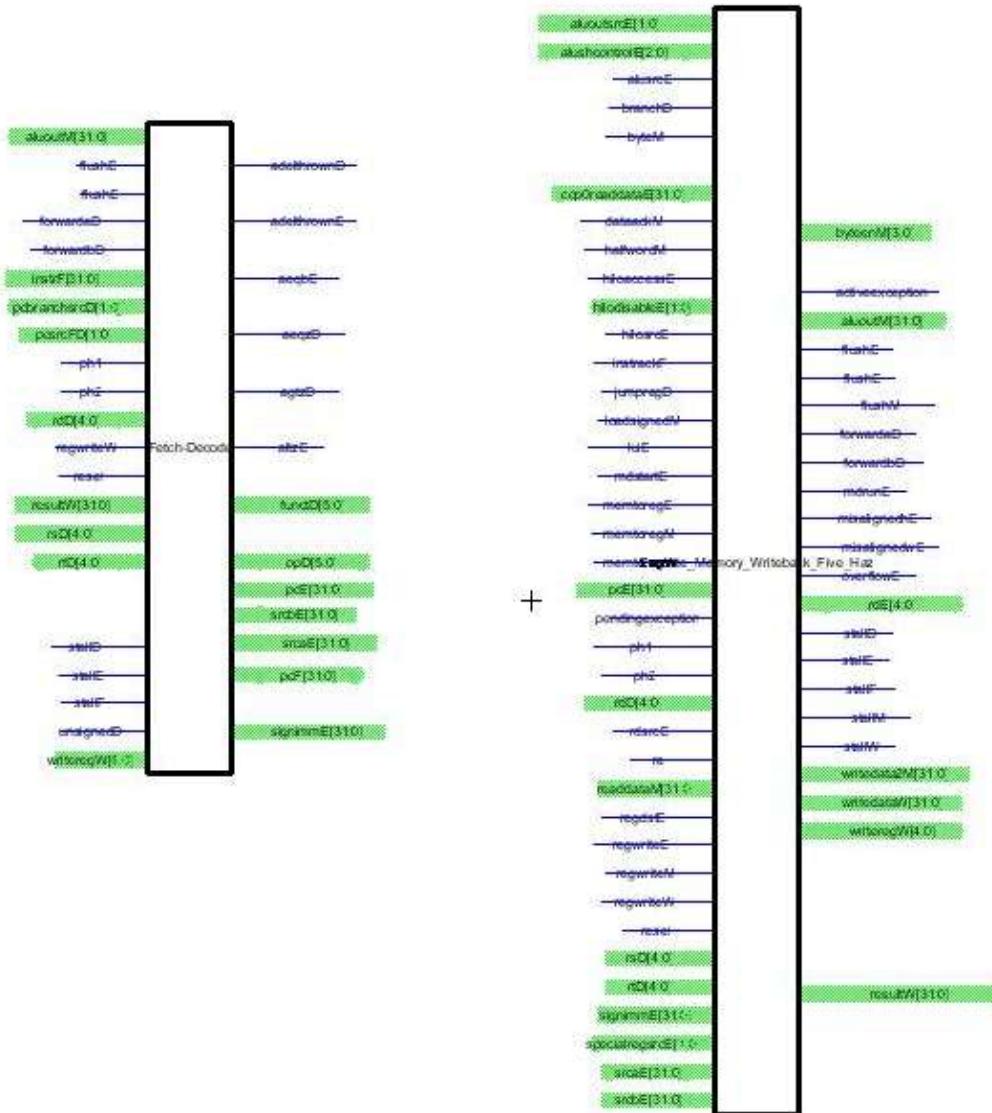
Input	Origin	Output	Destination
aluoutsrCE[1:0]	Controller	activeexception	Coprocessor
alushcontrolE[2:0]	Controller	adelthrownE	Coprocessor
alusrcE	Controller	aeqbD	Controller
branchD	Controller	aeqzD	Controller
byteM	Controller	agtzD	Controller
cop0readdataE[31:0]	Coprocessor	altzD	Controller
dataackM	Memory	aluoutM[31:0]	Memory
halfwordM	Controller	byteenM[3:0]	Memory
hi洛accessD	Controller	flushE	Controller
hilodisableE[1:0]	Controller	flushM	Controller
hilosrcE	Controller	functD[5:0]	Controller
instrF[31:0]	Memory	mdrunE	Controller
instrtrackF	Memory	misalignedhE	Coprocessor
jumpregD	Controller	misalignedwE	Coprocessor
loadsingedM	Controller	opD[5:0]	Controller
luiE	Controller	overflowE	Coprocessor
pcbranchsrcD[1:0]	Controller	pcE[31:0]	Coprocessor
mdstartE	Controller	pcF[31:0]	Memory
memtoregE	Controller	rdE[4:0]	Coprocessor
memtoregM	Controller	rsD[4:0]	Controller
memtoregW	Controller	rtD[4:0]	Controller
pcsrcFD[1:0]	Controller	stallD	Controller
pendingexception	Coprocessor	stallE	Controller
ph1	Chip input	sallM	Controller
ph2	Chip input	stallW	Controller
rdsrcD	Controller	writedataM[31:0]	Memory
re	Coprocessor	writedataW[31:0]	Coprocessor
readdataM[31:0]	Memory	writeregW[4:0]	Coprocessor
regdstE	Controller		
regwriteE	Controller		
regwriteM	Controller		
regwriteW	Controller		
reset	Chip input		
specialregsreE[1:0]	Controller		
unsignedD	Controller		

12.3. PLA Generation

The datapath unit has two PLAs in it. These PLAs are in the multdiv controller located in the execute stage which controls the multiplications. The PLAs were created with the PLA generator written by the library team. The PLA uses a pseudo-nMOS design, with weak pMOS pull-up transistors.

12.4. Schematic

The unit schematics were divided into the six subcomponents: Fetch, Decode, Execute, Memory-Writeback, Hazard, and Fivebit-Datapath. The top-level schematic is shown below illustrating the integrated subcomponents. Integration of the datapath occurred in a parallel manner to speed up the process. The schematics were integrated into small clusters of blocks which were NCC tested to match similar block clusters in layouts. These clusters were Fetch-Decode and Execute-Memory-Writeback, which were then integrated into Execute-Memory-Writeback-Hazard-Fivebit-Datapath, and finally the whole datapath was combined.



12.5. Layout

The unit layout is best understood when starting at the center of the left side of the datapath at the Fetch stage and moving horizontally right going through the Decode stage, two 32-bit busses connected to memory, the Execute Stage's ALU, the shifter, three 32-bit busses connected to memory, the remainder of the Execute stage including the multdiv section, and finally the Memory-Writeback stage. The multdiv controller is located above the multdiv section of the Execute stage. The hazard subcomponent is located left and up from the multdiv controller. The Fivebit-datapath is located directly below the hazard and appears much larger than the hazard.

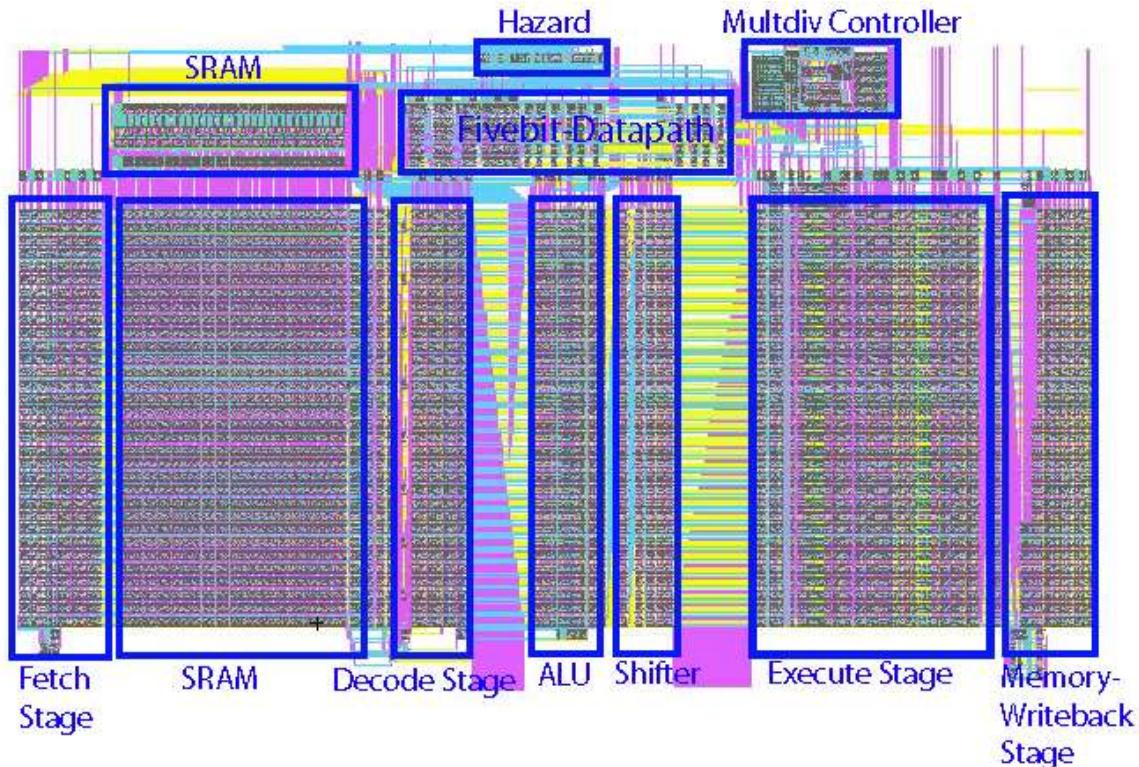


Figure 6. Datapath Layout with Subcomponent Names Labeled

12.6. Testing

The unit was tested in a Verilog test fixture. Twenty-seven chip test vectors were generated with the MIPS RTL Verilog code, and then run on the Verilog deck generated from the schematics created in Electric. The datapath Verilog decks replaced their equivalent modules in the original RTL code leaving the RTL code for the controller, coprocessor, and memory unchanged. The tests covered a variety of datapath functionality including: addi, add, sub, and, or, slt, sw, lw, beq, j, shift, multiply, divide, along with hazards. These tests are described in further detail at the chip level and in micro-architecture report.

12.7. Subcomponents

The subcomponents are organized in a logical manner tracing a horizontal path left-to-right through the center of the datapath with Hazard and the Fivebit-datapath last. Thereby the order is: Fetch, Decode, Execute, Memory-Writeback, Hazard, and Fivebit-datapath.

12.7.1. Fetch Stage

Function

The Fetch stage is responsible for determining what address the program counter is currently pointing to, and thereby which instruction will be executed at which time. It is part of the datapath unit, and is adjacent to the Decode stage; hence, many of the inputs and outputs are shared.

I/O Table

Input	Origin	Output	Destination
InstrF[31:0]	Off Chip	InstrD[31:0]	Decode
pcnexbrFD[31:0]	Decode	pcD[31:0]	Decode
pcsrcFD[1:0]	Controller (Branch)	pcF[31:0]	Off Chip
flushD[0:0]	Hazard	pcplus4D[31:0]	Decode
stallF[0:0]	Hazard	adelthrownD[0:0]	Decode
stallD[0:0]	Hazard		

Table 2. A list of all the inputs and outputs from the unit, and their origin/destination.

Special Units

The only new cells created for this block was a 1-bit flopencr, and a 1x invbuf to drive the inputs of said flopencr.

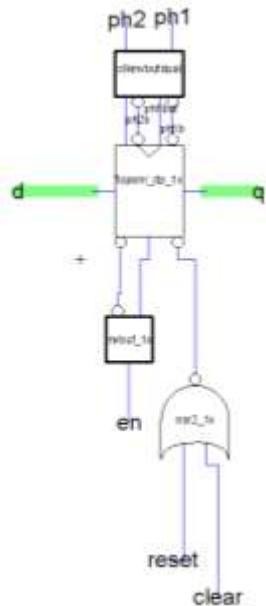


Figure 7. fopenrc_1x_1{sch}

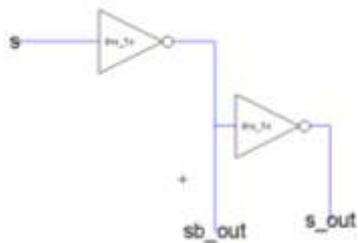


Figure 8. invbuf_1x

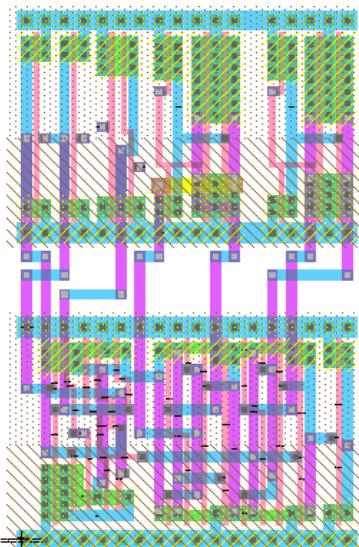


Figure 9. fopenrc_1x_1{lay}

Schematic

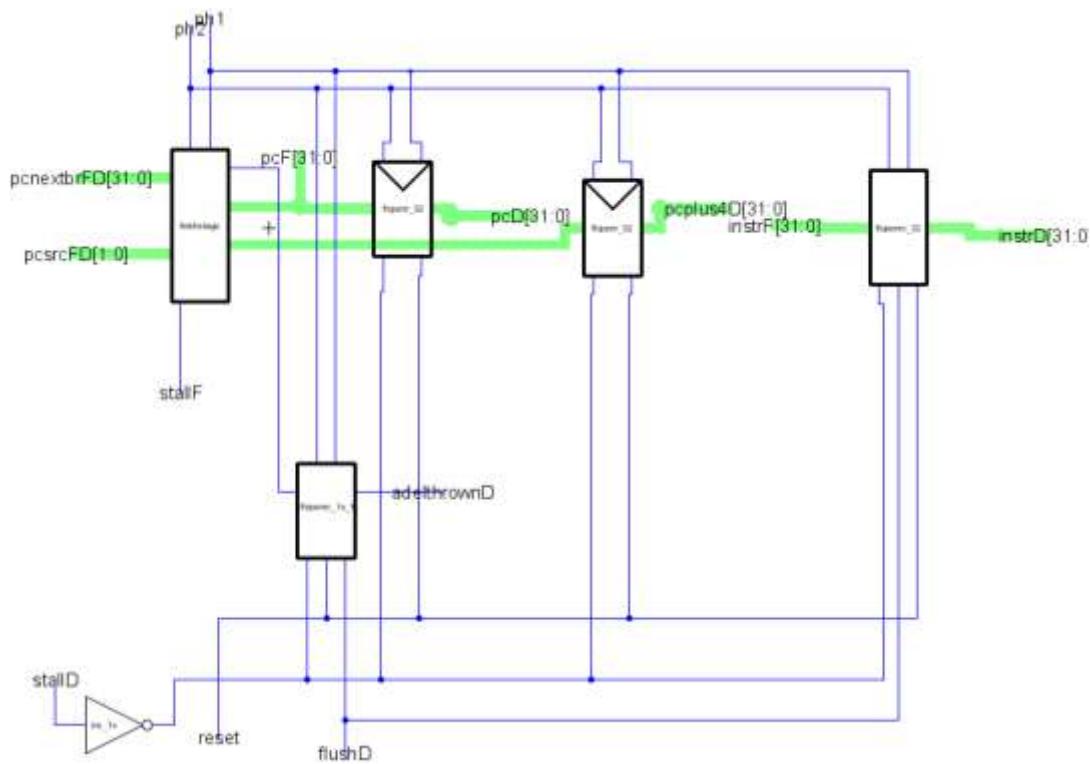


Figure 10. Fetch_Stage{sch}

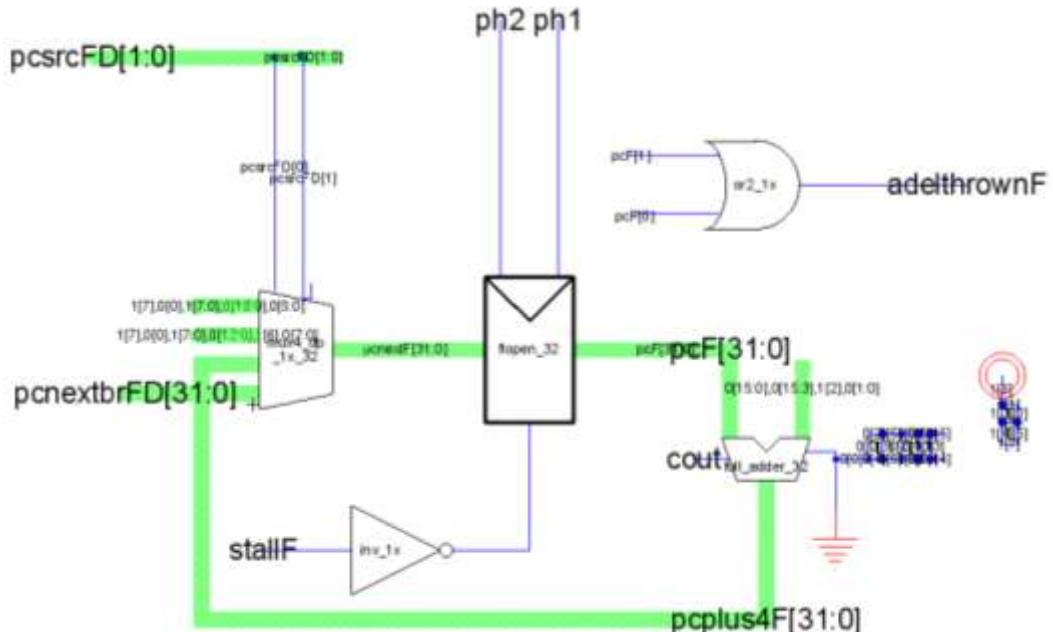


Figure 11. fetchstage{sch}

Layout

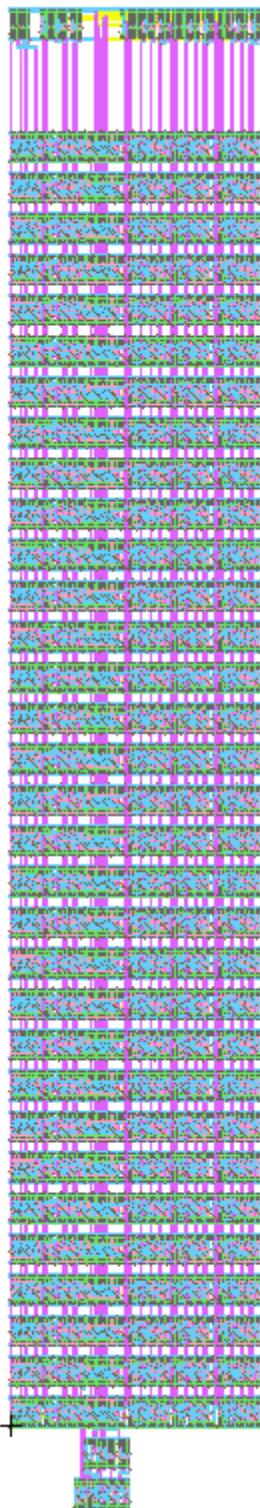


Figure 12. Fetch_Stage{lay}

Luckily there were enough open tracks of metal-2 in the flopopen and adder layouts to run the control tracks for the adelthrownF logic through the layout with no issues.

Testing

The following code was run on the Verilog generated from the fetchstage schematic:

```
// fetchstagetest.v
// dlindblad@hmc.edu 2-27-07
//
/////////////////////////////// Module: fetchstagetest ///////////////////
// Test fixture for Fetchstage module of pipelined MIPS processor
// Tested 20 January 2007
/////////////////////////////// /////////////////////////////////////////////////
module fetchstagetest();
    reg          ph1, ph2;
    reg          reset, stallF;
    reg [1:0]    pcsrcFD;
    reg [31:0]   pcnextbrFD;
    wire [31:0]  pcplus4F,pcF;
    wire adelthrownF;

    reg [31:0]  vectornum, errors;
    reg [127:0] testvectors[10000:0];
    reg [31:0]  pcFexpected, pcplus4Fexpected;
    reg adelthrownFexpected;
    reg          ready, start;

    // device under test
    fetchstage dut(.pcnextbrFD(pcnextbrFD), .pcsrcFD(pcsrcFD), .ph1(ph1), .ph2(ph2),
                  .reset(reset), .stallF(stallF), .adelthrownF(adelthrownF), .pcF(pcF),
                  .pcplus4F(pcplus4F));

    // generate clock
    always begin
        ph2 = 1; #4; ph2 = 0; #1;
        ph1 = 1; #4; ph1 = 0; #1;
    end

    // generate reset
    initial begin
        start = 1;
        reset = 1; #27; reset = 0;
        start = 0;
        ready = 1;
    end

    // load testvectors
    initial begin
        $readmemh("fetchstage.tv", testvectors);
        $display("test vectors read");
        vectornum = 0;
        errors = 0;
    end

    always @(posedge ph1)
        if (ready) begin
            #1 {pcnextbrFD, pcsrcFD, stallF, adelthrownFexpected, pcFexpected,
                 pcplus4Fexpected} = testvectors[vectornum];
            $display("passed vector number %d", vectornum);
            vectornum = vectornum+1;
            start = 1;
            ready = 0;
        end
        else if (~reset) begin
            #1 start = 0;
        end
    end

    always @(posedge ph2)
        if (~start&~ready) begin
            #1 ready = 1;
        end
    end

```

```

if (testvectors[vectornum][0] === 1'bx) begin
    $display("Finished %d test vectors with %d errors\n", vectornum, errors);
    $stop();
end
$display("Passed vector %d: pcnextbrFD = %x, pcsrcFD = %x, stallF= %x\n Expected (%x %x
%x) observed (%x %x %x)\n",
vectornum-1, pcnextbrFD, pcsrcFD, stallF, adelthrownExpected,
pcFexpected, pcplus4Fexpected, adelthrownF, pcF, pcplus4F);
if (adelthrownF !== adelthrownExpected || pcF !== pcFexpected ||
pcplus4F !== pcplus4Fexpected) begin
    $display("ERROR!");
/*$display("Error on vector %d: pcnextbrFD = %x, pcsrcFD = %x, stallF= %x\n Expected (%x
%x %x) observed (%x %x %x)\n",
vectornum-1, pcnextbrFD, pcsrcFD, stallF, adelthrownExpected,
pcFexpected, pcplus4Fexpected, adelthrownF, pcF, pcplus4F);*/
errors = errors+1;
end
end
endmodule

```

That code was used with these test vectors:

```

//fetchstage.tv
//testvectors for FetchStage of MIPS pipelined processor
//
//Format:
// [pcnextbrFD, pcsrcFD, stallF, adelthrownExpected, pcFexpected, pcplus4Fexpected]
//
//
FFFFFFFFFF_C_FFFFFFC_00000000
11110000_C_11110000_11110004
11111114_A_11110000_11110004
FCABDEFC_4_BFC00100_BFC00104
FCABDEFC_0_BFC00000_BFC00004
FCABDEFC_C_FCABDEFC_FCABDF00
FCABDEFD_E_FCABDEFC_FCABDF00
FCABDEFD_D_FCABDEFD_FCABDF01
FCABDEFF_D_FCABDEFF_FCABDF03
00000000_F_FCABDEFF_FCABDF03

```

This code was designed to test the fetchstage cell; however, other tests were run on the complete Fetch Stage. After it was verified that the schematics were correct, the layouts were checked by verifying that they passed DRC, ERC, and NCC.

12.7.2. Decode Stage

Function

The Decode stage splits up and organizes, (perhaps we could just say that it “decodes”) the incoming outputs from the instrD, reg file and aluoutM and additionally performs comparisons, branch calculations, etc. The decode stage (along with its reg file) is in the datapath and sits between the fetch stage and the execute stage. As designed it has over 17000 networks covering 20,000,000 square λ which is under the original estimate.

Input	Origin	Output	Destination
aluoutM[31:0]	Exicute	aeqbD	Controller
clear	Chip	aeqzD	Controller
en	Chip	agtzD	Controller
forwardaD	five-bit datapath	altzD	Controller
forwardbD	five-bit datapath	functD[5:0]	Controller
instrD[31:0]	Fetch	opD[5:0]	Controller

pcbranchsrcD[1:0]	Controller	pcnextbrFD[31:0]	Controller
pcplus4D[31:0]	Fetch	rdD[4:0]	five-bit datapath
ph1	Chip	rsD[4:0]	five-bit datapath
ph2	Chip	rtD[4:0]	five-bit datapath
rewriteW	five-bit datapath	signimmE[31:0]	Execute
reset	Chip	srcaE[31:0]	Execute
resultW[31:0]	Mem-writeback	srcbE[31:0]	Execute
unsignedD	Controller		
writeregW[4:0]	five-bit datapath		

Special Units

Two comparators, A equal to B and a equal to zero were created for this unit. They are shown below.

A equal to B :

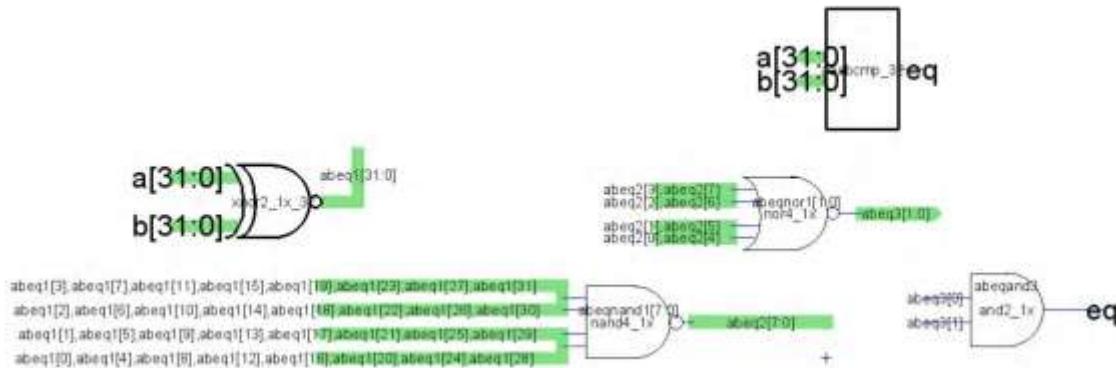


Figure 13. $A == B$ Comparator {ic} {sch}



Figure 14. $A == B$ Comparator {lay}

A equal to Zero:

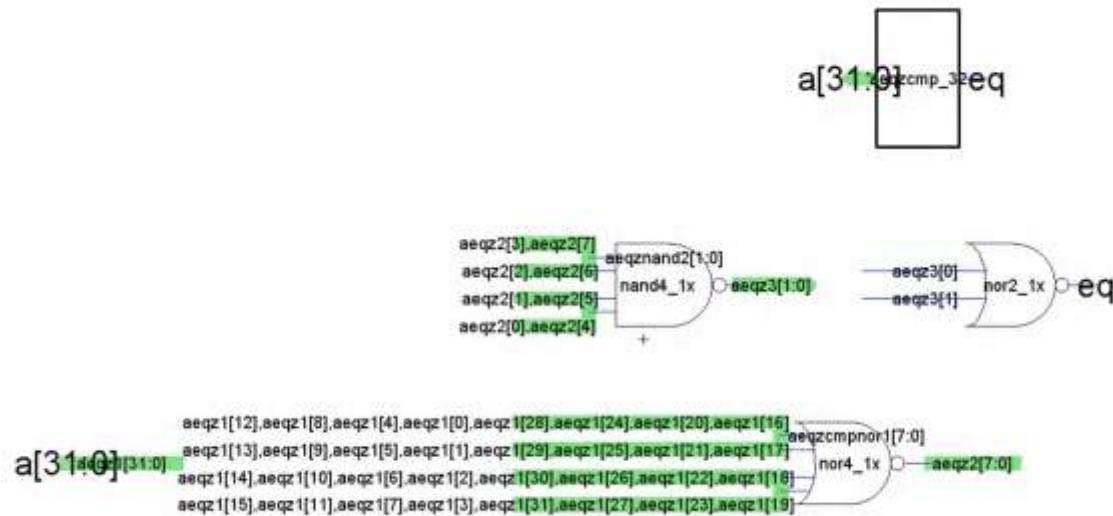


Figure 15. $A==0$ Comparator {ic} {sch}



Figure 16. $A==0$ Comparator {lay}

Additionally a unit called signext32 was created to sign extend a 16 bit number to 32 bits. This was nothing more than a 4x AND gate to help drive all the outputs, but due to peculiar, buggy behavior in Electric this unit had to be created. The bug in question creates connection and DRC errors within Electric when more than one wire in a bus is pulled from a single source. In this case, the most significant bit of the 16 bit input was to be extended another 16 bits, however Electric would not allow all of the distinctly named networks (signext[15] to signext[31]) to share a common source. Apparently Electric has a hard time checking when multiple bits from one bus are connected to a single driving bit from another bus. The workaround was to create signext32 which takes in an enable and the most significant bit of the number to be extended and produces well powered (4x) output that can then be assigned to bits 15-31 manually.

Schematics

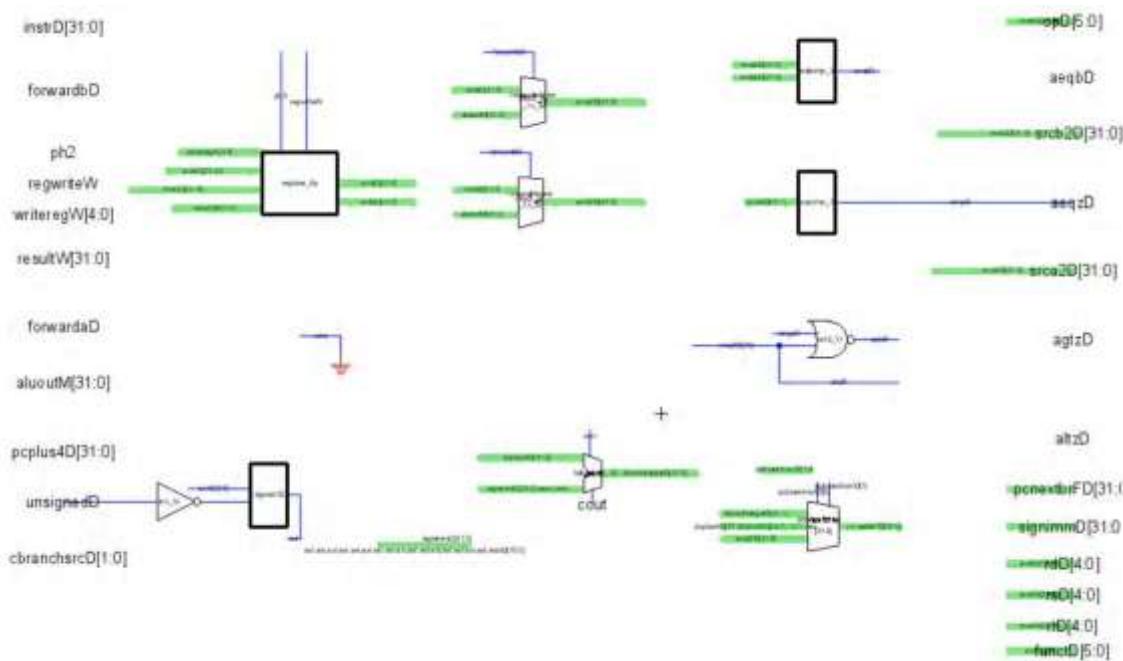


Figure 17. Decode{sch}

Icon

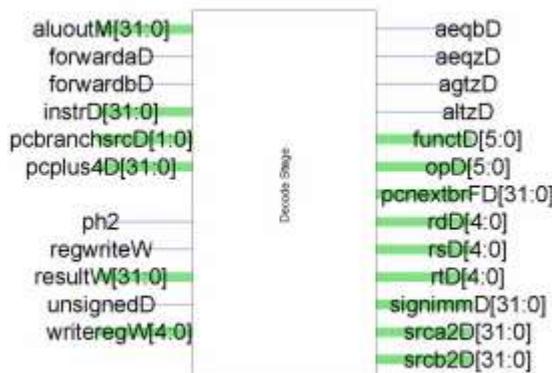


Figure 18. Decode {ic}

Layout

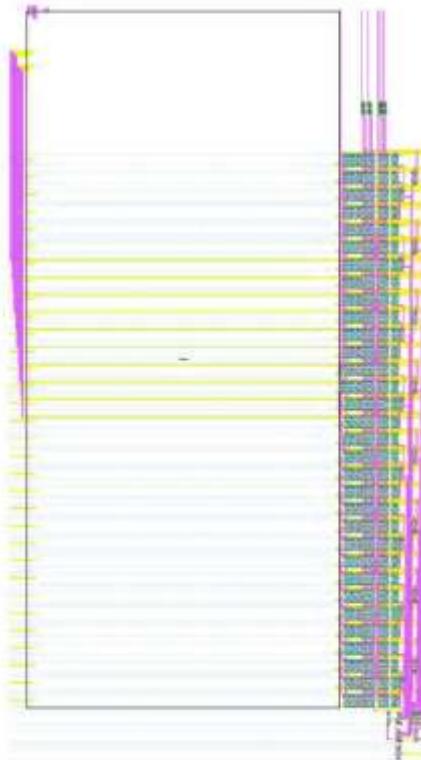


Figure 19. Decode {lay}

Testing

To test these blocks we used Electric to generate a Verilog deck of the design and then used existing chip tests we knew to be valid. (We replaced the original Verilog for the decoder with the Electric generated copy and ran the tests) Additionally the design passes DRC, ERC, and NCC checks within Electric.

Note: The decode stage has a set of flops between it and the execute stage and the following figures show their placement.

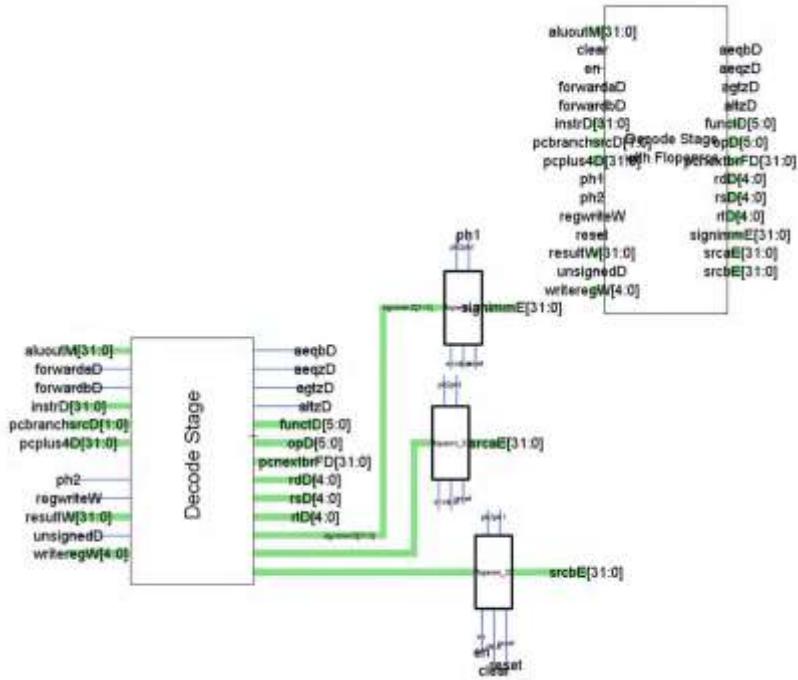


Figure 20. Decode with following flops {sch} {ic}

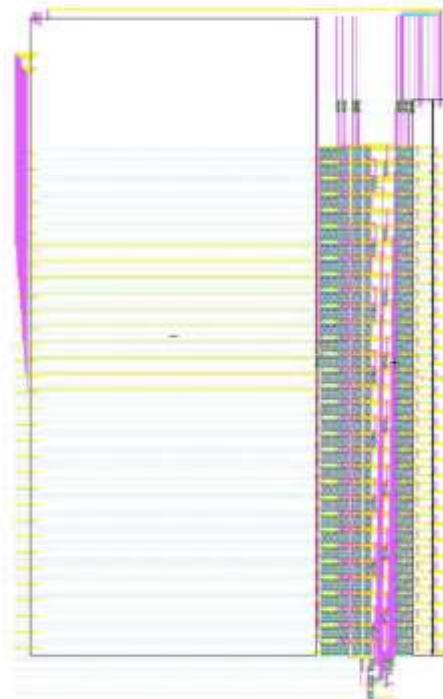


Figure 21. Decode with following flops {lay}

12.7.3. Register File SRAM Array (in Decode Stage)

Function

The register file SRAM array is part of the Decode stage. It is bordered on the left by the Fetch stage, on top by the register decoder, and on the right by the rest of the Decode stage.

The register file stores 32-bit numbers for use in the execution of commands. One register can be written and two registers can be read (or one register can be read twice) in each clock cycle.

I/O Table

Input	Origin	Output	Destination
write[31:1]	Register Decoder	r1[31:0]	Decode Stage
writeb[31:0]	Register Decoder	r2[31:0]	Decode Stage
read1[31:0]	Register Decoder		
read1b[31:1]	Register Decoder		
read2[31:0]	Register Decoder		
read2b[31:1]	Register Decoder		
w[31:0]	Writeback Stage		

Special Units

A custom 16-transistor SRAM cell was developed to be a reliable digital SRAM cell that would not require testing as an analog circuit. It has two read ports and one write port. The cells are arrayed vertically into RAM vectors, and the vectors are arrayed horizontally to form the array.

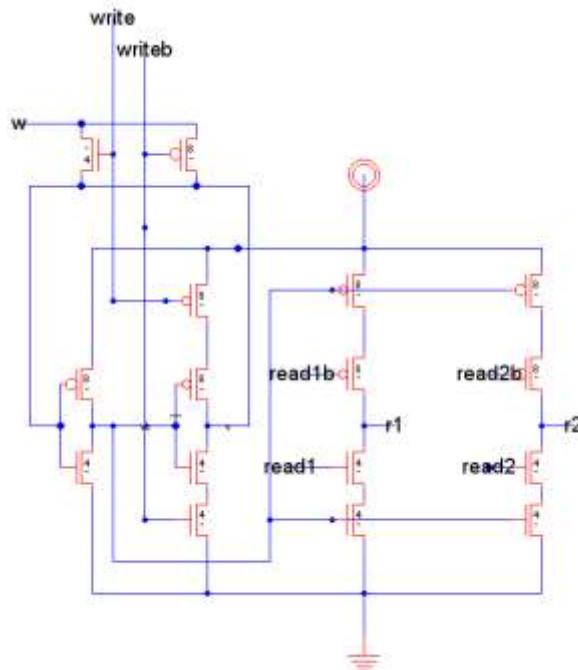


Figure 22. SRAM bit schematic.

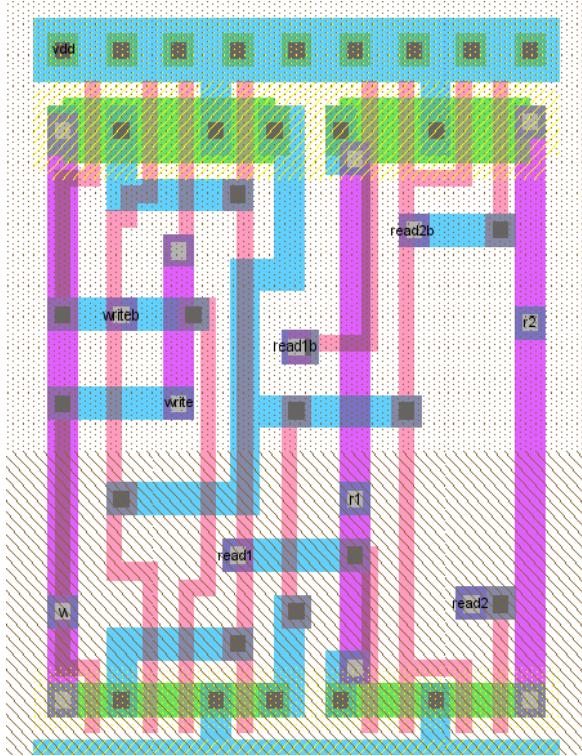


Figure 23. SRAM bit layout.

The zero register has a different “SRAM” cell than the others (regram_dp_bit0) which returns 0 if the read lines are high, but cannot be written.

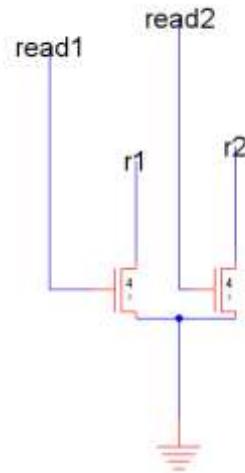


Figure 24. Bit 0 schematic.

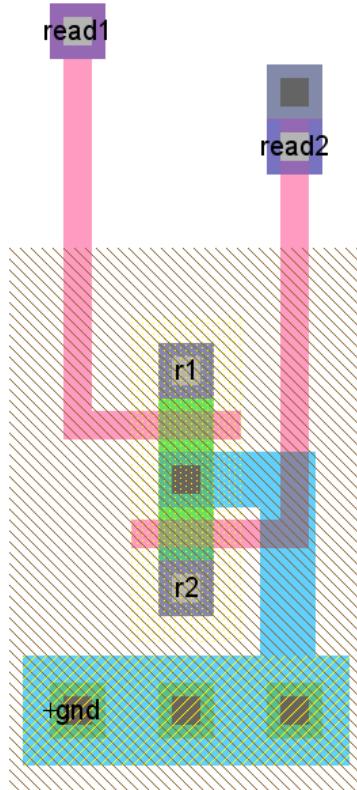
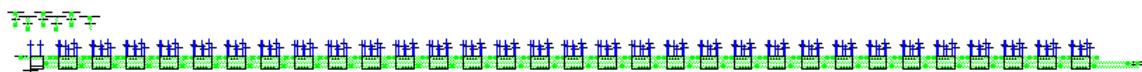


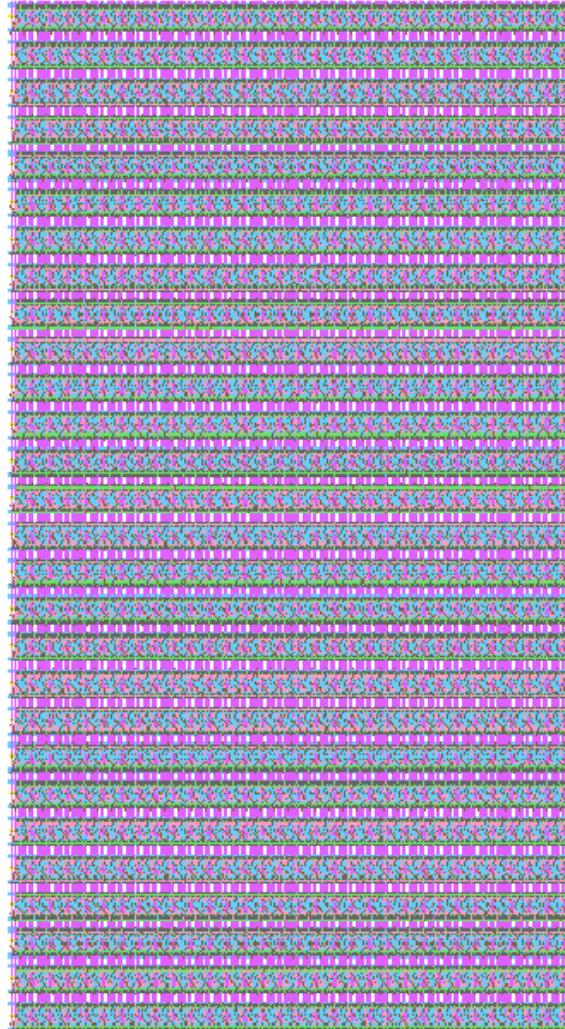
Figure 25. Bit 0 layout.

Schematic



The hierarchical build of the RAM array made layout straightforward.

Layout



The hierarchical build of the register RAM array made this layout much more manageable. Note that when arraying exports, under Preferences>General>Nodes, make sure “Duplicate/Array/Paste copies exports” is checked. Also note that it is better (for a large block with many sub-blocks) to create a set of new metal1 pins to the left of the block and export those as powers and grounds than it is to “re-export power and ground”.

Testing

The schematic for the register was turned into a Verilog file and simulated in place of the original Verilog “register”; the chip with this register passed 27 self-checking test vectors.

12.7.4. Register Array Decoder

Function

The reg_decoder block is above the register file for the datapath in the Decode stage. It takes in the 5 bit addresses of the registers to read and write, and produces the one-hot encoding of the address to activate the appropriate bitlines of the SRAM array that makes up the register file. In order to write, the RegWrite control signal needs to be high, and ph2 of the clock must be high: If the register file were written in ph1, the value from the previous instruction in the pipeline would begin to be written, and then later the new value from the current instruction would be written, so the correct value might not be written to the register. To avoid this, writing and reading occurs in ph2. Reading is not clocked, so as long as the writing is done and the correct read value passes through the slave latch (ph2 controlled latch) before ph2 falls, writing and reading in the same phase does not affect the system. If the clock speed is too fast, reading and writing in the same phase may not occur quickly enough, and the chip will not function properly.

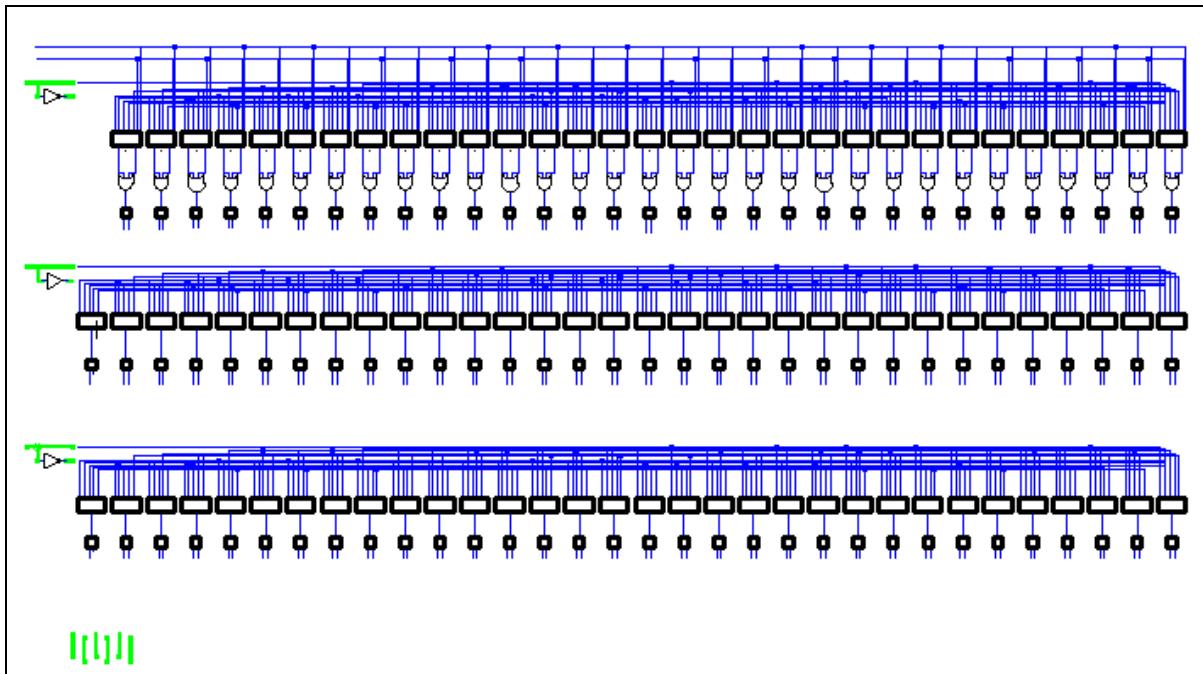
I/O Table

Input	Origin	Output	Destination
RegWrite	Controller	wdo[31:1]	regramarray_dp
ph2	Clock	wdoi[31:1]	regramarray_dp
wadd[4:0]	Five Bit Datapath	r1do[31:0]	regramarray_dp
r1add[4:0]	Fetch	r1doi[31:1]	regramarray_dp
r2add[4:0]	Fetch	r2do[31:0]	regramarray_dp
		r2doi[31:1]	regramarray_dp

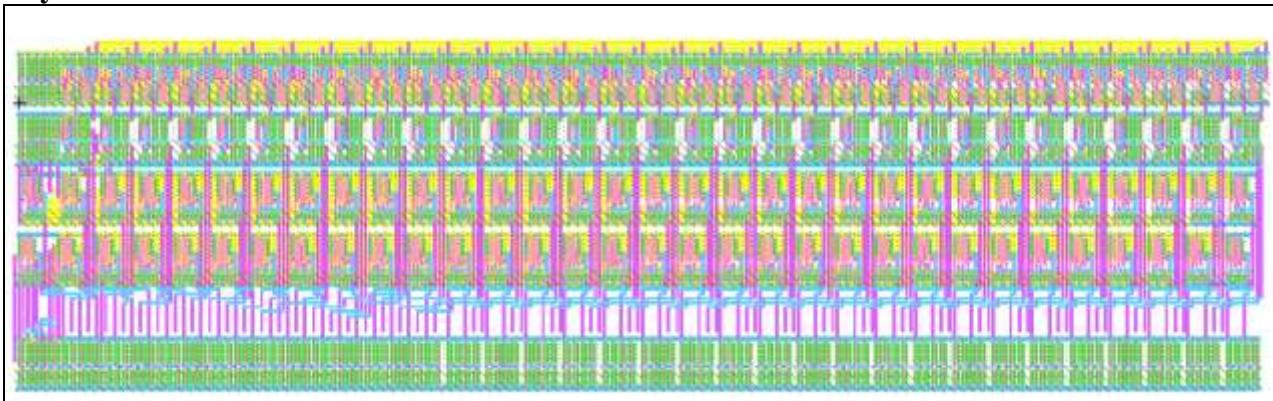
Special Units

A static CMOS nor5_1x gate was used in this module for the read decoder module. The transistor sizes were increased to 30 for the pMOS and 11 for the nMOS, and the p-n ratio was kept about the same.

Schematic



Layout



Testing

This block was tested using a self-checking Verilog testbench. Test vectors were created that covered every possible input for the two read decoders (just each possible one-hot 32 bit vector). The same was done for the write decoder with ph2 and RegWrite high and then both low. Also, having ph2 low and RegWrite high and the opposite case were tested on randomly selected vectors that were used in the previous tests.

12.7.5. ALU (Subunit in Execute Stage)

Function

The Arithmetic Logic Unit (ALU) performs several operations on the A[31:0] and B[31:0] inputs. The ALU always performs AND, XOR, NOR, and OR. Depending on the control bits alushcontrolE[2:0], the carry-ripple adder in the ALU computes either an addition or subtraction. From subtraction, the ALU derives sltS (signed comparison) and sltU (unsigned comparison) based on the MSB of the sum or the inverse of the MSB of the carryout, respectively. Each of these results is sent to inputs of a multiplexer, and the output is selected with alushcontrolE[2:0]. That result is fed into a multiplexer at a later stage in Execute. The ALU also computes if there was an overflow in adder using the logic specified in the RTL.

Schematic

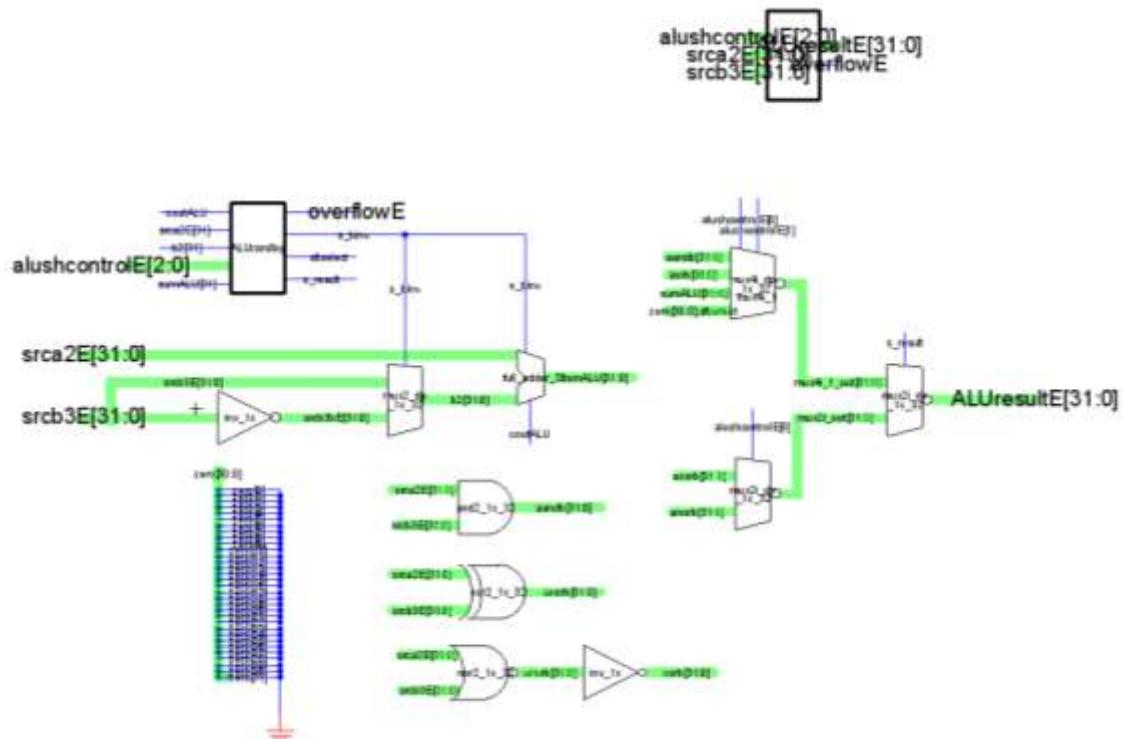


Figure 26. alu{sch}

The schematic of the ALU is broken down into two pieces, the bitslice and the random logic. The bitslice is the bulk of the datapath and constitutes the functions of the ALU, such as AND or addition. The random logic takes in the control signals alushcontrolE[2:0] and other bits to compute the overflow and select signals needed to control the bitslice. For example, s_binv, an output of the random logic, determines whether the ALU does a subtraction or addition.

Layout

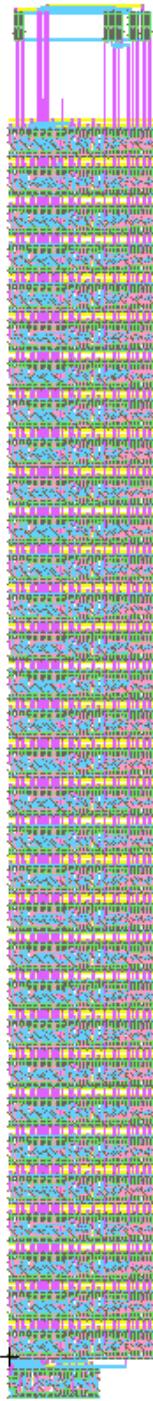


Figure 27. alu{lay}

The ALU layout is straightforward. The random logic is below the 32-bit wordslice and the zipper is above it. Additionally, the ALU uses almost every available metal-2 tracks to make all necessary wiring. In hindsight, the random logic could have been moved to bit 33 and bit 34 of the datapath, as these bits are not used in the ALU and most of the signals feeding into and out of

the random logic are the MSBs of a wordslice. However, space allocated for random logic is under the wordslice, so it was placed there.

Test Code with Test Vectors

```

// alutest.v
// achin at hmc dot edu 2/26/07
//
/////////////////////////////////////////////////////////////////
// Module: alutest
//
// Test fixture for serial multiplier
// Tested 26 February 2007
/////////////////////////////////////////////////////////////////

module alutest();
    wire [2:0] alushcontrolE;
    wire [31:0] srca2E;
    wire [31:0] srcb3E;
    reg [31:0] vectornum, errors;
    reg [99:0] testvectors[10000:0];
    wire [31:0] ALUresultE;
    wire overflowE;
    reg         ready;
    reg         start;
    wire [31:0] ALUexpectedresult;
    reg         error;

    // device under test
    ALU_dp dut(.alushcontrolE(alushcontrolE), .srca2E(srca2E), .srcb3E(srcb3E),
               .ALUresultE(ALUresultE), .overflowE(overflowE));

    // load testvectors
    initial
        begin
            $readmemh("alutestvectors.tv", testvectors);
            vectornum = 0;
            errors = 0;
            error = 0;
        end

    assign #1 {extra, alushcontrolE, srca2E, srcb3E, ALUexpectedresult} = testvectors[vectornum];
    //assign #1 {extra, alushcontrolE, srcb3E} = testvectors[vectornum];

    always @(ALUresultE)
        begin
            if (ALUexpectedresult == ALUresultE)
                begin
                    vectornum = vectornum+1;
                    wait (1000);
                    error = 0;
                end
            else
                error = 1;
        end
    endmodule

    // alu test cases

0_10101010_01010111_00000010      // a&b
1_11111111_33333333_33333333      // a|b
2_ABCDEF00_00ABCDEF_AC79BCEF      // add
3_FFFFFFFF_00000000_00000000      // sltu
4_11111111_01010111_10101000      // a^b
5_11111111_33333333_CCCCCCCC      // ~(a|b)
6_ABCDEF00_FFFFFFFF_ABCDEF01      // sub
3_80000000_0FFFFFFF_00000000      // slts
6_F0000000_0FFFFFFF_E0000001      // slts
7_F0000000_0FFFFFFF_00000001      // slts

0_11111111_01011011_01011011      // a&b
0_00000000_01011011_00000000      // a&b
1_11111111_01011011_11111111      // a|b

```

```

1_00000000_01011011_01011011 // a|b
2_FFFFFFFF_00000001_00000000 // add (-1+1)
2_70038209_7FFFFFFF_F0038208 // add with overflow
3_19092733_F8490390_00000001 // sltu
3_F8490390_19092733_00000000 // sltu
3_19092733_19092733_00000000 // sltu
3_19092732_19092733_00000001 // sltu
7_19092733_F8490390_00000000 // slts
7_F8490390_19092733_00000001 // slts
7_F9092733_F9092733_00000000 // slts
7_F9092732_F9092733_00000000 // slts
7_F9092734_F9092733_00000001 // slts

6_61747266_34125123_2D622143 // sub
6_00000000_00000005_FFFFFFFB // sub
6_F0086422_F8392898_F7CF3B8A // sub

/*
//check if sltu is working
3_FFFFFFFF_00000000_00000000 // sltu
3_00000000_00000001_00000001 // sltu
3_FFFFFFFF_FFFFFFFF_00000000 // sltu
3_00000001_00000000_00000000 // sltu
3_00000000_00000000_00000000 // sltu

```

12.7.6. Shifter

Function

The variable shifter can shift input B[31:0] by 0 to 31 places based on one of three inputs: A[4:0], signimmE[10:6], or a constant shift of 5'b10000. B[31:0] can be shifted logical left (replace LSB with 0), logical right (replace MSB with 0), or associative right (replace MSB with sign bit). The shifted result is routed to a multiplexer in a later stage of execute.

Schematic

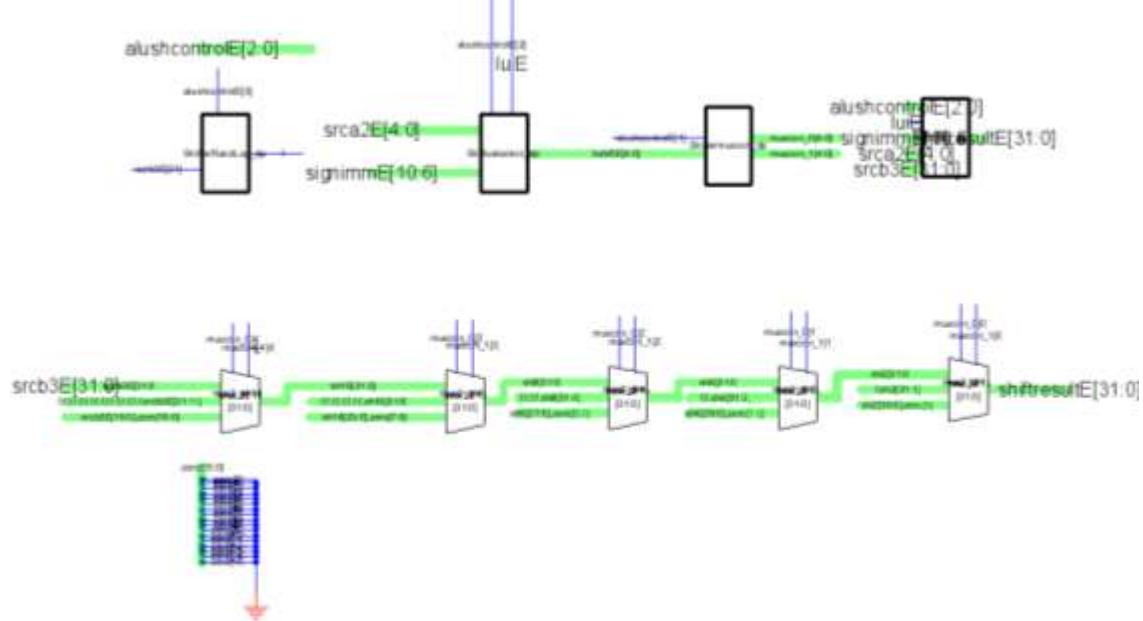


Figure 28. shifter{sch}

The schematic is composed of 5 stages of shifting: shift by 16, by 8, by 4, by 2, by 1. alushcontrolE[2:0] determines if B[31:0] is shifted right or left while the 5-bit shift value

determines if $B[31:0]$ will be shifted in each stage. The last piece of random logic assigns the empty spot of the logical and associative right shifts to either 0 or the sign bit $B[31]$, respectively.

Layout

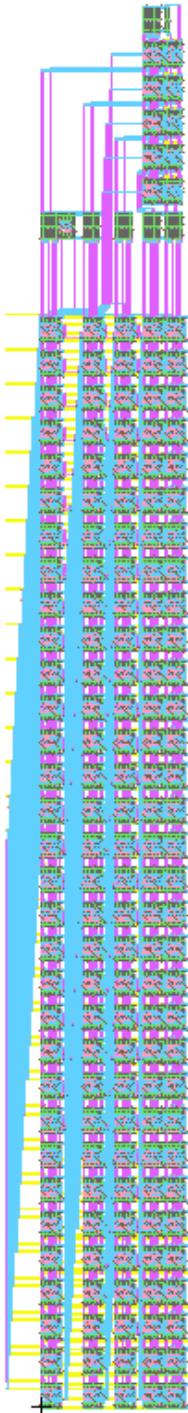


Figure 29. shifter{lay}

Because the multiple tracks of vertical wire was not taken into account in the schematic, the layout of the shifter takes up more space than it was allocated for in the preliminary floorplan. For example, the shift-by-16 stage requires 16 wires running down the wordslice to shift right and 16 wires running up to shift left, a total of 32 wires on an 8λ pitch. To reduce the total amount of space of wiring, metal-1 and metal-2 tracks are run over each other to cut the total number of vertical wire tracks in half. However, using metal-1 vertically between cells means that the Vdd/GND lines must be in metal-3, increasing the total resistance of the Vdd/GND lines in the datapath. Nevertheless, this is deemed to be better than allocating more than 1000 λ for the shifter.

Test Code with Test Vectors

```
// shifftest.v
// achin at hmc dot edu 2/26/07
//
///////////////////////////////
// Module: shifftest
//
// Test fixture for serial multiplier
// Tested 26 February 2007
///////////////////////////////

module shifftest();
    wire [2:0] alushcontrolE;
    wire luiE;
    wire [31:0] srcb3E;
    wire [10:6] signimmE;
    wire [4:0] srca2E;
    wire [31:0] shiftresultE;
    reg [31:0] vectornum, errors;
    reg [91:0] testvectors[10000:0];
    wire [31:0] shiftexpectedresult;
    wire [2:0] moreextra1, moreextra2;
    reg         error;

    // device under test
    shifter_dp dut(.alushcontrolE(alushcontrolE), .luiE(luiE), .srcb3E(srcb3E),
    .signimmE(signimmE),
    .srca2E(srca2E), .shiftresultE(shiftresultE));

    // load testvectors
    initial
    begin
        $readmemh("shifftestvectors.tv", testvectors);
        vectornum = 0;
        errors = 0;
    end

    assign #1 {luiE, alushcontrolE, moreextra1, srca2E, moreextra2, signimmE,
    srcb3E, shiftexpectedresult} = testvectors[vectornum];

    always @(shiftresultE)
    begin
        if (shiftresultE == shiftexpectedresult)
        begin
            vectornum = vectornum+1;
            wait (1000);
            error = 0;
        end
        else
        begin
            error = 1;
            errors = errors+1;
        end
    end
end
```

```

/*
  always @(*)
    if (ready) begin
      #1 {alushcontrolE, srca2E, srcb3E} = testvectors[vectornum];
      vectornum = vectornum+1;
      start = 1;
      ready = 0;
    end else begin
      #1 start = 0;
    end

  always @(*)
    if (done & ~start) begin
      #1 ready = 1;
      if (testvectors[vectornum][0] === 1'bx) begin
        $display("Finished %d test vectors with %d errors\n", vectornum, errors);
        $stop();
      end
      if (prodh !== prodhexpected || prod1 !== prodlexpected) begin
        $display("Error on vector %d: %x * %x: expected (%x %x) observed (%x %x)\n",
          vectornum-1, x, y, prodhexpected, prodlexpected, prodh, prod1);
        errors = errors+1;
      end
    end
  end
*/
endmodule

// shifter test vectors

/*assign #1 {extra, alushcontrolE, moreextra1, srca2E, moreextra2, signimmE,  srcb3E,
           shiftresultE, shiftexpectedresult} = testvectors[vectornum];
*/
8_00_00_80000000_00008000 // rightshift16 (constant)
A_00_00_80000100_01000000 // rightshift16 (constant)
A_08_1F_00000020_00200000 // leftshift16 (constant)
5_00_03_80000010_F0000002 // rightshift3 associative, 1 sign bit
5_00_03_00000010_00000002 // rightshift3 associative, 0 sign bit
0_00_01_00001111_00001111 // righshift increment
0_01_01_00001111_00000888
0_02_01_00001111_00000444
0_03_01_00001111_00000222
0_04_01_00001111_00000111
0_05_01_00001111_00000088
0_06_01_00001111_00000044
0_07_01_00001111_00000022
0_08_01_00001111_00000011
0_09_01_00001111_00000008
0_0A_01_00001111_00000004
0_0B_01_00001111_00000002
0_0C_01_00001111_00000001
0_0D_01_00001111_00000000
0_17_01_80000000_00000100
2_00_01_00001111_00001111 // leftshift increment
2_01_01_00001111_00002222
2_02_01_00001111_00004444
2_03_01_00001111_00008888
2_04_01_00001111_00011110
2_05_01_00001111_00022220
6_00_01_00001111_00002222
6_00_0F_00000111_08888000
6_00_1F_00000001_80000000
4_00_1F_80000000_00000001
5_00_1F_80000000_FFFFFFFF
5_00_1C_24795928_00000002
5_00_1C_A4795928_FFFFFFA
5_00_07_24795928_0048F2B2

```

12.7.7. Execute Stage

Function

The execute block contains 3 major units, the ALU, the 32 bit shifter and the multiply/divide unit. This block is responsible for any operations requiring a multiple or divide function as well as providing ALU and shifter functionality, along with the proper registers/multiplexers to handle pipelining. The PC counter is also incremented by 8 by an adder within Execute.

I/O Table

The following table outlines the inputs and outputs of the Execute stage.

Input	Origin	Output	Destination
ph1	Chip	srcb2E[31:0]	Memory
ph2	Chip	aluoutE[31:0]	Memory
reset	Chip	overflowE	Co-Processor 0
alusrcE	Controller	misalignedwE	Co-Processor 0
luiE	Controller	misalignedhE	Co-Processor 0
mdstartE	Controller	mdrunE	Controller
hilosrcE	Controller		
hilodisableE[1:0]	Controller		
specialregsrcE[1:0]	Controller		
aluoutsrce[1:0]	Controller		
forwardaE[1:0]	Hazard		
forwardbE[1:0]	Hazard		
alushcontrolE[2:0]	Controller		
srcaE[31:0]	Decode		
srcbe[31:0]	Decode		
resultW[31:0]	Mem-Writeback		
aluoutM[31:0]	Execute		
signmmE[31:0]	Decode		
pcE[31:0]	Decode		
cop0readdataE[31:0]	Co-Processor 0		

Special Units

Within the Execute block, 32-bit word slices of multiplexers/registers/logic gates were used in schematic to ease layout. Inside the Multdiv unit, 34-bit word slices are used at times and required only minor modifications to the 32-bit slices. There is one register that was modified to accommodate for the xsavedsign signal. Since this signal was introduced after the layout was mostly complete, the area in which this register could be implemented was very limited. A design decision was made to merge the register containing xsavedsign, xreg, with a 32-bit register within Multdiv, prodlreg, so that the inverters used for control signals such as ph1 and ph2 could be reused. The enable signal differs between the two registers, but does not raise any layout issues.

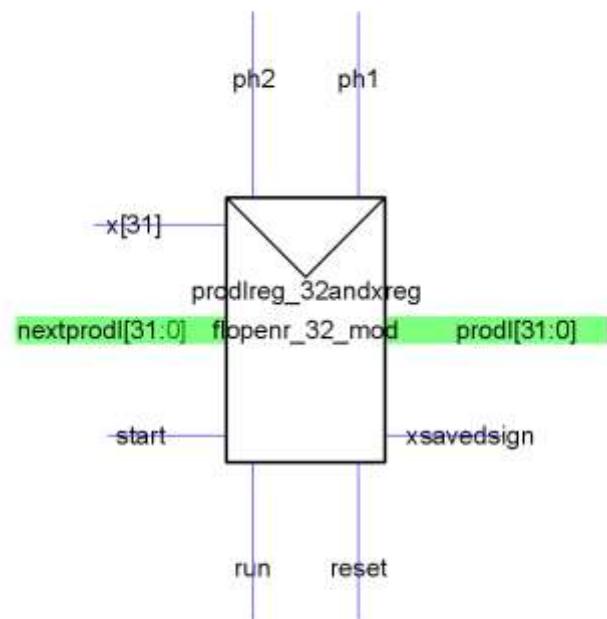


Figure 30. Merged prodreg register and xreg register.

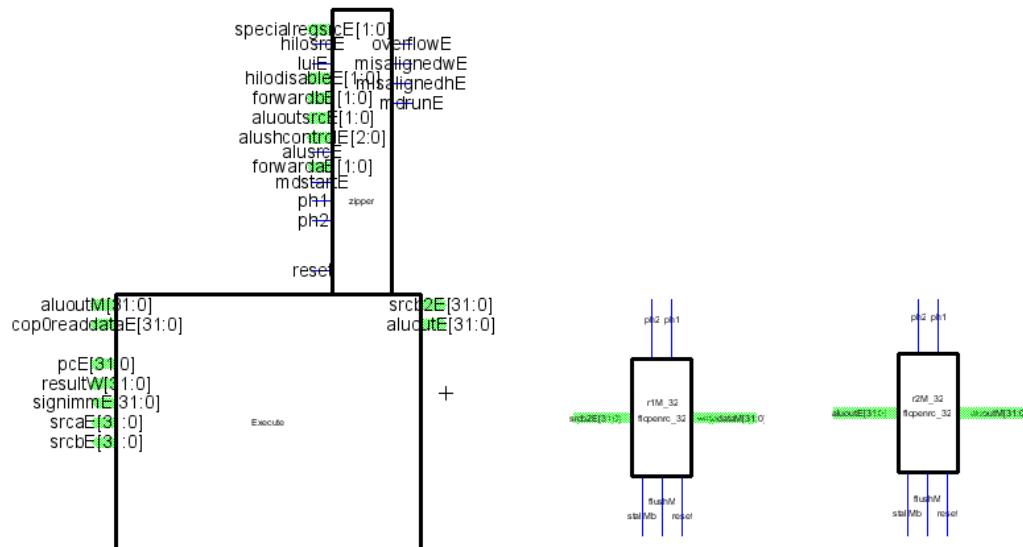


Figure 31. Layout for merged prodlreg and xreg.

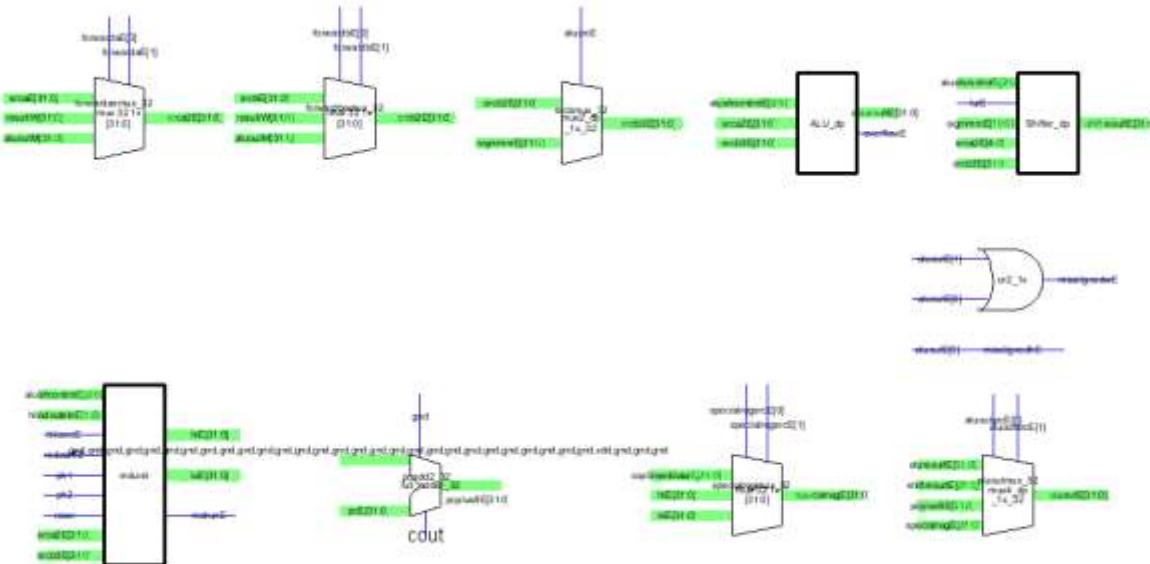
Schematic

The Execute block is broken up into a multitude of sub-blocks. The top level is execute stage. This unit contains the ALU, shifter, mdunit and PC plus 8 adder. The ALU and shifter will be discussed in a separate section. Within the mdunit block resides the Multdiv unit. The Multdiv unit contains its controller, mdcon, as well as registers and multiplexers used in multiply and divide operations. Mdcon contains the control logic for the multdiv unit and contains a programmable logic array (PLA) as well as assorted logic and will be discussed in further detail later in the report. The following schematics outline the various levels of execute.

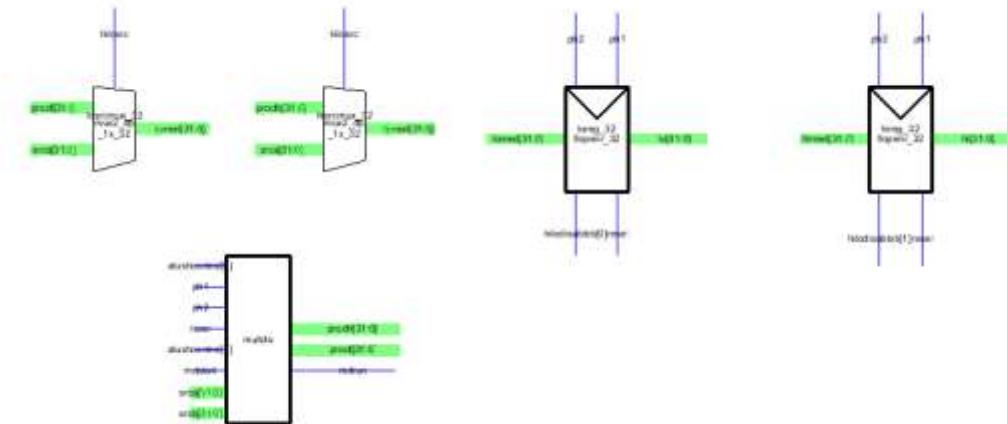
Execute – Top level



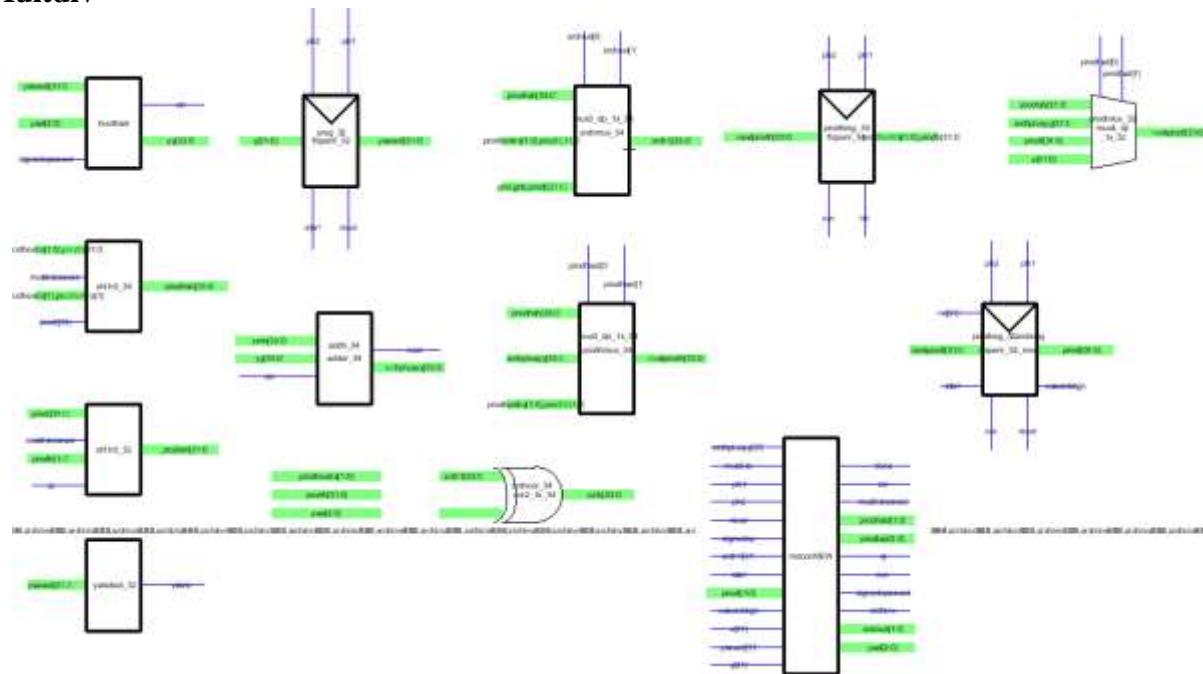
Execute Stage



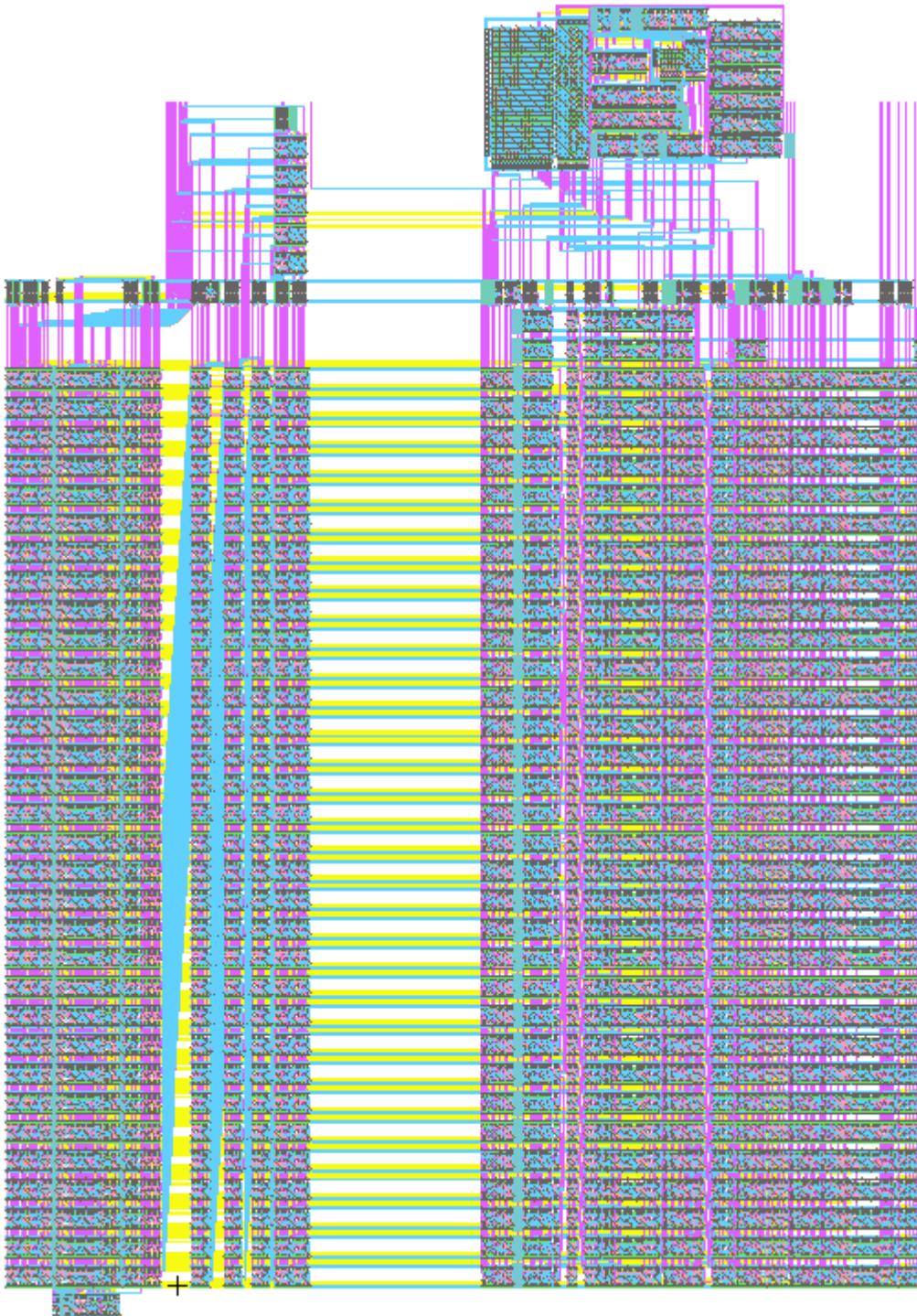
Mdunit



Multdiv



Layout



The large space seen in the middle of the Execute block is a channel used to move data from regions above and below the datapath on the chip. The mdcontroller PLA and associated logic is seen in the top right while the 32-bit word slices take up the majority of the area.

Testing

Two sets of tests were used to check the functionality of execute. One test exclusively tested the multiply and divide capabilities of the multdiv unit. The overall chip tests were also used to verify the functionality of Execute. The Verilog file and test vectors for the exclusive multdiv test are shown here:

```
// multdivtest.v
// David_Harris@hmc.edu 1/2/07
//
///////////////////////////////
/////
// Module: multdivtest
//
// Test fixture for serial multiplier/divider
// Tested 20 January 2007, passed 44 test vectors for signed and unsigned
// multiplication and division.
// This unit would still benefit from rigorous corner case testing.
Generating
// the 64-bit results will take a bit of work. Some corner cases to consider
// include:
// 0, 1, 2, 7FFFFFFF, 80000000, FFFFFFFE, FFFFFFFF, ABCDEF01, 23456789
///////////////////////////////
/////
`timescale 1 ns / 1 ps

module multdivtest();
    reg            clk;
    reg            clk2;
    reg            reset;
    reg [31:0]     x, y;
    reg            multdivb, signedop;
    wire [31:0]    prodh, prod1;
    wire           run, dividebyzero;

    reg [31:0]    vectornum, errors;
    reg [129:0]   testvectors[10000:0];
    reg [31:0]    prodhexpected, prodlexpected;
    reg            ready;
    reg            start;

    // device under test
    //multdiv dut(clk, clk2, reset, start, multdivb, signedop, x, y, prodh,
    prod1, run, dividebyzero);
    multdiv dut(multdivb, clk, clk2, reset, signedop, start, x, y,
    dividebyzero, prodh, prod1, run);
        // (multdivb, ph1, ph2, reset, signedop, start, x, y, dividebyzero,
    prod1, run)

    // generate clock
    always begin
        clk = 1; #5; clk = 0; #5;
    end

```

```

always begin
    clk2 = 0; #5; clk2 = 1; #5;
end

// generate reset
initial begin
    start = 0;
    reset = 1; #17; reset = 0;
    ready = 1;
end

// load testvectors
initial
begin
    $readmemh("multdiv.tv", testvectors);
    vectornum = 0;
    errors = 0;
end

always @(posedge clk)
if (ready) begin
    #1 {multdivb, signedop, x, y, prodhexpected, prodlexpected} =
testvectors[vectornum];
    vectornum = vectornum+1;
    start = 1;
    ready = 0;
end else begin
    #1 start = 0;
end

always @(negedge clk)
if (~run & ~start & ~reset) begin
    #1 ready = 1;
    if (testvectors[vectornum][0] === 1'bx) begin
        $display("Finished %d test vectors with %d errors\n", vectornum,
errors);
        $stop();
    end
    if (prodh != prodhexpected | prod1 != prodlexpected) begin
        if (multdivb)
            $display("Error on vector %d: %x * %x (signed = %d): expected (%x
%x) observed (%x %x)\n",
vectornum-1, x, y, signedop, prodhexpected, prodlexpected,
prodh, prod1);
        else
            $display("Error on vector %d: %x / %x (signed = %d): expected (%x
rem %x) observed (%x rem %x)\n",
vectornum-1, x, y, signedop, prodlexpected, prodhexpected,
prod1, prodh);
        errors = errors+1;
    end
end
end

endmodule

```

Test Vectors:

```

// unsigned multiplication tests
// corner cases: 0, 1, FFFFFFFF, 80000000, 7FFFFFFF
2_00000000_00000000_00000000_00000000
2_00000000_00000001_00000000_00000000
2_00000001_00000000_00000000_00000000
2_00000001_00000001_00000000_00000001
2_00000001_FFFFFFFF_00000000_FFFFFFFF
2_FFFFFFFF_00000001_00000000_FFFFFFFF
2_0000000F_0000000F_00000000_000000E1
2_0000000F_000000FF_00000000_0000FE01
2_0000FFFF_0000FFFF_00000000_FFFE0001
2_FFFFFFFF_FFFFFFFF_FFFFFFFF_00000001
2_80000000_80000000_40000000_00000000
2_0000ABCD_0000EF81_00000000_A0BAF54D

// signed multiplication tests
3_00000001_00000001_00000000_00000001
3_00000001_FFFFFFFF_FFFFFFFF_FFFFFFFF
3_00000010_ABCDEF12_FFFFFFFA_BDEF120
3_FFFFFFFF_FFFFFFFF_00000000_00000001
3_80000000_80000000_40000000_00000000
2_0000ABCD_0000EF81_00000000_A0BAF54D

// unsigned division tests
0_00000000_00000001_00000000_00000000
0_0000000B_00000002_00000001_00000005
0_0000000F_00000018_0000000F_0000000A
0_0000ABCD_0000EF81_0000ABCD_00000000
0_ABCDEF01_0000EF81_000038DE_0000B7A3
0_FEDCBA98_00000020_00000018_07F6E5D4
0_00000003_00000001_00000000_00000003
0_00000003_FFFFFFFF_00000003_00000000
0_FFFFFFFC_00000001_00000000_FFFFFFFC
0_FFFFFFFC_FFFFFFFF_FFFFFFFC_00000000
0_00000005_00000002_00000001_00000002
0_00000005_FFFFFFFE_00000005_00000000
0_FFFFFFFB_00000002_00000001_7FFFFFFD
0_FFFFFFFB_FFFFFFFE_FFFFFFFB_00000000

// signed division tests
1_00000000_00000001_00000000_00000000
1_0000000B_00000002_00000001_00000005
1_0000000F_00000018_0000000F_0000000A
1_0000ABCD_0000EF81_0000ABCD_00000000
1_00000003_00000001_00000000_00000003
1_00000003_FFFFFFFF_00000000_FFFFFFFD
1_FFFFFFFC_00000001_00000000_FFFFFFFC
1_FFFFFFFC_FFFFFFFF_00000000_00000004
1_00000005_00000002_00000001_00000002
1_00000005_FFFFFFFE_FFFFFFFF_FFFFFFFE
1_FFFFFFFB_00000002_00000001_FFFFFFFE
1_FFFFFFFB_FFFFFFFE_FFFFFFFB_00000002

```

12.7.8. Multiplier/Divider Controller

Function

The purpose of the multiplier/divider unit is to perform signed and unsigned multiplication and division functions in the chip. The multdiv controller is responsible for assigning signals to different multiplication and division operations, and these outputs are sent back to the multiply/divide unit, which then processes these signals in a series of multiplexers and registers in order to carry out the proper operations.

I/O Table

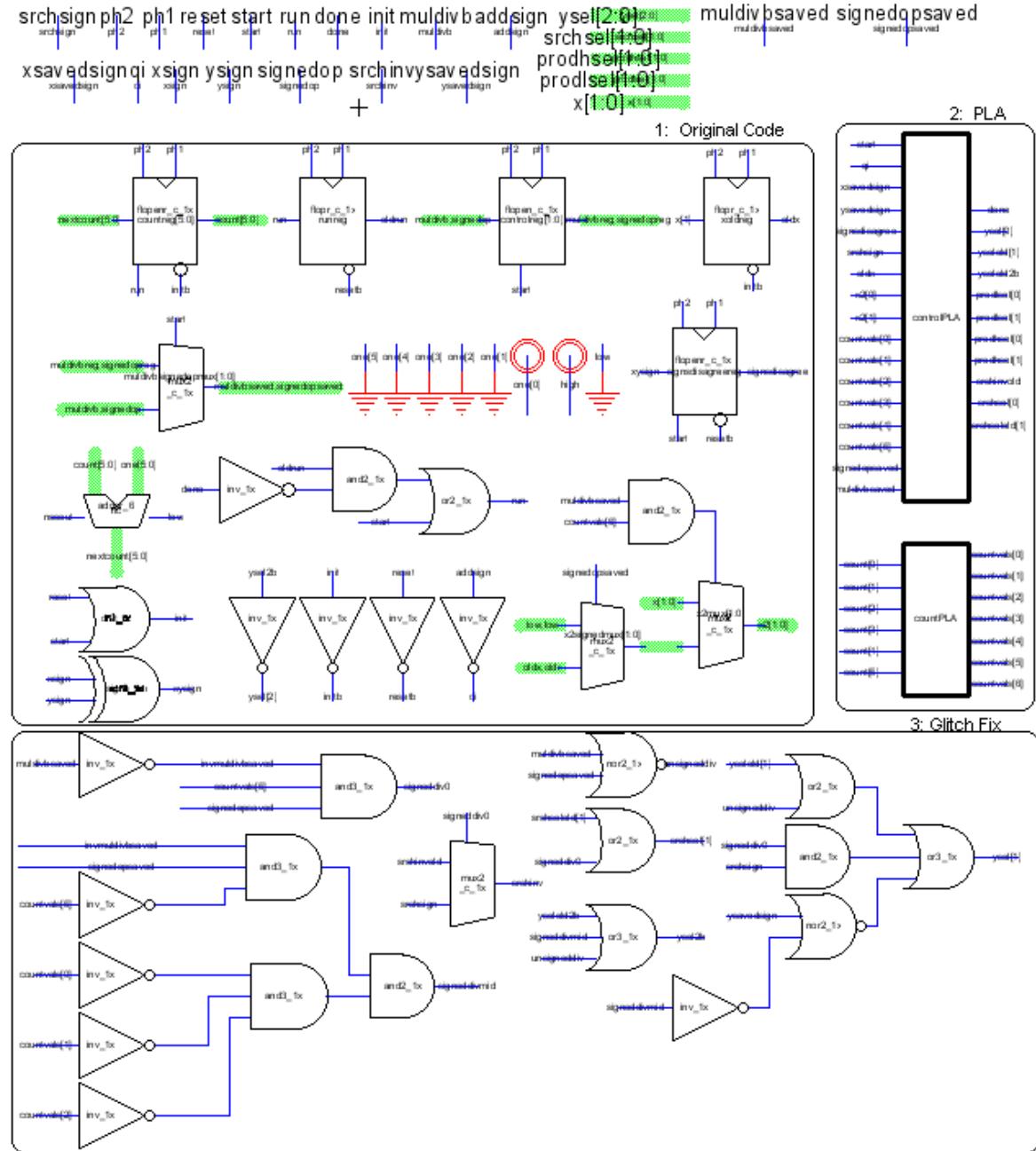
The inputs and outputs on the controller include the following:

Input	Origin	Output	Destination
ph1	multdiv	run	multdiv
ph2	multdiv	done	multdiv
reset	multdiv	init	multdiv
start	multdiv	ysel[2:0]	multdiv
muldivb	multdiv	srchsel[1:0]	multdiv
signedop	multdiv	srchinv	multdiv
x[1:0]	multdiv	prodhsel[1:0]	multdiv
xsign	multdiv	prodlsel[1:0]	multdiv
ysign	multdiv	qi	multdiv
ysavedsign	multdiv	muldivbsaved	multdiv
xsavedsign	multdiv	signedopsaved	multdiv
srchsign	multdiv		
addsign	multdiv		

Special Units

In order to design for a controller that would be purely combinational and not priority-based, a large part of the controller Verilog was rewritten in the form of two case statements, to be implemented in hardware as programmable logic arrays. One case statement handled count and acted as an encoder to interpret the number of cycles within an operation that have passed, and one case statement handled the conditional outputs given by the controller. Converting the behavioral Verilog to structural code was difficult; due to mistakes made in this conversion, debugging the code took countless hours. Additionally, the second PLA was difficult to deal with on a layout level, because of the amount of space it took. However, after rearranging the position of multdiv controller to occupy part of the hazard/5-bit datapath block, space was no longer a concern.

Schematic



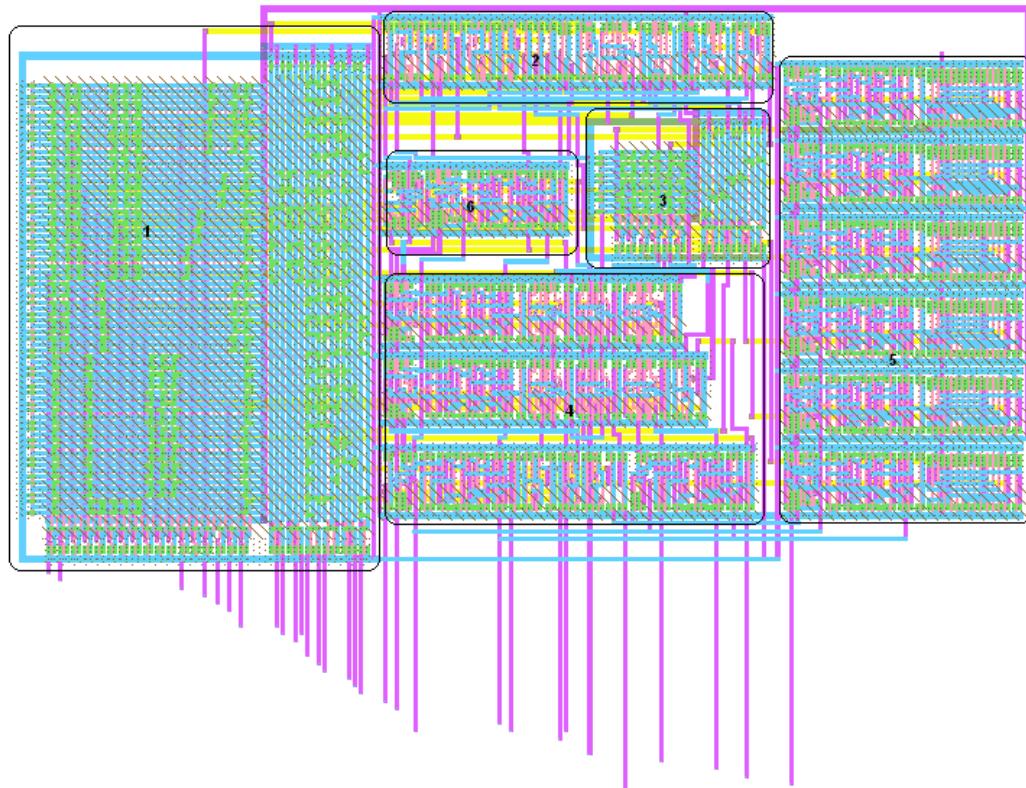
The block schematic for the pad frame is shown above. The schematic is split into three sections: original Verilog, PLA, and glitch fix implemented schematic. The three sections were created about a week from each other, as revisions arose in the RTL schematic.

The third block shown in the schematic above is the glitch fix. It was discovered a week after creating parts 1 and 2 in the schematic, and was due to combinational loops that arose when

glitching occurred. In order to prevent such glitching from causing indeterminate loops in logic sequences, the multdiv controller was changed to contain a multiplexer and additional combinational logic that would prevent momentary glitches from propagating through the signal datapath. These additional fixes are shown below in Verilog:

```
assign signeddiv0 = ~muldivbsaved & signedopsaved & countvals[5];
assign signeddivmid = ~muldivbsaved & signedopvesaved & (~countvals[5] &
~countvals[0] & ~countvals[1] & ~countvals[2]);
assign unsigneddiv = ~muldivbsaved & ~signedopsaved;
assign srchsel[1] = srchselold[1] + signeddiv0;
assign srchinv = signeddiv0 ? srchsign : srchinvold;
assign ysel2b = yselold2b + signeddivmid + unsigneddiv;
assign ysel[1] = yselold[1] + unsigneddiv + (signeddiv0 & srchsign) +
(signeddivmid & ~ysavedsign);
```

Layout:



Because of additional space allocation to the multdiv controller unit, there was little difficulty in creating the controller layout. A large majority of the issues with the controller unit involved preliminary releases of the PLA generator; these issues were resolved once the final revision of the PLA generator was released.

Six sections have been outlined in the pad frame layout. Sections 1 and 3 correspond to the Control PLA and the Count PLA. Sections 4 and 6 correspond to random logic and flip flops

used in the original Verilog code for multdiv controller and section 5 corresponds specifically to the count/next count sequence of logic in the original Verilog code. Lastly, section 2 was created to fix the glitches that caused combinational loops in the multiply/divide unit.

Testing

Multdiv passed DRC, ERC, and NCC tests in Electric without difficulty. The majority of testing took place in ModelSim in debugging the multdiv circuit. Initial problems involved the interpretation of the original Verilog code into combinational structural code. After these problems were resolved, multdiv had problems with signed and unsigned division, which was resolved by the micro-architecture team. Lastly, more testing took place when multdiv was experiencing indeterminate loops when integrated into the chip.

Initially, the controller was generated with its own Verilog net list and combined with the RTL schematic on the multiply/divide unit for testing. The multiply divide unit was put through 10,000 test vectors, generated by the micro-architecture team. However, when errors started to occur in multiply/divide in the schematic simulation of the entire chip, the controller and multiply/divide unit net lists were combined for testing and then compared to the multdiv RTL schematic.

Several test vectors were used to test the multdiv unit, including David Harris' corner-case test vectors, the chip test suite test vectors, and the micro-architecture team's randomly generated test vectors. However, the test vectors that were ultimately used in testing are shown below. They are a combination of the randomly generated test vectors and the test vectors that specifically correspond to chip test 16. The randomly generated test vectors test for transitions between operations, which had been problematic in the past, and test vectors from test 16 were included because test 16 had been the specific test that multdiv failed.

```
// multdivtest.v
// David_Harris@hmc.edu 1/2/07
//
///////////////////////////////
/////
// Module: multdivtest
//
// Test fixture for serial multiplier/divider
// Tested 20 January 2007, passed 44 test vectors for signed and unsigned
// multiplication and division.
// This unit would still benefit from rigorous corner case testing.
Generating
// the 64-bit results will take a bit of work. Some corner cases to consider
// include:
// 0, 1, 2, 7FFFFFFF, 80000000, FFFFFFFE, FFFFFFFF, ABCDEF01, 23456789
/////////////////////////////
`timescale 1 ns / 1 ps

module multdivtest();
    reg          clk;
    reg          reset;
```

```

reg [31:0] x, y;
reg multdivb, signedop;
wire [31:0] prodh, prodl;
wire run, dividebyzero;
reg [31:0] vectornum, errors;
reg [129:0] testvectors[10000:0];
reg [31:0] prodhexpected, prodlexpected;
reg ready;
reg start;
reg ph1, ph2;
// generate clock to sequence tests
always
begin
    ph1 <= 1; # 4; ph1 <= 0; #1;
    ph2 <= 1; # 4; ph2 <= 0; #1;
end

// device under test
multdiv dut(multdivb, ph1, ph2, reset, signedop, start, x, y, prodh,
prodl, run);

// generate reset
initial begin
    start = 0;
    reset = 1; #17; reset = 0;
    ready = 1;
end

// load test vectors
initial
begin
    $readmemh("multdiv.tv", testvectors);
    vectornum = 0;
    errors = 0;
end

always @(posedge ph1)
if (ready) begin
    #1 {multdivb, signedop, x, y, prodhexpected, prodlexpected} =
    testvectors[vectornum];
    vectornum = vectornum+1;
    start = 1;
    ready = 0;
end else begin
    #1 start = 0;
end

always @(negedge ph1)
if (~run & ~start & ~reset) begin
    #1 ready = 1;
    if (testvectors[vectornum][0] === 1'bx) begin
        $display("Finished %d test vectors with %d errors\n", vectornum,
        errors);
    $stop();
end

```

```

if (prodh !== prodhexpected | prodl !== prodlexpected) begin
    if (multdivb)
        $display("Error on vector %d: %x * %x (signed = %d): expected (%x
                  %x) observed (%x %x)\n", vectornum-1, x, y, signedop,
                  prodhexpected, prodlexpected, prodh, prodl);
    else
        $display("Error on vector %d: %x / %x (signed = %d): expected (%x
                  rem %x) observed (%x rem %x)\n", vectornum-1, x, y, signedop,
                  prodlexpected, prodhexpected, prodl, prodh);
        errors = errors+1;
    end
end
endmodule

// multdiv test vectors
// random sample of test vectors - operation transition test
2_f8b4f804_b834ffc6_b2f58f81_dfd3cf18
1_f8865608_fffff4e30_fffffe778_00000ac3
1_c6c65c53_00007043_fffffa34d_fffff7d82
2_b45cd7a5_f89cf994_af288951_d3f82864
3_c8e322d0_6abcb5a2_e9056c1d_681917a0
2_1e44ad0c_5bc1368b_0ad941bc_a5d67d84
1_00000001_fffffabad_00000001_00000000
0_92cc6cbd_0000be89_000057a1_0000c53c
3_af672019_5db52f19_e27f6bbe_d6a3b971
1_e3cb48a4_fffff4afc_fffff6730_000027e3
2_7fffffff_fffffff7_7fffffff_80000001
1_7fffffff_fffff4afc_0000a2eb_fffff4afb
3_80000000_c3c635e7_1e1ce50c_80000000
0_f2e4521d_000030ce_0000193d_0004fa10
// these test vectors specifically correspond to test 16
0_00000137_00000047_0000001B_00000004
2_00000001F_00000005_00000000_0000009B
2_00000047_00000002_00000000_0000008E
0_0000009B_0000009C_0000009B_00000000
2_0000009B_0000008E_00000000_000055FA
0_000055FA_0000000A_00000000_00000899

```

12.7.9. Memory - Writeback Stage

Function

The memory and writeback stages are responsible for determining what data should be written and read from and written to memory. The unit takes inputs to determine a byte mask, to determine whether it should write a full word, half word, or just a byte, and where in memory it should write it. It also uses that byte mask to determine which word, half word, or byte should be read, whether the byte or half word should be sign-extended. In the writeback stage, it determines whether the data that memorystage read or aluout should be the information that moves on to the next stage.

I/O Table

A table listing all inputs and outputs from the unit, and their origin/destination.

Input	Origin	Output	Destination
readdataM[31:0]	Cache	writedata2M[31:0]	Cache
writedataM[31:0]	Execute	writedataW[31:0]	Coprocessor0
aluoutM[31:0]	Execute	resultW[31:0]	Decode, Execute
byteM[0:0]	Controller (Pipeline)	byteenM[3:0]	Cache
halfwordM[0:0]	Controller (Pipeline)		
loadsingedM[0:0]	Controller (Pipeline)		
re	Coprocessor0		

Special Units

Quite a few special units were required for this block.

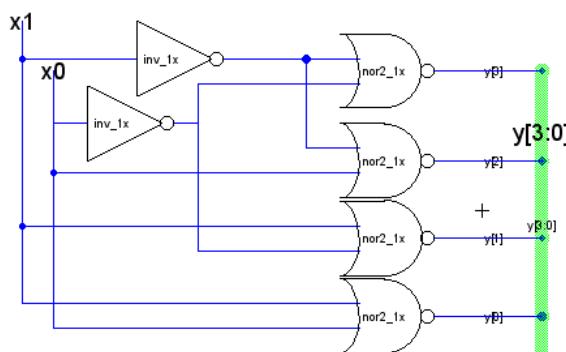


Figure 32 - dec2{sch}

For determining bytebyteenM from
aluoutereM[1:0]

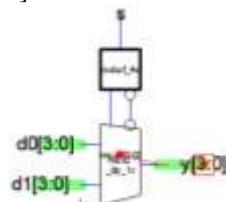


Figure 33 - mux2_1x_4{sch}

For determining halfwordbyteenM from
aluoutereM[1]

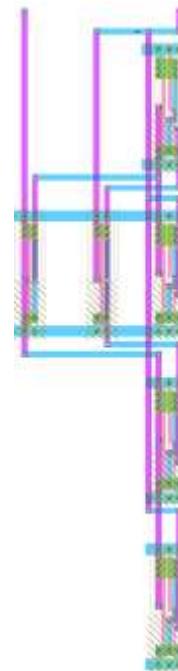


Figure 34 - dec2{lay}

This layout was designed around several already placed and routed cells.

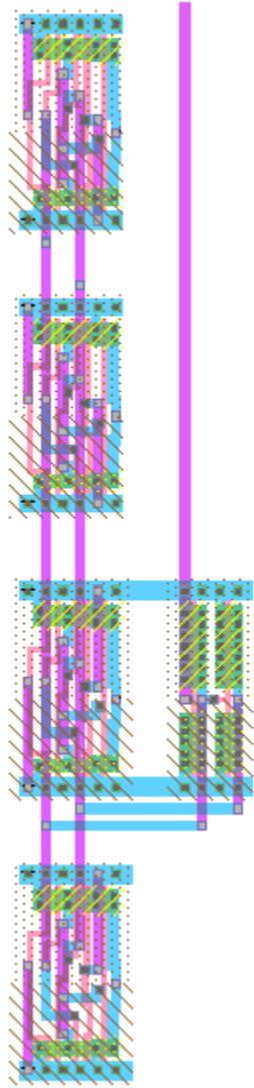


Figure 35 - mux4_1x_4{lay}

This layout was also designed around placed cells.

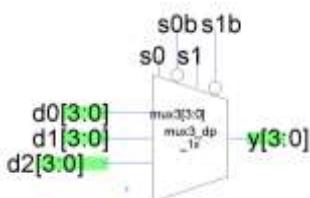


Figure 36 - mux3_1x_4{sch}

For determining byteenM from bytebyteenM, halfwordbyteenM, and [1,1,1,1]

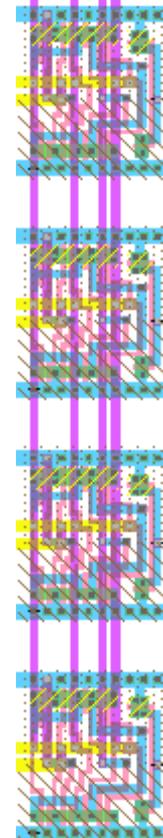


Figure 37 - mux3_1x_4{lay}

This layout was designed without a zipper, because a 32-bit mux3 was placed directly above it, the same select signals could be routed from there.

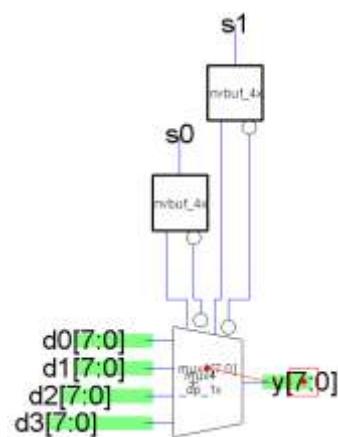


Figure 38 - mux4_1x_8{sch}

For determining rbyteM from readdataM.

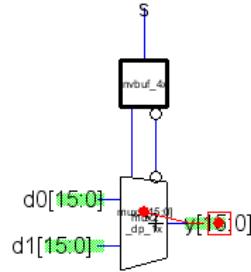


Figure 39 - mux2_1x_16{sch}

For determining rhalfwordM from readdataM.

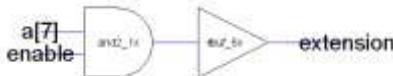


Figure 40 - signext_8_32{sch}

For doing the sign extension of rbyteM to rbyteextM. Since the schematic of signext_16_32 is the same thing with a 4x buffer in place of the 6x buffer that schematic has not been included, and because the layouts are rather boring, they've been included in Figure 42 as the items in the top two rows.

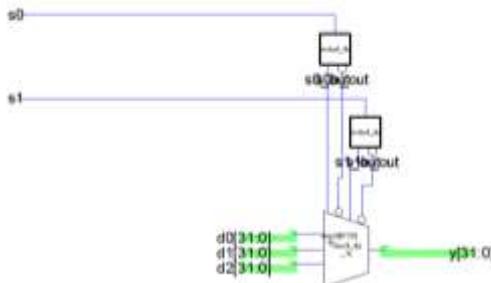


Figure 41 - mux3_dp_1x_32_special{sch}

The only difference between this and the regular mux3_dp_1x_32 was that this version also provides outputs for the select signals to drive the mux3 pictured in Figure 37.



**Figure 42 - mux4_1x_8{lay},
mux2_1x_16{lay}**

Obviously the most difficult part of this layout was routing the wires to the proper locations. Luckily, some of the wire tracks to the mux4_1x_8{lay} could be shared with the mux2_1x_16{lay} adjacent to it.

Schematic

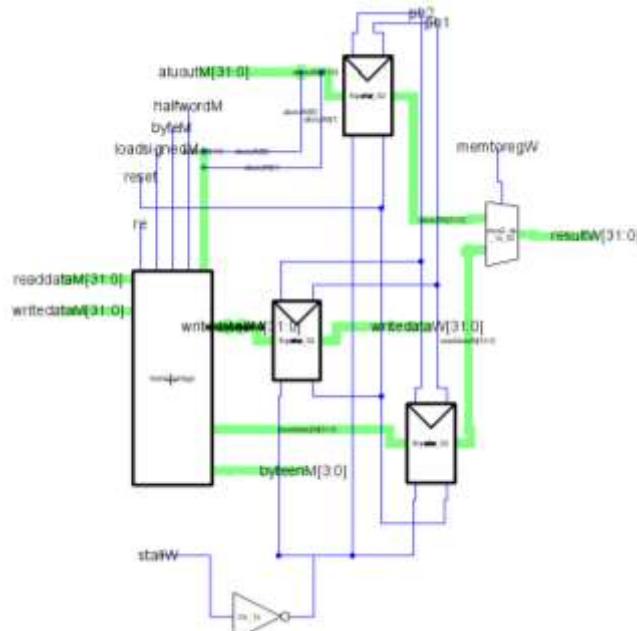


Figure 43. Memory_Writeback{sch}

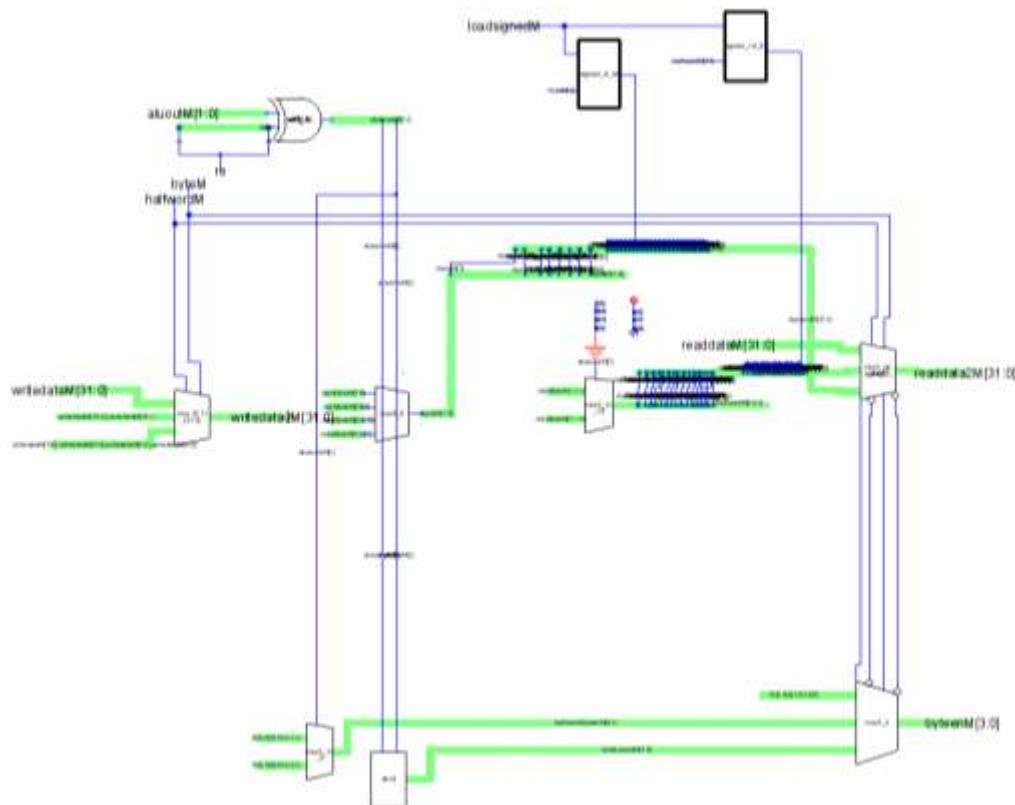


Figure 44. memorystage{sch}

The mess of wire routing between the mux4_8, mux2_16, and mux3_32_special were the only way around a particular bug in Electric. The program does not allow multiple CMOS VLSI Design, Spring 2007 HMC-MIPS Chip Report

outputs of cells to be connected to each other within the cell, so the routing of the extension bits had to be done outside of the signext units.

Layout

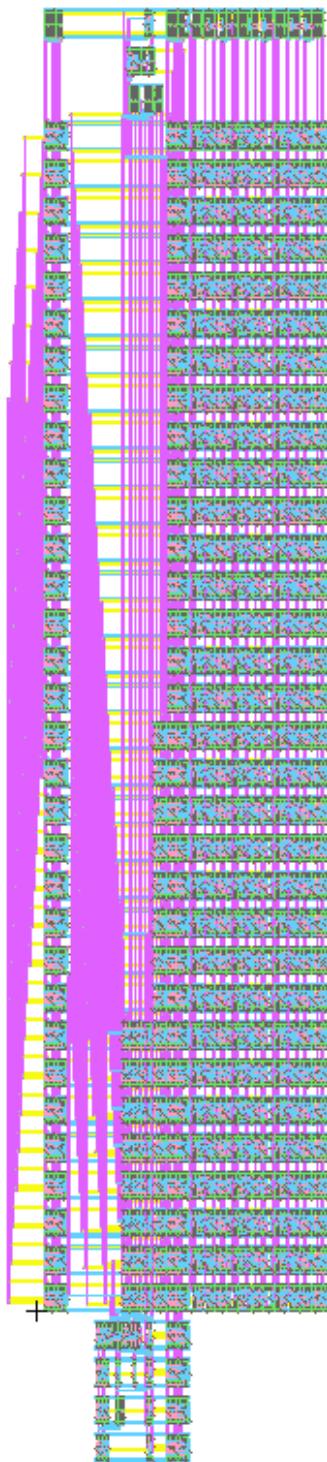


Figure 45. Memory_Writeback{lay}

The most challenging part of this layout was the wire routing, with sets of 16 and 24 bits that had to be run to other places within the block.

Testing

The following test was run on the Verilog file generated from the schematics of the Memory_Writeback cell:

```
// memorywritebackstagetest.v
// dlindblad@hmc.edu 2-27-07
//
///////////////////////////////
// Module: memorywritebackstagetest
//
// Test fixture for Fetchstage module of pipelined MIPS processor
// Tested 20 January 2007
///////////////////////////////

module memorywritebackstagetest();
    reg          ph1, ph2;
    reg          reset, stallw, byteM, halfwordM, re, loadsignedM, memtoregw;
    reg [31:0]   aluoutM;
    reg [31:0]   readdataM, writedataM;
    wire [31:0]  resultw, writedata2M, writedataW;
    wire [3:0]   byteenM;
    reg [1:0]    junk;

    reg [31:0]  vectornum, errors;
    reg [127:0] testvectors[10000:0], testvectors2[10000:0];
    reg [31:0]  resultwexpected, writedata2Mexpected, writedataWexpected;
    reg [3:0]   byteenMexpected;
    reg          ready, start;

    // device under test
    Memory_Writeback dut(.aluoutM(aluoutM), .byteM(byteM), .halfwordM(halfwordM),
                         .loadsighedM(loadsighedM), .memtoregw(memtoregw), .ph1(ph1), .ph2(ph2),
                         .re(re), .readdataM(readdataM), .reset(reset), .stallw(stallw),
                         .writedataM(writedataM), .byteenM(byteenM), .resultw(resultw),
                         .writedata2M(writedata2M), .writedataW(writedataW));

    // generate clock
    always begin
        ph2 = 1; #4; ph2 = 0; #1;
        ph1 = 1; #4; ph1 = 0; #1;
    end

    // generate reset
    initial begin
        start = 1;
        reset = 1; #27; reset = 0;
        start = 0;
        ready = 1;
    end

    // load testvectors
    initial
    begin
        $readmemh("memory_writeback_input_test.tv", testvectors);
        $readmemh("memory_writeback_output_test.tv", testvectors2);
        $display("test vectors read");
        vectornum = 0;
        errors = 0;
    end

    always @(posedge ph1)
    if (ready) begin
        #1 {aluoutM[31:0], byteM, halfwordM, re, loadsighedM, junk, memtoregw,
            stallw, readdataM[31:0], writedataM[31:0]} = testvectors[vectornum];
        {byteenMexpected[3:0], resultwexpected[31:0], writedata2Mexpected[31:0],
         writedataWexpected[31:0]} = testvectors2[vectornum];
        $display("passed vector number %d", vectornum);
        vectornum = vectornum+1;
        start = 1;
        ready = 0;
    end

```

```

    end else if (~reset) begin
        #1 start = 0;
    end

    always @(posedge ph2)
        if (~start&~ready) begin
            #1 ready = 1;
            if (testvectors[vectornum][0] === 1'bx) begin
                $display("Finished %d test vectors with %d errors\n", vectornum, errors);
                $stop();
            end
            $display("Passed vector %d: byteM = %x, halfwordM = %x, re = %x, loadsIGNEDM = %x, memtoregw = %x, stallw = %x\n Expected (%x %x %x %x) Observed (%x %x %x %x)\n",
                    vectornum-1, byteM, halfwordM, re, loadsIGNEDM, memtoregw, stallw,
                    byteenMexpected, resultWexpected, writedata2Mexpected,
                    writedataWexpected,
                    byteenM, resultW, writedata2M, writedataW);
            if (byteenM !== byteenMexpected | resultW !== resultWexpected | writedata2M !==
                writedata2Mexpected | writedataW !== writedataWexpected) begin
                $display("ERROR!");
                /*$display("Passed vector %d: byteM = %x, halfwordM = %x, re = %x, loadsIGNEDM = %x,"
                    " memtoregw = %x, stallw = %x\n Expected (%x %x %x %x) Observed (%x %x %x %x)\n",
                    vectornum-1, byteM, halfwordM, re, loadsIGNEDM, memtoregw, stallw,
                    byteenMexpected, resultWexpected, writedata2Mexpected,
                    writedataWexpected,
                    byteenM, resultW, writedata2M, writedataW);*/
                errors = errors+1;
            end
        end
    endmodule

```

This was used with the two following sets of test vectors:

```

//MemoryStage Test Vectors Input
//Format:
//[[aluoutM[31:0],byteM,halfwordM,re,loadsIGNEDM,0,0,memtoregw,stallw,readdatAM[31:0],
//      writedataM[31:0]]
//
15816fd2_92_62c4646f_c64df465
64cf4a3_12_6a2cf4a1_445bc654
2624fce0_70_64f46ed5_66224acd
2424ab50_00_64f46c6a_24f624fd
42faca46_52_642656a4_5c8ad6ad
64cf4a3_13_6a2cf4a1_445bc654

//MemoryStage Test Vectors Output
//Format:
//[[byteenM[3:0],resultW[31:0],writedata2M[31:0],writedataW[31:0]]
//
4_FFFFFFC4_65656565_65656565
F_6A2CF4A1_445bc654_445BC654
C_2624FCE0_4acd4acd_4acd4acd
F_2424AB50_24f624fd_24F624FD
C_00006426_d6add6ad_d6add6ad
F_00006426_445bc654_5c8a5c8a

```

12.7.10. Hazard

Function

The Hazard unit performs checks on the control and datapath units to make sure the actions of the datapath correctly match the control inputs. If they do not match, the hazard sends an “active exception” signal to the control unit. Furthermore, the hazard checks that the pipelined actions do not interfere with each other, outputting stalls to all the datapath blocks when necessary. Finally, the hazard also clears (or flushes) the bits in the datapath when there is an exception.

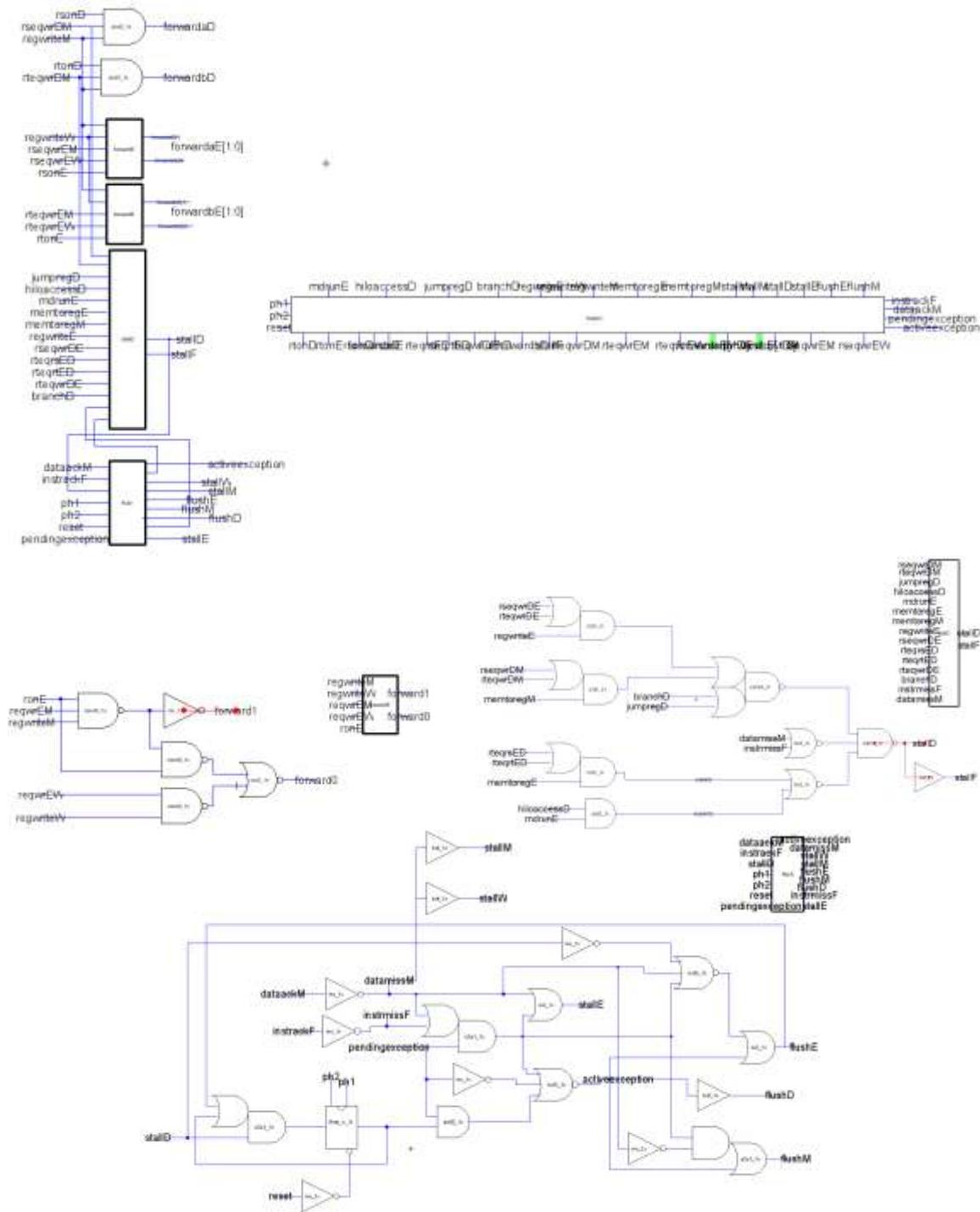
I/O Table

Input	Origin	Output	Destination
ph1	Chip	activeexception	coprocessor
ph2	Chip	flushD	decode
reset	Chip	flushE	execute
branchD	control	flushM	memory
dataackM	control	stallD	decode, control
hiлоaccessD	control	stallE	execute, control
instrackF	control	stallF	fetch, control
jumpregD	control	stallM	memory, control
mdrunE	execute	stallW	writeback, control
memtoregE	control	forwardaD	decode
memtoregM	control	forwardbD	decode
pendingexception	coprocessor	forwardaE[1:0]	execute
regwriteE	control	forwardbE[1:0]	execute
regwriteM	control		
regwriteW	control		
rsonD	five-bit datapath		
rsonE	five-bit datapath		
rtonD	five-bit datapath		
rtonE	five-bit datapath		
rseqwrDE	five-bit datapath		
rseqwrDM	five-bit datapath		
rseqwrEM	five-bit datapath		
rseqwrEW	five-bit datapath		
rteqrsED	five-bit datapath		
rteqrtED	five-bit datapath		
rteqwrDE	five-bit datapath		
rteqwrDM	five-bit datapath		
rteqwrEM	five-bit datapath		
rteqwrEW	five-bit datapath		

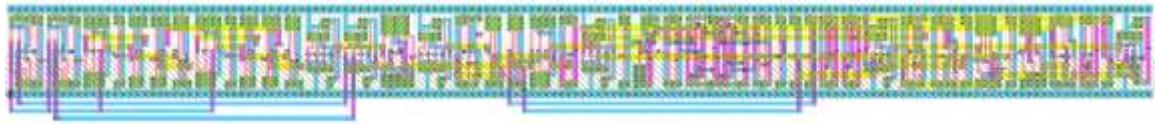
Special Units

Although the block was broken up into smaller, large components, no special units were created.

Schematic



Floorplan



Testing

To test the block, Verilog test decks were generated from the Electric and passed previously-designed chip tests. Both schematic and layout passed DRC, ERC, and NCC.

12.7.11. Five Bit Datapath

Function

The five-bit datapath contains all the 5-bit registers in the datapath, taking 5-bit inputs from the decode stage and outputting them to later stages, such as execute and writeback stages(after going through the appropriate registers. Additionally, the five-bit datapath contains all the 5-bit comparators necessary for the hazard unit, since they fit better. It is located above the decode and execute units, and below the hazard unit.

I/O Table

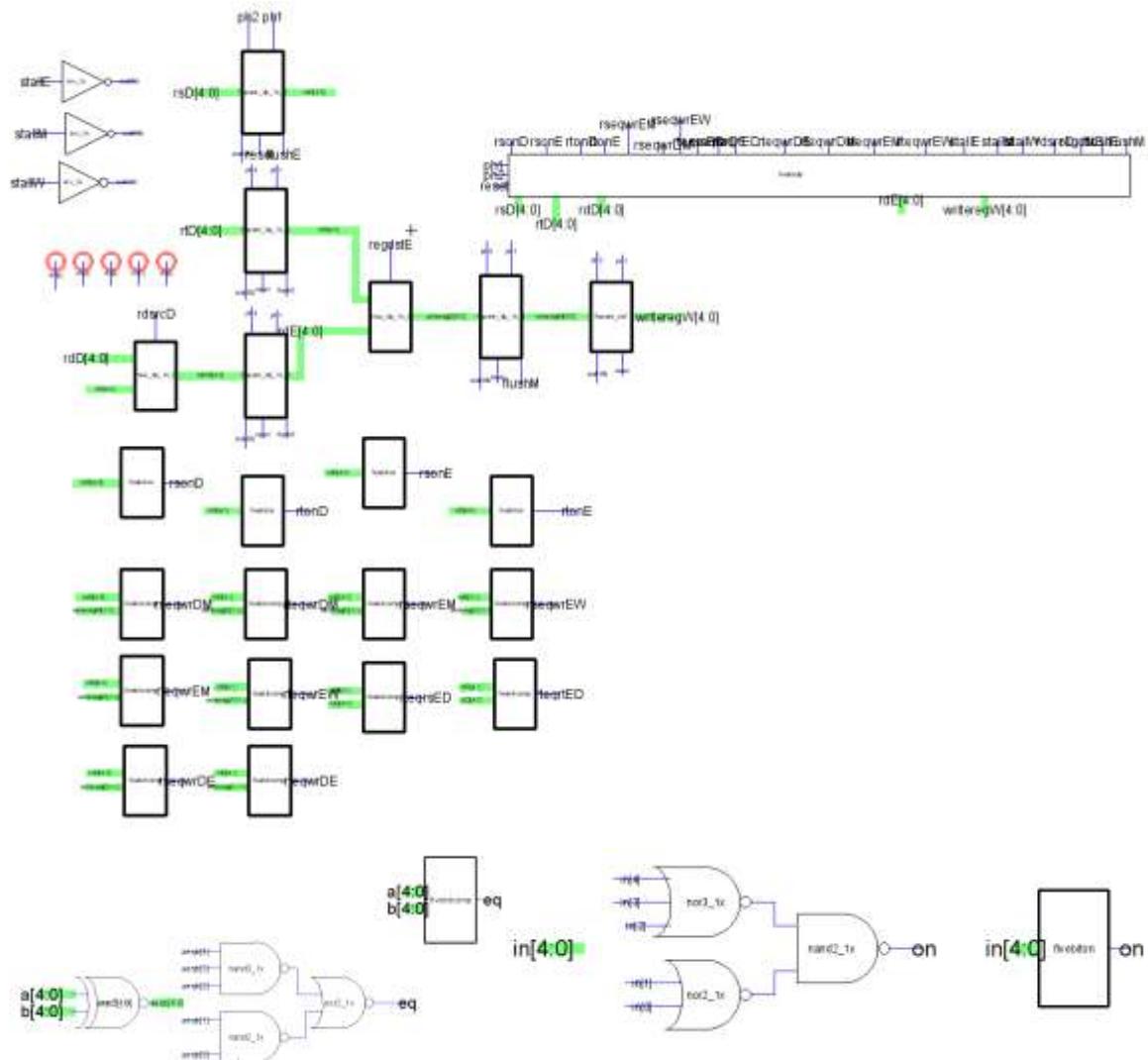
A table listing all inputs and outputs from the unit, and their origin/destination.

Input	Origin	Output	Destination
ph1	Chip	rdE[4:0]	Coprocessor
ph2	Chip	writeregW[4:0]	Coprocessor
reset	Chip	rsonD	Hazard
rdD[4:0]	Decode	rsonE	Hazard
rsD[4:0]	Decode	rtonD	Hazard
rtD[4:0]	Decode	rtonE	Hazard
rdsrcD	Control	rseqwrDE	Hazard
regdstE	Control	rseqwrDM	Hazard
stallE	Hazard	rseqwrEM	Hazard
stallM	Hazard	rseqwrEW	Hazard
stallW	Hazard	rteqrsED	Hazard
flushE	Hazard	rteqrtED	Hazard
flushM	Hazard	rteqwrDE	Hazard
		rteqwrDM	Hazard
		rteqwrEM	Hazard
		rteqwrEW	Hazard

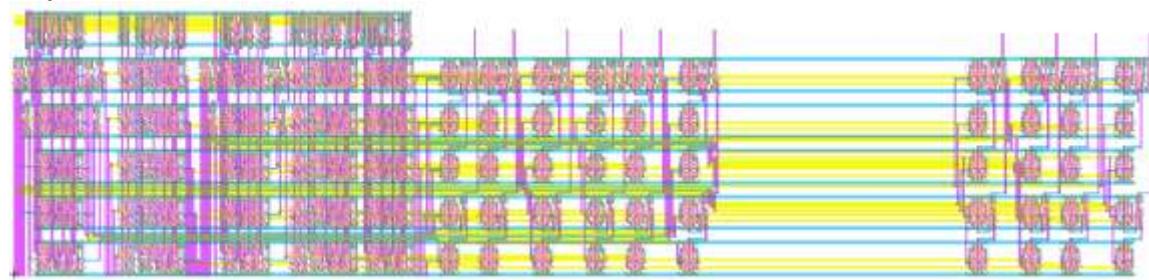
Special Units

Although the block was broken up into smaller units (such as five-bit comparators), no special units were created.

Schematic



Layout



The long space was introduced to allow room for an element in the execute block that required use of the five-bit datapath area.

Testing

To test the block, Verilog test decks were generated from the Electric and passed previously-designed chip tests. Both schematic and layout passed DRC, ERC, and NCC.

13. Controller

13.1. Function

The MIPS controller generates all of the control signals for the Datapath portion of the processor, as well as control signals for the Coprocessor and Memsys. The signals are delayed in control registers and are available for the other units at the proper stages. All of the signals are lined up with the wires coming from Datapath to simplify the wiring between the two systems.

13.2. Unit I/O

Input	Origin	Output	Destination
ph1	Chip Input	memtoregE	Datapath
ph2	Chip Input	memtoregM	Datapath
Reset	Chip Input	adelableE	Coprocessor
stallDb	Datapath	adesableE	Coprocessor
stallEb	Datapath	aluoutsrce[1:0]	Datapath
stallMb	Datapath	alushcontrolE[2:0]	Datapath
stallWb	Datapath	alusrcE	Datapath
flushE	Datapath	bdsE	Coprocessor
flushM	Datapath	breakE	Coprocessor
functD[5:0]	Datapath	byteM	Datapath
opD[5:0]	Datapath	fpuE	Coprocessor
rsD[4:0]	Datapath	halfwordM	Datapath
rtD[4:0]	Datapath	hiloaccessD	Datapath
aeqbD	Datapath	JumpregD	Datapath
aeqzD	Datapath	hilosrcE	Datapath
agtzD	Datapath	loadsingedM	Datapath
altzD	Datapath	luiE	Datapath
mdrunE	Datapath	mdstartE	Datapath
Pendingexception	Datapath	memtoregW	Datapath
		memwriteM	Memsys
		overflowableE	Coprocessor
		regdstE	Datapath
		halfwordE	Coprocessor
		rfeE	Coprocessor
		riE	Coprocessor
		specialregsrcE[1:0]	Datapath
		syscallE	Coprocessor
		regwriteW	Datapath
		regwriteE	Datapath

	Cop0writeW	Coprocessor
	regwriteM	Datapath
	rdsrcD	Datapath
	unsignedD	Datapath
	pcbranchsrcD[1:0]	Datapath
	pcsrcFD[1:0]	Datapath

13.3. PLA Generation

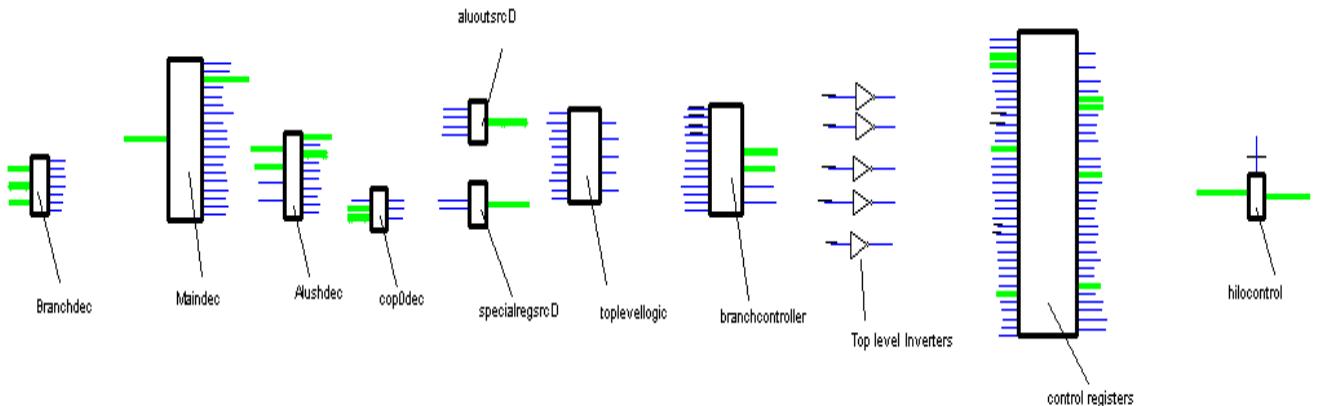
The controller unit has three PLAs in it. The blocks containing PLAs are the maindec, branchdec, and alushdec. The PLAs were created with the PLA generator written by the library team. The PLA uses a pseudo-nMOS design, with weak pMOS pull-up transistors. These weak transistors had a width of 3, and also a length of 3. The nMOS transistors in the PLA have a width of 4, and the input and output inverters have pMOS sizes of 10 and nMOS sizes of 5.

13.4. Special Unit

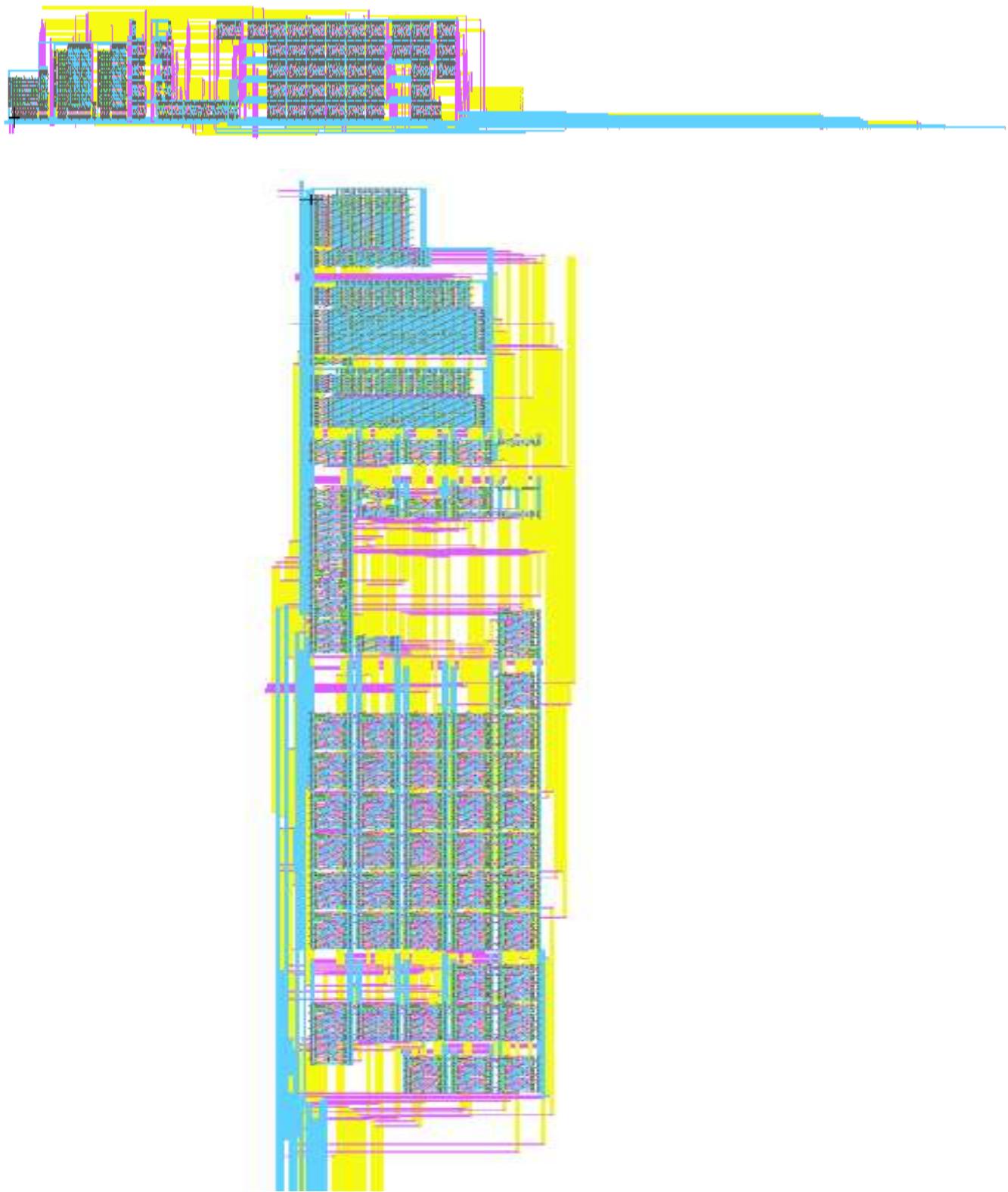
The top level controller generates the inverted reset signals and inverted stall signals for the registers. This was an arbitrary design choice that was intended to simplify the unit layout.

13.5. Schematic

The schematic was organized in the order that it would be placed in the layout to simplify the process.



13.6. Layout



13.7. Testing

The unit was tested with a Verilog test fixture. Test vectors were generated with the original mips Verilog code, and then run on the Verilog deck generated from Electric. The test vectors supplied to the controller have been provided. The unit passed all the vectors and it also passed testing when included in the chip schematic.

```
//Mips Controller Test Vectors
0_0_000000_100000_00000_00000_00000_xxxxxx0x0xxxxx00xxxxxx0x01000xxxxxxxxxxxxx0xx
0_0_000000_100000_00000_00000_00000_0xxxxx000x11xx001000100001000xx0000000000000000
0_0_000000_100000_00000_00000_00000_00000_00x0000000111x001000100001000x0000000000000000
0_0_000000_100000_00000_00000_00000_00000_000000000000111001000100001000000000000000000
0_0_000100_100000_00000_00000_00000_00000_000000000000100011100100010000100000000000000000
0_0_001111_100000_00000_00000_00000_00000_000000000000100011000001100001000000000000000000
0_0_100011_100000_00000_00000_00000_00000_000000000000100000010001010000101001010000100000000000
0_0_100001_100000_00000_00000_00000_00000_100000010001100000001000010000000000010000000000
0_0_001101_100000_00000_00000_00000_00000_110000011101110000001000010000000000011000000000
0_0_000001_100000_00000_00000_00000_00000_01100111010110000000100010000000000000000000000000
0_0_101011_001100_00000_00000_00000_00000_001000000010110000001000010000000000000000000000000
0_0_000001_000111_00000_00000_00000_00000_00000000000011000000100000001000010000100000000000000
0_0_000000_000110_00000_00000_00000_00000_00000000000000001000000000000000000000000000000000000
0_0_000000_101011_00000_00000_00000_00000_00000000000000001000000000000000000000000000000000000
0_0_000000_100001_00000_00000_00000_00000_0000000000001100000000001100010000000000000000000000000
1_0_000000_000110_00000_00000_00000_00000_0000000000001110000000010000000000000000000000000000000
0_1_000000_101011_00000_00000_00000_00000_0000000000001000000010000000100000001000000000000000000
```

```
//reset, pendingexception, opD[5:0], functD[5:0], rsD[4:0], rtD[4:0], stallD, stallE,
//stallM, stallW, flushE, flushM, aeqzD, aeqbD, agtzD, altzD, mdrunE, expectedresult
```

13.8. Subcomponents

13.8.1. Branch Decoder

Function

The branch decoder decodes the instructions for branching, and generates the control signals to define the branching. Most of the outputs from this block are passed to the branch controller, which uses them to generate the necessary signals for the specific type of branching indicated by the instruction.

I/O Table

The inputs and outputs of the branch decoder are shown below.

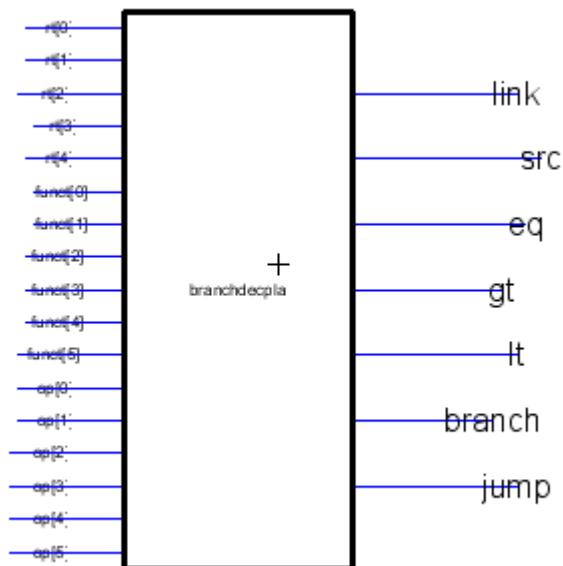
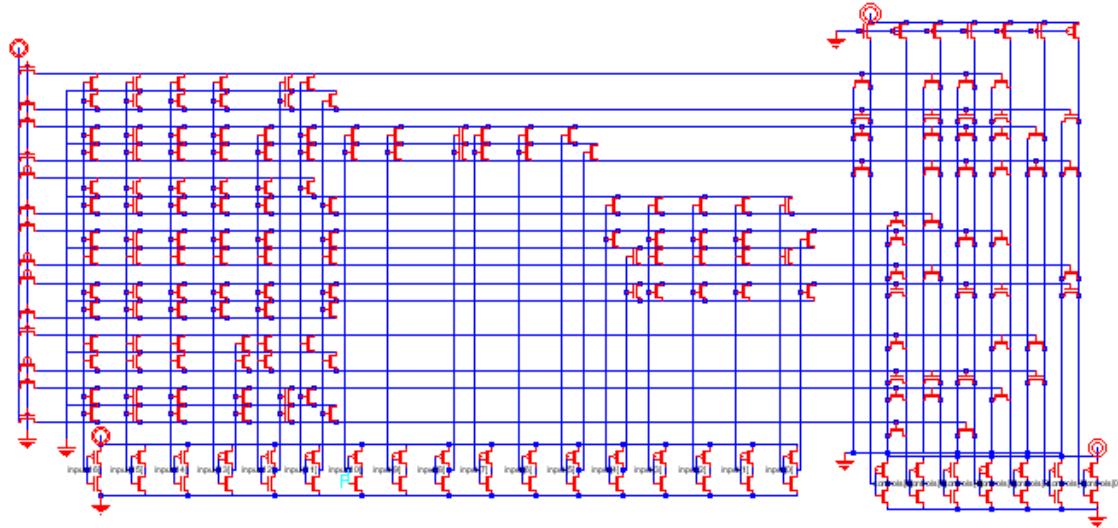
Input	Origin	Output	Destination
op[5:0]	Datapath	jump	Branch Controller
rt[4:0]	Datapath	branch	Branch Controller, Datapath
funct[5:0]		lt	Branch Controller
		gt	Branch Controller
		eq	Branch Controller
		src	Branch Controller
		link	Branch Controller

Special Units

The branch decoder used the PLA generator created by the library team for both the schematic and layout.

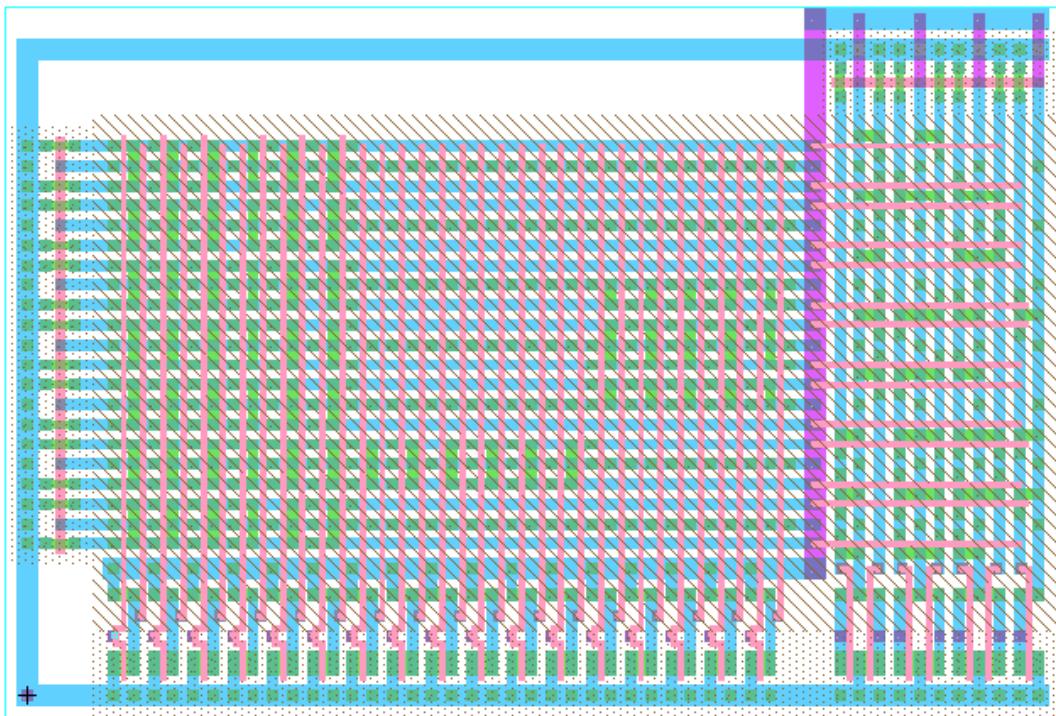
Schematic

The schematic for the entire block is shown below. The branch decoder only contained a case statement, and so the schematic consists only of the PLA.



Layout

The layout for the branch decoder is shown below. It is a fairly standard PLA, with the inputs in the lower left and the outputs in the lower right.



Testing

The schematic was first tested with IRSIM to determine if the unit was functioning properly. In this testing specific examples of the case statement were tested and the outputs were verified.

The unit was then tested in Verilog with the test bench for the entire controller. In order for the Verilog to properly test the pseudo-nMOS configuration, the pull-up pMOS transistors had to be labeled as “weak” in Electric. The entire controller passed the test bench, and so it was concluded that the branch decoder was functioning properly.

13.8.2. Main Decoder

Function

The maindec block is the main decoder for the control unit. It comes after the branchdec and before the alushdec blocks. It is mainly made up of the maindecpla, but it also includes a NOR gate that creates the last control signal needed, riD. The PLA generator did not handle the default case of all don't cares and one 1, so the logic for the default case had to be created outside of the PLA. Four signals were put into a NOR gate since at least one of them will be 1 for a valid op code. If all were zeroes, then the last control signal should be a 1. At a higher level, the main decoder takes in the op code of the instruction and produces many of the control signals that will be used to tell the datapath what to do with the current instruction. Many of the control signals produced

are passed through the pipeline registers so that they move along the pipeline with the instruction that they belong to.

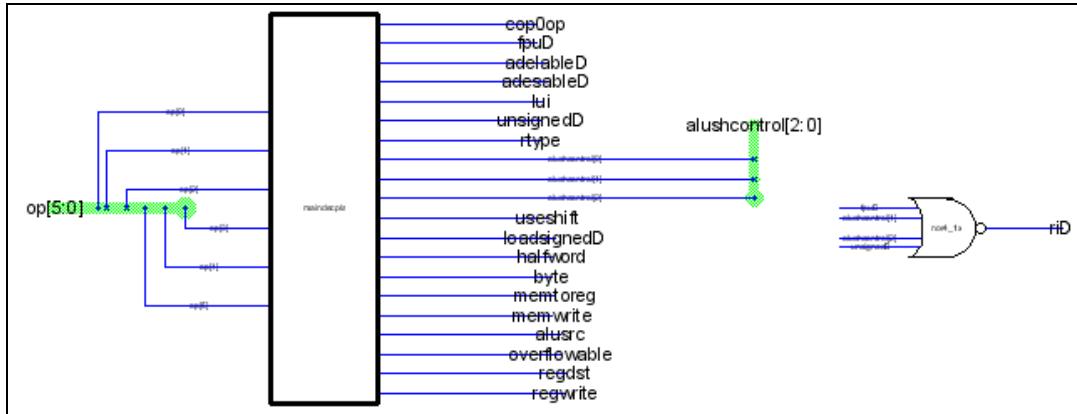
I/O Table

Input	Origin	Output	Destination
op[5:0]	Decode (Datapath)	adelableD	registers
		adesableD	registers
		alushcontrol[2:0]	alushdec
		alusrc	registers
		byte	registers
		cop0op	cop0dec
		fpuD	registers
		halfword	registers
		loadsingedD	registers
		lui	registers
		memtoreg	registers
		memwrite	registers
		overflowable	toplevellogic
		regdst	toplevellogic
		regwrite	toplevellogic
		riD	registers
		type	alushdec
		unsignedD	datapath
		useshift	alushdec

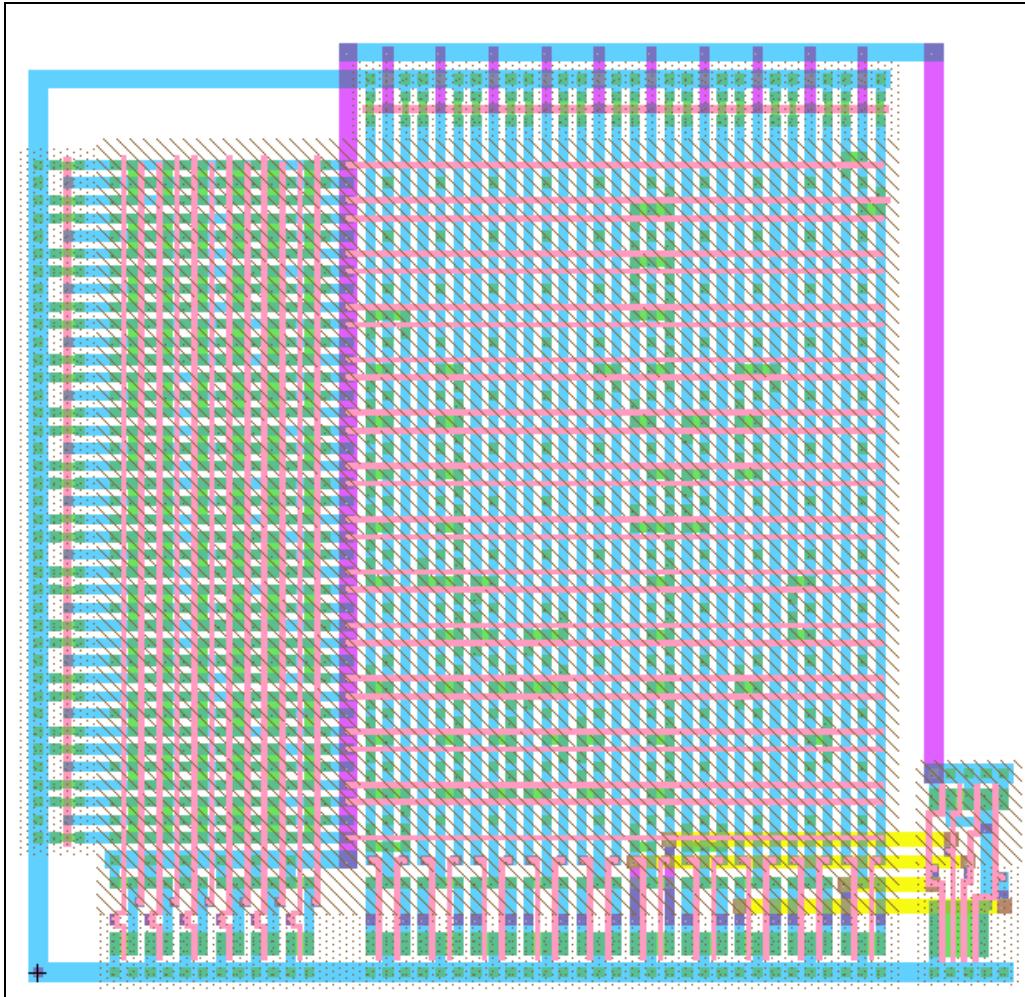
Special Units

Used the maindecpla block which had the PLA schematic and layout generated by the PLA generator created by the Library team.

Schematic



Layout



Testing

This block was tested as part of the overall controller module, which passed the chip tests created by the microarchitecture team.

13.8.3. Main Decoder PLA

Function

The maindecpla block is the main part of the main decoder of the control unit. It comes after the branchdec and before the alushdec blocks. It takes in the op code of the instruction and produces most of the control signals that will be used to tell the datapath what to do with the current instruction. Many of the control signals produced are passed through the pipeline registers so that they move along the pipeline with the instruction that they belong to.

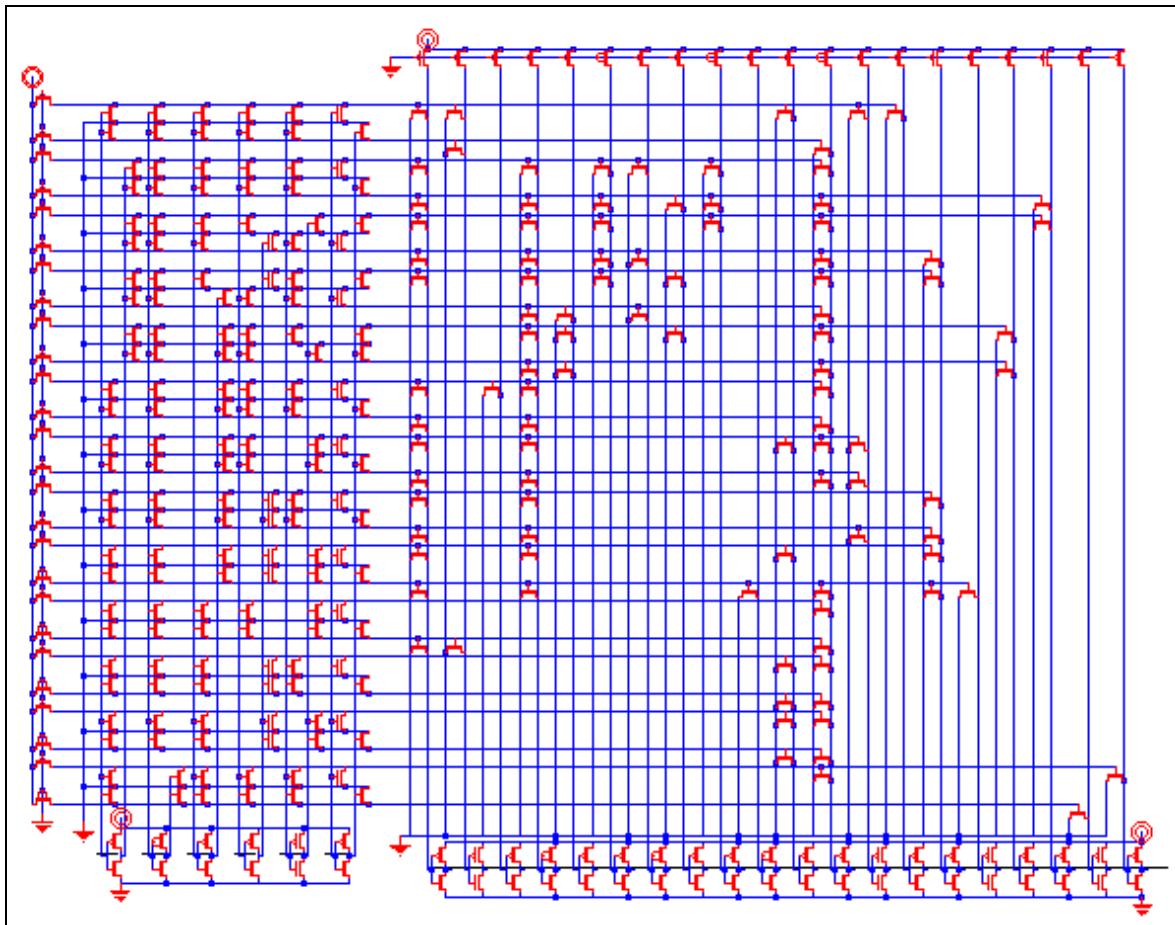
I/O Table

Input	Origin	Output	Destination
op[5:0]	Decode (Datapath)	controls[0]	maindec
		controls[1]	maindec
		controls[2]	maindec
		controls[3]	maindec
		controls[4]	maindec
		controls[5]	maindec
		controls[6]	maindec
		controls[7]	maindec
		controls[8]	maindec
		controls[9]	maindec
		controls[10]	maindec
		controls[11]	maindec
		controls[12]	maindec
		controls[13]	maindec
		controls[14]	maindec
		controls[15]	maindec
		controls[16]	maindec
		controls[17]	maindec
		controls[18]	maindec
		controls[19]	maindec

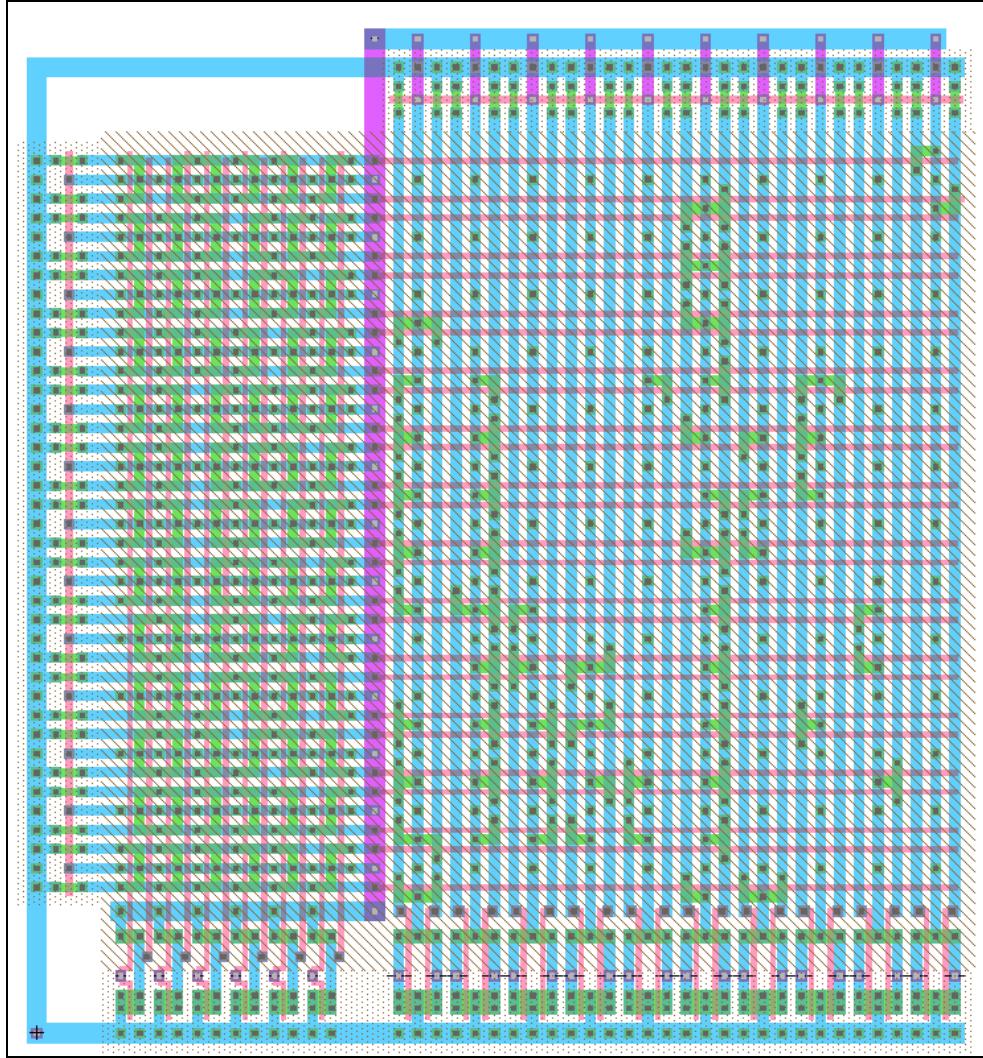
Special Units

The PLA schematic and layout was created using the PLA generator developed by the Library team.

Schematic



Layout



Testing

Another PLA was tested using HSPICE to verify that the weak pMOS pull-up transistors were weak enough, and the pMOS transistors were found to be weak enough. This block was tested as part of the overall controller module, which passed the chip tests generated by the microarchitecture team.

13.8.4. ALU and Shifter Decoder

Function

The ALU and Shifter Decoder decodes instructions for the ALU and shifter, and generates the control signals for these blocks. The outputs from this block go into aluoutsrsrcD, specialregsrcD, toplevellogic, or registers blocks in the control unit.

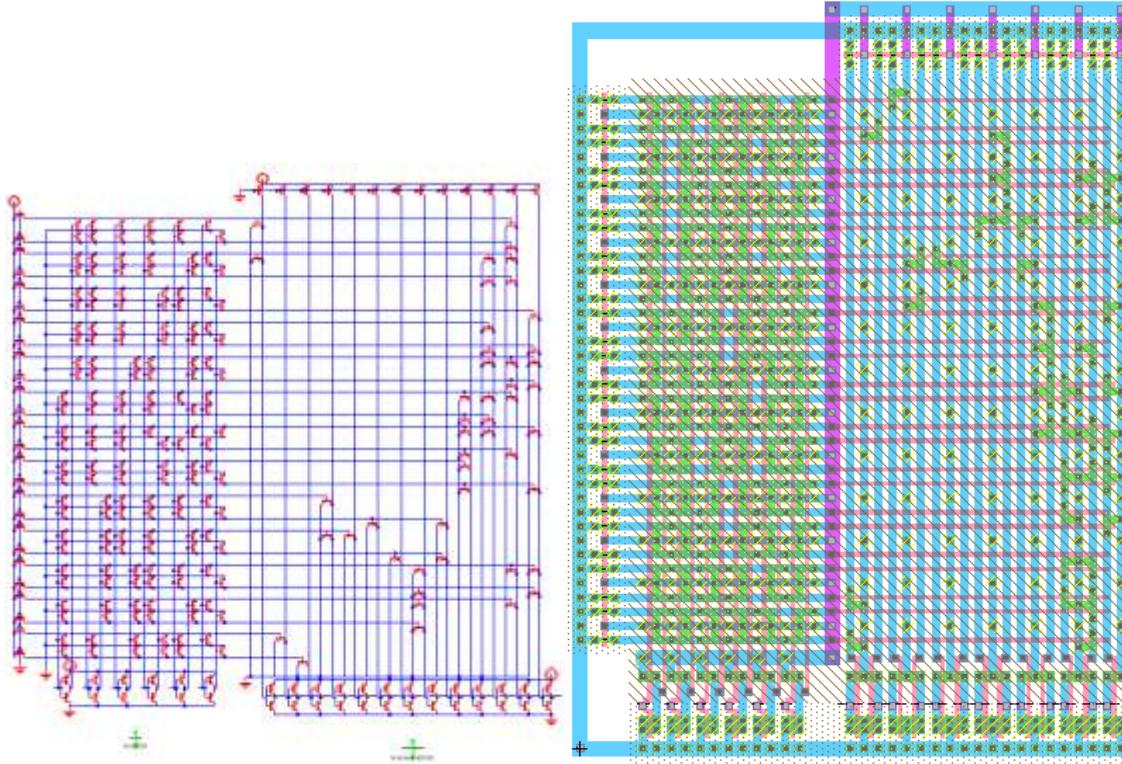
I/O Table

The inputs and outputs of the ALU and Shifter Decoder are listed below

Input	Origin	Output	Destination
Alushmaincontrol[2:0]	Main Decoder	Alushcontrol[2:0]	Registers
Funct[5:0]	Memsys	BreakD	Registers
Maindecuseshifter	Main Decoder	Hilodisable[1:0]	Registers
Rtype	Main Decoder	Hiloread	Aluoutsrd, specialregsrcD, toplevellogic
		Hilosel	SpecialregsrcD, Registers
		Hilosrc	Registers
		Mdstart	Toplevellogic, Registers
		Overflowable	Toplevellogic
		SyscallD	Registers
		Useshifter	Aluoutsrd

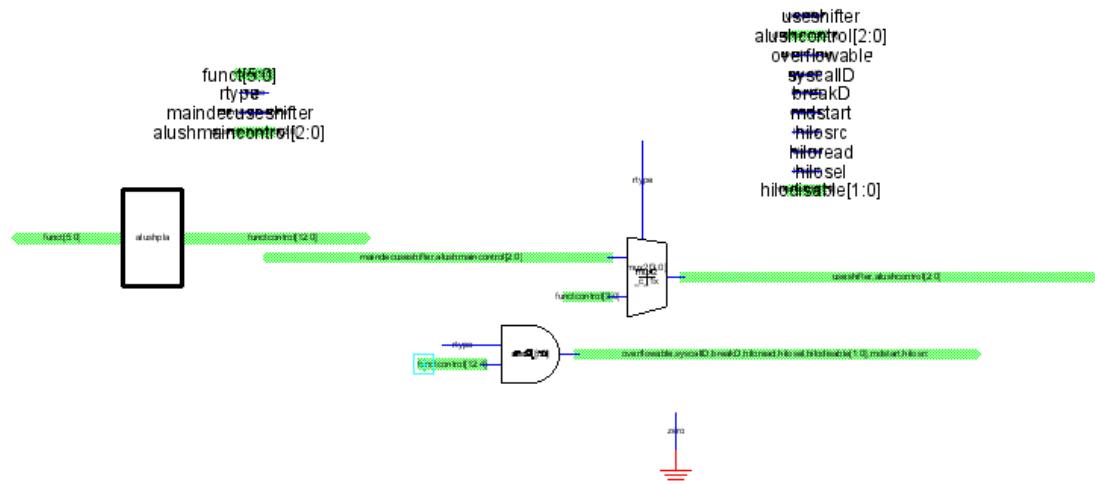
Special Units

The ALU and Shifter Decoder contains a PLA, which was created using the PLA generator written by the library team. The PLA schematic and layout are shown below. This PLA uses pseudo-nMOS nor gates to generate the minterms and the output. The pMOS pullup transistors have length and width 3λ , and are labeled as weak so that they simulate correctly.



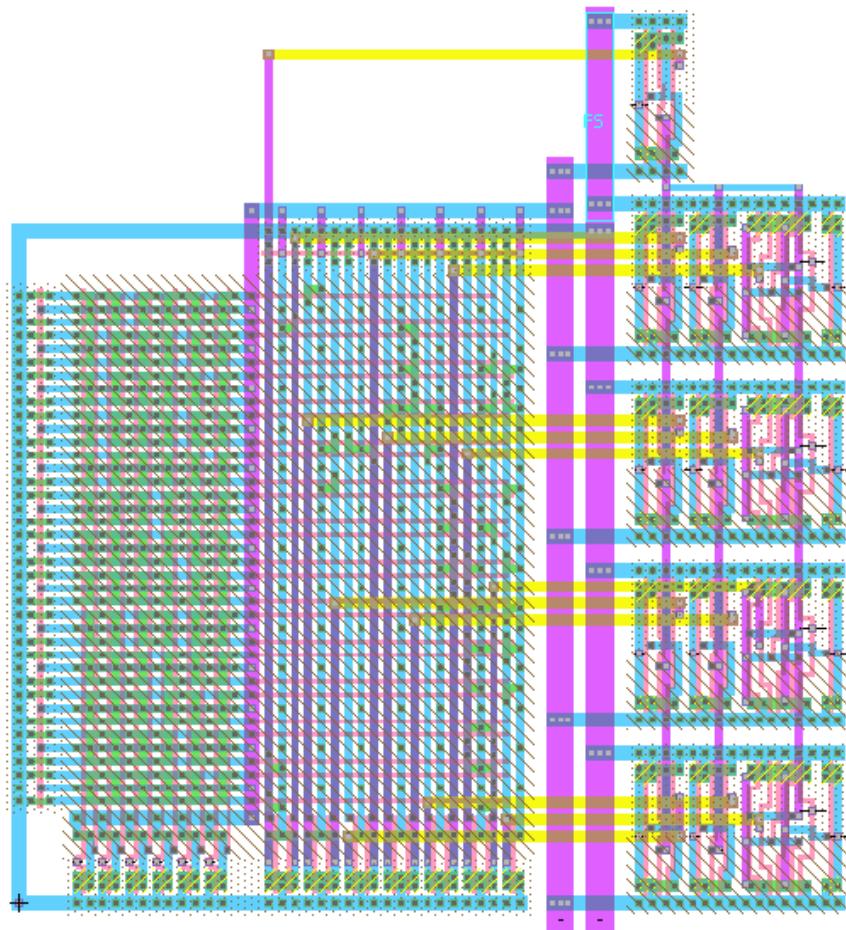
Schematic

The schematic of the entire block is shown below. The and gates perform the task of a 2-input multiplexer with a grounded first input.



Layout

The layout of the entire block is shown below. Metal 3 was used to connect the outputs of the PLA to the inputs of the multiplexers and and gates.



Testing

The Schematic and Layout were both tested using a Verilog testbench. In this testing, each of the valid inputs to the PLA was tested, as well as all inputs to the multiplexers.

13.8.5. Coprocessor Decoder

Function

The Coprocessor Decoder decodes the instructions for the coprocessor, and generates control signals for the coprocessor. The outputs from this block are passed to aluoutsrd, registers, and toplevellogic, other blocks in the Controller. The inputs are taken from the instruction, and from the main decoder.

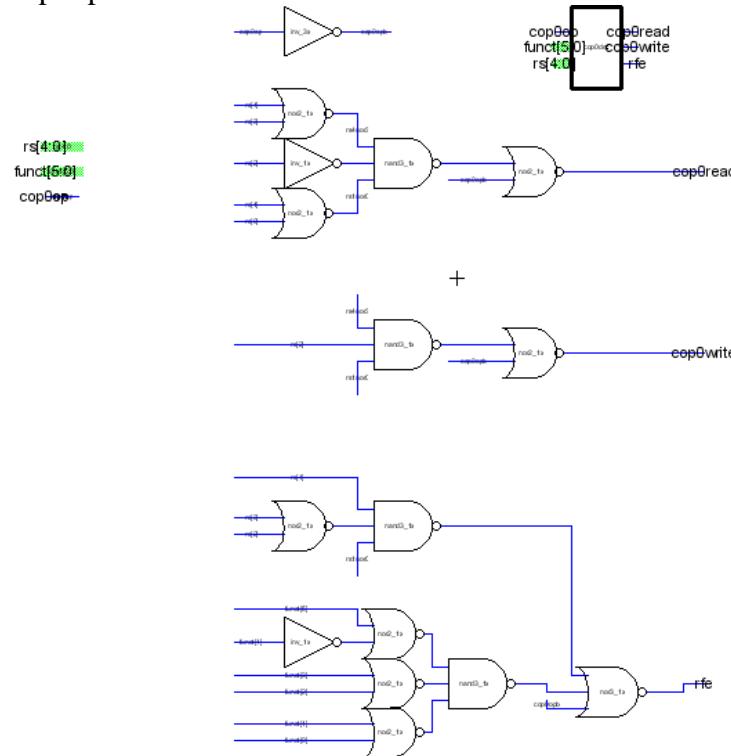
I/O Table

The inputs and outputs of the ALU and Shifter Decoder are listed below

Input	Origin	Output	Destination
Cop0op	Main Decoder	Cop0read	Toplevellogic, aluoutsrd
Funct[5:0]	Instruction	Cop0write	Toplevellogic, registers
Rs[4:0]	Instruction	rfe	Registers

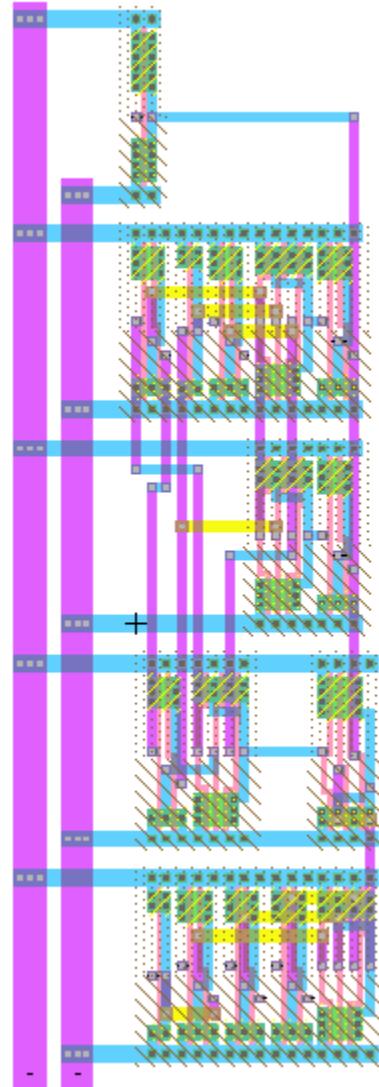
Schematic

The schematic of the entire block is shown below. Some of the nor gates are shared between output paths to reduce the total size of the cell.



Layout

The layout of the entire block is shown below. The inverter at the top inverts the cop0op signal so that nor gates could be used in the design.



Testing

The Schematic and Layout were both tested using a Verilog testbench. In this testing, all values that should produce a high reading on one of the output signals was tested, as well as many cases that should not.

13.8.6. aluoutsrCD

Function

The aluoutsrCD module is in between the toplevellogic and specialregsrcD modules in the control unit. This block produces a two-bit output that is eventually used in the datapath to select the output a 4:1 32-bit multiplexer that chooses between the ALU results and other 32 bit busses.

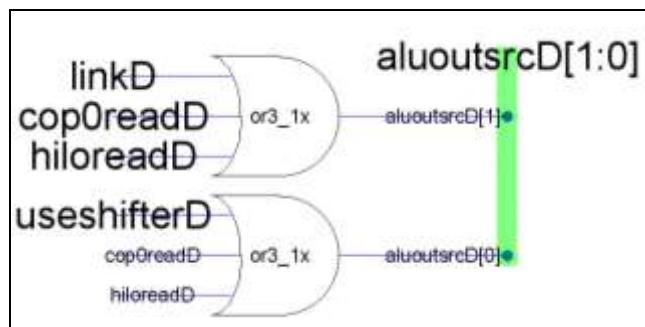
I/O Table

Input	Origin	Output	Destination
linkD	branchdec	aluoutsrcD[1:0]	registers (Controller)
cop0readD	cop0dec		
hiloreadD	alushdec		
useshifterD	alushdec		

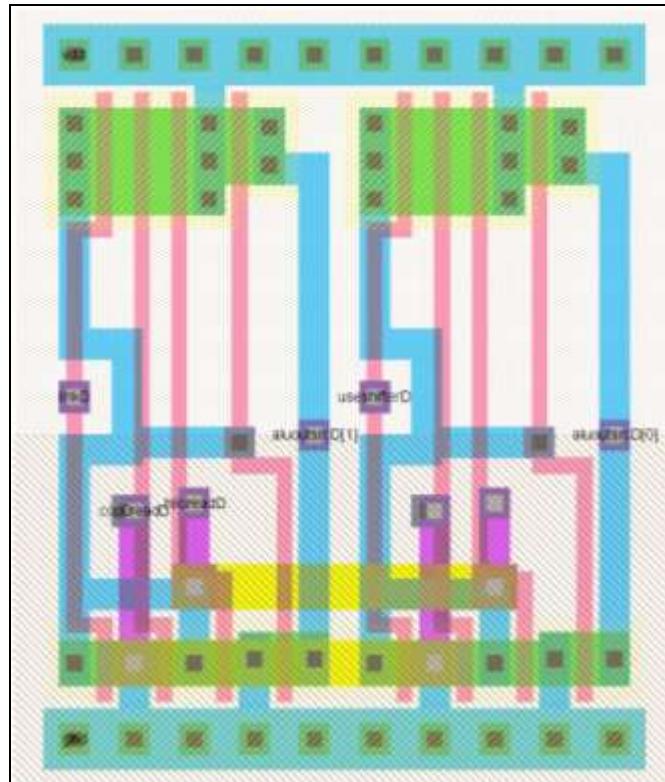
Special Units

Only standard static CMOS gates from muddlib were used.

Schematic



Layout



Testing

This module was tested in IRSIM. All 16 possible inputs were generated, and the module passed all of the tests.

13.8.7. specialregsrcD

Function

The specialregsrcD module is in between the aluoutsrCD and branchcontroller modules in the control unit. This module takes in two input signals generated by the alushdec module and produces a control signal used by the Execute stage of the datapath. Specifically, it controls a 3:1 32-bit multiplexer that chooses between hi and lo registers and data from Coprocessor 0.

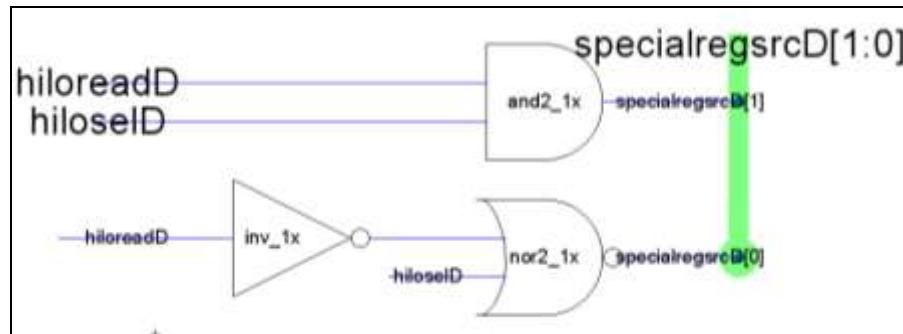
I/O Table

Input	Origin	Output	Destination
hiloread	alushdec	specialregsrcD[1:0]	registers (Controller)
hiloselD	alushdec		

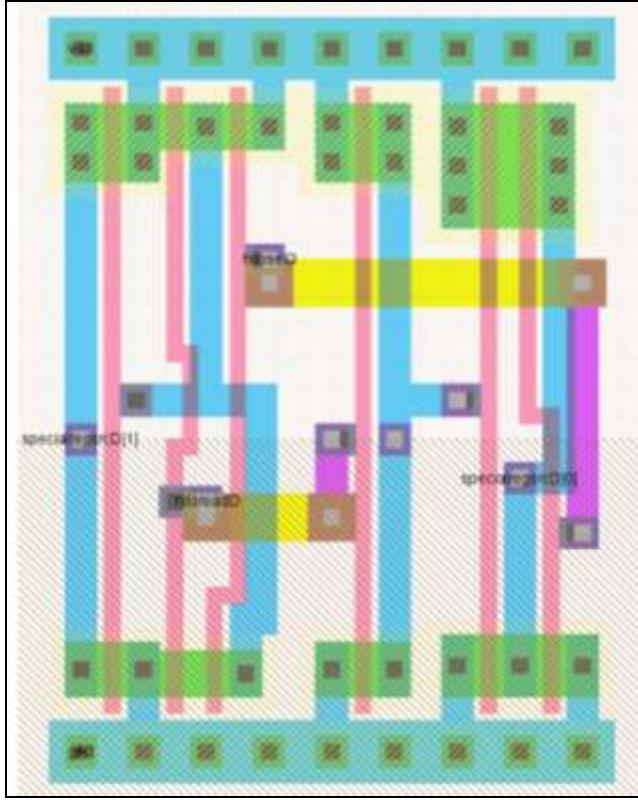
Special Units

Only standard static CMOS gates from muddlib were used.

Schematic



Layout



Testing

This module was tested in IRSIM by testing all four possible cases. The module passed all of the tests.

13.8.8. Control Top Level Logic

Function

The Top Level Logic in the Controller generates outputs to be sent to the Datapath. It Generates the regdst, regwrite, overflowable, bds, and hiloaccess signals, all of which either go directly to the Datapath, or go into registers, to be sent to the Datapath in later stages.

I/O Table

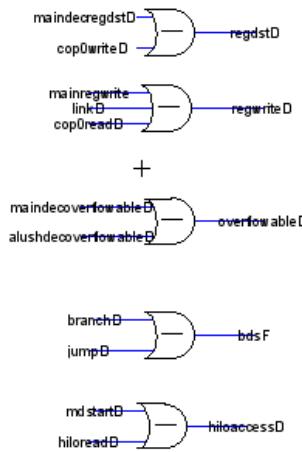
The inputs and outputs of the ALU and Shifter Decoder are listed below

Input	Origin	Output	Destination
maindecregdstD	maindec	bdsF	registers
cop0writeD	cop0dec	hioloaccessD	Datapath
mainregwrite	maindec	overflowableD	registers
linkD	branchdec	regdstD	registers
cop0readD	cop0dec	regwriteD	registers
maindecoverflowableD	maindec		

alushdecoverflowableD	alushdec		
branchD	branchdec		
jumpD	branchdec		
mdstartD	alushdec		
hiloreadD	alushdec		

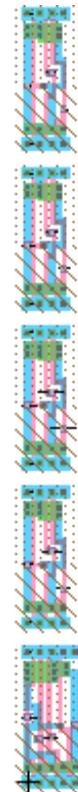
Schematic

The schematic of the entire block is shown below. This block is simply a series of two and three input or gates, directly between inputs and outputs.



Layout

The layout of the entire block is shown below. It is five individual or gates.



Block testing

The Schematic and Layout were both tested using a Verilog testbench.

13.8.9. Branch Controller

Function

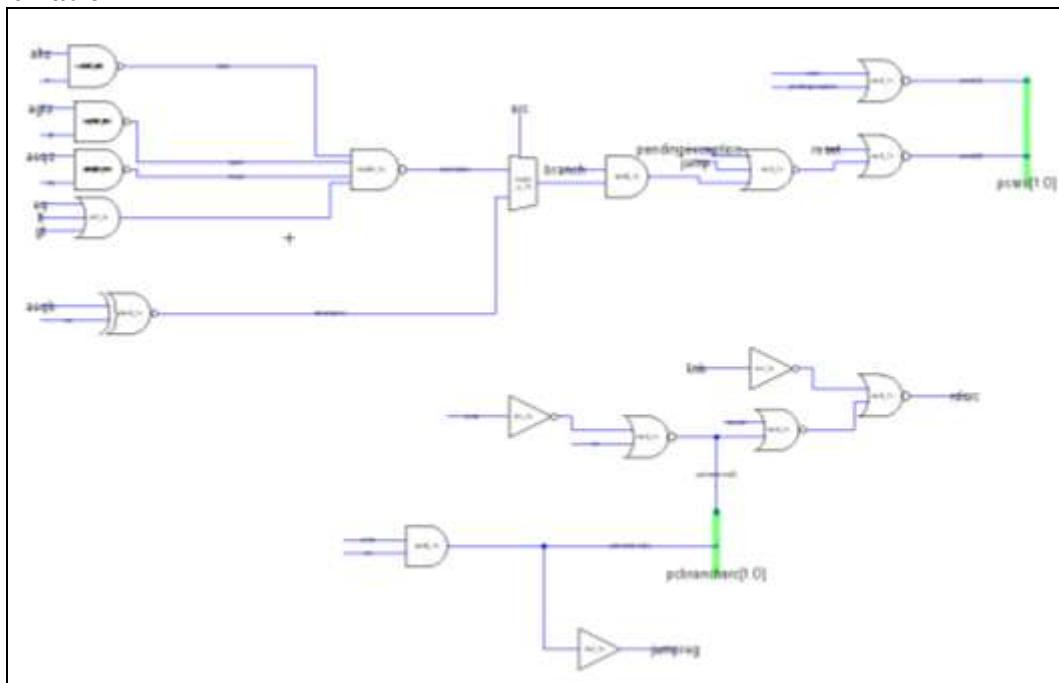
The branchcontroller module is part of the controller unit and is located in between the specialregsrd module and the register files. It generates four outputs that go to the datapath, which control various branching functions of the processor.

I/O Table

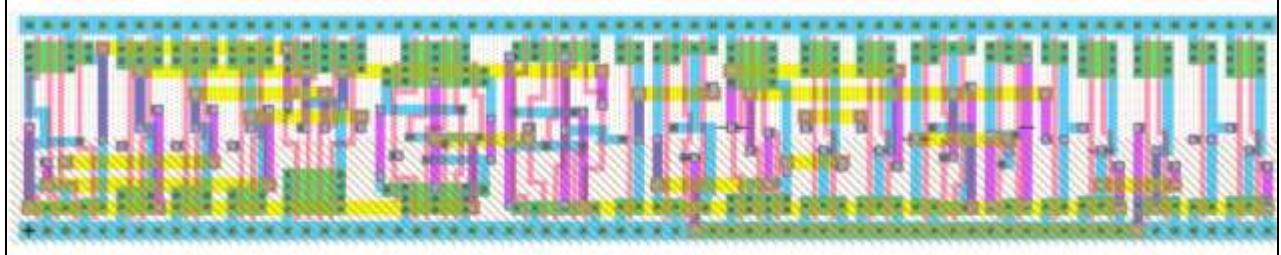
Input	Origin	Output	Destination
aeqbD	Decode (Datapath)	pcbranchsrcD[1:0]	Decode (Datapath)
aeqzD	Decode (Datapath)	pcsrcFD[1:0]	Fetch (Datapath)
agtzD	Decode (Datapath)	rdsrcD	Five Bit Datapath
altzD	Decode (Datapath)	jumpregD	Hazard Unit
branchD	branchdec		
eqD	branchdec		
gtD	branchdec		
jumpD	branchdec		
linkD	branchdec		
ltD	branchdec		
pendingexception	Coprocessor0		
reset	Reset		
brsrcD	branchdec		

Special Units

This module used all standard static CMOS cells from muddlib.

Schematic

Layout



Testing

A self-checking Verilog testbench with over 15 test vectors that represent possible inputs to the unit was created and run on the module, and all test vectors passed. Also, this block was tested by running the overall chip test on the controller unit. The unit passed, which indicates that the branchcontroller unit works.

13.8.10. Control Registers

Function

The registers in the controller are used to store values of control signals that will be used in later stages. There are 4 registers in the controller: 1-bit RegD, 31-bit RegE, 6-bit RegM, and 3-bit RegW.

I/O Table

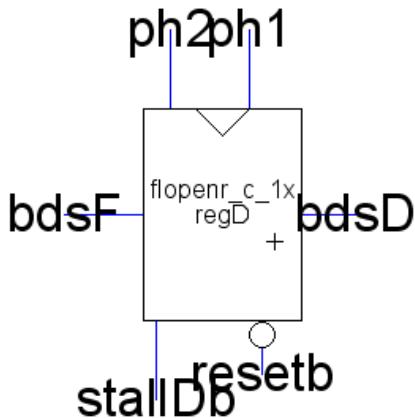
Input	Origin	Output	Destination
ph1	Chip Input	memtoregE	Datapath
ph2	Chip Input	memtoregM	Datapath
Reset	Chip Input	adelableE	Coprocessor
Resetb	Controller	adesableE	Coprocessor
stallDb	Datapath	aluoutsrce[1:0]	Datapath
stallEb	Datapath	alushcontrolE[2:0]	Datapath
stallMb	Datapath	alusrcE	Datapath
stallWb	Datapath	bdsE	Coprocessor
flushE	Datapath	breakE	Coprocessor
flushM	Datapath	byteM	Datapath
adelableD	Maindec	fpuE	Coprocessor
adesableD	Maindec	halfwordM	Datapath
aluoutsrCD[1:0]	AluoutsrCD	hilodisablealushE[1:0]	Controller
Alushcontrol[2:0]	Alushdec	hiloselE	Controller
alusrcD	Maindec	hilosrcE	Datapath
bdsF	Toplevellogic	loadsingedM	Datapath
breakD	Alushdec	luiE	Datapath
byteD	Maindec	mdstartE	Datapath
Cop0writeD	Cop0dec	memtoregW	Datapath
fpuD	Maindec	memwriteM	Memsys

halfwordD	Maindec	overflowableE	Coprocessor
hilodisablealushD[1:0]	Alushdec	regdstE	Datapath
hiloselD	Alushdec	halfwordE	Coprocessor
hilosrcD	Alushdec	rfeE	Coprocessor
loadsingedD	Maindec	riE	Coprocessor
luiD	Maindec	specialregsrcE[1:0]	Datapath
mdstartD	Alushdec	syscallE	Coprocessor
memtoregD	Maindec	regwriteW	Datapath
memwriteD	Maindec	regwriteE	Datapath
overflowableD	Toplevellogic	Cop0writeW	Coprocessor
regdstD	Toplevellogic	regwriteM	Datapath
regwriteD	Toplevellogic		
rfeD	Cop0dec		
riD	Maindec		
specialregsrcD[1:0]	SpecialregsrcD		
syscallD	Alushdec		

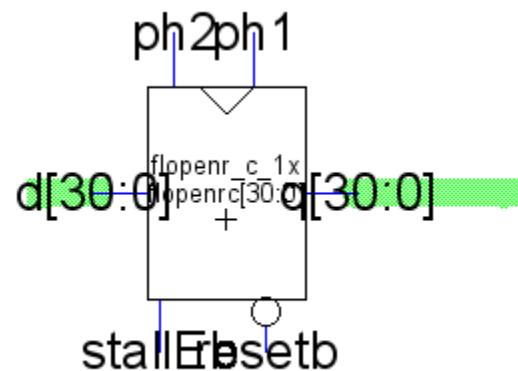
Schematic

The schematic for each individual register block, as well as the entire control register block are shown below. The icon for the registers is also shown.

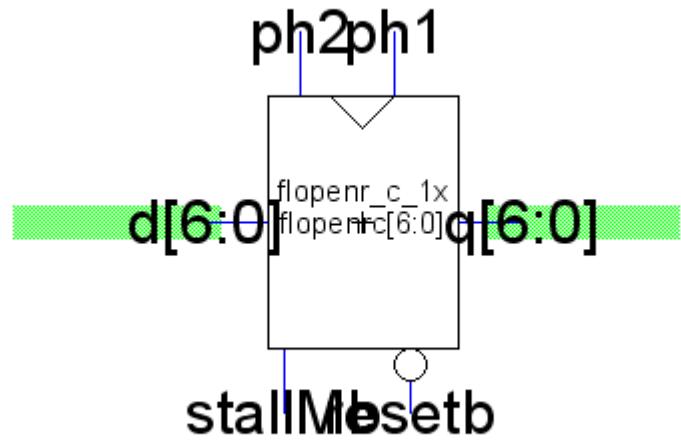
RegD



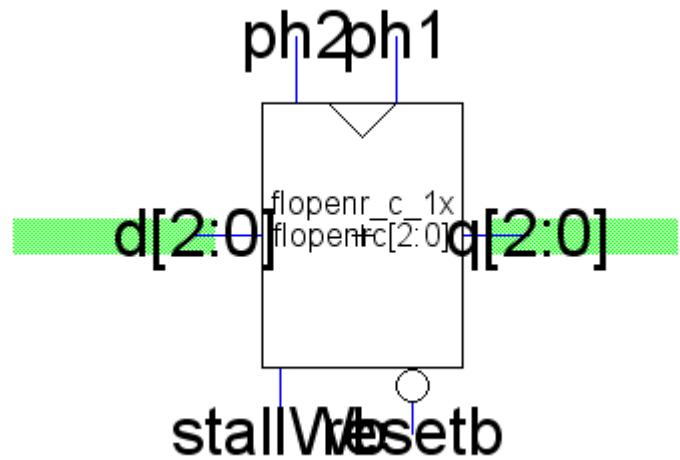
RegE



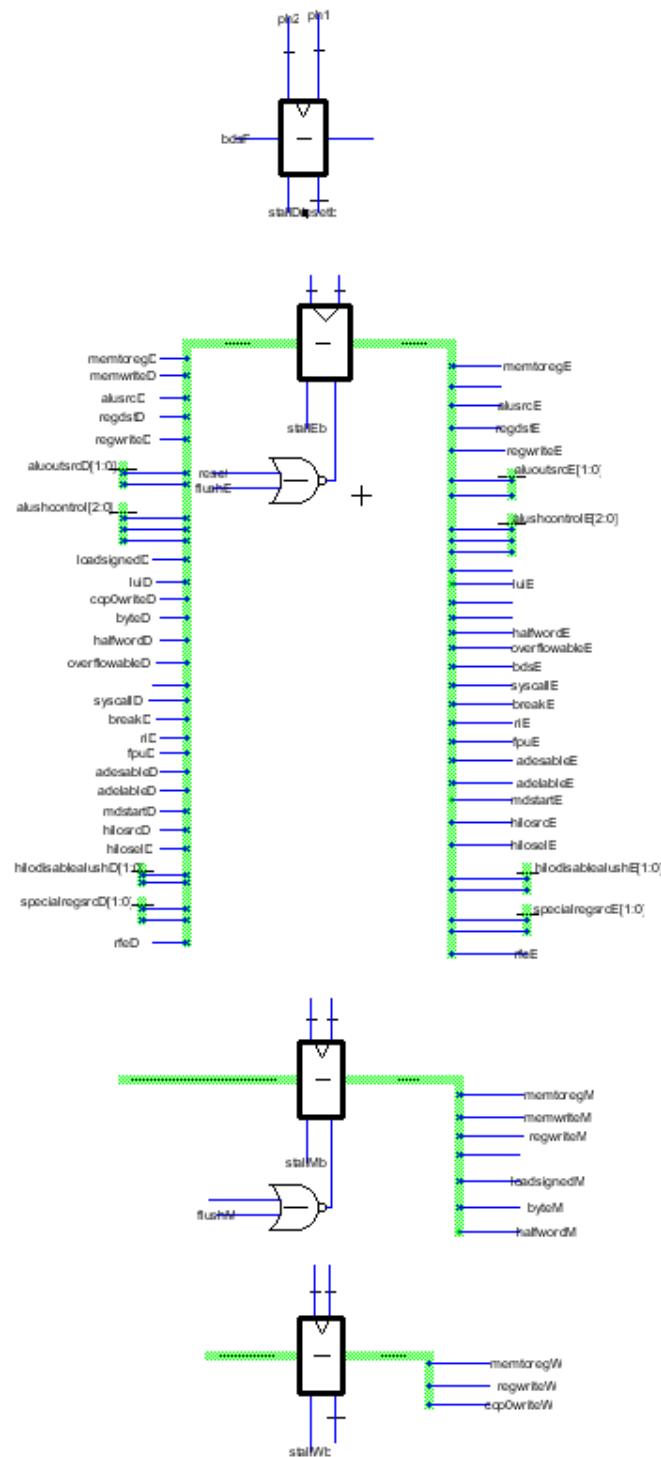
RegM



RegW



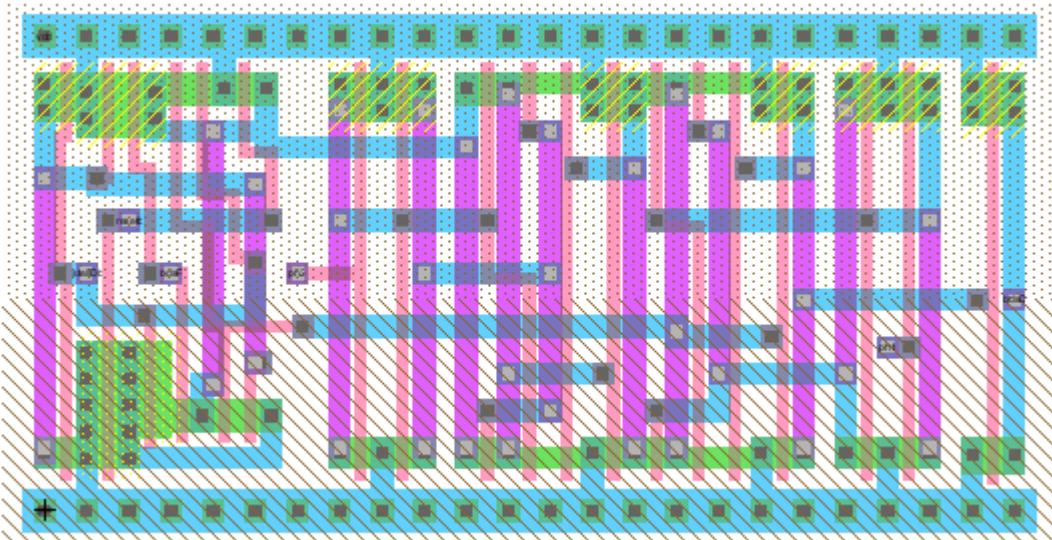
All Registers



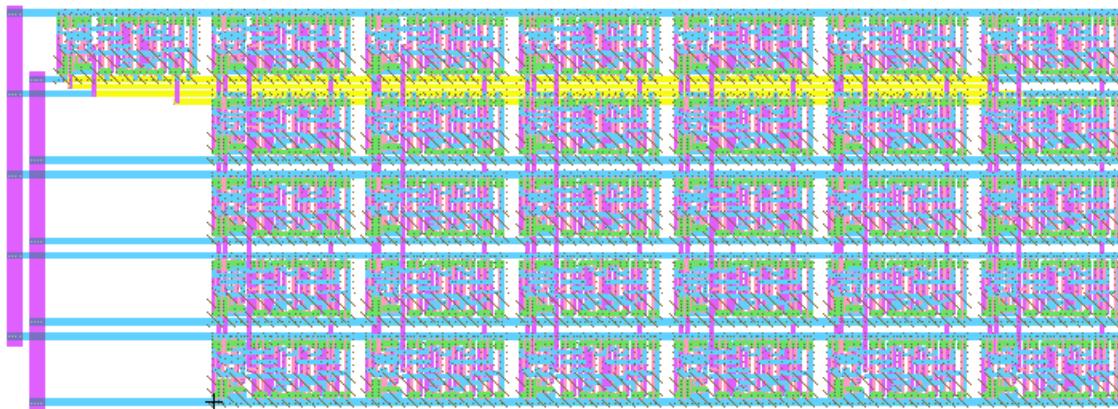
Layout

The layout for each individual register is shown as well as for the entire block.

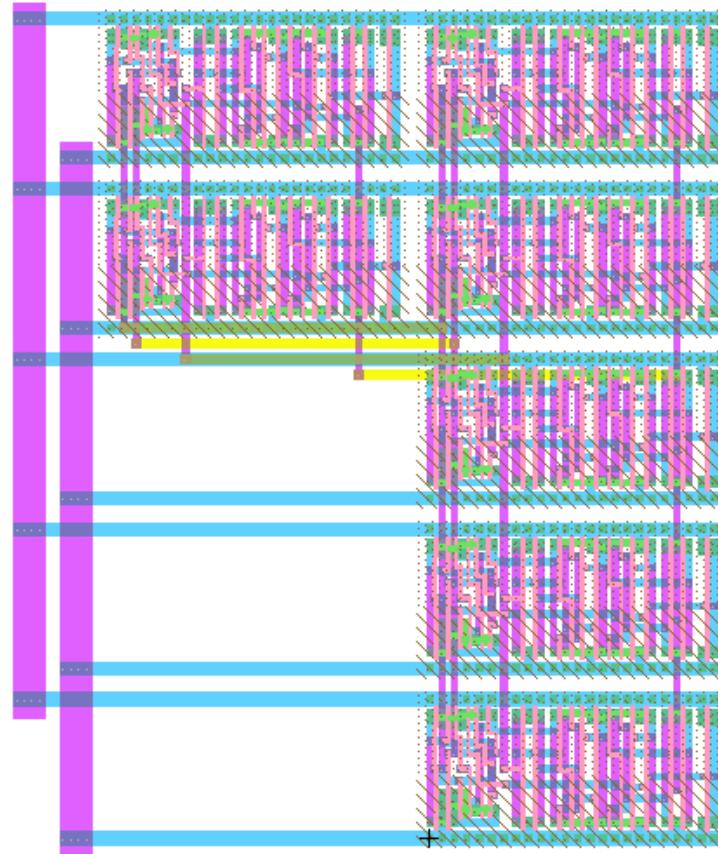
RegD



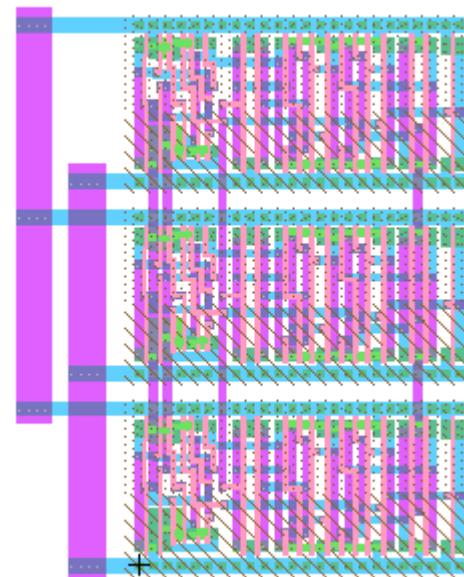
RegE



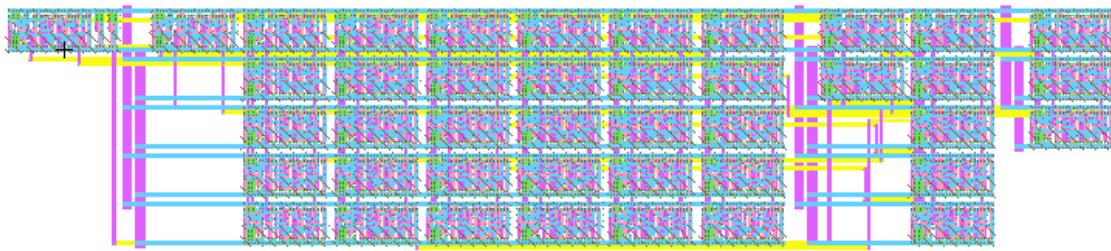
RegM



RegW



All Registers



Block Testing

This block was tested in Verilog with simple tests on a representative register. Because the unit consists only of registers, simply showing that the register was functional was enough to know that the entire block was functional.

13.8.11. hilocontrol

Function

The hilocontrol module is to the right of the pipeline registers in the control unit. This block takes in two input signals and produces a control signal for the Execute stage of the datapath, specifically for the mult/div unit.

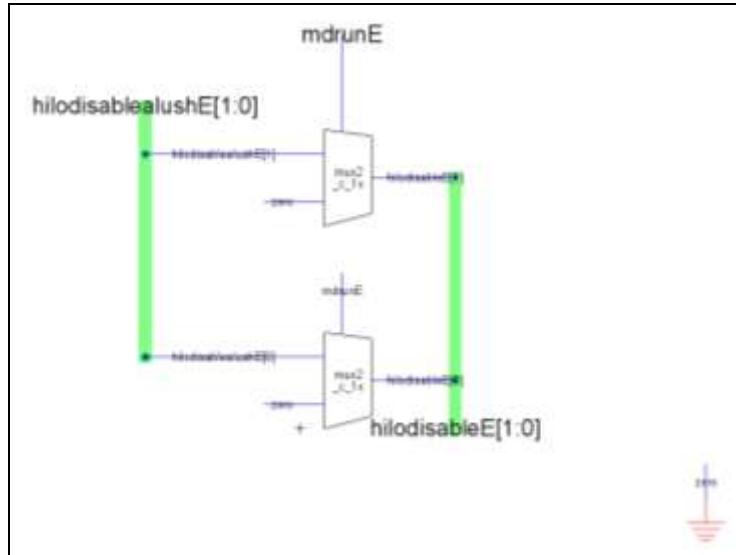
I/O Table:

Input	Origin	Output	Destination
mdrunE	Hazard Unit	hilodisableE[1:0]	Execute (Datapath)
hilodisablealushE[1:0]	registers (Controller)		

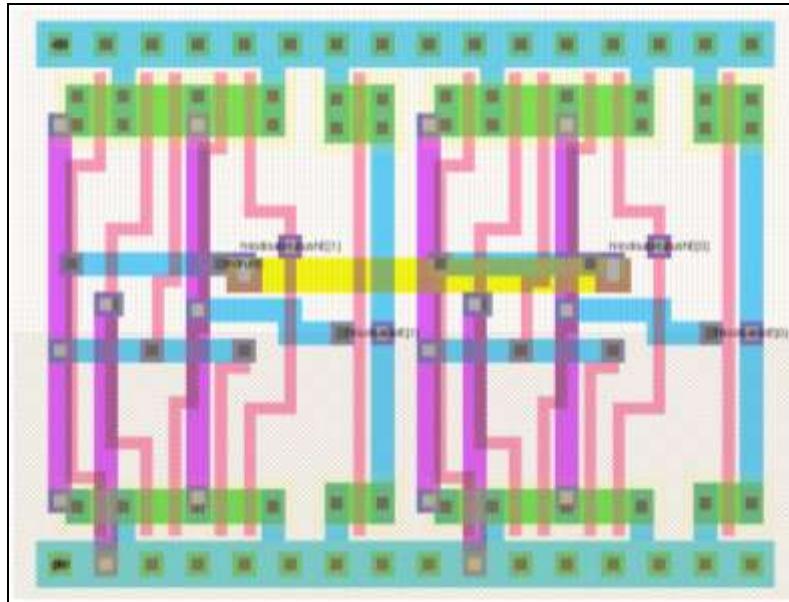
Special Units

Only standard static CMOS gates from muddlib were used.

Schematic



Layout



Testing

This module was tested in IRSIM. All 8 possible inputs were generated, and the module passed all of the tests.

14. Memory

14.1. Function

The memory subsystem, known formally as the memsys module, consists of two 512-byte caches, control logic, and a write buffer. The primary function of this module is to cache instructions and data for use by the processor. Since the cache system implemented is specified as write-through, the memory system not only interacts with the processor but also outputs information to be stored in main memory. The write buffer and memsys control logic manage the interface between the caches and main memory.

14.2. Unit I/O

Input	Origin	Output	Destination
[31:0] memdata	External memory	[31:0] instrF	Datapath
[31:0] writedataM	Datapath	[31:0] readdataM	Datapath
[31:2] pcF	Datapath	[31:0] memdata	External Memory
[29:0] adrM	Datapath	[26:0] memadr	External Memory
[3:0] byteenM	Datapath	[3:0] membyteen	External Memory
memwriteM	External Memory	dataackM	Control
memdone	External Memory	instrackF	Control
reF	Control	memen	External Memory
reM	Control	memrwb	External Memory
swc	Control		

ph1	Clock		
ph2	Clock		
reset	Reset		

Table 3. Memory Unit Input-Output table.

14.3. Special Units

All special units used are described in the unit sub-component section below.

14.4. Schematic

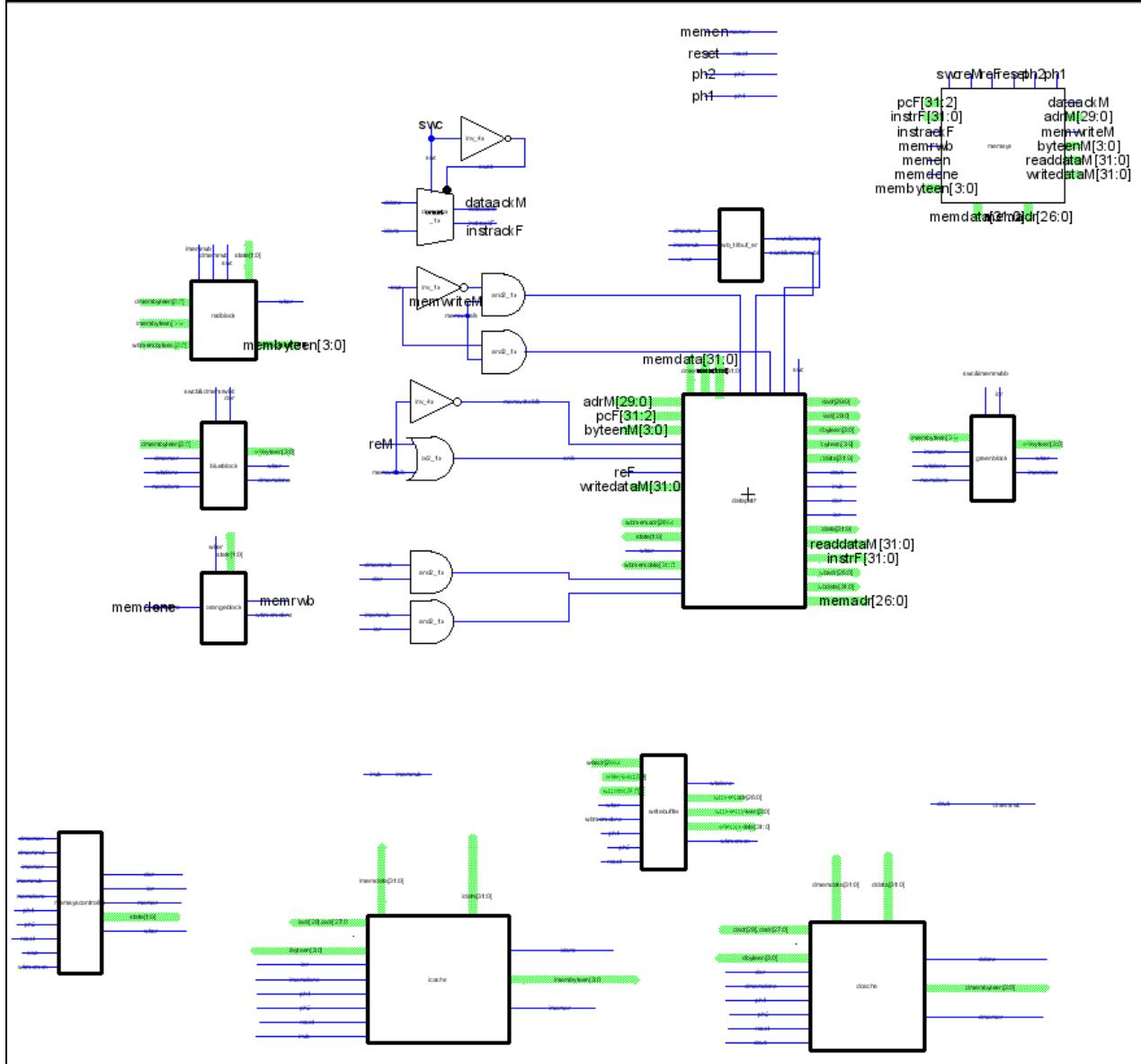


Figure 46. Memory Unit Schematic.

This schematic displays the overall connections between all of the sub-components of the memory unit. The sub-components are described in detail in the following sections.

14.5. Layout



Figure 47. Complete layout of memsys module showing all unit sub-components.

14.6. Subcomponents

14.6.1. Memsys Module

Function

The memsys block consists primarily of a datapath section and secondarily of small blocks of combinational logic. All of these pieces are concerned only with routing busses containing data and their associated addresses among the cache, the write buffer, external memory, the rest of the chip. The memsys block is the top level block in the memory unit and connects to the datapath and external memory. Internally, the memsys block connects to the instruction and data caches, the write buffer, and the memory control logic.

I/O Table

Input	Origin	Output	Destination
adrM[29:0]	Datapath	dataackM	Control
byteenM[3:0]	Datapath	instrF[31:0]	Datapath
memdone	External Memory	memadr[26:0]	External Memory
memwriteM	External Memory	membyteen[3:0]	External Memory
pcF[31:2]	Datapath	memen	External Memory
ph1	Clock	memrwb	External Memory
ph2	Clock	readdataM[31:0]	Datapath
reF	Control	memdata[31:0]	External Memory
reM	Control		
reset	Reset		
swc	Control		
writedataM	Datapath		
memdata[31:0]	External Memory		

Table 4. Memsys block input-output table

Special Units

The memsys block is split up into 5 main sections, the datapath and four smaller blocks that are given names from colors. Each colored block is associated with a single sub-block like the data cache or the write buffer.

The datapath requires the use of a complementary multiplexer cell. This cell takes two input signals and a control signal and has two outputs. The first output behaves exactly as a regular multiplexer, choosing one of the inputs based on the control signal. The second output chooses the other input, opposite to the first output, also based on the control signal. The cell was built basically as two multiplexers with complementary control signals. It was important that the layout be done correctly because the cell is used in the datapath and the two inputs and outputs always travel in opposite directions. Therefore, if the layout was done correctly, both input and both output signals could lie in the same track of metal 3, thus saving space. As can be seen in Figure 48. Layout of **Cmux2_dp_1x** cell, d1 and y1 live on the left side of the cell, while d0 and y0 live on the right side.

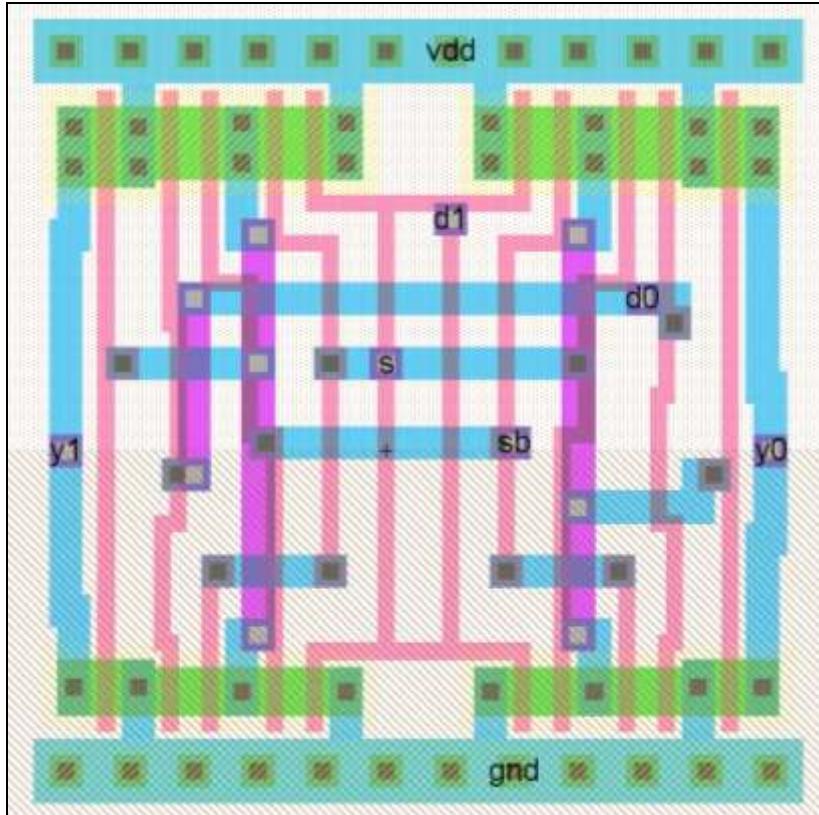


Figure 48. Layout of Cmux2_dp_1x cell

The green and blue blocks correspond to the instruction and data caches and contain the routing cells to allow the 4-bit byte mask bus to connect to these blocks. The orange block corresponded to the write buffer and contains some control logic. The red block connects to all the sub-blocks and allows for some control signals to pass between them all. As an example, Figure 6 shows the schematic for the blue block used in the main memsys layout.

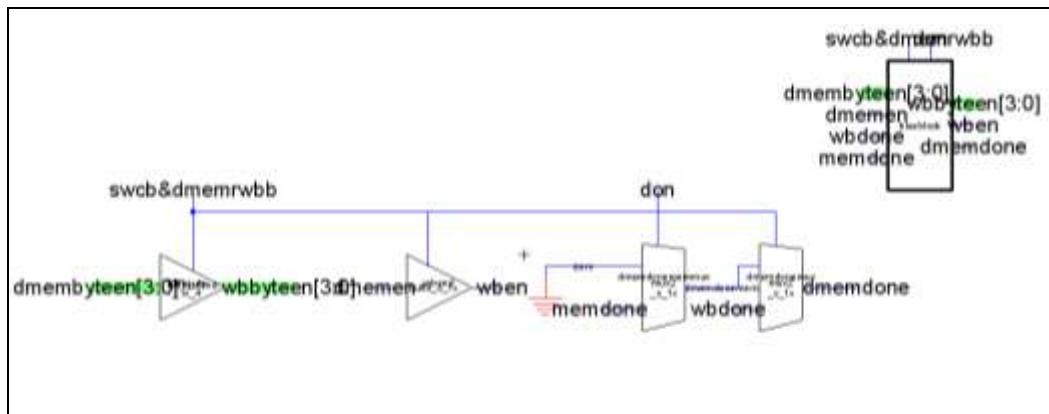


Figure 49. Blue block schematic

Schematic

The schematic was rather busy before the colored sub-blocks were made. This extra partitioning made the overall schematic much simpler and made debugging problems easier. Each block was assembled and tested on its own, and then all the working blocks were connected together with just a few wires.

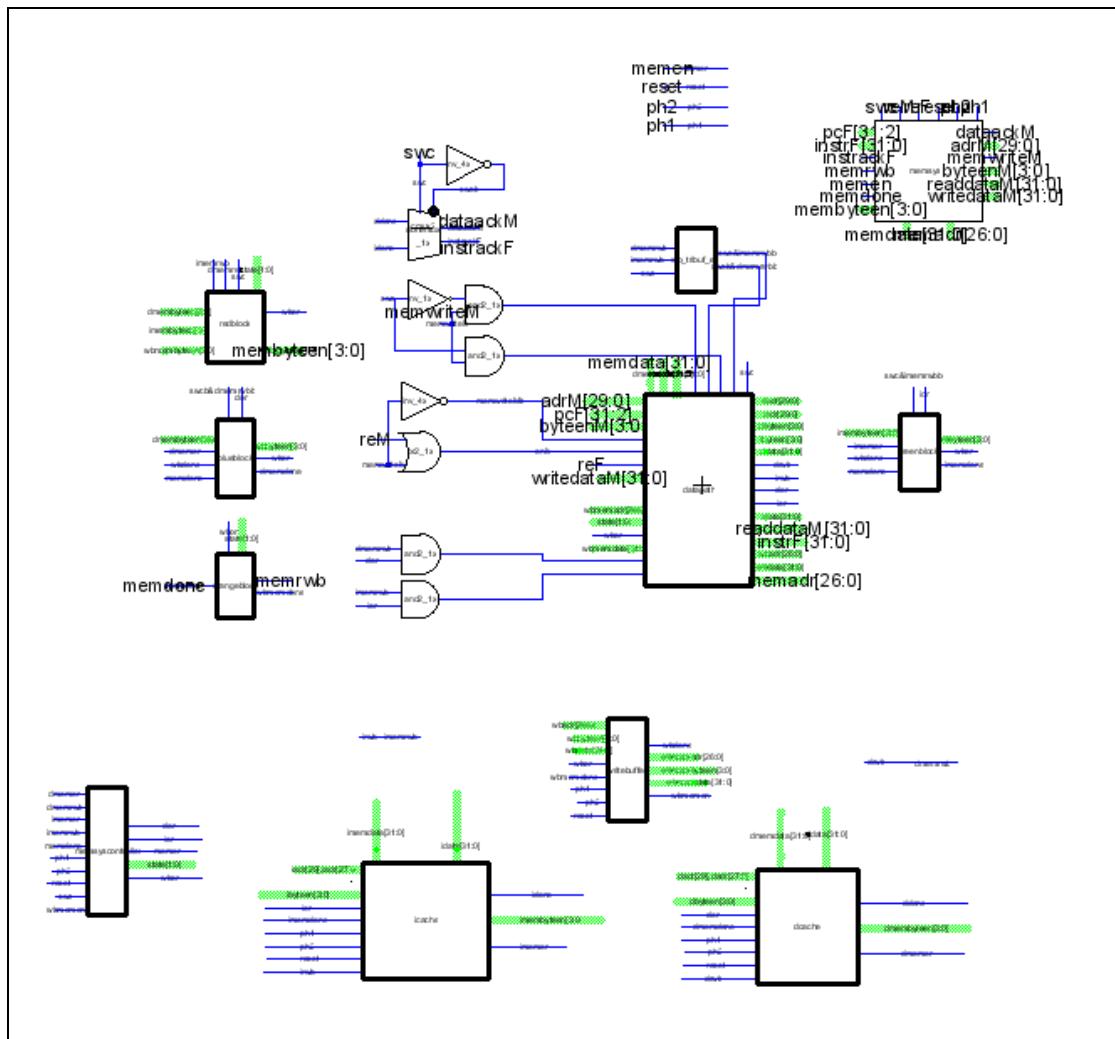


Figure 50. Memsys schematic



Figure 51. Datapath sub-block schematic.

This block groups together the datapath portions of the memsys module and expedites connectivity on the main memsys schematic level.

Layout

The block layout was very straightforward and regular. The most work was in the datapath layout, which was significantly more complicated than the rest of the block. The hardest individual task was wiring the busses that came into and out of the datapath from above or below. These busses had to run vertically into the block, but horizontally inside the block, so every wire had to turn a ninety degree corner, all of which had to be laid down by hand. These corners also took up most of the space in the block, since a 32 bit bus takes $32 \times 8 = 256\lambda$ of space. The overall layout is roughly 4000λ tall and 2500λ wide, including all the vertical busses. The colored blocks are obviously much smaller and were very easy to layout. In the final unit layout they were moved to other, better locations, to mesh with their sub-blocks.

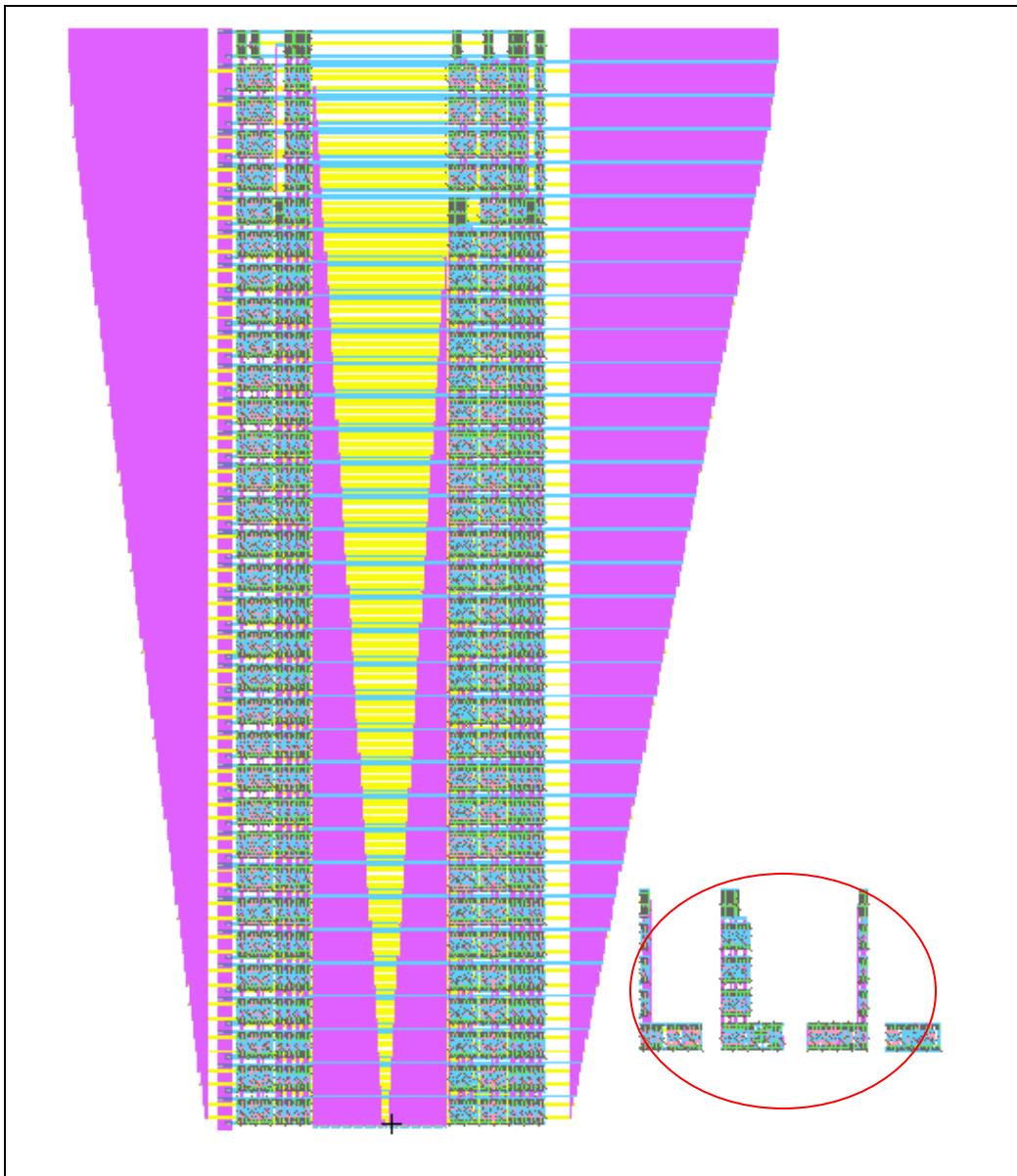


Figure 52. Memsys block layout.
The colored block layouts are encircled in red. The datapath constitutes the majority of the rest of the layout.

Testing

The block was tested extensively in Verilog. A Verilog netlist was written for the entire block from the schematic and was tested in two different and separate ways. The first set of tests consisted of a self-checking test-bench that ran a small set of random test vectors through the block. The second test was to integrate the Verilog netlist with the RTL written by the micro-architecture team and to run their full test suite on the full chip, with the memsys block included. The block passed both test sets with no warnings or errors.

14.6.2. Memsyscontroller Module

Function

The memsys controller is a small set of controls to determine which block reads and writes through to the writebuffer and onto the memory. It takes inputs from the datapath, icache, and dcache. The majority of the functionality is a finite state machine cycling between four states. Each one is used to control the read and write functions of the cache modules.

I/O Table

Input	Origin	Output	Destination
ph1,ph2,reset	Memsys	state[1:0]	Memory Datapath Orangeblock Redblock
wbmemen	Blueblock	don	Blueblock
imemrwb	Datapath	ion	Greenblock
imemen	Icache	wbon	Memsys Datapath Orange Block
dmemrwb	Memsys Datapath	memen	Main Memory
dmemen	Dcache		
memdone	Main Memory		
Swc	Main Datapath		

Table 5. Memsys controller input-output table

Special Units

The nextstatelogic sub-block (Figure 54) was added to help simplify the main cell. Based on the inputs and the current state it computes the next state. This logic was given as a case statement in the Verilog module and was challenging to implement without resorting to large muxes. A large number of logic gates needed to be used to correctly separate each case. The critical path in this logic is about 8 gates long, so the delay is considerable. However, this is an improvement over the first design which used a string of muxes.

Schematic

Figure 8 and Figure 9 are the schematics for the memsyscontroller module and the nextstatelogic block respectively. The decoder block is a simple 2:4 decoder that decodes the nextstate into the appropriate outputs. The nextstate schematic reflects the complicated logic required to determine the appropriate next state. The large number of gates suggests that this may be an area that is critical to the delay within this module.

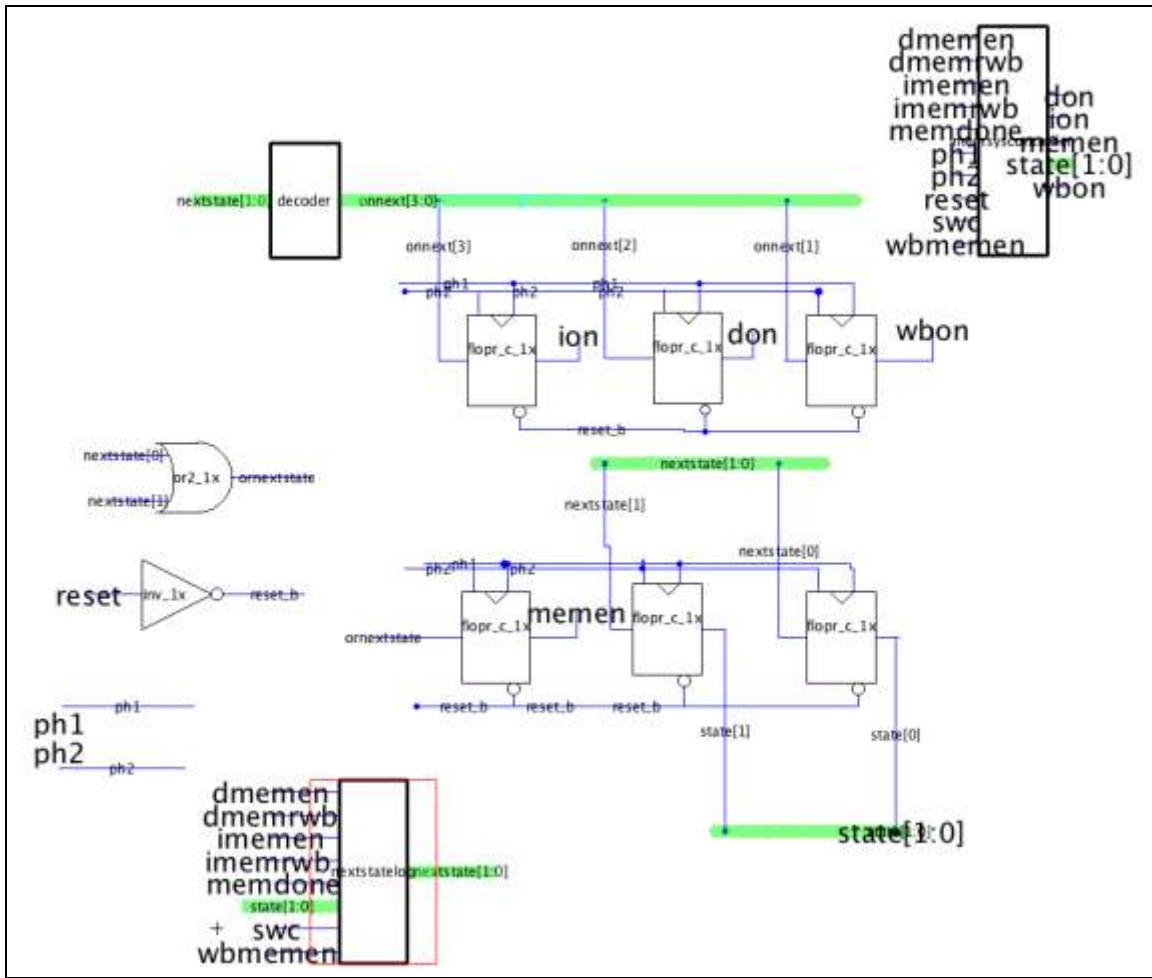


Figure 53. Memsyscontroller schematic and icon

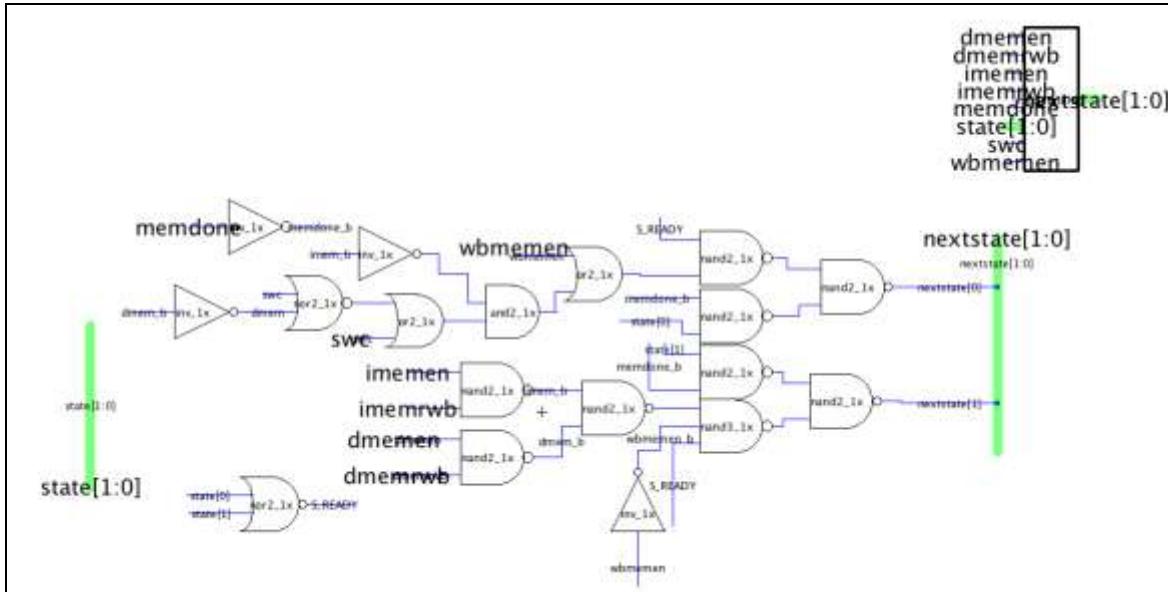


Figure 54. Nextstateologic schematic and icon

Layout

The large number of inputs and outputs for such a small block created a few issues in wiring the part together as many wires needed to be routed from one area to another. Without using much metal-3 except when allowable and needed, most wires were connected with vertical metal-2 wires and horizontal metal-1 wires. The wiring also posed issues because many of these inputs came from different parts of the chip and it required a lot of space to make these connections.

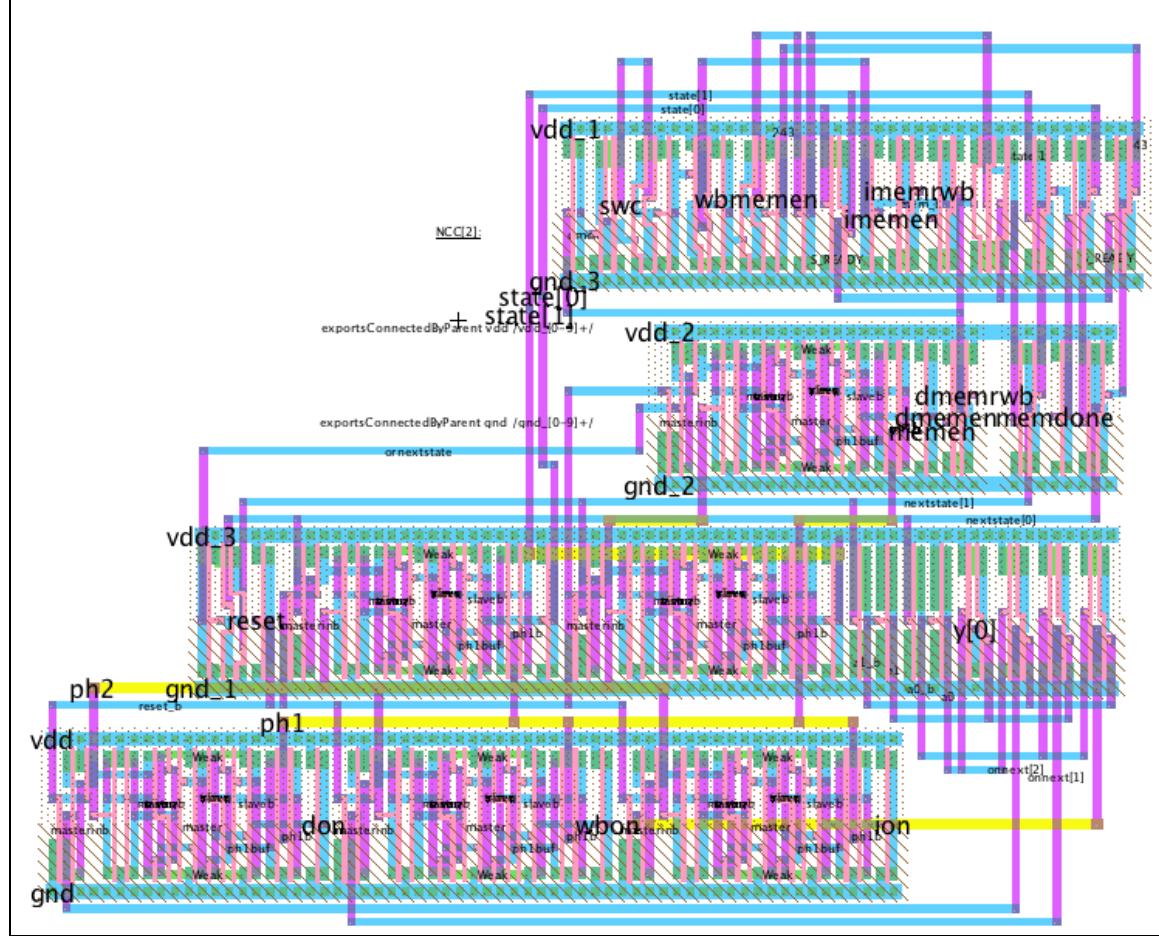


Figure 55. Memsyscontroller layout

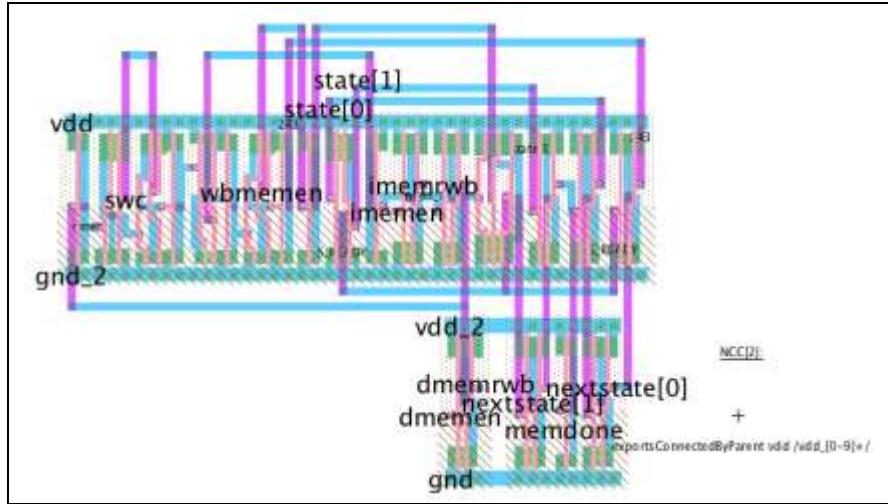


Figure 56. Nextstatelogic layout

Testing

All testing was performed in IRSIM and in Verilog using Modelsim. Tests were designed to check all possible states and input combinations to ensure proper functioning of the block. Since timing delays are not an issue as long as the clock is long enough to allow sufficient setup times for nextstate logic, the functioning of the chip should be accurately predicted by the simulation. Final verification was ensured by running this part with the Verilog HDL in the full chip simulation tests designed by the micro-architecture team.

14.6.3. Cacheram Module

Function

The function of the cacheram is to either read or write data from the cache array given an input address. Regardless of whether the instruction received is a read or write, the 53 bit value stored in the input address location is always output. The state of the cache (reading or writing) is determined from the cache controller. Cacheram will write to the cache if the value of control signal rwb is 0. It will read at all other times.

Cacheram is designed to be a single-ported direct-mapped cache and as such each address will at any time correspond to only one 53-bit signal. The 53-bit output signal (dout) is composed of one valid bit (1 representing that there is valid data at that particular memory location), 20 tag bits, and 32 address bits. The output of cacheram is used in the cache module and cachecontroller module. The 20-bit tag and valid bit are used in the cachecontroller, but the 32 bit address is muxed with a signal from external memory. Depending on the state of the cache, either the data read from the cache or the data from the external memory will be output to the memsys module.

I/O Table

Inputs	Origin	Outputs	Destination.
ph1	cache	dout[52:0]	cache
ph2	cache		
adr[6:0]	cache		
rwb	cache		
din[52:0]	cache		

Table 6. Cacheram input-output table

Special Units

In order to achieve the same functionality of the idealized Verilog instance of cacheram, the module required the addition of a decoder, bit conditioning, write driver, array of sram cells, and signal buffers.

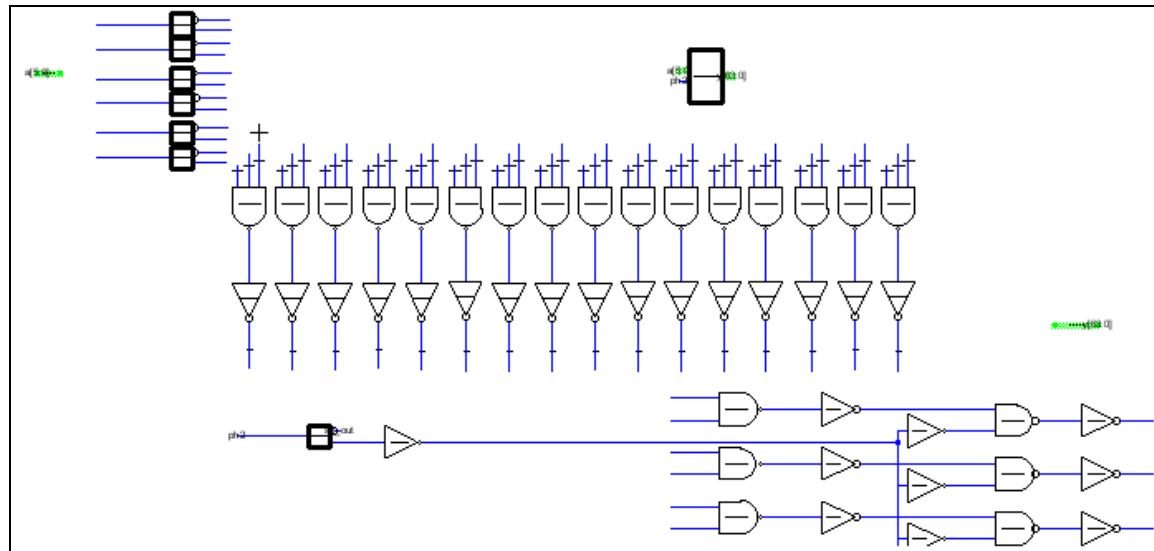


Figure 57. Partial schematic of the decoder showing only three wordlines.

The Lower right portion of the schematic repeats for 64 wordlines. Vertical gates represent the predecoding phase. A special layout for the NAND gates had to be employed to pitch match to the small SRAM cell.

The decoder ensures that read or write occurs to a single memory location in the cache by turning on at most one wordline when clock ph2 is high. It is composed of multiple 3 and 2 input NANDs and inverters that will determine which wordline (0-63) to turn on based on a 5 bit signal (00000 represents wordline 0 and 11111 represents wordline 63). The result of these NANDs is ANDed with ph2 to ensure that a wordline only goes high when ph2 is high. The implementation of sixty-three six-input AND gates was avoided through the use of predecoding.

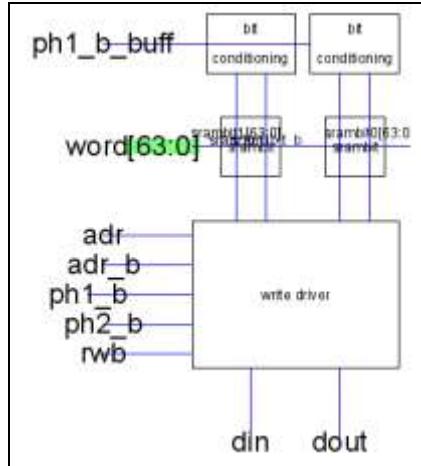


Figure 58. Cacheram array schematic.

Consisting of two columns of 64 SRAM cells, bit conditioning and a write driver.

Wordlines are only turned high when ph2 is high because ph1 is reserved for bit conditioning. The bit conditioning unit pulls the bit lines (bit and bit_b) of columns of SRAM cells high when ph1 is high. A write is then accomplished by pulling down the appropriate bit line, bit or bit_b. The bit conditioning cell is simply composed of two strong pmos pull up transistors with ph1_b as the gate signal and two weak keeper pmos pull transistors governed by ph1. This arrangement allows for the precharged value to be held high enough to keep its value without being strong enough to overcome the pull down transistor if the line needs to be changed. The source and drain are attached to vdd and either bit or bit_b.

The SRAM array is composed of 106 columns of 64 SRAM cells requiring one bit conditioning cell per column as the bit lines are tied together vertically. Not all SRAM cells on a horizontal line represent a single address location even though they share the same wordline. As the cacheram represents a 128 word cache but is only 64 SRAM cells tall, each horizontal line of SRAM cells represents two different addresses. Specifically, looking at the first row of SRAM cells, the first cell (row 0) is location zero as is every other cell on the line. The remaining cells represent address location 63. In a similar fashion every horizontal line represents address location x and x+63. As a row is tied together with the same wordline, a write driver (Figure 59) is required to make sure a write only occurs to every other SRAM cell along a horizontal line.

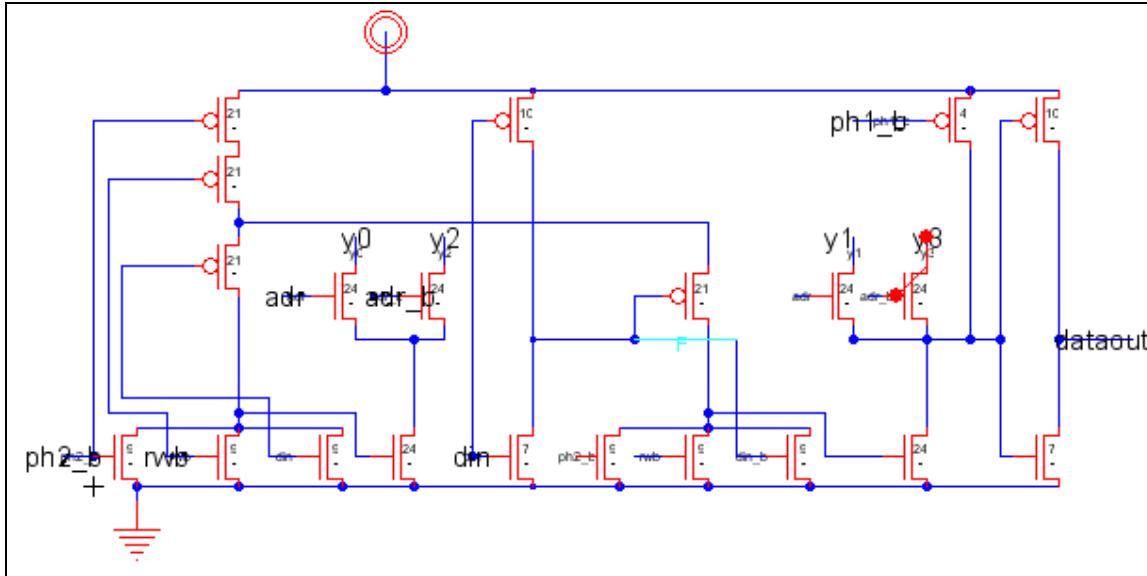


Figure 59. Schematic of write driver.

Special attention had to be paid to transistor sizes since a single write driver drives two columns of SRAM cells. It is important to note that when ph1 is high, there will always be a path from dataout to vdd making the output of dataout valid only on ph2. This is a key difference between the Verilog and the actual implementation.

The write driver consists of two three-input NAND gates to ensure that writing and pulling down occur only on ph2. The third input is the value that is actually being written, and its complement appears on the other gate. There exists one write driver for every two columns of SRAM cells and as such an nmos transistor tied to the most significant bit of address (adr[6]) is used in series with the output of the write driver to determine if we will be writing to location x or x+63.

Finally the SRAM ram array itself is composed of SRAM cells, each of which consists of a wordline, two bit lines, and two cross-coupled inverters. Data is stored between the cross coupled inverters and can only be read or written if wordline is high. To write, the bit lines must be complements of each other, bit and bit_b, since they are tied to inverters and will cause contention otherwise. No problems occur under bit conditioning (when both bit and bit_b are high) because transistor sizes were carefully chosen to ensure a bitline carrying a one is not strong enough to write a value. A value can only be written by making either bit or bit_b zero, which is the function of the write driver.

Schematic

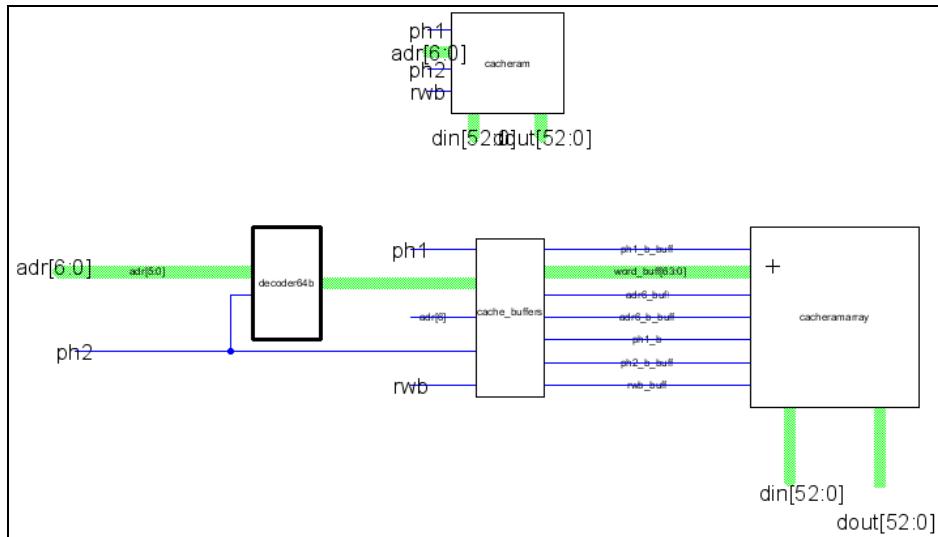


Figure 60. Schematic of cacheram made up of the decoder, buffers, and cacheram array. The buffers buffer each signal going into the cacheram array since they have to drive a large load in the array. The array is composed of the SRAM cells, bit conditioning, and the write drivers.

Layout

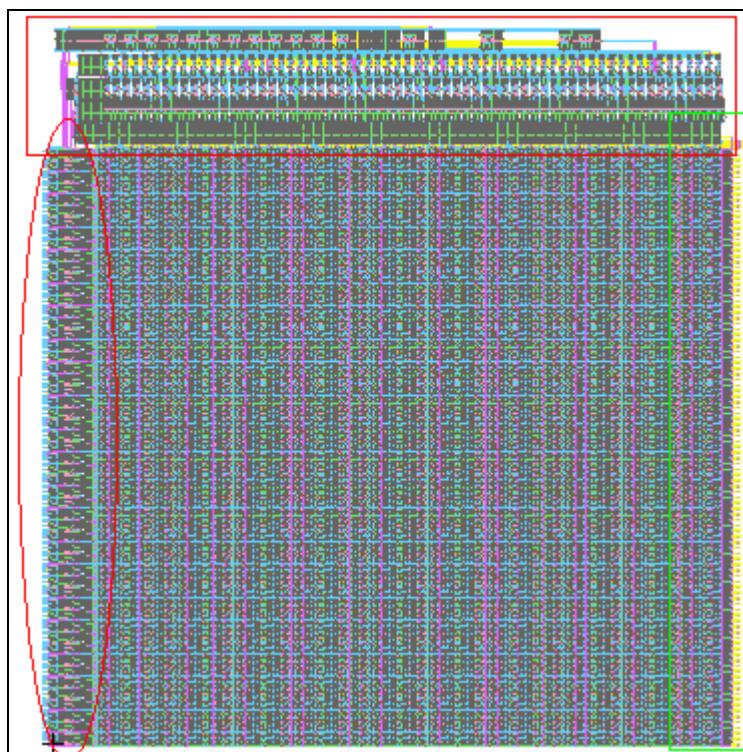


Figure 61. Layout of the cacheram module.

The decoder and buffers are outlined with a red rectangle, the write drivers with a red oval, and bit conditioning cells with a green rectangle. The majority of the layout is comprised of the 6784 SRAM cells. Final dimensions were $3218 \times 3326\lambda$

The above module required careful layout as an SRAM cell is only $46 \times 26\lambda$ and must be pitch matched to the write driver, decoder, and bit conditioning. Since bit conditioning is only four transistors this was not a problem; however, the write driver consisted of two three-input NAND gates and inverters which were too large in muddlib07 to be pitch matched. The decoder also had to be pitch matched, which required the laying out of a new NAND gate that was at most only 46λ tall.

Testing

Cacheram was tested by using a test vector that wrote and read from every address in the cache. This test was successful. Cacheram however would not initially pass the chip tests when its netlist was substituted for the Verilog code. This was believed to be related to the fact that the Verilog models an ideal memory and the schematic is not ideal. The schematic also differs from the Verilog in that the schematic contains reads and writes that occur on ph2 whereas in Verilog they occur on ph1. This change was made to ensure the correct value was passed through the flip flops which operate off a multi-phase clock. However after changing the Verilog to accept reads on ph2 instead of ph1, we still saw failure in tests 8, 12, 16 and 19-26. It was eventually discovered that the test clock the debug.fdo file ran off of was too fast for the module alone to handle. However, once the clock was slowed down by a factor of 10 cacheram failed every test. It was then found that in addition to simply changing the declaration of the clock by a factor of ten in mipstest.v, it was also necessary to change clock delays in imem.v and mipstest.v by a factor of ten as well. Once these changes were made cacheram passed all chip tests.

14.6.4. Cachecontroller Module

Function

The function of the cachecontroller is to determine if a read or write is occurring and to produce the control signals that will read or write using the cache or the external memory. It is called from the cache module and the values of its four output control signals determine the output of cache.

I/O Table

Input	Origin	Output	Dest.
adr[29],adr[27:7]	cache	bypass	cache
en	cache	done	cache
memdone	cache	reading	cache
ph1	cache	waiting	cache
ph2	cache		
reset	cache		
rwb	cache		
tagdata[19:0]	cacheram		
valid	cacheram		

Table 7. Cachecontroller input-output table

Special Units

The cache controller needs to check if the 20 bit tag data signal is equal to bits 26 through 7 of address. To accomplish this it was necessary to construct a 20 bit comparator that would check to see if each bit of these signals was the same.

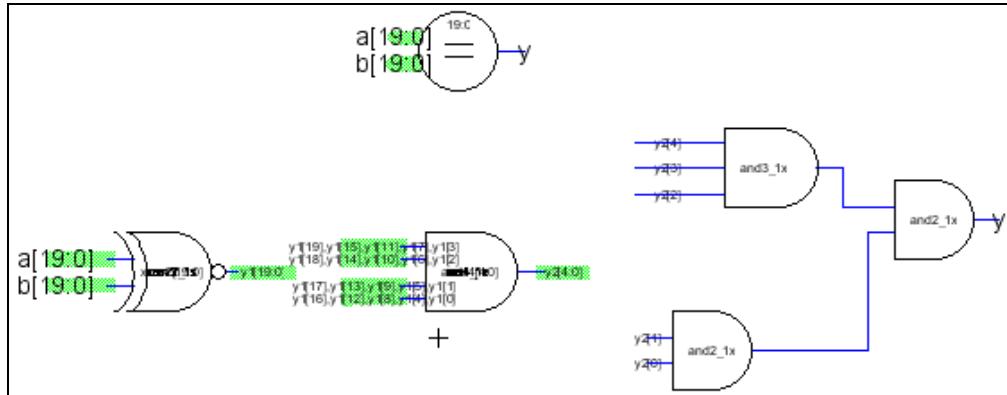


Figure 62. Schematic and icon for 20 bit comparator.

Consisting of 20 XNOR gates and a ones decoder returning a one if a bit-wise equals b , and 0 otherwise.

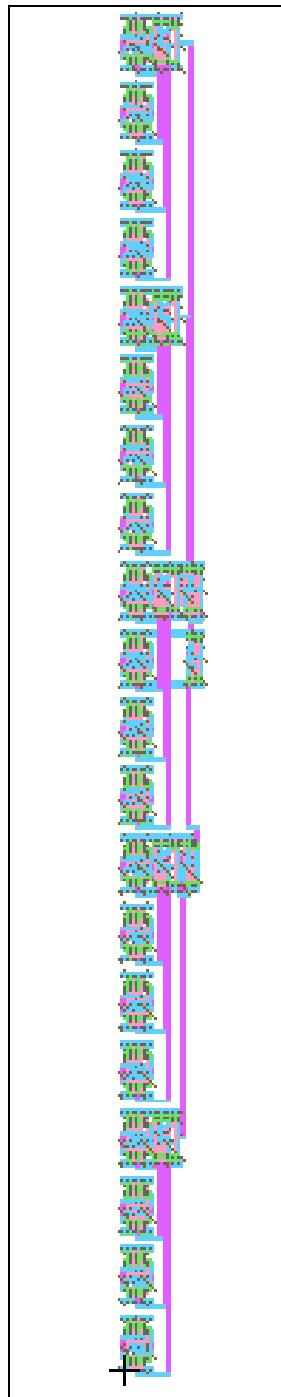


Figure 63. Layout of the comparator.

The output of the 20 XNOR Gates is fed into a ones decoder to see if any XNOR did not output a 1. The ones decoder is implemented by cascading NANDS from 20 signals into one and passing the result through an inverter.

Schematic

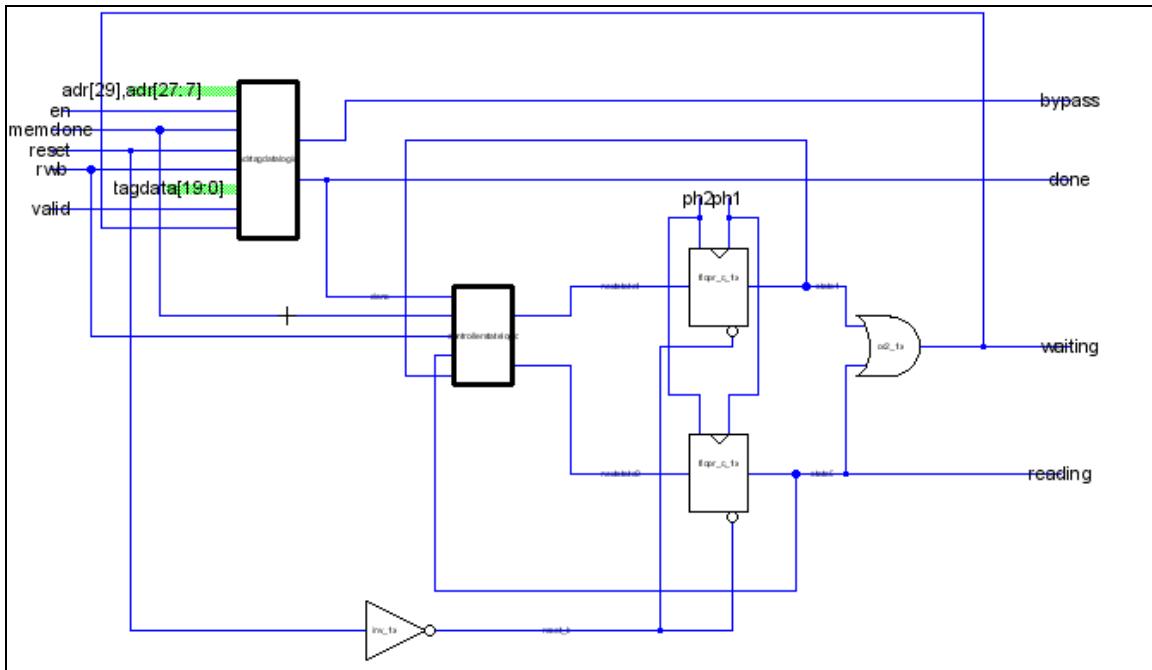


Figure 64. Schematic of cachecontroller broken into modules.
adrtagdatalogic and *controller state logic* for readability.

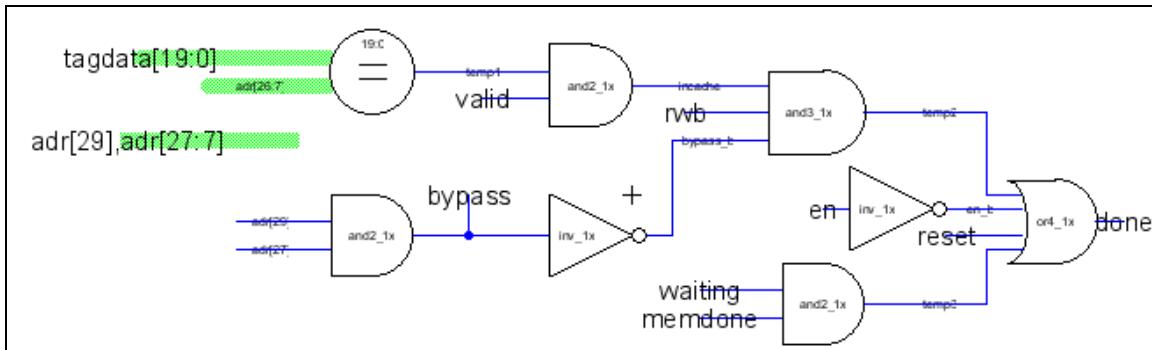


Figure 65. Schematic of adrtagdatalogic utilizing the special 20 bit comparator cell.

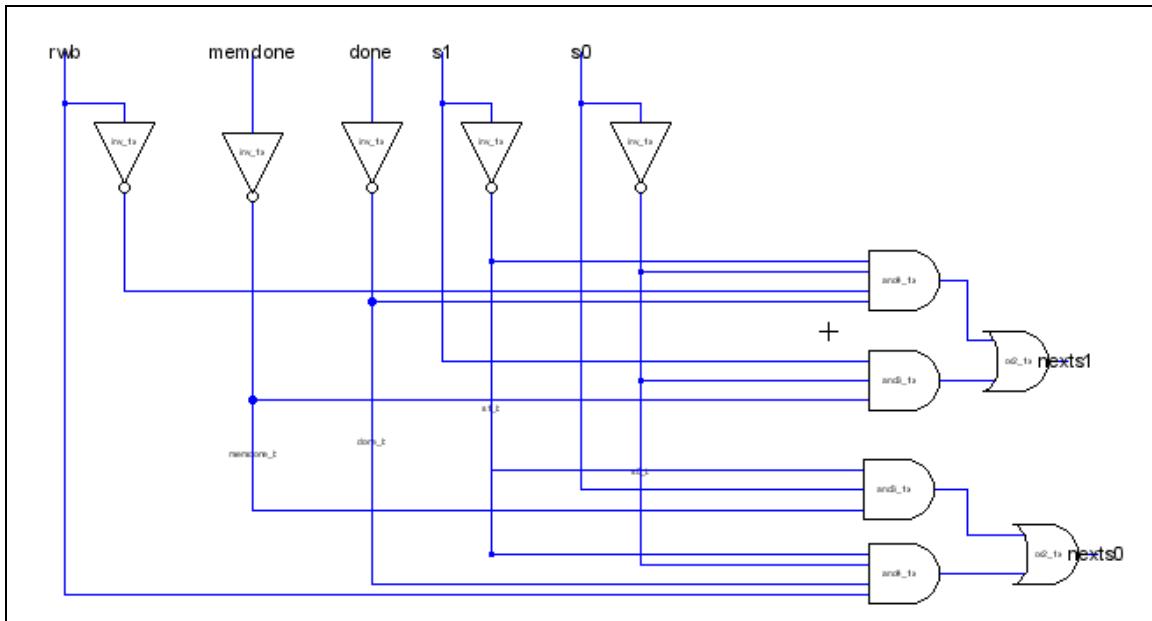


Figure 66. Schematic of controllerstate logic.
It determines the next state to be input to the flip flops from the
previous state and output of adrtagdatalogic

No problems were encountered while making this unit. It is simply a finite state machine that toggles between the three states steady, sread, and swrite, which are encoded as 00, 01, and 10 respectively.

Layout

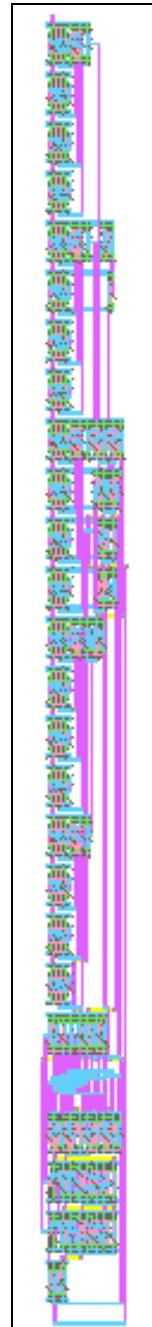


Figure 67. Block layout for cachecontroller having dimensions $206 \times 3164 \lambda$.
The large height was due to the comparator that required the use of 20 stacked XNOR gates

Testing

The block was tested by replacing the Verilog code for the module with the generated netlist from Electric. All chip tests passed without difficulty as well as NCC, DRC, and ERC for the layout.

14.6.5. Write Buffer Module

Function

The cache system of the R2000 chip has a write-through specification. This means that whenever a miss occurs in the cache, new data is written both to the cache as well as main memory. Since the write to main memory takes longer than a write to cache, a write buffer is implemented to manage the data being written to main memory. The write buffer temporarily stores the data until the memory controller is ready to execute the write. It is a four-entry deep array that functions as a first-in-first-out buffer. It communicates with the memsyscontroller module of the memsys unit in order to correctly queue data and store data to main memory. It is possible to ensure that the writebuffer is empty by performing an uncached load from anywhere. A write to memory followed by an uncached read from the same address flushes out the write buffer queue.

I/O Table

The following table lists all the inputs and outputs from the write buffer, including their origin and destination.

Input	Origin	Output	Destination
ph1	Memsys	wbmemadr [26:0]	Memsys
ph2	Memsys	wbmemdata [31:0]	Memsys
reset	Memsys	wbmembbyteen [3:0]	Memsys
wbadr [26:0]	Memsys	wbmemen	Memsys
wbdata [31:0]	Memsys	wbmemdone	Memsys
wbbyteen [3:0]	Memsys		Memsys
wben	Memsys		Memsys
wbdone	Memsys		Memsys

Table 8. List of Inputs and Outputs of the Write Buffer

Special Units

The following units were created using standard library cells from muddlib07.

2:4 Decoder

A 2:4 decoder was created in order to select a location in the write buffer array to which data can be written or from which data can be read. The schematic and layout of the 2:4 decoder appear below.

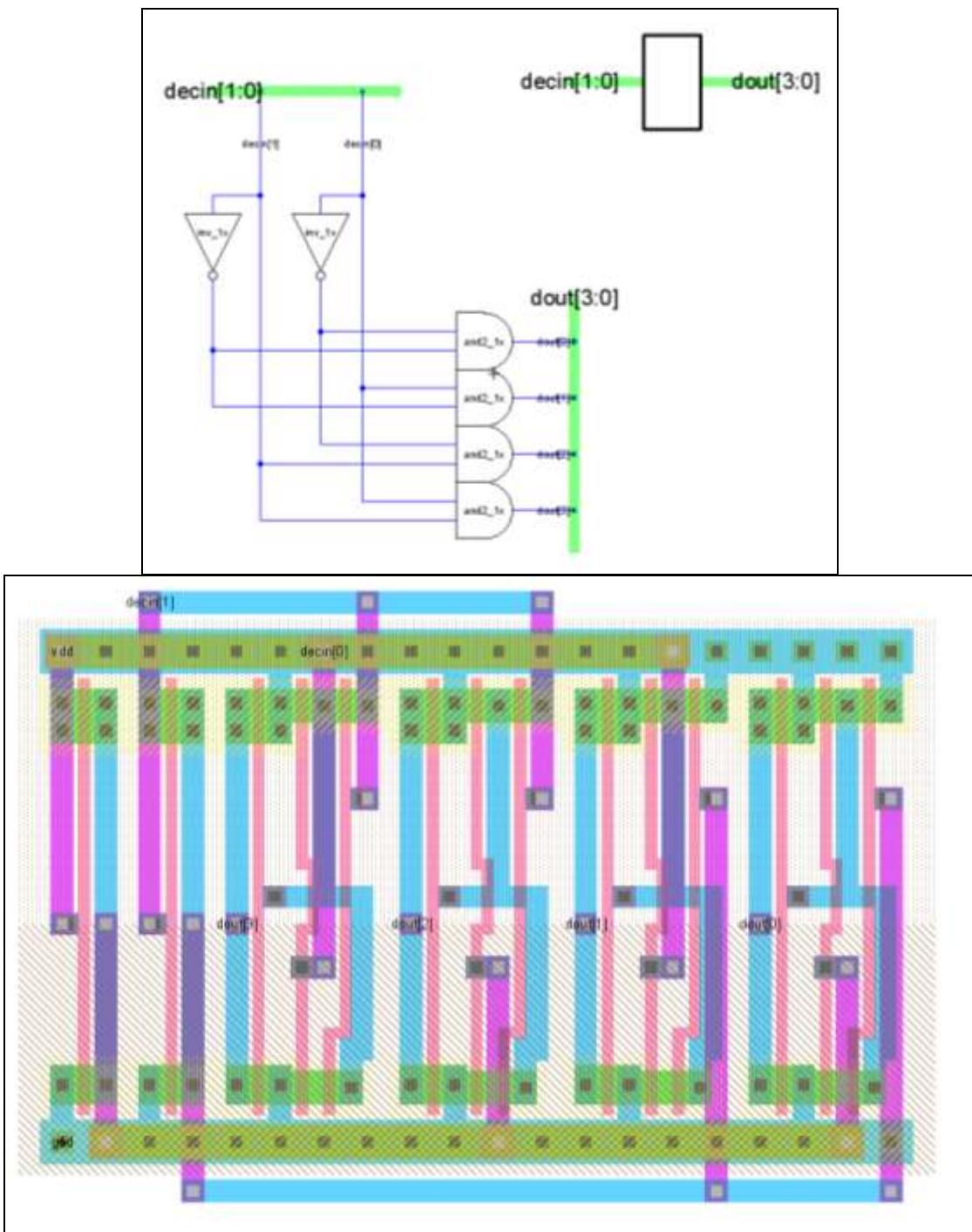


Figure 68. 2:4 Decoder Schematic and Layout

One-Hot Four-Mux

The write buffer requires a four-input multiplexer whose select signal consist of four bits of which one is high at any given time. In order to accommodate this type of select signal, combinational logic was used to convert the four bits into the expected two-

bit select signal. Converting the four-bit select to a two-bit select allowed the use of a library mux4 cell. The schematic and layout for the four-mux appear below.

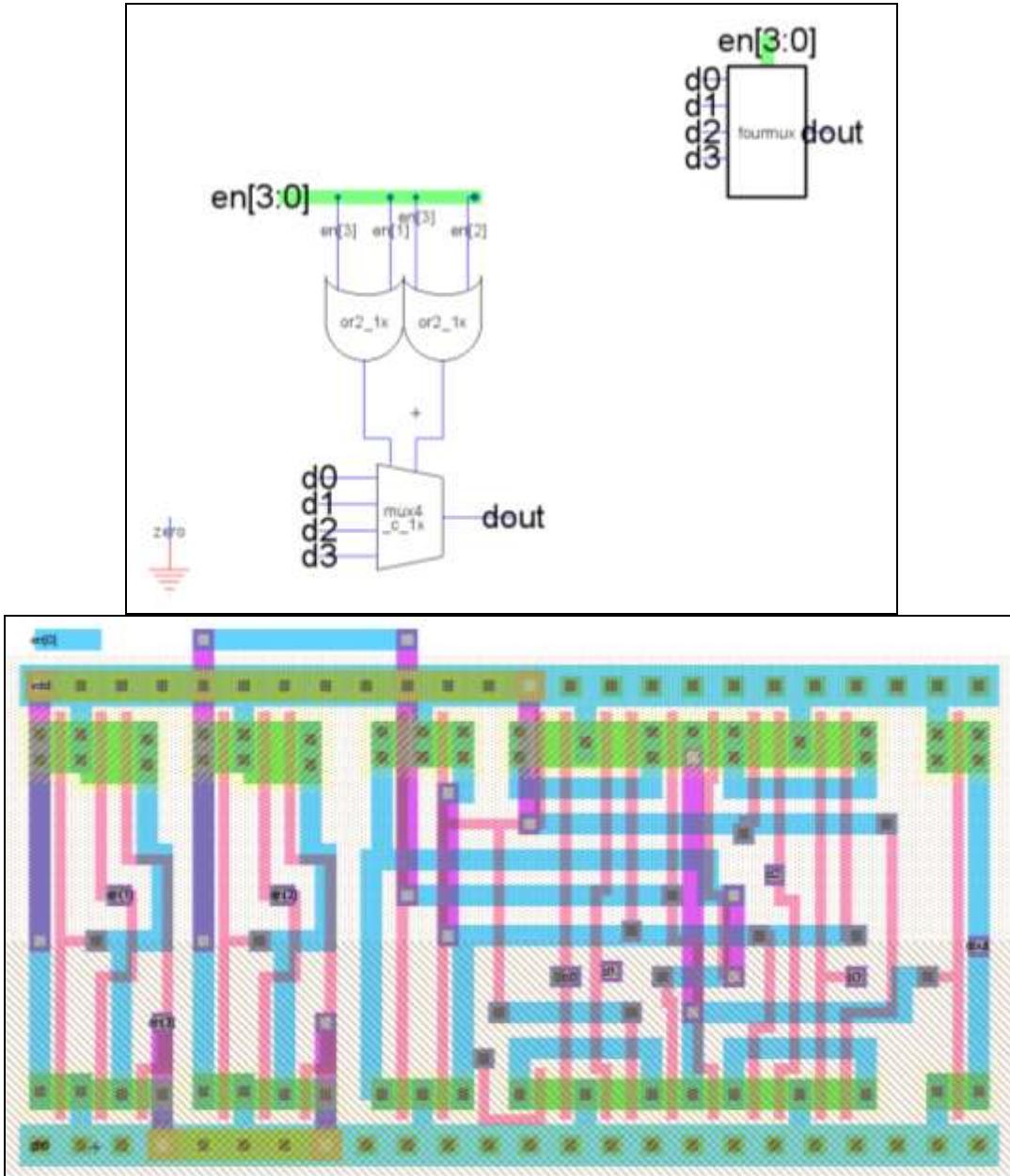


Figure 69. Four-mux Schematic and Layout

Note: the dangling export $\text{en}[0]$ was needed to pass NCC since all four enable bits are input into the fourmux. However, only three are needed to determine the two-bit select signal. All four signals were input into the mux to facilitate ease of connection between logic components on the schematic level.

Incrementer

In order to correctly read and write to the array, the write buffer contains a block of flops and incrementers that cycle through the array positions sequentially. The *ptr* and *writeptr* signals that are used to determine array location are generated from this block. The two-bit incrementers needed to implement these signals were designed using two full-adders as seen below.

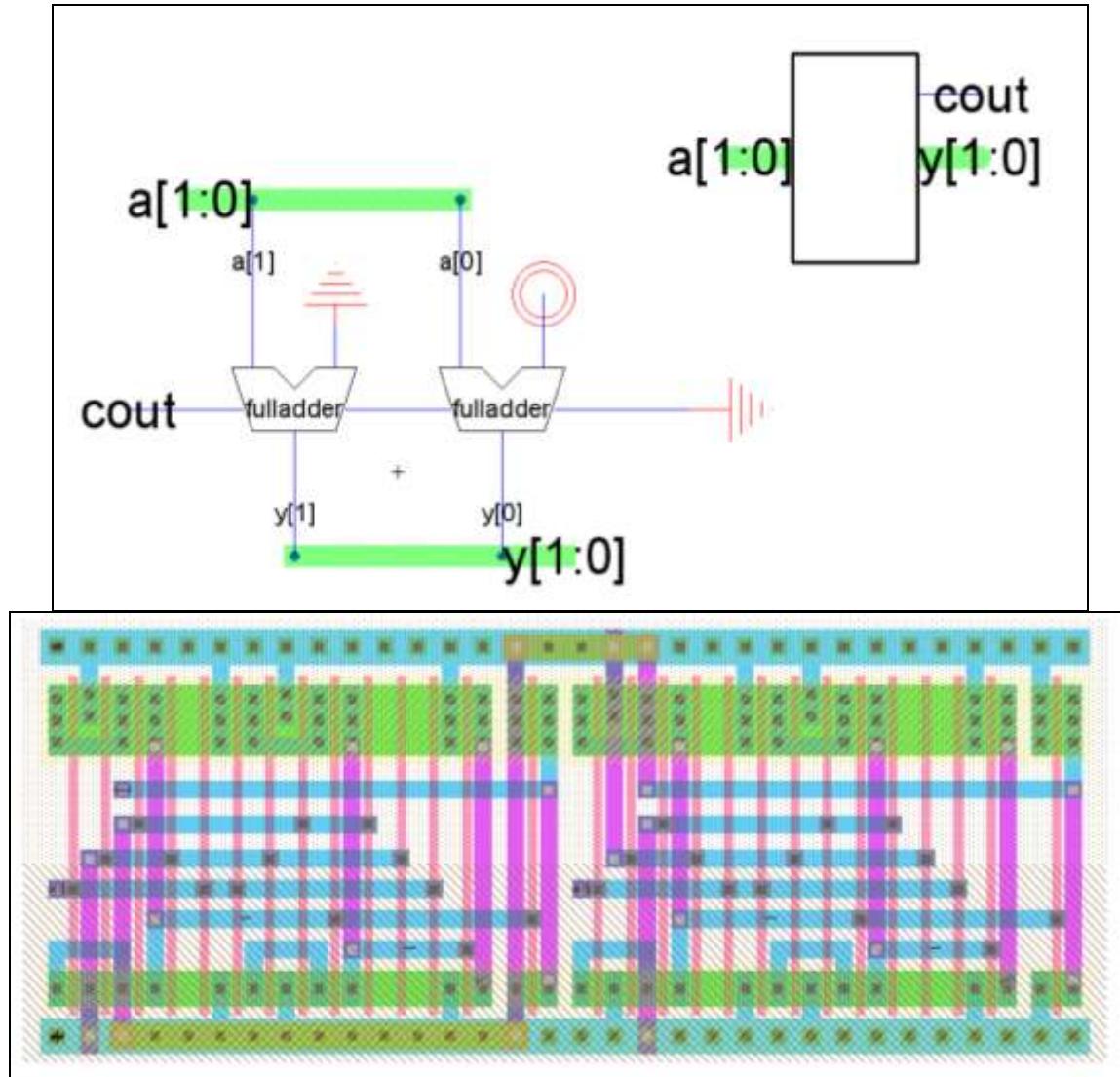


Figure 70. Incrementer Schematic and Layout

12T SRAM

Since the write-buffer needed to be implemented such that horizontal usage of chip real estate was minimized, it was necessary to redesign the 12T SRAM bit shown in *CMOS VLSI Design* so that it is vertically oriented. In addition, the write-buffer required dual-ported SRAM with a read as well as write port. The following transistor schematic shows the implementation of this dual-ported SRAM bit. The layout that follows shows the re-design of the bit with the new read port.

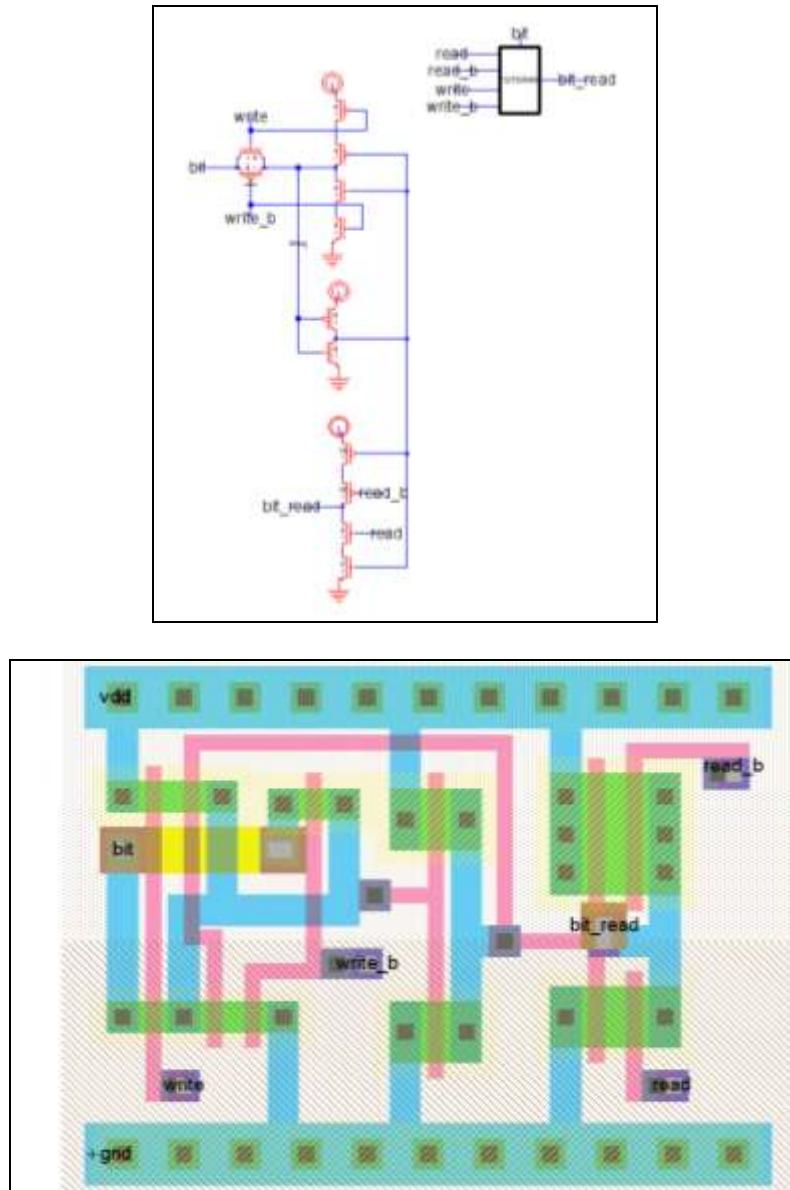


Figure 71. 12T SRAM Schematic and Layout

The SRAM layout has a center-to-center height of 60λ . The bits in the array were flipped so that vertically adjacent bits would share power and ground lines. This height and arrangement was optimal given the space constraints of the memory subsystem. The layout width of 92λ also met design constraints since only four adjacent horizontal bits were needed to implement the array.

Schematic

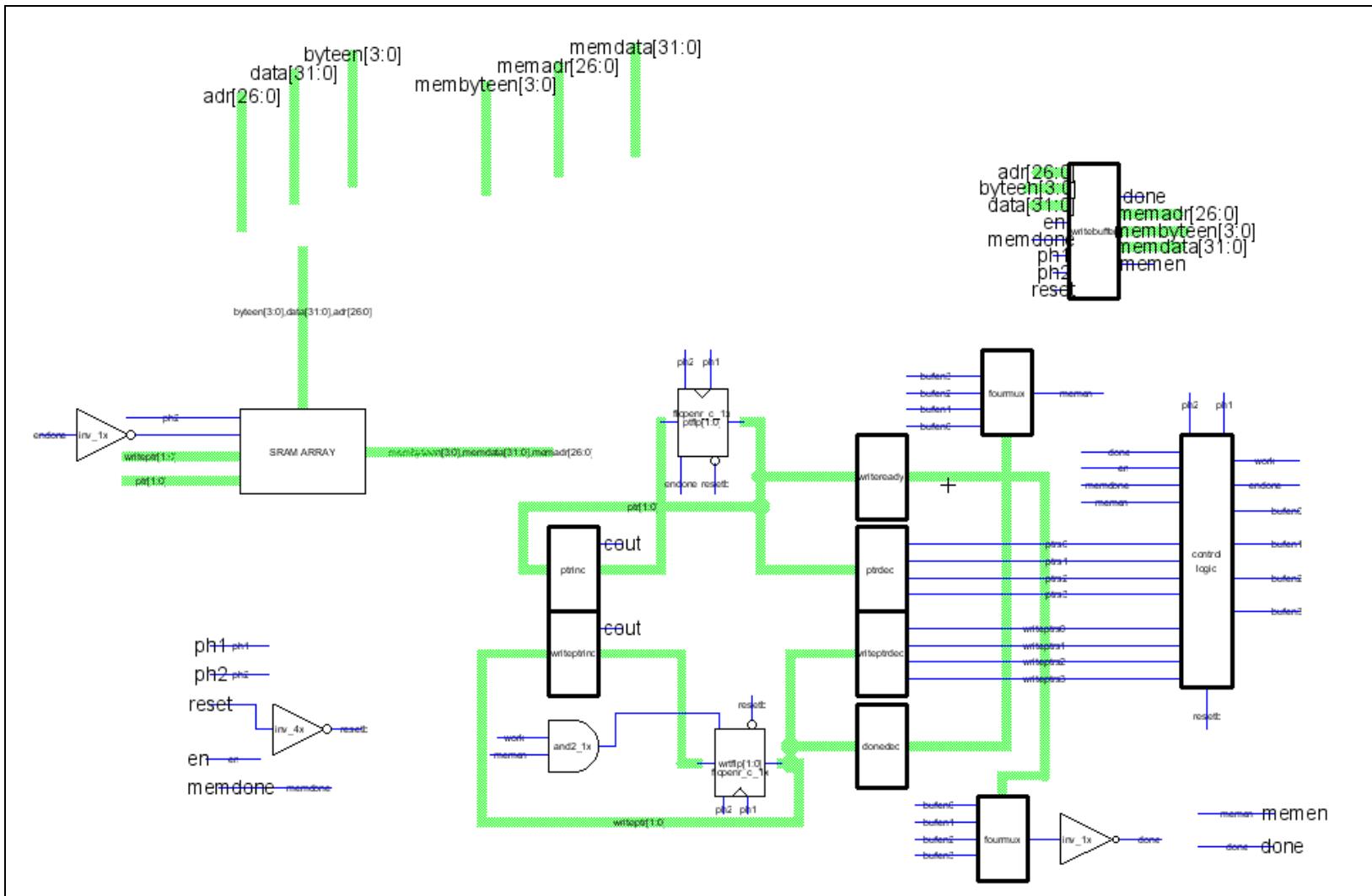


Figure 72. Overall Write Buffer Schematic

Block Schematic Hierarchy

In order to facilitate an easy layout process and pass NCC, the write buffer block schematic consists of several levels of hierarchy. Grouping logic blocks together in this manner allows the layout process to be done in smaller portions and be checked for NCC errors in a systematic fashion.

SRAM Array

The SRAM array consists of the 63×4 12T SRAM bits and the decoders used to select the wordlines based on the signals *ptr* and *writeptr*. While a read from the array could occur at any time, a write could happen only when *clk* and *rwb* are high.

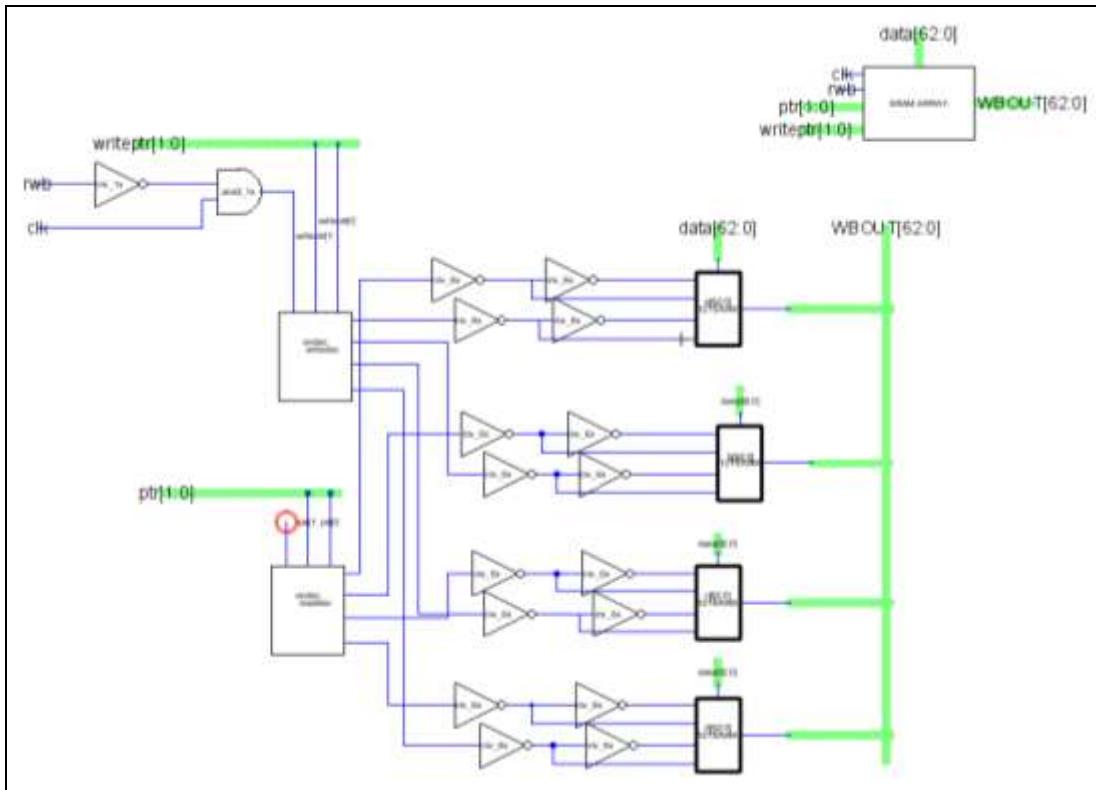


Figure 73. SRAM Array Schematic

The following figure displays the layout of the SRAM Array.

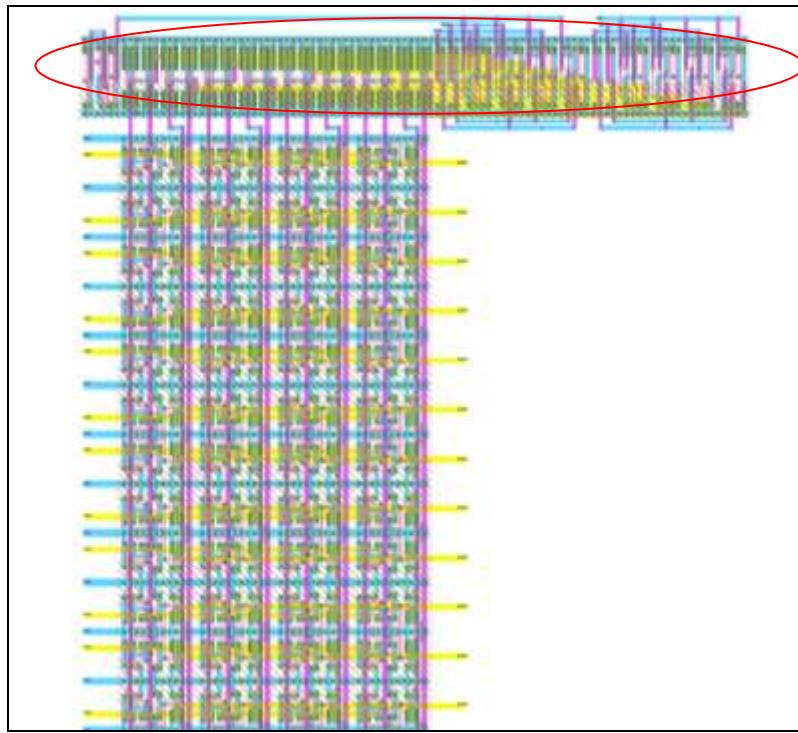


Figure 74. Layout of the SRAM Array.

Only part of the array is shown for conciseness. The encircled region contains the control combinational logic that drives the SRAM Array

Control Logic

The control logic block groups together the random logic that generates the control signals that govern the functionality of the write buffer.

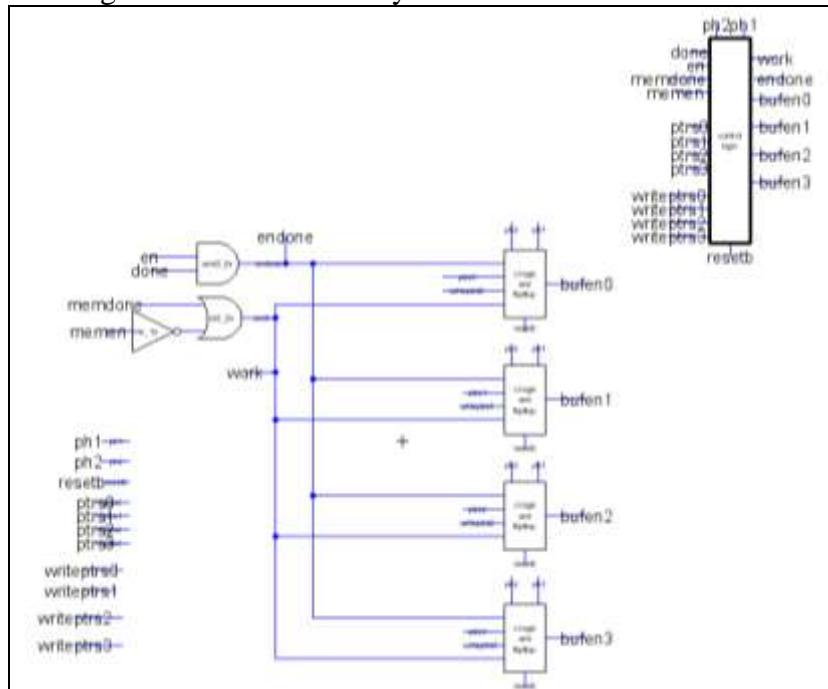


Figure 75. Control Logic Schematic

The following figure shows the layout of the control logic block

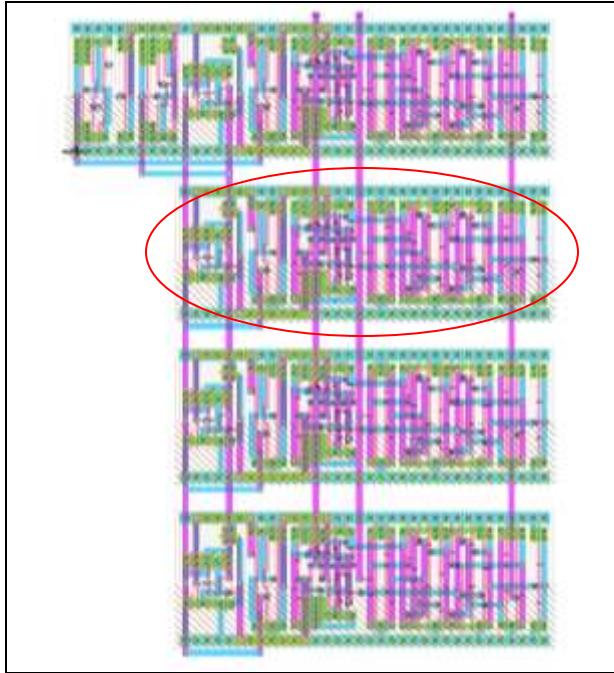


Figure 76. Control logic block layout.

The majority of this layout is occupied by the four instances of the ‘combinational logic and flipflops’ block described below (one instance is encircled in red)

Combinational Logic and FlipFlops

A further level of hierarchy that groups together repeating blocks of logic within the control logic block.

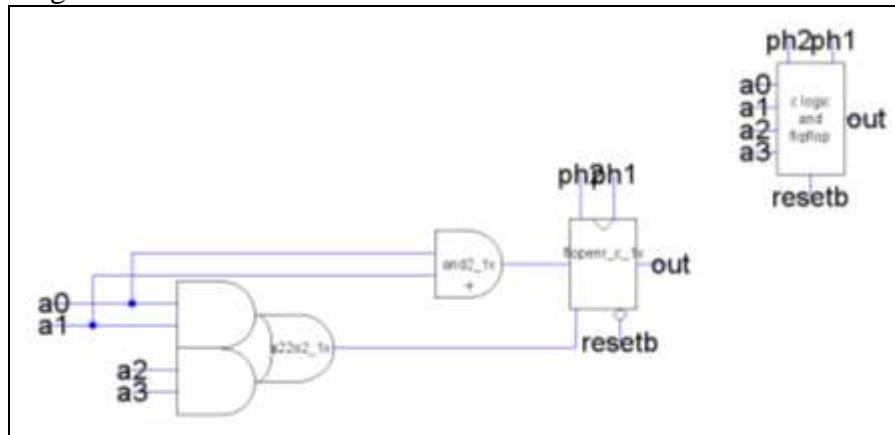


Figure 77. Combinational logic and Flop Schematic

The following figure illustrates the layout of the above schematic.

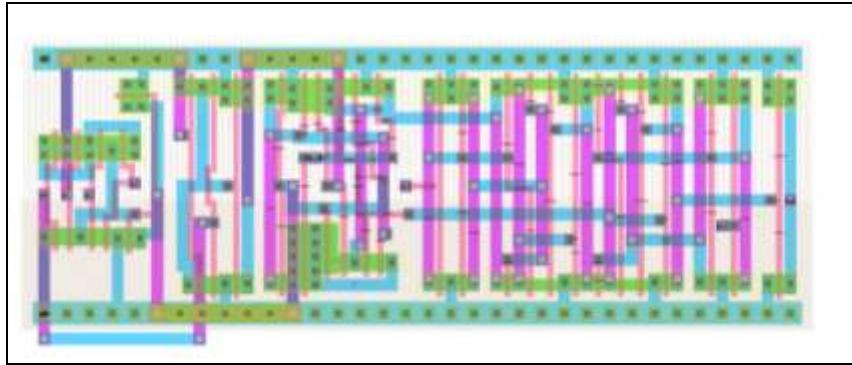


Figure 78. Combinational logic and Flipflops Layout

Layout

During the preliminary design stages of the chip, the write buffer was slated to be implemented using registers rather than SRAM bits. The initial floor plan devised for the write buffer is presented in Figure 34. The minimum dimensions of $1400\lambda \times 10000\lambda$ given this arrangement and the size of the flipflops far exceeded the design constraints of the memory system. Therefore, the write buffer was redesigned using 12T SRAM bits and oriented vertically to maximize horizontal space for the caches. The floor plan for this design appears in Figure 80. Finally, since the control logic blocks along the side of the main array did not occupy the entire length of the module, all of these random logic blocks were gathered and placed to the right near the top of the main array. The finalized layout using this arrangement appears in Figure 81. This layout utilizes the layouts of blocks described in the previous section. Thus partitioning the design into hierarchies allowed for relative ease of wiring within the main layout. However, the creation of these blocks also dictated that some space inefficiencies exist in the write buffer layout. Some of the white space in the control logic section could have been reduced if the groupings of logic were not used. However, this would have entailed greater time in connecting single gates or small logic blocks within the main layout. Therefore, the decision was made to use the blocks as they were and sacrifice some space, which was sufficiently available at that location in the memory subsystem. This arrangement also allowed the cachecontroller and cache to fit neatly below the control logic blocks and take full advantage of the limited space allocated to the memory system.

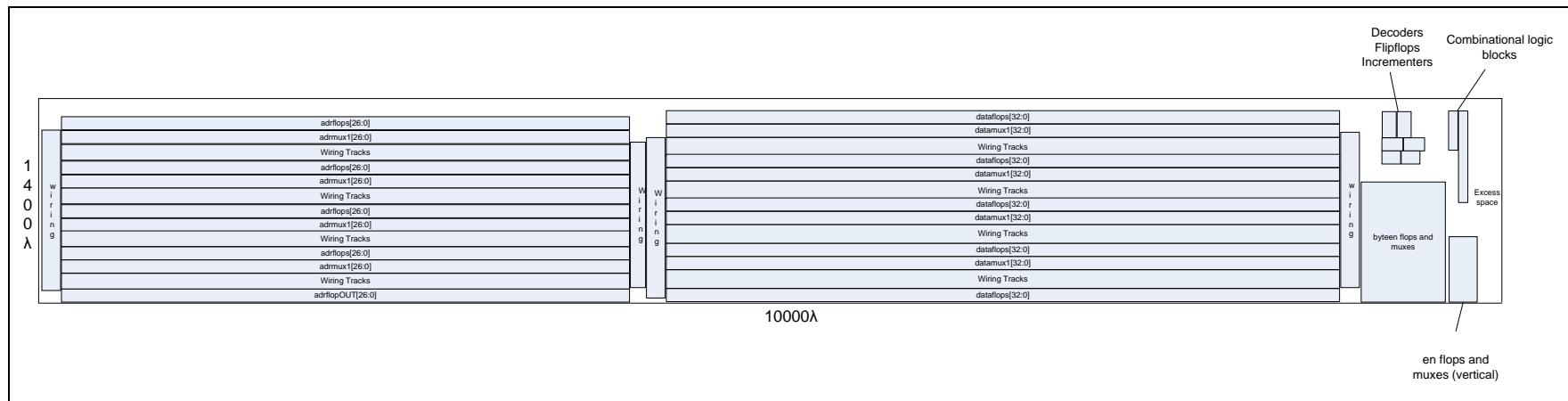


Figure 79. Preliminary WriteBuffer Floor Plan.

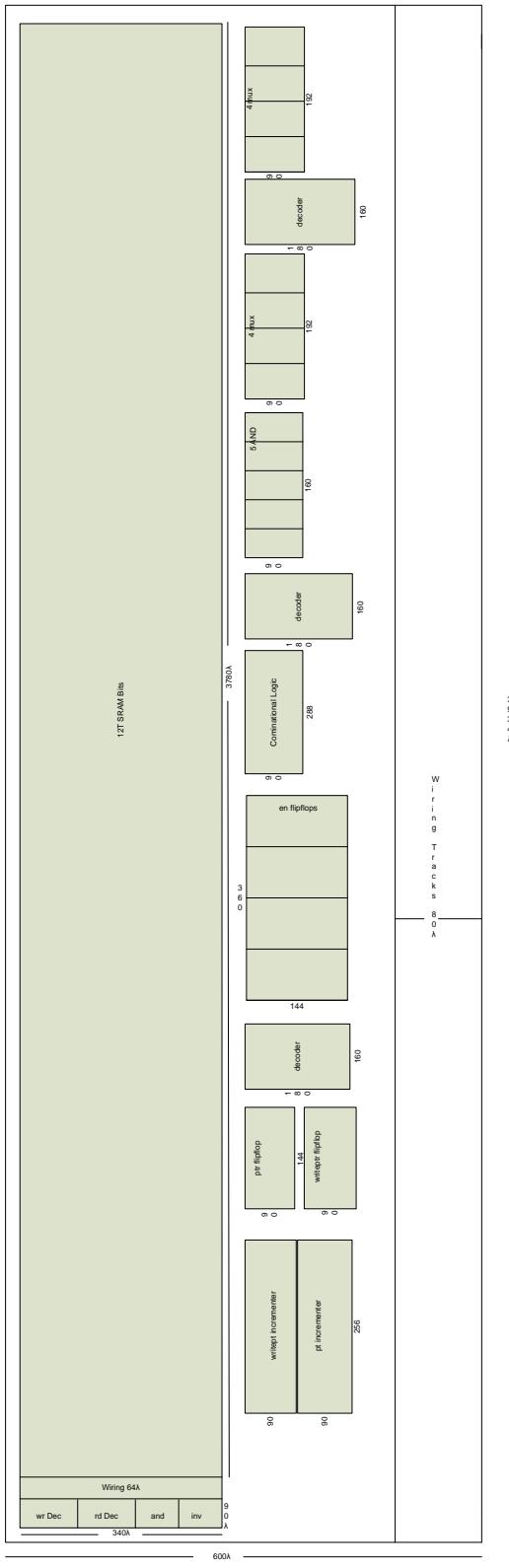


Figure 80. Revised Write Buffer Floorplan

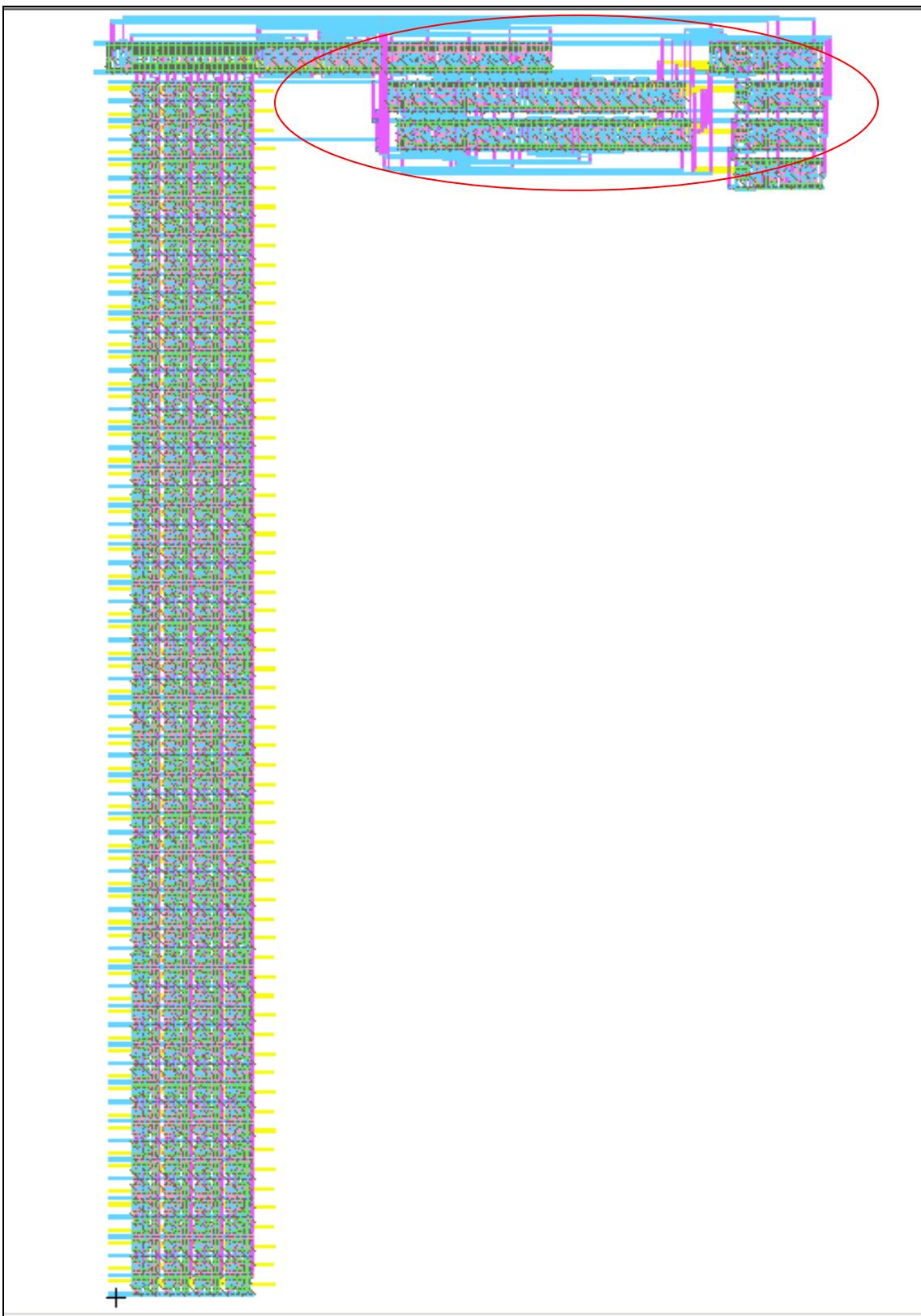


Figure 81. Finalized WriteBuffer Layout. The control logic is encircled in red.

Testing

The complete block schematic was tested using the chip tests designed by the micro-architecture team. Electric was used to generate a netlist of the write buffer, and this netlist was used to substitute for the write-buffer code in the overall chip Verilog code. This substituted version was then subject to the twenty-seven chip function tests designed by the micro-architecture team. The schematic passed all of these tests. Having verified that the schematic functions as expected, the layout was created based on it. The layout's accuracy was verified using DRC, ERC, and NCC within Electric. The layout and schematic pass all of these tests as well.

15. Coprocessor 0

15.1. Function

Coprocessor0 handles exceptions. It takes inputs from datapath, control and external sources and tells the chip whether an exception has been detected. If so, what kind of exception and how much to decrement the program counter by. It also stores the status of the processor.

15.2. Unit I/O

Input	Origin	Output	Destination
activeexception	hazard(datapath)	cop0readdataE[31:0]	execute (datapath)
adelableE	control	isc	n/a
adelthrownE	datapath (Fetch-decode)	pendingexception	control (branchcontroller)
adesableE	control	re	memorystage(datapath)
bdsE	control	swc	memsys
branchdelay	hazard (datapath)		
breakE	control		
fpuE	control		
halfwordE	control		
interrupts[7:0]	chip		
misalignedhE	execute (datapath)		
misalignedwE	execute (datapath)		
overflowableE	control		
overflowE	execute(datapath)		
pce[31:0]	datapath (decode)		
ph1	chip		
ph2	chip		
Reset	chip		
rfeE	control		
riE	control		
syscallE	control		
writedata[31:0]	datapath		

Table 9. I/O table for Coprocessor0.

15.3. Special Units

All cells used, apart from the blocks themselves were standard muddlib/wordlib cells.

15.4. Schematic

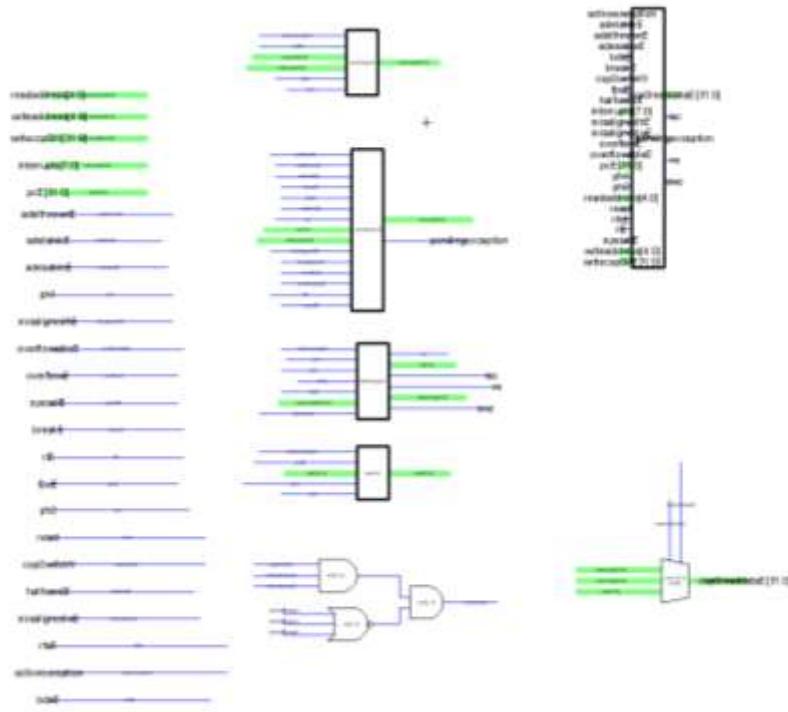


Figure 82. Coprocessor0 schematic

15.5. Layout



Figure 83. Coprocessor0 layout

15.6. Unit sub-components

15.6.1. Staturegunit

Function:

The statusregunit serves as a register that stores the status of the processor. It is used to determine the mode of the chip (kernel/user) and has the ability to disable further interrupts. Many features normally found in the statusregunit of a MIPS processor are not implemented, such as floating point handling and parity checking.

Statusregunit takes the main inputs from the datapath through the 32-bit writedata bus. The result of the status register is sent to the cop0readdatadE mux at the end of coprocessor. The unit also has other interactions with the hazard as well as other coprocessor0 blocks.

I/O Table:

Input	Origin	Output	Destination
writedata[31:0]	datapath	statusreg[31:0]	coprocessor0
rfee	control	im[7:0]	exceptionunit
activeexception	hazard (datapath)	re	memorstage(datapath)
writenable	coprocessor0	swc	memsys
reset	chip	isc	n/a
ph1	chip	iec	exceptionunit
ph2	chip		

Table 10. I/O table for statusregunit

Special Units

Only conventional CMOS circuits from muddlib07 was used in this unit

Schematic:

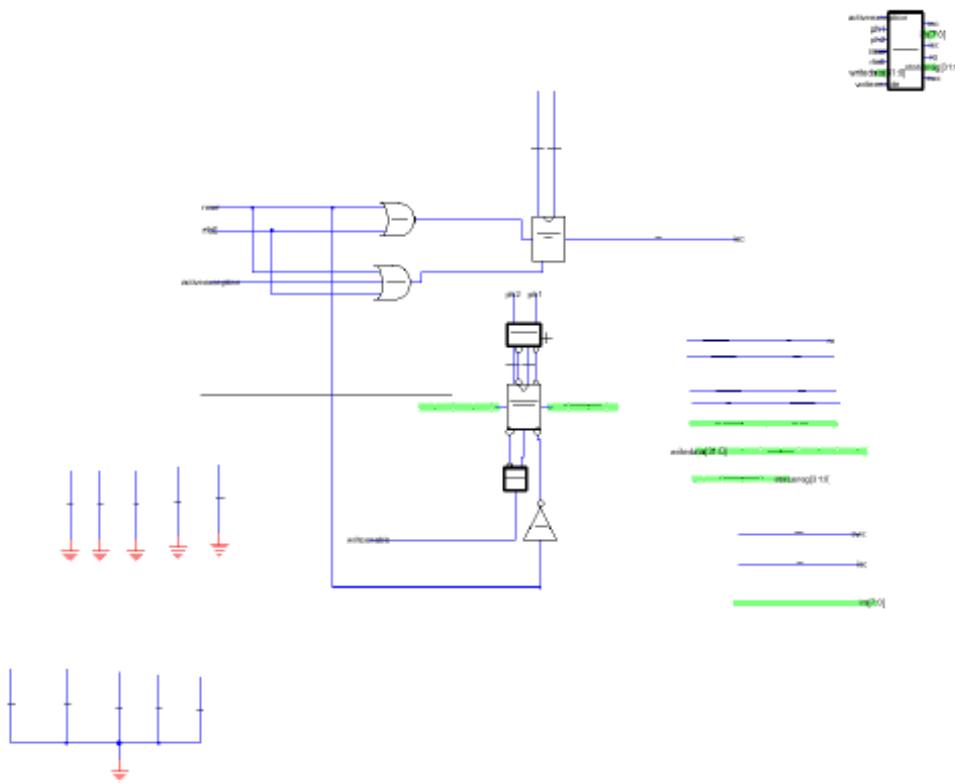


Figure 84. Statusregunit schematic

The block consists of 2 gates and 32 flip flops. Note, despite having 32 flops in this unit, the wordlib's 32 bit flops were not used. The bottom srlo block must not be resettable. Thus, a 31 bit flopnr was used for srhigh and a control flopopen was used for srlo because of the built in clock inverters.

Layout:



Figure 85. Statusregunit layout

As mentioned before a control flop was used for srlo. Flopen_c layouts are wider than the flopenr_d used for srhigh. Thus srlo was pushed into random logic, and bit 0 of the statusreg

datapath is left empty. Fortunately, srlo is the 0th bit of the statusreg signal and thus should be on the very bottom of the layout. However, long clock and reset wires had to be brought down from the zipper.

Testing:

The Verilog netlist generated from Electric was used to replace the statusregunit module in the RTL code. The modified version of mipspipelined.v was put through the 27 microarchitecture team tests and passing all 27 tests.

15.6.2. Causeregunit

Function:

The causeregunit is used to identify and store the cause of the interruption or exception received by the chip. The unit takes in the 8 bit interrupt signal from sources external to the chip as well as exception code from the exception unit. The output of the block is fed into the mux at the end of the coprocessor.

I/O Table:

Input	Origin	Output	Destination
interrupts[7:0]	chip	causereg[31:0]	coprocessor0
exccode[4:0]	exceptionunit		
activeexception	hazard (datapath)		
branchdelay	hazard (datapath)		
ph1	chip		
ph2	chip		
reset	chip		

Table 11. I/O table for causeregunit

Special Units

Only conventional CMOS circuits from muddlib07 and worldlib was used in this unit.

Schematic:

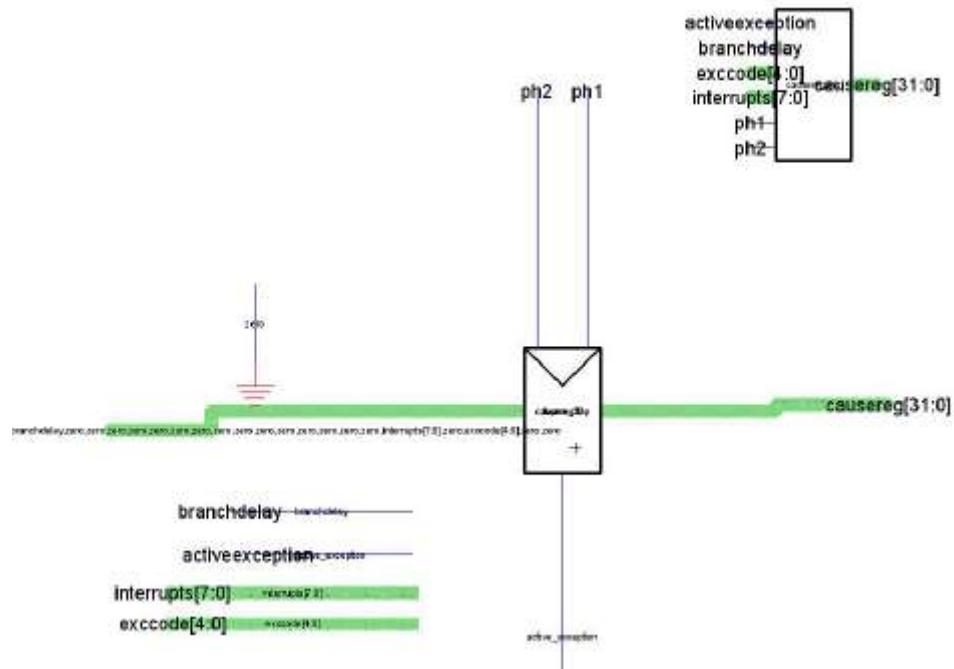


Figure 86. Causeregunit schematic

The worldlib's 32-bit flop was used in this block, no other logic is present.

Layout



Figure 87. Causeregunit layout

The block was laid out following standard datapath layout procedures.

Testing:

The Verilog netlist generated by Electric was used to replace the causeregunit module in the RTL code. The modified version of mipspipelined.v was put through the 27 microarchitecture team tests and passing all 27 tests.

15.6.3. Epcunit

Function:

The epcunit calculates and stores the exception program counter. It takes the program counter from after the decode stage and outputs to the end mux in coprocessor0. One of the inputs of the adder is hard-coded 0xFFFFFFFFC or -4 in 2's complement.

I/O Table:

Input	Origin	Output	Destination
pcE[31:0]	Decode (datapath)	Epc[31:0]	Coprocessor0
Activeexception	Hazard(datapath)		
bdsE	Control		
Ph1	Chip		
Ph2	Chip		
Reset	Chip		

Table 12. I/O table for epcunit

Special Units

Only conventional CMOS circuits from muddlib07 was used in this unit

Schematic:

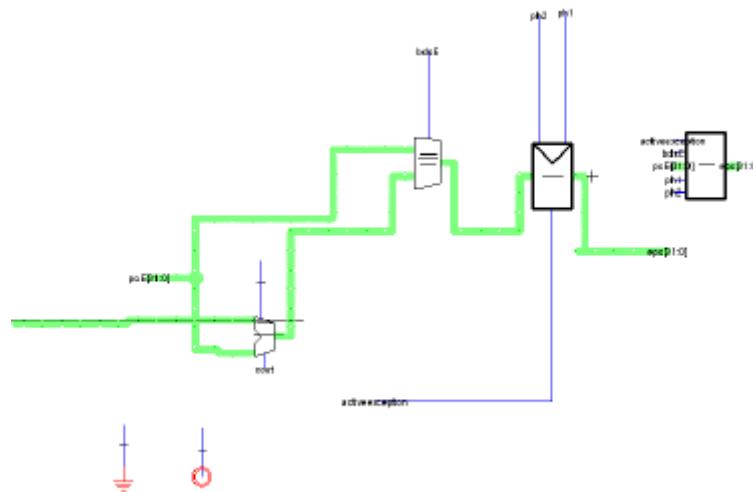


Figure 88. Epcunit schematic

Layout



Figure 89. Epcunit layout

The arrangement of the 3 modules are: adder, MUX2, and fopen. All other aspects of the block was laid out following standard datapath layout procedures.

Testing:

The Verilog netlist generated by Electric was used to replace the epcunit module in the RTL code. The modified version of mipspipelined.v was put through the 27 microarchitecture team tests and passing all 27 tests.

15.6.4. Exceptionunit

Function:

Exceptionunit tells the chip if an exception has been detected (pendingexception) and what kind of exception (exccode[4:0]). The unit consists of random logic so it is not located in line with the datapath as the other units of Coprocessor0 are. It is located above the rest of Coprocessor0, in line with the 5-bit datapath and the zipper.

I/O Table:

Input	Origin	Output	Destination
im[7:0]	Statusregunit	pendingexception	Control (branchcontroller)
interrupts[7:0]	Chip	Exccode[4:0]	Causeregunit
adelableE	Control		
adelthrownE	Datapath (Fetch-decode)		
adesableE	Control		
breakE	Control		
fpuE	Control		
halfwordE	Control		
iec	statusregunit		
misaligneddhE	Datapath (execute)		
misaligneddwE	Datapath (execute)		
overflowE	Datapath (execute)		
overflowableE	Control		
riE	Control		
syscallE	Control		

Table 13. I/O table for exceptionunit

Special Units

The logic to produce exccode and the 8:3 priority encoder were made into individual cells but like the rest of exceptionunit, they consisted of standard cells from muddlib07.

Schematic:

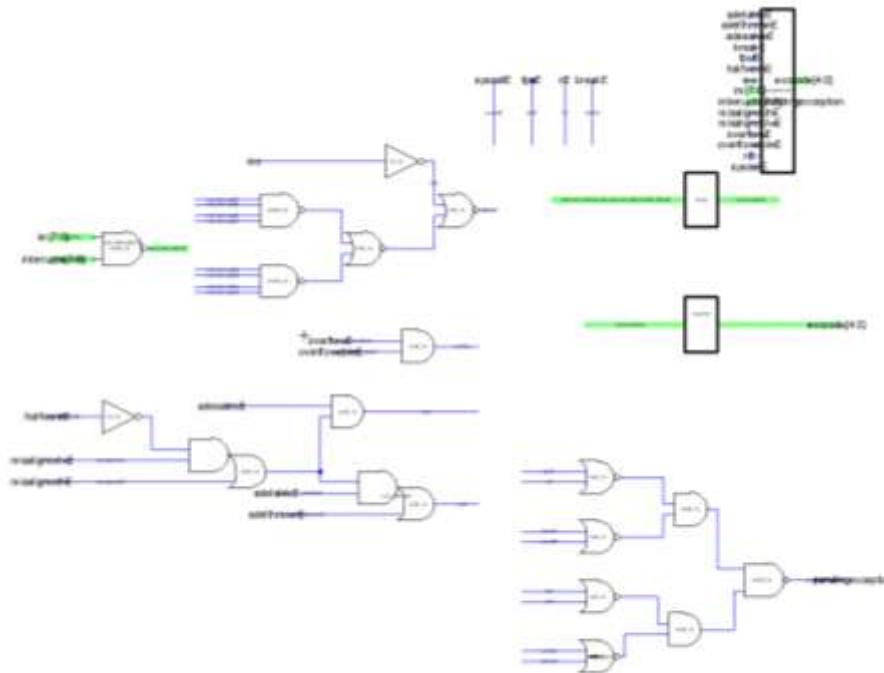


Figure 90. Exceptionunit schematic

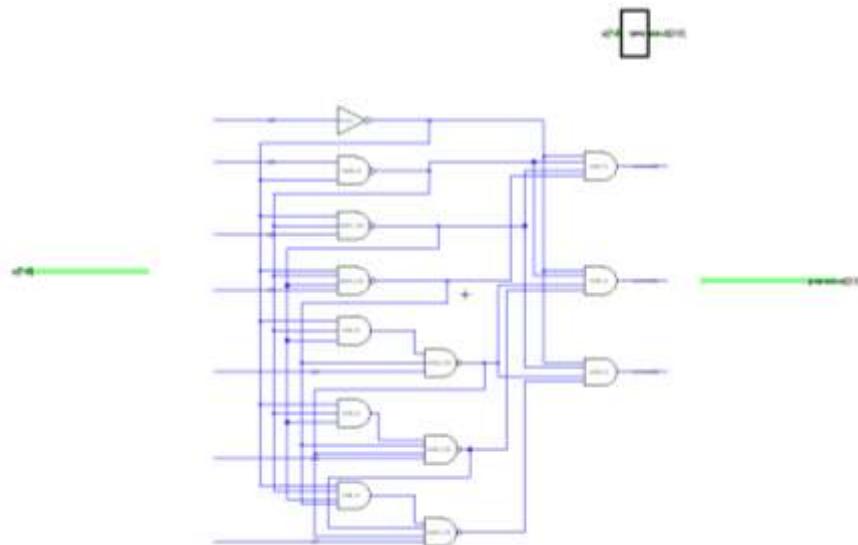


Figure 91. Prienc schematic

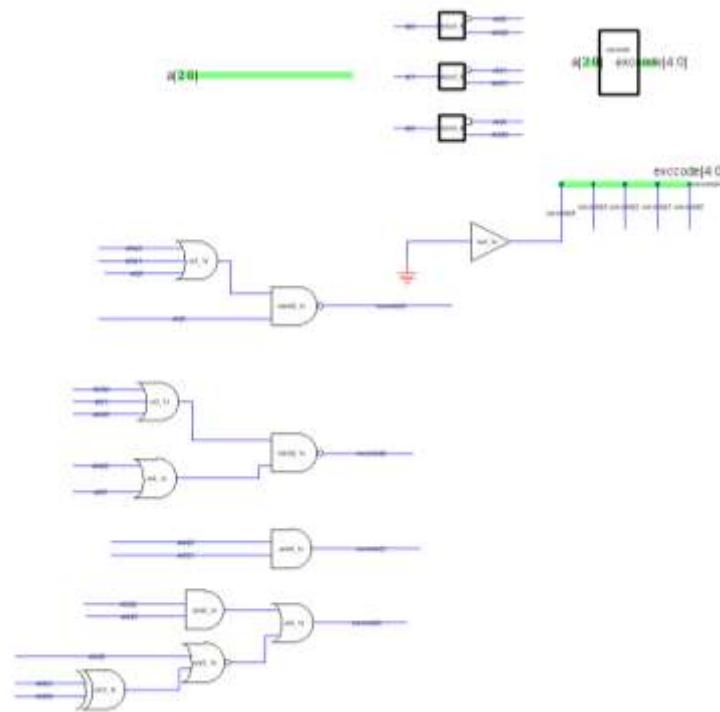


Figure 92. Exicode schematic

The block consists completely of random logic. Prienc is an 8:3 priority encoder but the LSB is never used. The MSB of exicode is always 0.

Layout

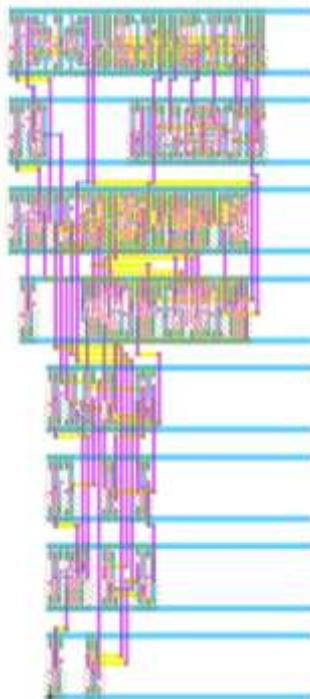


Figure 93. Exceptionunit schematic

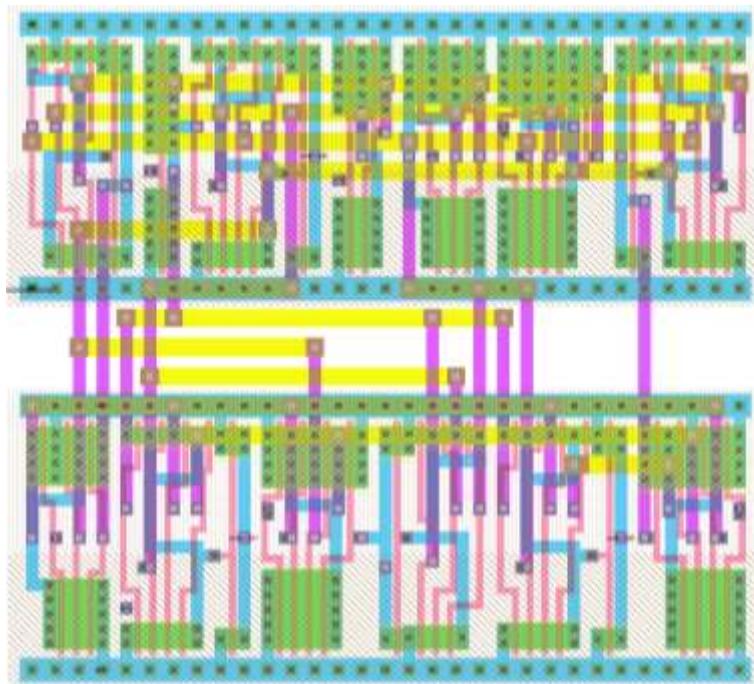


Figure 94. Prienc layout

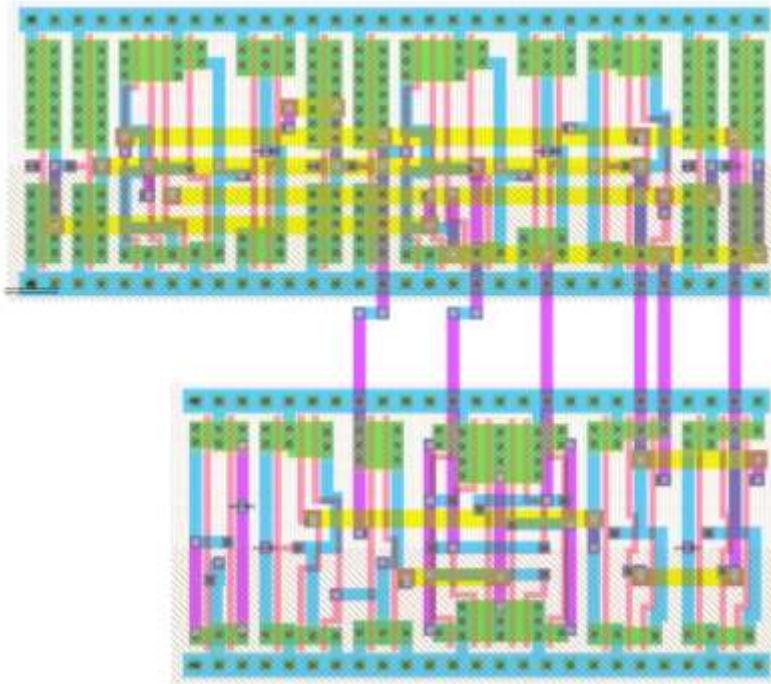


Figure 95. Exicode layout

Testing:

A Verilog netlist was generated using Electric and ten test vectors designed to test corner cases were run. All tests were passed.

Block testing:

The Verilog netlist generated from Electric was used to replace the epcunit module in the RTL code. The modified version of mipspipelined.v was put through the 27 microarchitecture team tests and passing all 27 tests.

16. Lessons learned and experience gained

16.1. Datapath Team

Dane Linbald

The most important thing that I learned through this whole chip-design process was that you definitely have to make sure that everybody has the same ideas of what the design rules are. If anybody makes a mistake it causes others even more time trying to fix it.

Examples:

A cell is designed with a horizontal track of Metal-2 in the vertical track where the input is supposed to run. People using the cell run wires in Metal-1 to compensate. The cell gets corrected, the other cells have to be redesigned.

Power and ground are supposed to be run in Metal-1 at set distances across the entire chip for the datapath and zipper. A block is designed using Metal-1 for a signal in the spaces where power is supposed to run. The block then needs to be redesigned to run power and ground across it.

The memory-writeback block had several examples of problems with moving individual wires and accidentally moving the cells that they were attached to. A handy way I found around this was by moving the node or via that the wire to be moved is attached to, until the wire is in position and then moving the node or via back to the proper location.

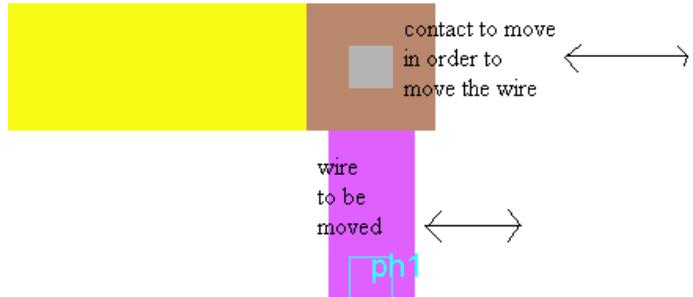


Figure 96. Wire moving example

Jacques Favreau

The most abundantly clear lesson I found was that hierarchical design pays off in an extreme fashion when the systems push past a couple small units. The ability to create a library such as our wordlib and access 32 bit arrayed elements without additional hassle was a great help.

The other thing that became clear as we entered the layout and integration stages of the project was that the design rules may seem arbitrary and fudgable in the beginning but without them chaos reigns. With 17,000 plus networks in the decode+reg alone this project would not have been possible without solid rules and hierarchical design.

The program Electric took quite a bit of getting used to. Even today while looking over my schematics for this block report I found a new feature. The biggest thing I think I pioneered within our section was my work with editing the design in Electric's text-based storage format. This was most useful when you needed to rename entire multi-bit networks, delete or replace a large number of exports, or any large-scale operation where doing it by hand would take too long. Additionally I was able to save Nate an entire night's work of replacing elements from an old library by changing the file to reference the newer library. With a few regular expressions, a good text editor and a couple sleepless nights spent understanding the storage format, this manual editing technique saved quite a few members of our group a lot of time (more than I spent learning it I guess, so it seems like we came out on top.)

William Schulze

Hierarchical structure, even when it wouldn't obviously reduce design time (making 2D arrays with Electric in one action is possible), can save in the long run (it's better to make a 1D array of 1D arrays).

Also, design rules which may seem arbitrary levels are instituted for good reason. For instance, it may seem irritating to only use metal2 vertically and on an 8l pitch, but when you are assembling a very large structure, you reap the benefit of the discipline.

Andy Chin

Some tasks have a much higher priority to get done than others and should get completed as quickly as possible to prevent any dead time. For example, the entire execute block hinged on

the completion of the mdcon, which was completely dependent on the PLA generator. Although the ALU and Shifter were completed weeks before the execute block was assembled, the mdcon held up the process, which slowed down production for many people in datapath and caused a lot of extra stress. Identification of these hotspots would have made layouts and overall assembly much smoother.

Additionally, the shifter has some non-ideal qualities about it, mainly the extra wiring and bypassing vdd/gnd in metal-3. This might have been avoided if those wires were taken into account during the creation of the schematic and not during layout. Again, better foresight and better floorplanning would have alleviated this problem.

Mike Saldana

Throughout this project, the idea that careful planning saves loads of time has been reinforced many times. From the initial floor plan estimates to the ordering of the elements in the Multdiv unit, I have repeatedly seen and experienced how careful initial planning has made life easier in layout. I have also gained a lot of experience with debugging. The Multdiv unit has had many problems throughout the process and through all the trials and tribulations, I feel like I have gained useful debugging knowledge that I can use in the future. Using the list command in Modelsim helped us debug the infinite combinational loops found when combining the schematics for Multdiv and mdcontroller.

Cassie Chou

This unit took plenty of time to complete. Several unforeseen circumstances arose, causing the unit to take longer than expected. I took several lessons from designing the multdiv controller. Mostly, I developed a better sense of how to debug any circuit and how to design digital circuits. I consolidated my knowledge of sequential and combinational circuits in creating the final revision of multdiv controller. Also, I learned how to gauge the amount of time that would be required for future similar projects. I was overly optimistic in the beginning about multdiv controller and thought that I would be able to complete the unit by putting in around 10 hours per week, when in actuality it took closer to around 20 to 25 hours per week. I also learned about creating wise test vectors and learned to find corner cases within Verilog code, which is a very important skill in debugging.

Nate Schlossberg

Designing the hazard unit required careful planning, since there was a large amount of random logic. Moving the 5-bit comparators to the five-bit datapath was an excellent idea, and saved a lot of grief and wiring tracks in the long run. Every now and then, as when I designed the five-bit datapath, some components would seem to get accidentally shifted by $\frac{1}{2}$ or 1λ , which would be hard to fix. It most likely resulted from accidentally moving components with the arrow key.

The biggest challenge in designing the unit is in planning the order of the cells, especially since a large portion of it works with random logic. Electric is a tricky program, it would become difficult to move large portions of the unit (as I did in making that gap in the floorplan). The amount of prior planning in creating a layout was unexpectedly long.

John Parker

I recommend having everyone write a description of the block they plan to build before they begin (on the first day). In this description, they should describe the purpose of each signal, which seems tedious, but will pay-off later into the design. I learned that the time necessary for redoing any single block schematic or layout in the datapath design process will slow down the entire team often by an equivalent amount of time.

Planning the integration carefully will save considerable time when integrating layouts. The planning includes a bitslice plan with carefully designated tracks for all wires. The tracks should include all signals and the tracks should be optimized to produce the densest packing. Even with careful planning in our design, one to two signals were found within each stage which were not included on the slice-plan. Therefore I recommend leaving a metal3 and a metal1 track available in each stage after signal optimization occurs.

When initially constructing a floor-plan and slice plan for the unit, I recommend that a Visio template is made with a set sizing scale. After this, the integration works best if there is only one central file that everyone works from. Having several Visio documents (no matter how stringent the guidelines are) will be difficult to connect together and tedious to edit once connected.

I learned through integration to run DRC before each save (even if the cell had already passed DRC or I was trying to complete NCC), and to increase the number of undo's to 100 minimum in the preferences menu. I learned that selecting the wrong section will often distort the entire wiring network and that a DRC is usually advisable before continuing. The F1 option for mimic stitching is also advisable to speed up the process.

Lastly, I recommend that future teams divide the datapath into as many parallel sections for each task as possible. The parallel sections will pass tests more readily, will be easier to debug, and will be completed faster allowing the critical path to be recognized at an early stage in the design. Spending the time to break apart complex blocks into simple ones was worth it for us, as was structuring the datapath to promote pitch matching and intuitive block clumping.

16.2. Controller Team

Dan Pivonka

Each team seemed to have an entirely different set of challenges, and for the controller most of the complexity was in the wiring and layout. I especially discovered this in doing the layout of the control registers, in which I was constrained to a height of 600λ . This required the registers to be arranged in columns, and so wiring the inputs and outputs become even more complex. I found that it was greatly beneficial to devote time early on to planning out the design and in keeping the wires as organized as possible. Even our fairly well-planned out designed proved tedious to wire, which I think is unavoidable for a unit like ours.

It was somewhat difficult to be so reliant on the work of other teams in the progress of our own unit, but I think it was a good experience to work closely with them to find the optimal solution. The PLA generator from the library team needed some revamping, but in the end it worked very well. Additionally, in the final stages of wiring, my team committed to lining up our wires with Datapath, and the controller would never have been within its size constraint if we had only made changes to the controller. After looking at the layout for Datapath, it became obvious that it would be much simpler to move some of their wires as well as some of the control

wires. I am confident that this decision saved us several hours of work, and it was the result of simple communication between our teams.

Matt Weiner

I was part of the controller group in the chip cluster. Personally, I was responsible for the main decoder, the branch controller, two units of random logic, and wiring some of the controller together. I also helped the decode stage of the datapath by creating the register file decoder. One of the most important lessons that I learned in this project was that the preliminary planning stages have a large effect on how difficult the layout stage will be. For instance, the way that we laid out the controller minimized the amount of extra wiring that was needed. We kept the modules that had common inputs next to each other, and put modules that used a previous input generated in the controller after the module that generated the signal. As it was, the wiring was complicated, but if we had not planned ahead of time for this, it would have been unmanageable. Another important lesson learned was to make the layout of large modules as regular as possible. For example, in the register decoder, the layout was broken down in blocks of read decoders and write decoders. These could then be arrayed, along with pins used for wiring and exports, which could be connected quickly with mimic stitching. Also, using the mimic stitching with dialogue option lets you choose if you want to mimic stitch with each wire created so that you do not accidentally create stitches you do not want. Finally, communication is necessary between the different units. For the controller group, we needed to keep in contact with the datapath team so that we would know where to run the control wires down so that they could be connected to the datapath easily. Overall, the project took a lot of effort from everybody, and if any group had not pulled their weight, the whole project would not have been successful.

Making the layout as regular as possible is the best way to make larger cells. It allows the use of the array command and mimic stitching. The array command was useful for placing the 31 repeated decoder_write cells, the two rows of 32 repeated decoder_read cells, contacts, and exports. Mimic stitching could then be used to connect all of the ports to the contacts or exports.

16.3. Memory Team

Michael Dayringer:

The most important lesson that I learned from this experience was that planning will set you free. A lot of time was wasted during this project redoing parts that did not work well together with other parts. Parts that were planned out ahead of time to match specifically with other parts saved a lot of time down the road when they were joined together. Simply assuming that two blocks sitting next to each other will be easy to join does not actually work. We could have saved ourselves a lot of work by planning out all the wiring ahead of time instead trying to find room for it all at the very end.

Another important thing I learned from this project is that transistors and gates aren't a problem to make and place; rather, the hard part of making a chip is the wiring. Wires take up a lot of space and are generally difficult to place and route. If you don't plan for how they are going to get around the chip, you will likely end up not having space for them all.

On the less technical side, this project was great experience in working with different people. Prior to this project, I didn't know any of the people on my team very well, so I had to

quickly learn to work with them so we could accomplish our jobs. In addition, our team worked with students from Australia, so we all had to learn how to communicate across the ocean and many time zones. The best advice I took from this experience is to get to know the people you are working with outside of the technical requirements of the project. You will work together better if you are friends and understand each other on many levels.

Stephen Brawner:

In designing this part, I initially started by translating the Verilog directly to chip schematics. A large part of my block was a case statement and I initially figured the easiest way to design this was using muxes with a small amount of select logic. It turned out that using a mux for each *if* statement led to a large cell that would have been expensive in power and space. By redesigning the block with combinational logic for the *nextstate* logic, I was able to save a large amount of space and power.

Richard Priddell:

The cacheram was possibly the most painful module of memory. I learned that just because something can be modeled with four lines of Verilog code, the actual implementation might be significantly more difficult and require further research into the element. I learned that a good floor plan early can be very helpful later as it was not understood exactly how the caches would integrate with the rest of memsys until very late. I also learned that just because something simulates it doesn't necessarily mean it is right, as you must remember that ModelSIM doesn't take transistor sizing into consideration.

In addition, I gained much experience in debugging as it was necessary to compare dozens of signals over long periods of time to track down bugs. The importance of patience in chip design was also learned as finding a single error amongst tens of signals can seem very overwhelming unless you approach the problem calmly and methodically. Overall the block was extremely troublesome, time consuming, challenging, and unsatisfying as the majority of man hours that went into it were spent figuring out why it didn't work.

Whereas the cacheram was a never ending nightmare, the cachecontroller was a breath of fresh air that was easy to understand, easy to design, and worked the first time. I learned that VLSI can cause many headaches and much grief, but it is also a rather satisfying experience.

Anthony Weerasinghe:

VLSI was my first introduction to a large scale project that involved an entire class working together. As such, it gave me valuable experience in working with a small team within the scope of a large group. This project gave me a glimpse of the struggles and difficulties I will likely face when I step into industry. As designer of the write-buffer, I had to change my implementation several times in order to meet the constraints of the overarching memory unit. Such redesign often took long hours and required a lot of new thought. I have come to realize, in hindsight, that such redesign is likely going to be part of any major project that I undertake in the future. Therefore, I think VLSI has given me at least a foundation on how to manage my time effectively and work efficiently as part of a large project team.

16.4. Coprocessor 0 Team

Nikhil Sonde

The strategy of simulating and testing designs on different levels of the hierarchy (cell → block → unit → chip) was a useful technique to make sure that potentially catastrophic errors were caught early, before correcting them would become impossible or prohibitively time-consuming. A useful tip to keep in mind while doing layout for a block/unit would be to budget for columns/rows for wires on paper before starting, especially in unit layouts where multiple buses may run between blocks. This avoids the frustrating situation where most of the wiring is done save for a small bus but there is no empty space remaining within the block/unit.

Ted Jiang

This is quite a project to coordinate. To have about 8 weeks to put a chip together from a sketch drawing to a full blown 160K transistor chip and to direct four units of about twenty people was quite the experience for me. With tremendous help from everyone and the great sacrifices put forth by everyone, I am glad they are all paying off and we have a successful chip. I will try to keep my reflections on the project in two categories, technical and team work.

The very first task I had as the chip lead was to establish a time table for the entire chip. I was definitely not prepared. It was I and Prof. Harris in front of a white board and I was to come up with a schedule for the whole team in about 15 minutes. The schedule was established more for my convenience than a really thought out schedule. The team could have done stuff the first 2-3 weeks of class to relieve the countless sleepless nights we all had. However, circumstance was such that the chip team could not have started without a locked-down RTL, so I feel my schedule was justified for how much time I had to work with. It was not possible to have multiple tasks in one week, say block schematic and unit schematic. What was really lacking and really hurt the teams was the lack of review and correction time. When there were mistakes found in the schematics people has to double up their work load because layout was due next week too. Thus any “going back and fix stuff” really hurt the team and cut into sleep.

Furthermore on team work, the teams could have restructured a little better such that everyone suffered about the same amount of fatigue. We had many wonderful volunteers who recognized the problem and switched or helped out. But things could have been easier if teams were “balanced” from the start.

On the technical side, I cannot stress enough the importance and the art of floorplanning. Despite the amount of preliminary floorplanning or detailed floorplanning we have done it was still not enough and created serious problem at the end. One significant problem is that one must understand the function of a unit before one can plan it. I had very little clue about memory and focused most of my planning on datapath. But there were many subtle details in memory that I had no idea of until much later in the projects. Thus I grossly underestimated the complexity and position of the memory unit that probably have caused the team a lot pain. Even with the amount of work I put on datapath planning, things still wasn’t right. Two crucial aspect of the datapath was pitch and width. For pitch between datapath rows, we went over them again and again to determine that 130λ was enough for wiring running across the whole unit, but in the end it still didn’t work. This brings in another important aspect of planning. Once things are planned large changes cannot happen without extensive rework. I originally planned the 5 32-bit busses between datapath and memory to come from the two edges. However, this was changed by memory during spring break. First, this significant change should not have happened that late

in the project. Datapath pitch was set and thus if we needed more horizontal tracks we might as well scrap the entire datapath layout and start from page 1. Second, I did not analyze the effect of this change until when datapath had to be put together and we realized that we are suddenly short on tracks. This probably cost many people days of sleep to rectify this grave mistake.

Single wire signal or 4-5 bit signal looks like nothing on paper, but in the grand scheme of things, when there are 30-50 of them, then it is something to be reckon with. For the longest time I concentrated on those massive busses but ignored the small wires. This cost the interface between control and datapath quite a bit of wiring around, when we discovered there were 5, 5-bit busses from datapath to control that had no where to turn. A follow up advice to this is, leave wiring space. When I calculated the width of Coprocessor0, it was tiny. But when the layout came out, its width was increased by about 50% due to wires that has to travel up to the 5 bit datapath. You may think there are “plenty of metal 2 tracks to bring signal up from the datapath, but in reality, there are stretches of $3k-4k \lambda$ of space in datapath that has 0 vertical metal2 tracks. Thus the only way wires can reach up to the hazard or 5 bit datapath was through extra wiring space.

17. Chip Cluster Appendix

17.1. Preliminary Floorplanning

This document should help you understand how the numbers in Prelim FP.xls, and figures in Prelim FP.vsd were estimated.

Total Space available: 4mm X 4mm or $13 \text{ k}\lambda \times 13 \text{ k}\lambda$

Core space available: Assume 7501 pads, $11.5 \text{ k}\lambda \times 11.5 \text{ k}\lambda$

Control

PLA control was calculated under the following assumption (Section 11.7 VLSI book):

1. Each input output wire takes 2 metal tracks or 16λ
2. Each case statement takes about 1.5 metal tracks or 12λ
3. about 100λ of extra logic around the main PLA arrays

One exception is the Branchdecoder, which has case statement embedded in case statements which was accounted for by tripling the branchdecoder PLA area

The rest of the control elements is 38 LUT and about 42 registers, the following assumption was used

1. Each LUT is equivalent to 10gates, or 40 transistors. Each transistor for control logic is $1000 \lambda^2$ (Section 1.10 VLSI book)
2. Each registers takes $20K \lambda^2$ (Muddlib library)

In addition 30% of safety/wiring space was added to the final total.

Cache

Using the follow assumption

1. One bit of SRAM is $45\lambda \times 32 \lambda$
2. We will have 1024bytes *8 bits/byte SRAM
3. There are two identical caches

The result of the above assumption is $23.6M \lambda^2$ per KB implemented, which include the fact that there are 2 identical caches.

In addition, 100% decode and wiring space was added to the final total

COP0

Part of COP0 is appended to the end of datapath. Its area is accounted in the datapath area.

The rest of COP0 elements is 26 LUT and about 0 registers, the following assumption was used

1. Each LUT is equivalent to 10gates, or 40 transistors. Each transistor for control logic is $1000 \lambda^2$ (Section 1.10 VLSI book)
2. Each registers takes $20K \lambda^2$ (Muddlib library)

In addition 30% of wiring space was added to the final total.

Datapath

The following assumption was used to estimate datapath

1. Divided into 32 bitslices a zipper unit above and hazard unit below
2. See Preliminary FP.xls sheet 2
3. All component dimensions follows muddlib library
4. All sizes were rounded up to the nearest 10
5. Only 1x components were used in the bitslices
6. Bitslices pitch is 130λ , most wiring required is 11-12 tracks
7. Size estimation for zipper is 30% of bitslices
8. $1.5K\lambda$ of wiring spaces are left on each side of the bitslices for bus wiring

In addition following points of interests are observed

1. Since $\text{pitch} > \text{cell size}$, additional metal 1 wires could be used for wiring if needed, for tracks outside of the cell
2. Instruction cache should be on the left, memory cache on the right to facilitate bus wiring
3. Part of COP0 is attached to the datapath at the end
4. control should be a thin long strip that lies above the datapath's zipper, this facilitates wiring.

Hazard unit has 60 LUT estimated using

1. Each LUT is equivalent to 10 gates, or 40 transistors. Each transistor for control logic is $1000 \lambda^2$ (Section 1.10 VLSI book)

Zipper is assumed to be 30% of the bitslice area.

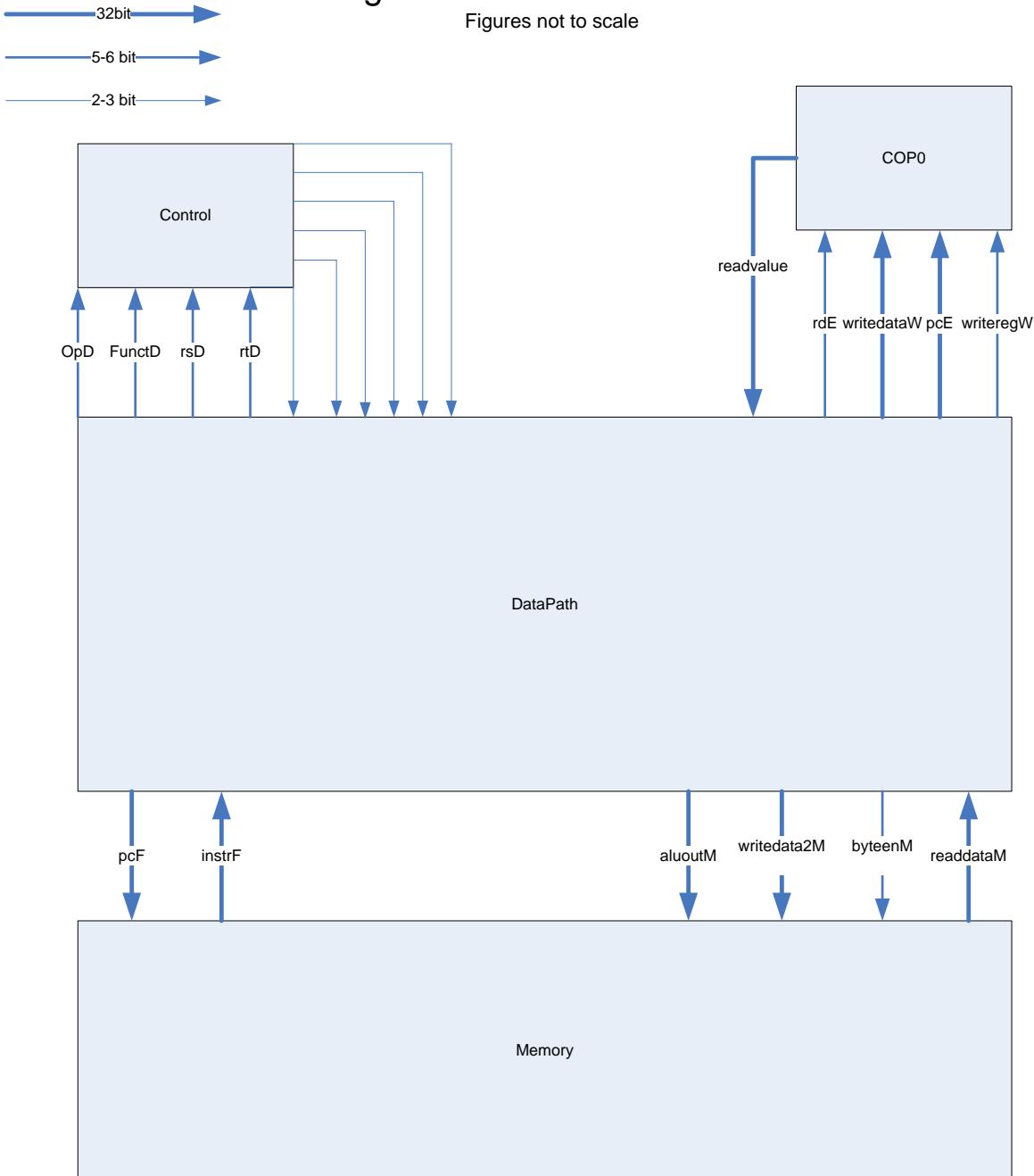
Pads

With $11.5 K\lambda$ available on each side of the chip and pad been 350λ wide, there will be approximately 32 pads available on each side of the chip for a total of 128 pads

Most I/O pins are concentrated on the bottom, because that's where the memory are. But still has some vdd/gnd pad mixed in.

High level block interactions

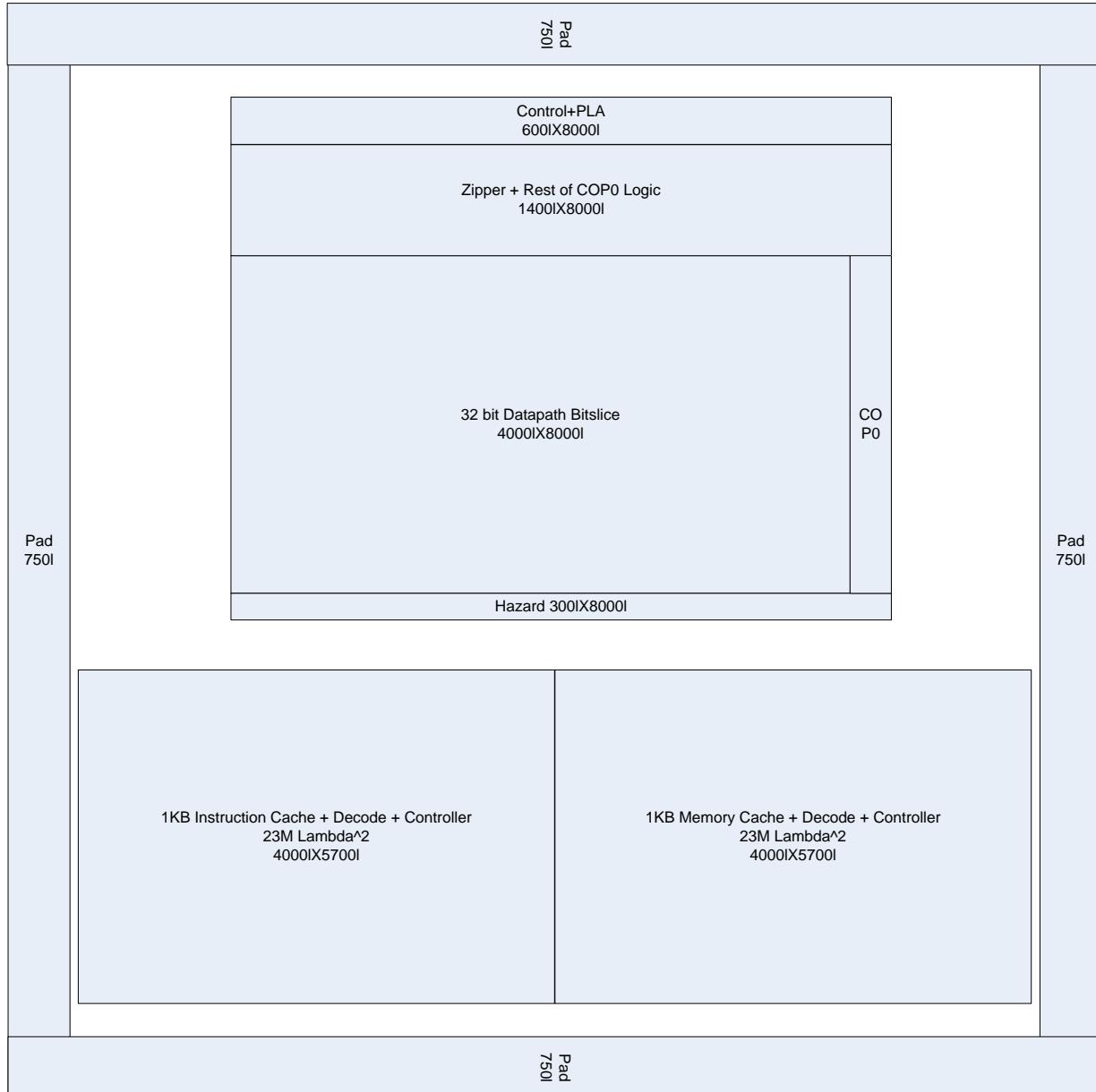
Figures not to scale



This figure shows the high level interactions between each units and major wiring between them. Size, position of the component have no relevance here merely for aesthetics.

Chip Layout

All Figures in Scale



The entire visio sheet is configured to use the unit Λ. All blocks dimensions are accurate to the hundred place (e.g 5200). Note the following

1. Memory blocks include the 100% area increase due to decoder/wiring space
2. memories are folded 64bits X 128 bits
3. Zipper, control, hazard were all sized to have the same width as the datapath bitslice. Their thickness was calculated by (Area estimation)./(bitslice width)

17.2. Pad generation File

Note: Even though this is the pad frame used to generate the pad frame for our chip, it is incorrect. The memdata and memadr pins are on the opposite sides. The memadr pins should be on the right, because the memadr exports in the core is on the right. The same applies to the memdata pins. This mistake was discovered after the PCB for this chip was sent for manufacturing. Thus to gracefully fit the chip in the pads, the core is rotated 180 degrees such that the pins align correctly with the chip's output. This change not only affect the artistic nature of the chip, but also cause the clock and reset wires to run from one end of the chip to the other which may affect performance. Fortunately the clocks and reset are then passed into a large buffer before delivered to the rest of the chip.

```
; Pad Frame arrangement file
; Cassie_Chou@hmc.edu
; 27 February 2007
; Generate a padframe for a 108-pin MOSIS package
; Based on code by David Harris (David_Harris@hmc.edu)

; specify the cell library with the pads
celllibrary muddpads13_ami05.jelib

; make layout and schematic views
views lay sch

; create a top-level cell containing the padframe and core
cell chip

; place this cell as the "core"
core core

; set the alignment of the pads (specifying input and output port names)
align pad_corner      dvdd-1 dvdd
align pad_in          dvdd-1 dvdd
align pad_out         dvdd-1 dvdd
align pad_analog      dvdd-1 dvdd
align pad_dvdd        dvdd-1 dvdd
align pad_dgnd        dvdd-1 dvdd
align pad_blank       dvdd-1 dvdd

;; replace the pad_in and pad_out statements with the pad frame arrangement
you want.
;; keep at least two pad_dvdd and pad_dgnd pads somewhere on the chip to
supply
;; power and ground to the pads and core

; place the top edge of pads counterclockwise, starting with upper-right
place pad_corner
place pad_blank
place pad_blank
place pad_blank
place pad_blank
place pad_blank
```

```

place pad_dvdd
place pad_dgnd
place pad_in din=memdata[0] export pad=memdata[0]
place pad_in din=interrupts[7] export pad=interrupts[7]
place pad_in din=interrupts[6] export pad=interrupts[6]
place pad_in din=interrupts[5] export pad=interrupts[5]
place pad_dvdd
place pad_dgnd
place pad_in din=interrupts[4] export pad=interrupts[4]
place pad_in din=interrupts[3] export pad=interrupts[3]
place pad_in din=interrupts[2] export pad=interrupts[2]
place pad_in din=interrupts[1] export pad=interrupts[1]
place pad_in din=interrupts[0] export pad=interrupts[0]
place pad_dgnd
place pad_dvdd
place pad_in din=RESET export pad=RESET
place pad_in din=PH2 export pad=PH2
place pad_in din=PH1 export pad=PH1
place pad_in din=memdone export pad=memdone
place pad_in din=memen export pad=memen
place pad_in din=memRWB export pad=memrwb
place pad_dgnd
place pad_dvdd
place pad_in din=membyteen[0] export pad=membyteen[0]
place pad_in din=membyteen[1] export pad=membyteen[1]
place pad_dgnd
place pad_dvdd
place pad_blank
place pad_blank
place pad_blank
place pad_blank
place pad_blank
; place the left edge of pads
; from top to bottom
rotate cc
place pad_corner
place pad_blank
place pad_blank
place pad_blank
place pad_blank
place pad_blank
place pad_dvdd
place pad_dgnd
place pad_dgnd
place pad_in din=membyteen[2] export pad=membyteen[2]
place pad_in din=membyteen[3] export pad=membyteen[3]
place pad_in din=memadr[2] export pad=memadr[2]
place pad_in din=memadr[3] export pad=memadr[3]
place pad_in din=memadr[4] export pad=memadr[4]
place pad_in din=memadr[5] export pad=memadr[5]
place pad_in din=memadr[6] export pad=memadr[6]
place pad_in din=memadr[7] export pad=memadr[7]
place pad_in din=memadr[8] export pad=memadr[8]
place pad_dvdd

```

```

place pad_dgnd
place pad_in din=memadr[9] export pad=memadr[9]
place pad_in din=memadr[10] export pad=memadr[10]
place pad_in din=memadr[11] export pad=memadr[11]
place pad_in din=memadr[12] export pad=memadr[12]
place pad_in din=memadr[13] export pad=memadr[13]
place pad_in din=memadr[14] export pad=memadr[14]
place pad_in din=memadr[15] export pad=memadr[15]
place pad_in din=memadr[16] export pad=memadr[16]
place pad_in din=memadr[17] export pad=memadr[17]
place pad_in din=memadr[18] export pad=memadr[18]
place pad_in din=memadr[19] export pad=memadr[19]
place pad_dvdd export pad=VDD
place pad_dgnd export pad=GND
place pad_blank
place pad_blank
place pad_blank
place pad_blank
place pad_blank

; place the bottom edge of pads
; from left to right
rotate cc
place pad_corner
place pad_blank
place pad_blank
place pad_blank
place pad_blank
place pad_blank
place pad_blank
place pad_dgnd
place pad_dvdd
place pad_in din=memadr[20] export pad=memadr[20]
place pad_in din=memadr[21] export pad=memadr[21]
place pad_in din=memadr[22] export pad=memadr[22]
place pad_in din=memadr[23] export pad=memadr[23]
place pad_in din=memadr[24] export pad=memadr[24]
place pad_in din=memadr[25] export pad=memadr[25]
place pad_in din=memadr[26] export pad=memadr[26]
place pad_dgnd
place pad_dvdd
place pad_in din=memadr[27] export pad=memadr[27]
place pad_in din=memadr[28] export pad=memadr[28]
place pad_in din=memdata[31] export pad=memdata[31]
place pad_in din=memdata[30] export pad=memdata[30]
place pad_in din=memdata[29] export pad=memdata[29]
place pad_in din=memdata[28] export pad=memdata[28]
place pad_in din=memdata[27] export pad=memdata[27]
place pad_dgnd
place pad_dvdd
place pad_in din=memdata[26] export pad=memdata[26]
place pad_in din=memdata[25] export pad=memdata[25]
place pad_in din=memdata[24] export pad=memdata[24]
place pad_in din=memdata[23] export pad=memdata[23]
place pad_in din=memdata[22] export pad=memdata[22]
place pad_dgnd

```

```

place pad_dvdd
place pad_blank
place pad_blank
place pad_blank
place pad_blank
place pad_blank
place pad_blank

; place the right edge of pads
; from bottom to top
rotate cc
place pad_corner
place pad_blank
place pad_blank
place pad_blank
place pad_blank
place pad_blank
place pad_blank
place pad_dvdd
place pad_dgnd

place pad_in din=memdata[21] export pad=memdata[21]
place pad_in din=memdata[20] export pad=memdata[20]
place pad_in din=memdata[19] export pad=memdata[19]
place pad_in din=memdata[18] export pad=memdata[18]
place pad_in din=memdata[17] export pad=memdata[17]
place pad_in din=memdata[16] export pad=memdata[16]
place pad_in din=memdata[15] export pad=memdata[15]
place pad_in din=memdata[14] export pad=memdata[14]
place pad_in din=memdata[13] export pad=memdata[13]
place pad_in din=memdata[12] export pad=memdata[12]
place pad_dgnd
place pad_dvdd
place pad_in din=memdata[11] export pad=memdata[11]
place pad_in din=memdata[10] export pad=memdata[10]
place pad_in din=memdata[9] export pad=memdata[9]
place pad_in din=memdata[8] export pad=memdata[8]
place pad_in din=memdata[7] export pad=memdata[7]
place pad_in din=memdata[6] export pad=memdata[6]
place pad_in din=memdata[5] export pad=memdata[5]
place pad_in din=memdata[4] export pad=memdata[4]
place pad_in din=memdata[3] export pad=memdata[3]
place pad_in din=memdata[2] export pad=memdata[2]
place pad_in din=memdata[1] export pad=memdata[1]
place pad_dgnd
place pad_dvdd
place pad_blank
place pad_blank
place pad_blank
place pad_blank
place pad_blank

```

17.3. PLA Simulation Report

This report describes the HSPICE simulation used to verify the functionality of the Programmable Logic Arrays (PLAs) created with the PLA Generator program written by Justin Gries and Danny Lavelle. The PLA tested functions as a two-input xor, which duplicates its output. This gate was chosen because it is simple, with only two inputs, and has two identical outputs because the PLA generator requires at least two outputs. It takes $ab[1:0]$ as the two input bits, and outputs $y[1:0]$.

Using the SPICE tools built into Electric, Vdd was set to be 3.3V above ground, the second input bit was set to be a constant 0V, and the first input bit was made to pulse with a 2 ns delay, 200 ps

rise and fall time, 10 ns pulse width, 20 ns period, and 3.3V magnitude. Then, a SPICE netlist was generated using the built-in functionality in Electric. This netlist was edited to accept parameters from the library files located in the E158 directory. SPICE Simulations were then performed using Typical-Typical, Fast-Slow, and Slow-Fast transistors.

Figure 97 - TT Test

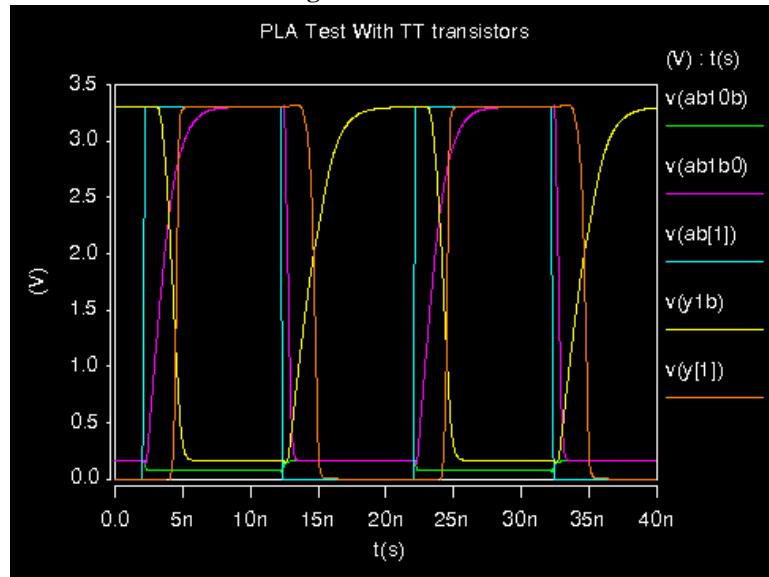
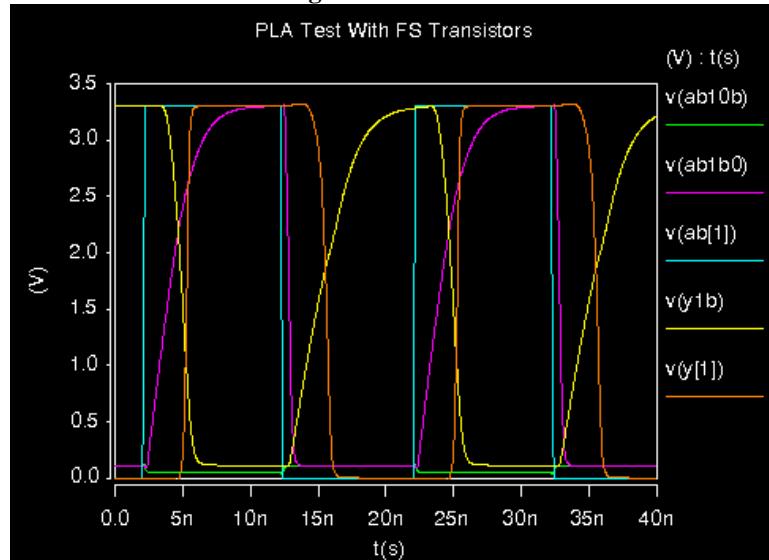


Figure 98 - FS Test



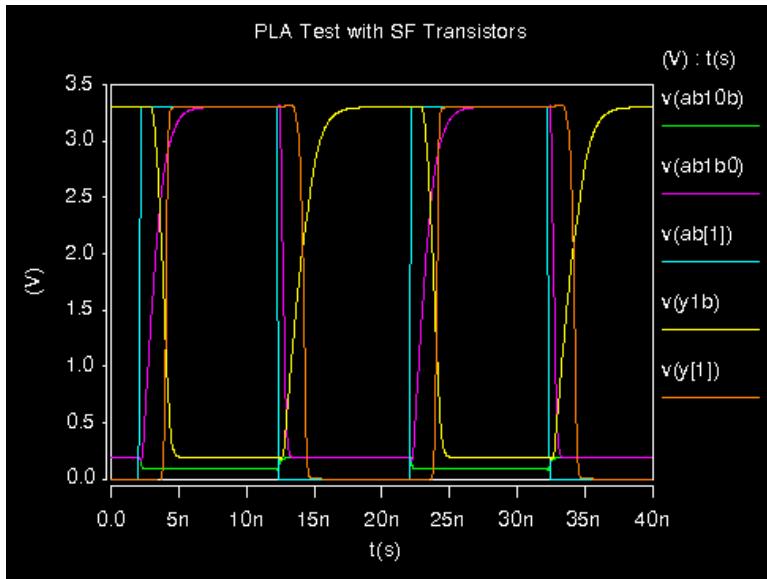
The Typical-Typical transistors yielded the results shown in Figure 97. This plot shows the response of $ab10b$, the minterm corresponding to $ab[1]$ nor $\sim ab[0]$, $ab1b0$, the minterm corresponding to $\sim ab[1]$ nor $ab[0]$, $\sim y[1]$, and $y[1]$ to the pulsed input $ab[1]$. The output, $y[1]$ rises and falls as expected, rising to 3.3V shortly after $ab[1]$ rises, and falling to 0V shortly after $ab[1]$ falls. The inner terms $ab1b0$ and $y1b$ rise slowly, but eventually reach 3.3V, as they should, and fall to below the threshold voltage of 0.7V, not rising above 0.12V when they are supposed to be low.

Figure 98 and Figure 99 show the results from using libraries containing parameters for FS and SF transistor pairs. The FS transistors drop the internal signals lower than Typical – Typical, but

cause the signals to rise more slowly, although the signals still reach 3.3V. The SF transistors only bring the internal signals down to 0.2V, but this is still low enough that it is below the threshold voltage of our transistors. The output signals for each of these transistor sets rise above 3.2V and drop below 0.1V, and closer than 2% of Vdd to the desired value.

These results indicate that the pmos transistors are weak enough, since they allow the signals to drop to below the threshold voltage of transistors, and allow the inverter

Figure 99 - SF Test



to power a valid output.

Systems Cluster

18. Package Definition

Owner: Cassie Chou

Revision: 1

Date: April 17, 2007

18.1. Packaging Function

The purpose of the packaging portion of the chip is to allow the chip to interface with the PCB, which then communicates with the external memory system. The packaging portion of the chip consists of two components: the pin grid array (PGA) and the pad frame. The package chosen for this chip is the PGA108A, with 27 pin outs on each side. This fully accommodates for all signals going into and out of the chip, and also allows for extra power and ground signals to be placed throughout the chip.

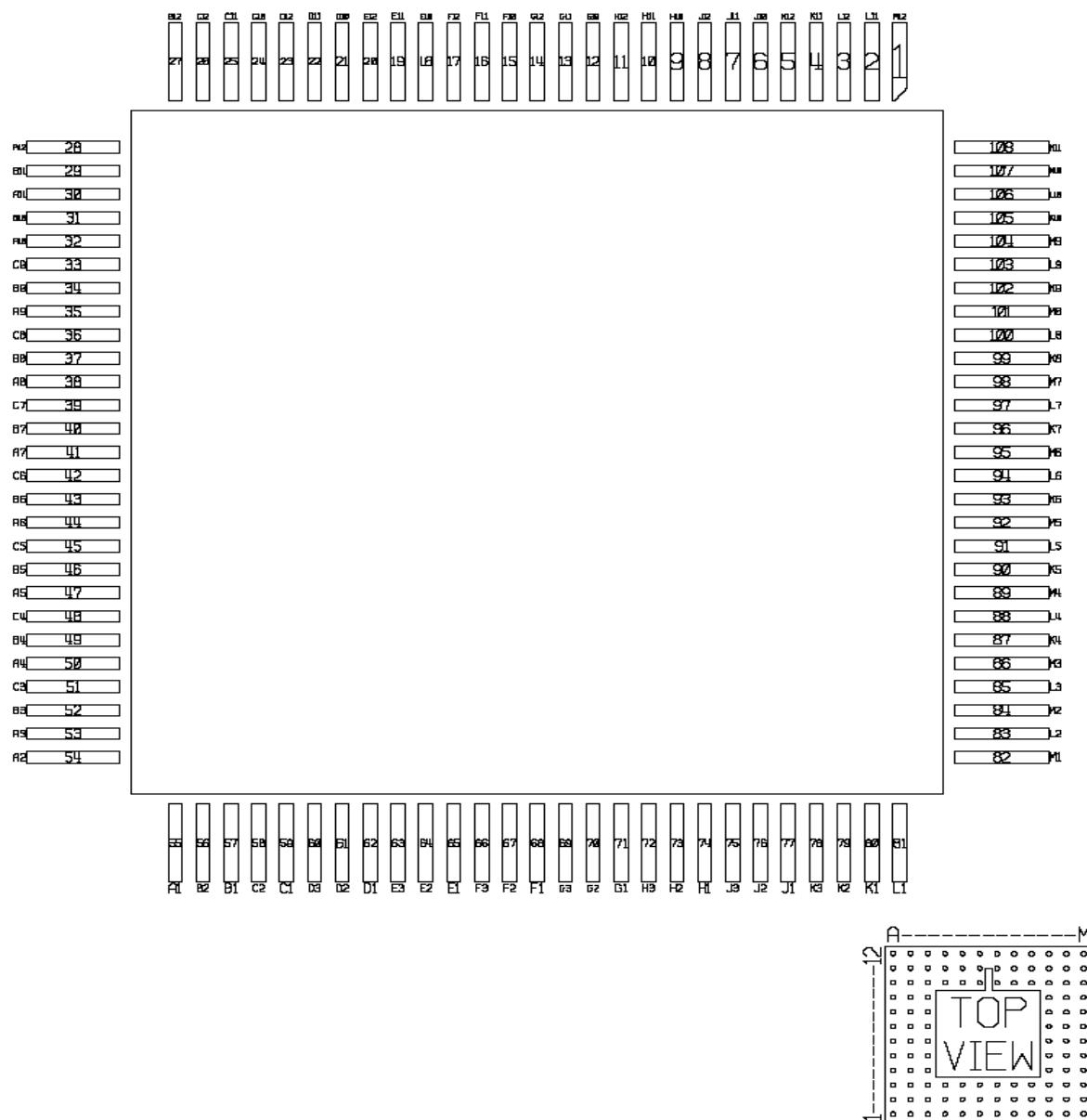
The pin grid array is manufactured by MOSIS according to the connections diagram that is shown below, and it communicates with the PCB layout. The pad frame can be considered as the top level portion of the processor, and it communicates with the different subsystems within the chip as well as the pin grid array. The pad frame can be represented by the bonding diagram that is also shown below. In summary, packaging, along with the PCB, is responsible for communication between the chip and external systems, such as memory.

M	82	84	86	89	92	95	98	101	104	107	108	1
L	81	83	85	88	91	94	97	100	103	106	2	3
K	80	79	78	87	90	93	96	99	102	105	4	5
J	77	76	75						6	7	8	
H	74	73	72						9	10	11	
G	71	70	69						12	13	14	
F	68	67	66						15	16	17	
E	65	64	63						18	19	20	
D	62	61	60						21	22	23	
C	59	58	51	48	45	42	39	36	33	24	25	26
B	57	56	52	49	46	43	40	37	34	31	29	27
A	55	54	53	50	47	44	41	38	35	32	30	28

1 2 3 4 5 6 7 8 9 10 11 12

Pin Side view
of package

Figure 100. Pine side view of PGA



PGA108M (350 MIL SQ CAVITY)

Figure 101. Pad Frame of PGA

18.2. I/O Table

The pin out table is shown below. It corresponds to both the connections and the bonding diagrams that have been shown in previous pages and includes all signals that will be connected to the PGA.

SIGNAL	PIN	PIN #	SIGNAL	PIN	PIN #	SIGNAL	PIN	PIN #
VDD	B12	27	MemData8	K8	99	MemAddr26	E3	63
GND	C12	26	MemData9	M7	98	MemAddr25	D1	62
ByteEn1	C11	25	MemData10	L7	97	MemAddr24	D2	61
ByteEn0	C10	24	MemData11	K7	96	MemAddr23	D3	60
VDD	D12	23	VDD	M6	95	MemAddr22	C1	59
GND	D11	22	GND	L6	94	MemAddr21	C2	58
RWB	D10	21	MemData12	K6	93	MemAddr20	B1	57
EN	E12	20	MemData13	M5	92	VDD	B2	56
DONE	E11	19	MemData14	L5	91	GND	A1	55
PH1	E10	18	MemData15	K5	90	GND	A2	54
PH2	F12	17	MemData16	M4	89	VDD	A3	53
RESET	F11	16	MemData17	L4	88	MemAddr19	B3	52
VDD	F10	15	MemData18	K4	87	MemAddr18	C3	51
GND	G12	14	MemData19	M3	86	MemAddr17	A4	50
Intrpt0	G11	13	MemData20	L3	85	MemAddr16	B4	49
Intrpt1	G10	12	MemData21	M2	84	MemAddr15	C4	48
Intrpt2	H12	11	GND	L2	83	MemAddr14	A5	47
Intrpt3	H11	10	VDD	M1	82	MemAddr13	B5	46
Intrpt4	H10	9	VDD	L1	81	MemAddr12	C5	45
GND	J12	8	GND	K1	80	MemAddr11	A6	44
VDD	J11	7	MemData22	K2	79	MemAddr10	B6	43
Intrpt5	J10	6	MemData23	K3	78	MemAddr9	C6	42
Intrpt6	K12	5	MemData24	J1	77	GND	A7	41
Intrpt7	K11	4	MemData25	J2	76	VDD	B7	40
MemData0	L12	3	MemData26	J3	75	MemAddr8	C7	39
GND	L11	2	VDD	H1	74	MemAddr7	A8	38
VDD	M12	1	GND	H2	73	MemAddr6	B8	37
VDD	M11	108	MemData27	H3	72	MemAddr5	C8	36
GND	M10	107	MemData28	G1	71	MemAddr4	A9	35
MemData1	L10	106	MemData29	G2	70	MemAddr3	B9	34
MemData2	K10	105	MemData30	G3	69	MemAddr2	C9	33
MemData3	M9	104	MemData31	F1	68	ByteEn3	A10	32
MemData4	L9	103	MemData28	F2	67	ByteEn2	B10	31
MemData5	K9	102	MemData27	F3	66	GND	A11	30
MemData6	M8	101	VDD	E1	65	GND	B11	29
MemData7	L8	100	GND	E2	64	VDD	A12	Tabl

Table 14. PGA I/O Table

18.3. Special Units

The pad arrangement file that was written for the pad frame generator is shown below. It is based on a lab completed in E158. The generated cell is written into a cell called proj_chip. This file can be found in Section 17.2

A new pad_blank cell was generated, in order to allow for placement of blank pads with no inputs or outputs. These are necessary because each pad is of a fixed width (710λ). In order to create a pad frame that will accommodate a 4.5 mm x 4.5 mm chip, empty pads are placed, without the actual metal-2-metal-3 node pads (the large brown pads on the layouts), side by side. The schematic, icon, and layout for this pad is shown below:

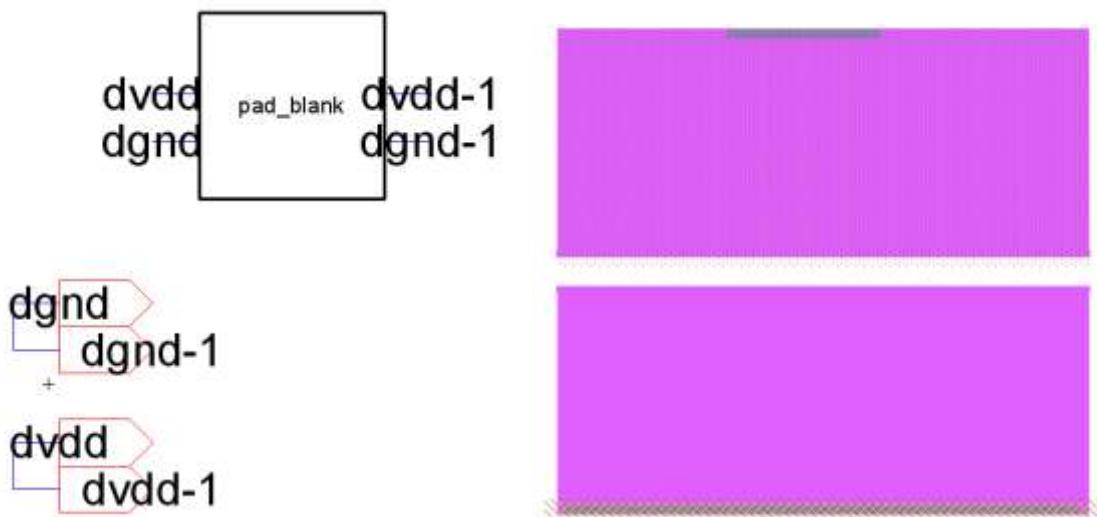


Figure 102. Blank Pad Schematic, Icon and Layout

18.4. Pad Frame Schematic

The block schematic for the pad frame is shown below, without the chip in the core of the pad frame. The one difficulty in generating the schematic was in connecting the blank pads to each other, which was done through mimic stitching. The schematic shows these mimic stitched connections in blue. As requested by the chip design team and the memory team, memory address and memory data signals branch out from the bottom middle portion of the chip, and all other signals are placed near the top.

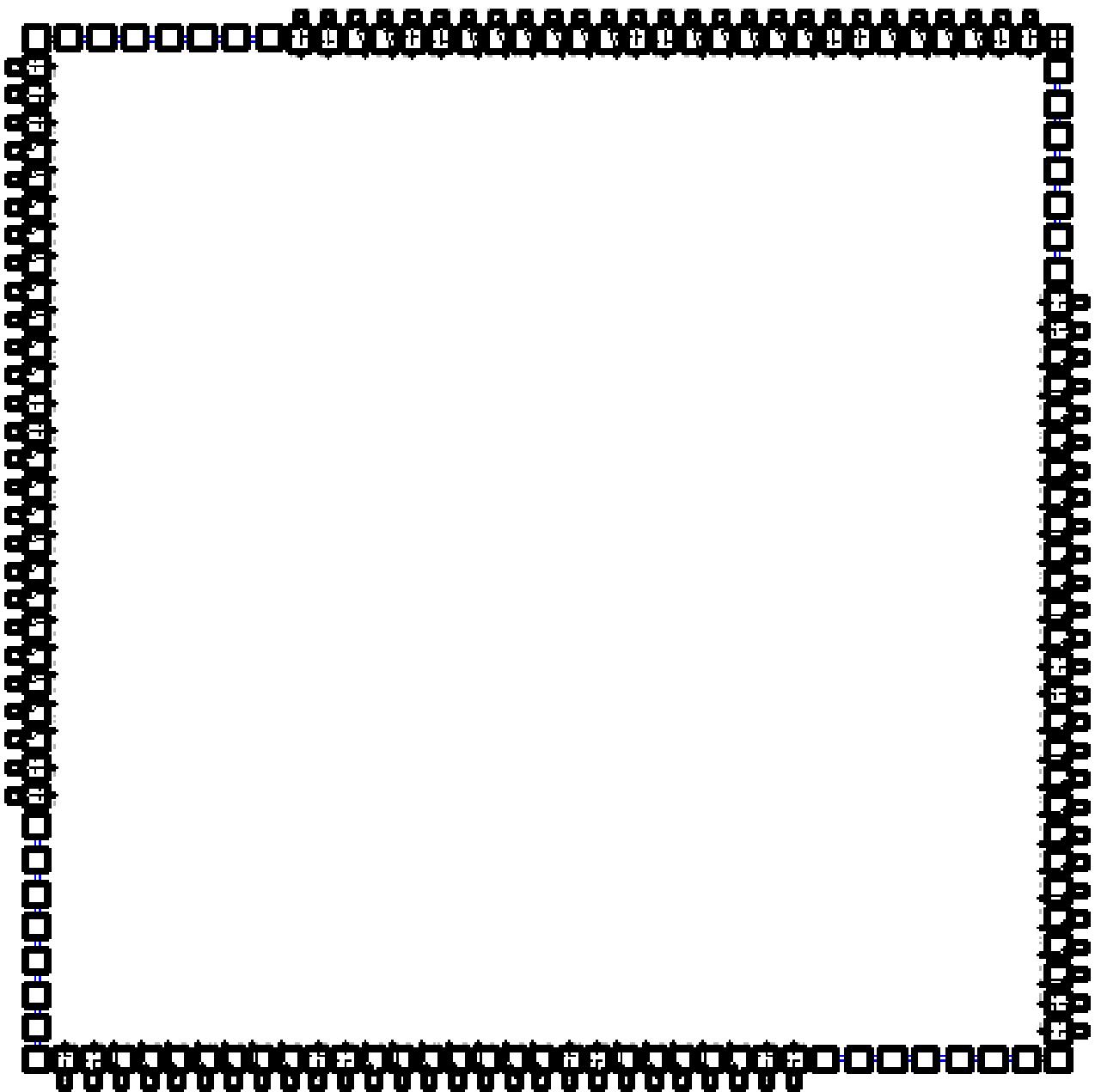


Figure 103. Pad Frame Schematic

18.5. Pad Frame Layout

The pad frame layout is shown below, without the chip in the core. There are ten spacer pads placed on each side of the pad frame ring, and the entire pad frame comes to a size of 14311.5λ x 14311.5λ . This translates to 4.293 mm x 4.293 mm.

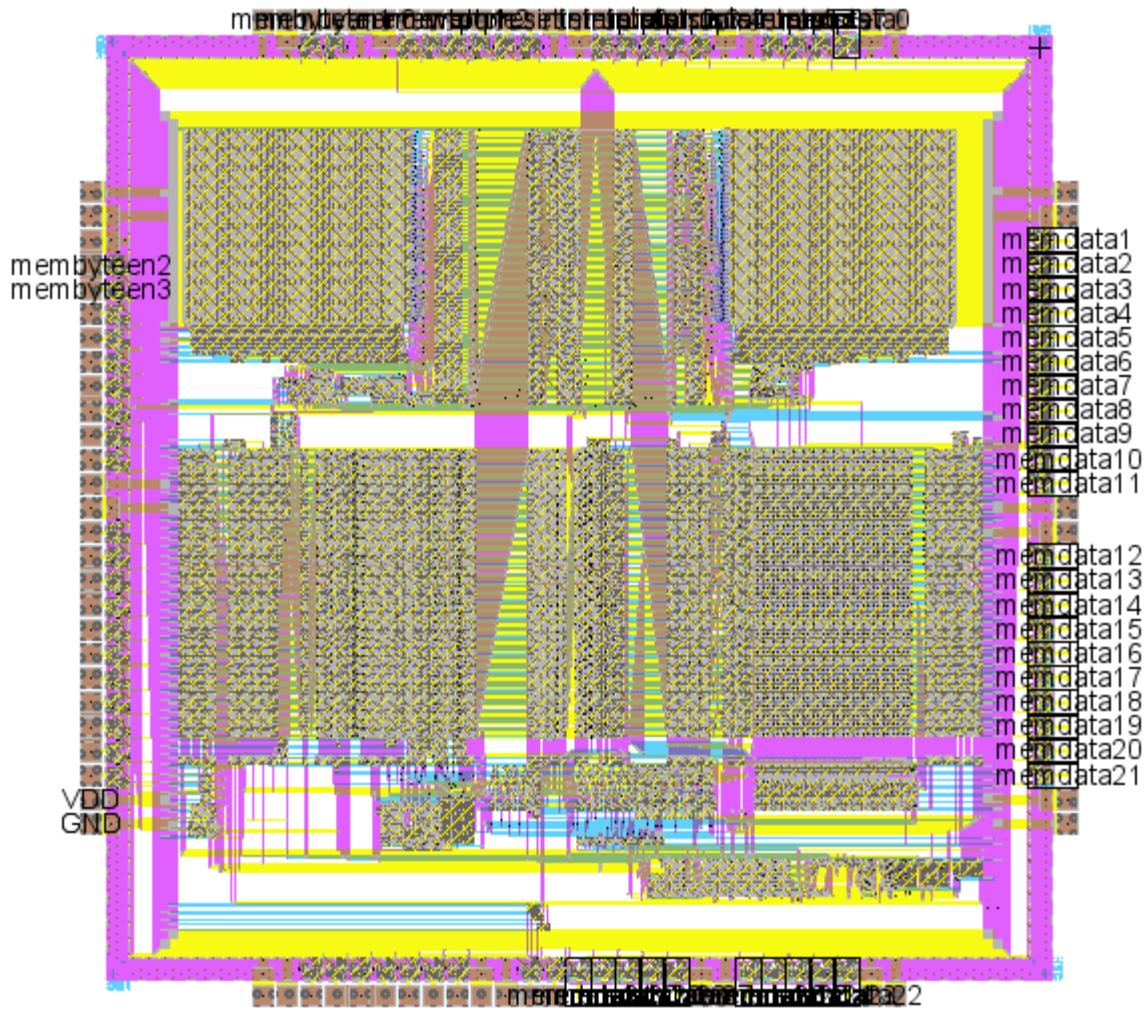


Figure 104. Pad Frame Layout

18.6. Lessons learned and experience gained

In creating the pad frame generator, I developed a better sense of how the chip interacted with the pin grid array, PCB, and memory system. Choosing the package pins helped me understand the signals that would be required by the MIPS processor and memory system, and it also helped me understand the size constraints imposed on the chip.

19. External Memory & I/O System

Owner: Howard Chen

Revision: 2

Date: April 27, 2007

19.1. Hardware

The external memory system is implemented using the Virtex II Pro FPGA XC2VP30 FF896 -7. The FPGA will be connected to the processor on the PCB through two 40-pin headers. Within this FPGA, the Logic and RAM blocks can be used to simulate the external memory system. According to the datasheet for this FPGA, there are a total of 2448Kb (136 blocks of 18Kb) Block RAM and 428Kb Distributive RAM (CLB). The external memory controller and the I/O mapping will be programmed into the Logic Blocks while the ROM and RAM will be programmed into the Block SelectRAM+. Due to the size limitation of the Block SelectRAM+, the external memory system will only be utilizing 17 bits of the address line.

19.2. Design Overview (RTL)

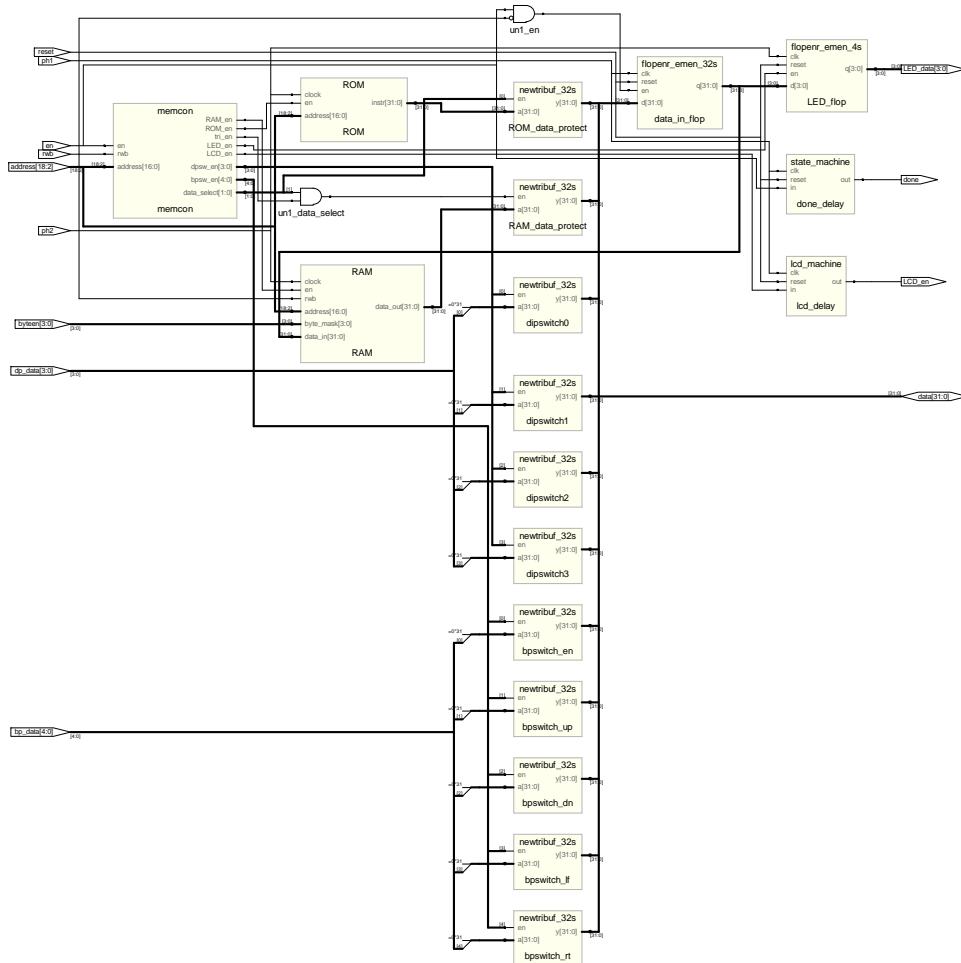


Figure 105. RTL of External Memory

19.3. Design

The design of the external memory system was based around two constraints: signal specifications from the processor and efficient usage of clock cycles. The biggest issue when dealing with signals is the data bus. Details of this issue are highlighted in the signals section. The second issue is the speed of the external memory system. In order to reduce the overall time needed by the external memory, all the RAM blocks are split into four byte addressable blocks. This reduces all writing and reading operations to one cycle time thus leaving the remainder of the clock cycles to deal with other issues such as timing constraints.

The external memory system also allows for quick ROM changes. In order to change the ROM file in the external memory system, just replace the ROM.v according to the template provided in ROM.v Template. If you are utilizing the compiler created by the Systems Cluster, the compiler automatically outputs the compiled file into a ROM.v file. You can directly copy that file into the project folder for immediate use.

NOTE: Sometimes Xilinx Project Navigator does not recognize that the ROM file is changed if you only replace the file. Therefore, you can do one of two things: Remove and then add the ROM.v file from within Xilinx Project Navigator or you can copy and paste the contents of the new ROM file into the ROM.v file within the project.

19.4. Signals

19.4.1. Inputs

Reset:

Restarts all the state machines that control output signals. This is a global reset that restarts the entire system.

Ph1, Ph2 Clocks:

A two phase clock controls the timing of the external memory system. The external memory system runs primarily on the ph1 clock, but the ph2 clock is utilized to help with timing constraints.

En:

Enables the entire external memory system. When it is off, all the control signals in the memory system are off.

RWB (Read Write Bar):

Controls whether the external memory system should be reading or writing. When it is high, it is reading, when it is low it is writing.

Address [18:2]:

Due to the amount of Block SelectRAM+ available, the external memory system only requires 17-bits of address. The first two bits of the address is unnecessary because external memory is word addressable and can be byte addressable through the usage of the byte_en input. Therefore, address only looks at [18:2].

Byte_en [3:0]:

Only during write operations into the RAM, this allows the external memory system to know what bytes to write the data into. This is used directly as a byte mask attached to each of the RAM blocks. When the bit is high, it means that it is writing to that specific byte.

dp_data [3:0]:

The dipswitch data provides data coming from the dip switches on the Virtex 2 Pro Board. They are relatively associated: meaning dp_data[0] = sw0 and etc.

bp_data [4:0]:

The push button data provides data coming from the push buttons on the Virtex 2 Pro Board.

- bp_data[0] = enter
- bp_data[1] = top
- bp_data[2] = bottom
- bp_data[3] = left
- bp_data[4] = right

19.4.2. Outputs

Done:

This bit goes high for one cycle as soon as external memory is done with the requested operation.

LED_data [3:0]:

This is directly mapped to the LEDs on the Virtex 2 Pro. The polarity coming from the external memory is not inverted, meaning that when the bit is high it means to turn the LED on. But according to the specification sheet of the Virtex 2 Pro, the LEDs on the board are inverted, so these bits are inverted in the top file before being assigned to the board LEDs. Again, these bits are relatively associated.

LCD_en:

This is a single enable bit that is directly tied to the LCD screen on the PCB board. The enable bit is controlled through a state machine within the external memory system. The timing of this is tricky because the LCD that is being utilized writes data on the negative edge of clock.

Therefore, the LCD_en bit is delayed until the 2nd cycle of the operation in order to provide proper hold and setup times.

19.4.3. InOuts

Data Bus:

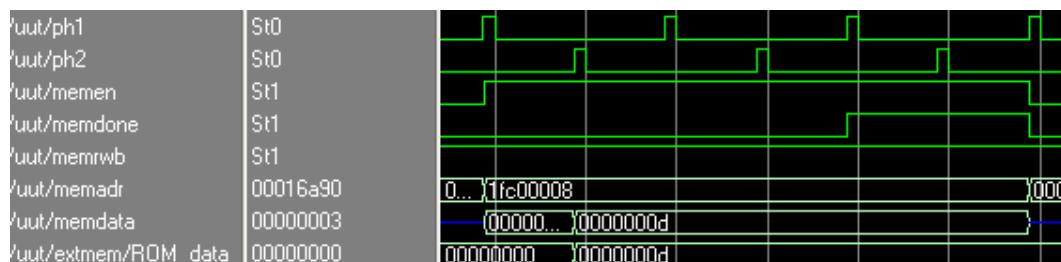
A major signal specification from the processor was the bidirectional data bus. The data bus was extremely crucial because it is being shared by both the processor and the external memory.

Before outputting to the data bus, all signals had to go through tristate buffers to prevent contesting signals. During read operations, the data bus should have high Z's coming from the processor and actual output from the external memory. During write operations, the data bus should have the data signal coming from the processor and high Z output coming from the external memory. The next issue was when writing to the external memory system. All data coming from the processor must be flopped because it is unclear how long the data in the data bus is valid.

19.5. Timing

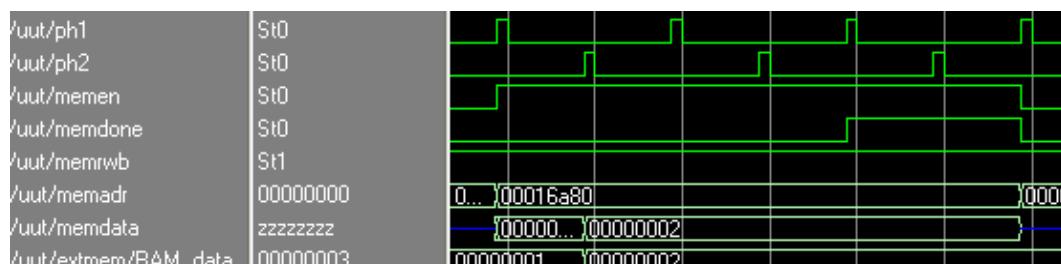
The external memory system runs in a constant 3 cycle process independent of memory operation with the last cycle returning the done bit back to the processor. A 3 cycle external memory is necessary because one cycle is always going to be used by the done bit. Secondly, two cycles are necessary in order to have complete cycles of ph1 and ph2 clocks in order to account for the different dependencies on the clock. Details of each of the operation timing are highlighted with the graphs below.

19.5.1. ROM Read



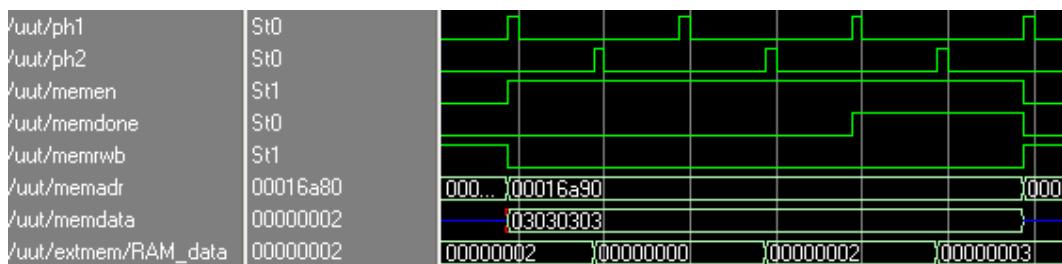
When reading from ROM, it is necessary to prepare the data before the ph2 because the cache system can only write on the posedge of ph2. Since it takes one cycle to read the data, another cycle is necessary to meet the setup time for writing into the cache.

19.5.2. RAM Read



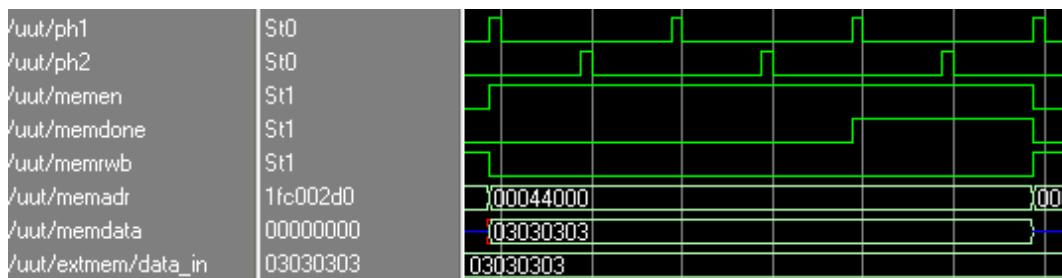
Same issue as reading from ROM.

19.5.3. RAM Write



Because the data line cannot be trusted after one cycle, the data is flopped into a data_in line. This requires one cycle. Then it requires another cycle to write the data into RAM to finally complete the process. After it is complete, the third cycle outputs the done bit. This cannot be reduced anymore because the cache system reacts on the posedge of the done bit.

19.5.4. I/O Write



19.6. I/O Devices

Address mapping for I/O Devices:

- 4x LED (0x0044000) Virtex II Pro FPGA
 - data[4:0] corresponds to each LED.
- 4x Dipswitches (0x0044004) Virtex II Pro FPGA
 - SW0 (0x0044004)
 - SW1 (0x0044008)
 - SW2 (0x004400C)
 - SW3 (0x0044010)
- 5x Push Buttons (0x0044014) Virtex II Pro FPGA
 - BP Enter (0x0044014)
 - BP Top (0x0044018)
 - BP Right (0x004401C)
 - BP Left (0x0044020)
 - BP Right (0x0044024)
- 2x16 CrystalFontz LCD Display (0x0011028) PCB
 - LCD_en

Each I/O device will be associated to a unique memory address except for the voice synthesizer chip because it requires two inputs. Therefore, only 7 memory addresses will be necessary for our implementation, but 32 memory addresses will be reserved for I/O devices. This will allow for future additions of I/O devices.

All the I/O devices will be located on the PCB in order to achieve a stand-alone design. This way only the external memory system is tied to a specific FPGA. This allows freedom in choosing another FPGA to replace the external memory system without affecting our programs or I/O devices.

19.7. Memory Map

The memory map shown in Figure 106 is unique because of limited RAM on the FPGA and because Virtual Memory is not being implemented. This allows the external memory controller to only look at 17 bits of the memory address that is coming from the processor. The 3 MSB are not necessary because it determines caching. The 2 LSB are not necessary because memory has to be word addressed. Finally, due to a 69,632 word limitation in the memory space only memadr[18:2] are utilized. The remaining bits will be left open.

On initialization, the MIPS processor will send a request to address 0xBFC0 0000 for Instructions. Because the memory controller only looks at certain bits within the address, the starting address is equal to 0x000 0000, which is convenient since it eliminates the need to jump to a separate address.

Due to limited space, 1/3 of the available word lines were given to ROM and the remaining 2/3s were given to RAM. Finally, 32 addresses were allocated for I/O devices, which is plenty since only 11 addresses will be used in the final implementation.

Two versions of the memory map is provided. The first version is based on a 32-bit address with the first four bits omitted. This version should be used globally by any other system other than the external memory controller. The second version provides the addresses that the external memory system looks at, which is based on a 17-bit address instead.

19.8. Test Plan

To ensure that the external memory will be functional after its implementation, a simple test utilizing the various spaces within the memory map and I/O devices will be tested. The test will consist of testing the extreme ranges of the memory map as well as various sections of the memory.

On initialization, the processor will access the starting address where it will jump the processor to the beginning of the ROM. The instructions will then load two numbers into the extreme ends of the RAM by saving the numbers into a global variable and the stack. Then it will perform an addition of the two numbers and attempt to store that value into the beginning of the heap. Finally, if the final value matches the expected value, it will send a signal to a memory mapped LED.

Multiple tests were performed afterwards on the external memory to test the system thoroughly. These tests include LEDs responding to dipswitches using global variables, LEDs responding to button presses, initializing the LCD and outputting “Hello World” on the screen, and finally the demo program.

The external memory successfully passed all tests mentioned above.

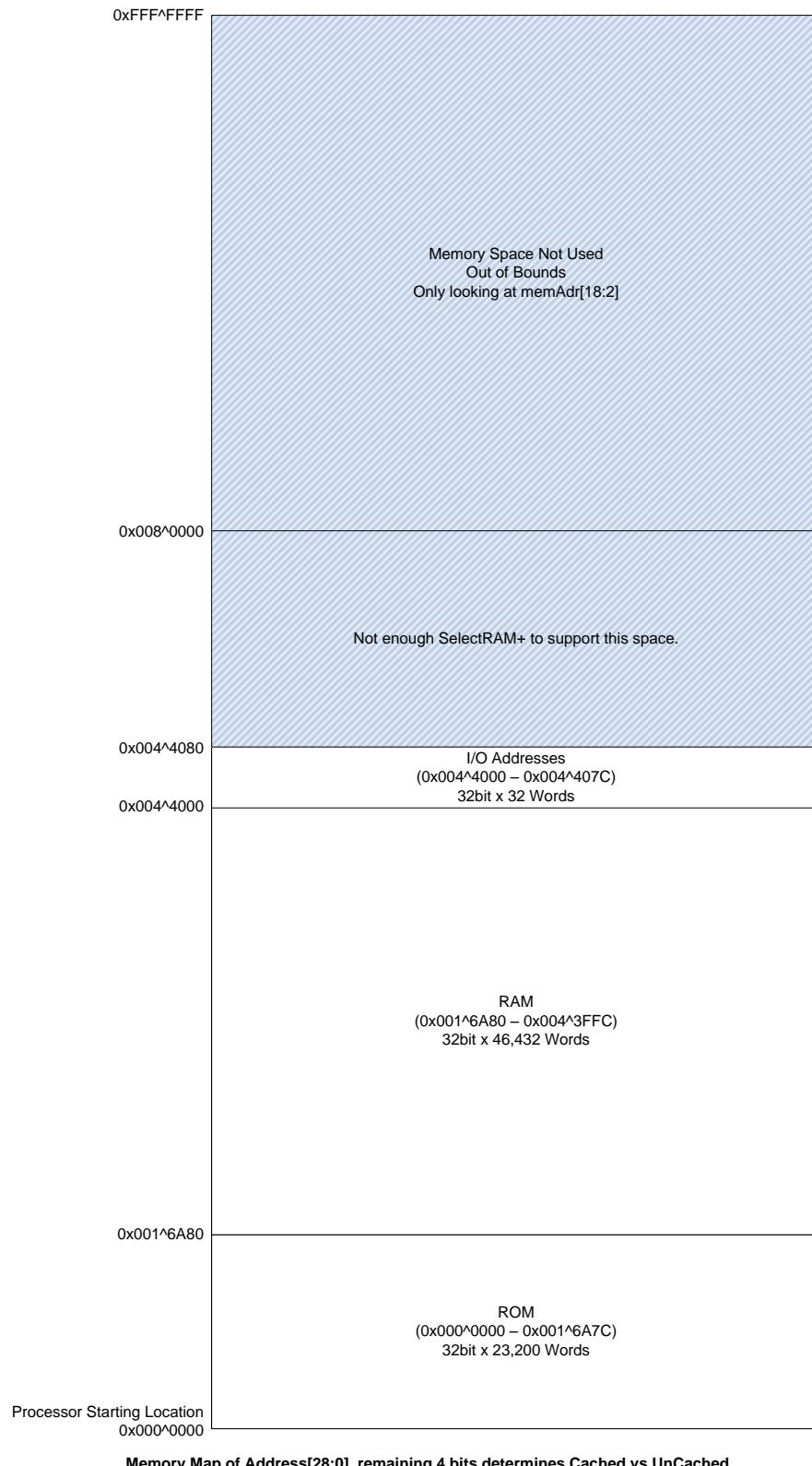


Figure 106. 32-Bit External Memory Map

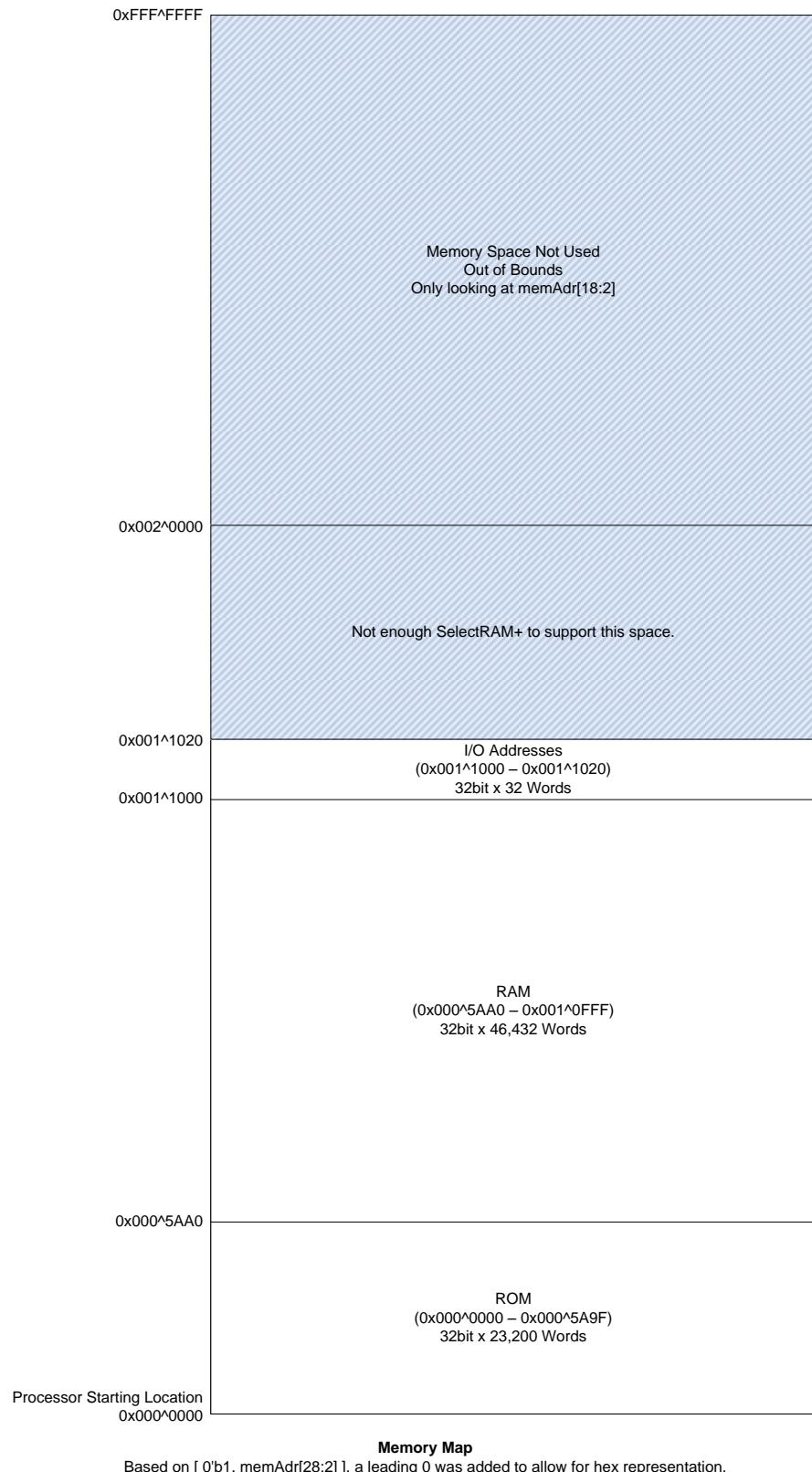


Figure 107. 17-Bit External Memory Map (Internal Use)

19.9. Verilog

19.9.1. External Memory Top (extmem.v)

```
module extmem_top(ph1, ph2, en, rwb, reset, address, byteen, data, dp_data,
bp_data, done, LED_data, LCD_en);

    input ph1;
    input ph2;
    input en;
    input rwb;
    input reset;
    input [18:2] address;
    input [3:0] byteen;
    inout [31:0] data;
        input [3:0] dp_data;
        input [4:0] bp_data;
    output done;
        output [3:0] LED_data;
        output LCD_en;

    wire [1:0] data_select;
    wire [31:0] ROM_data;
    wire [31:0] RAM_data;
    wire [31:0] data_out;
    wire [31:0] data_in;
    wire [3:0] dpsw_en;
    wire [4:0] bpsw_en;
    wire LCD_en;
    wire LCD_en_temp;

    // External Memory Components
    memcon memcon(en, rwb, address, RAM_en, ROM_en, tri_en, LED_en,
dpsw_en, bpsw_en, LCD_en_temp, data_select);
    ROM ROM(ph2, ROM_en, address, ROM_data);
    RAM RAM(ph2, RAM_en, rwb, address, byteen, data_in, RAM_data);

    // Tristate Buffers to protect the data line.
    newtribuf #(32) ROM_data_protect(data_select[0], ROM_data, data);
    newtribuf #(32) RAM_data_protect(data_select[1] & tri_en, RAM_data,
data);

    // Tristate buffers for Dipswitches
    newtribuf #(32) dipswitch0(dpsw_en[0], {31'b0, dp_data[0]}, data);
    newtribuf #(32) dipswitch1(dpsw_en[1], {31'b0, dp_data[1]}, data);
    newtribuf #(32) dipswitch2(dpsw_en[2], {31'b0, dp_data[2]}, data);
    newtribuf #(32) dipswitch3(dpsw_en[3], {31'b0, dp_data[3]}, data);

    // Tristate buffers for Button Presses
    newtribuf #(32) bpswitch_en(bpsw_en[0], {31'b0, bp_data[0]}, data);
    newtribuf #(32) bpswitch_up(bpsw_en[1], {31'b0, bp_data[1]}, data);
    newtribuf #(32) bpswitch_dn(bpsw_en[2], {31'b0, bp_data[2]}, data);
    newtribuf #(32) bpswitch_lf(bpsw_en[3], {31'b0, bp_data[3]}, data);
    newtribuf #(32) bpswitch_rt(bpsw_en[4], {31'b0, bp_data[4]}, data);

    // Flops
```

```

        fopenr_emen #(32) data_in_flop( ph1, reset, en & ~rwb, data,
data_in);
        fopenr_emen #(4) LED_flop( ph2, reset, LED_en, data_in[3:0],
LED_data);

// Using state machines to delay signals
state_machine done_delay(ph1, reset, en, done);
lcd_machine lcd_delay(ph1, reset, LCD_en_temp, LCD_en);

endmodule

module newtribuf #(parameter WIDTH = 32)
    (input en,
     input [WIDTH-1:0] a,
     output [WIDTH-1:0] y);
    wire [WIDTH-1:0] highz;
    assign highz = {WIDTH{1'b1}};
    assign #1 y = en ? a : highz;
endmodule

module fopenr_emen #(parameter WIDTH = 32)
    (      input clk, reset, en,
          input [WIDTH-1:0] d,
          output reg [WIDTH-1:0] q );
    always @ (posedge clk)
        q = reset ? 1'b0 : ( en ? d : q );
endmodule

module state_machine (input clk, reset, in,
                      output out);
    reg [2:0] nextstate, state;

    parameter S0 = 3'b000;
    parameter S1 = 3'b001;
    parameter S2 = 3'b010;
    parameter S3 = 3'b011;
    parameter S4 = 3'b100;

    always @ ( posedge clk)
        state = reset ? S0 : nextstate;

    always @ ( * )
        case(state)
            S0: nextstate <= in ? S1 : S0;
            S1: nextstate <= S2;
            S2: nextstate <= S0;
            S3: nextstate <= S0;
            S4: nextstate <= S0;
            default: nextstate <= S0;
        endcase
    assign out = (state == S2);
endmodule

```

```

endmodule

module lcd_machine (input clk, reset, in,
                     output out);

    reg [2:0] nextstate, state;

    parameter S0 = 3'b000;
    parameter S1 = 3'b001;
    parameter S2 = 3'b010;
    parameter S3 = 3'b011;
    parameter S4 = 3'b100;

    always @ ( posedge clk)
        state = reset ? S0 : nextstate;

    always @ ( * )
        case(state)
            S0: nextstate <= in ? S1 : S0;
            S1: nextstate <= S2;
            S2: nextstate <= S0;
//            S3: nextstate <= S0;
//            S3: nextstate <= S0;
            default: nextstate <= S0;
        endcase

        assign out = (state == S1);

endmodule

```

19.9.2. Memory Controller (memcon.v)

```

module memcon(en, rwb, address, RAM_en, ROM_en, tri_en, LED_en,
              dpsw_en, bpsw_en, LCD_en, data_select);

    // I/O Declarations
    input en;
    input rwb;
    input [16:0] address;
    output reg RAM_en;
    output reg ROM_en;
    output reg [1:0] data_select;
    output reg tri_en;
    output reg LED_en;
    output reg [3:0] dpsw_en;
    output reg [4:0] bpsw_en;
    output reg LCD_en;

    always @(*)
        if(en)
            begin
                if ( address < {1'b0, 16'h5AA0} & rwb == 1) // If it
is within the ROM address space

                    // Does not allow it to be written into
begin
                ROM_en          <= 1'b1;
                RAM_en          <= 1'b0;
                LED_en          <= 1'b0;
                dpsw_en         <= 4'b0;
                bpsw_en         <= 5'b0;
                LCD_en          <= 1'b0;
                data_select <= 2'b01;
                tri_en          <= 1'b1;
end
else if ( address >= {1'b0, 16'h5AA0} &
           address < {1'b1, 16'h1000} )

// If it is within the RAM address space
begin
    ROM_en          <= 1'b0;
    RAM_en          <= 1'b1;
    LED_en          <= 1'b0;
    dpsw_en         <= 4'b0;
    bpsw_en         <= 5'b0;
    LCD_en          <= 1'b0;
    data_select <= 2'b10;
    tri_en          <= rwb ? 1'b1 : 1'b0;
end
else if ( address == {1'b1, 16'h1000} ) // LED
Address
begin
    ROM_en          <= 1'b0;
    RAM_en          <= 1'b0;
    LED_en          <= 1'b1;
    dpsw_en         <= 4'b0;

```

```

        bpsw_en      <= 5'b0;
        LCD_en       <= 1'b0;
        data_select <= 2'b00;
        tri_en       <= 1'b0;
    end
else if ( address == {1'b1, 16'h1001} ) // Dipswtich
address SW0
begin
    ROM_en      <= 1'b0;
    RAM_en      <= 1'b0;
    LED_en      <= 1'b0;
    dpsw_en     <= 4'b0001;
    bpsw_en     <= 5'b0;
    LCD_en      <= 1'b0;
    data_select <= 2'b00;
    tri_en       <= 1'b0;
end
else if ( address == {1'b1, 16'h1002} ) // Dipswtich
address SW1
begin
    ROM_en      <= 1'b0;
    RAM_en      <= 1'b0;
    LED_en      <= 1'b0;
    dpsw_en     <= 4'b0010;
    bpsw_en     <= 5'b0;
    LCD_en      <= 1'b0;
    data_select <= 2'b00;
    tri_en       <= 1'b0;
end
else if ( address == {1'b1, 16'h1003} ) // Dipswtich
address SW2
begin
    ROM_en      <= 1'b0;
    RAM_en      <= 1'b0;
    LED_en      <= 1'b0;
    dpsw_en     <= 4'b0100;
    bpsw_en     <= 5'b0;
    LCD_en      <= 1'b0;
    data_select <= 2'b00;
    tri_en       <= 1'b0;
end
else if ( address == {1'b1, 16'h1004} ) // Dipswtich
address SW3
begin
    ROM_en      <= 1'b0;
    RAM_en      <= 1'b0;
    LED_en      <= 1'b0;
    dpsw_en     <= 4'b1000;
    bpsw_en     <= 5'b0;
    LCD_en      <= 1'b0;
    data_select <= 2'b00;
    tri_en       <= 1'b0;
end
else if ( address == {1'b1, 16'h1005} ) // Button
Address BP_Enter

```

```

begin
    ROM_en      <= 1'b0;
    RAM_en      <= 1'b0;
    LED_en      <= 1'b0;
    dpsw_en     <= 4'b0000;
    bpsw_en     <= 5'b00001;
    LCD_en      <= 1'b0;
    data_select <= 2'b00;
    tri_en      <= 1'b0;
end
else if ( address == {1'b1, 16'h1006} ) // Button
Address BP_Up
begin
    ROM_en      <= 1'b0;
    RAM_en      <= 1'b0;
    LED_en      <= 1'b0;
    dpsw_en     <= 4'b0000;
    bpsw_en     <= 5'b00010;
    LCD_en      <= 1'b0;
    data_select <= 2'b00;
    tri_en      <= 1'b0;
end
else if ( address == {1'b1, 16'h1007} ) // Button
Address BP_Down
begin
    ROM_en      <= 1'b0;
    RAM_en      <= 1'b0;
    LED_en      <= 1'b0;
    dpsw_en     <= 4'b0000;
    bpsw_en     <= 5'b00100;
    LCD_en      <= 1'b0;
    data_select <= 2'b00;
    tri_en      <= 1'b0;
end
else if ( address == {1'b1, 16'h1008} ) // Button
Address BP_Left
begin
    ROM_en      <= 1'b0;
    RAM_en      <= 1'b0;
    LED_en      <= 1'b0;
    dpsw_en     <= 4'b0000;
    bpsw_en     <= 5'b01000;
    LCD_en      <= 1'b0;
    data_select <= 2'b00;
    tri_en      <= 1'b0;
end
else if ( address == {1'b1, 16'h1009} ) // Button
Address BP_Right
begin
    ROM_en      <= 1'b0;
    RAM_en      <= 1'b0;
    LED_en      <= 1'b0;
    dpsw_en     <= 4'b0000;
    bpsw_en     <= 5'b10000;
    LCD_en      <= 1'b0;

```

```

        data_select <= 2'b00;
        tri_en           <= 1'b0;
    end
else if ( address == {1'b1, 16'h100A} ) // LCD_en
begin
    ROM_en           <= 1'b0;
    RAM_en           <= 1'b0;
    LED_en           <= 1'b0;
    dpsw_en          <= 4'b0000;
    bpsw_en          <= 5'b0;
    LCD_en           <= 1'b1;
    data_select <= 2'b00;
    tri_en           <= 1'b0;
end
else
begin
    ROM_en           <= 1'b0;
    RAM_en           <= 1'b0;
    LED_en           <= 1'b0;
    dpsw_en          <= 4'b0;
    bpsw_en          <= 5'b0;
    LCD_en           <= 1'b0;
    data_select <= 2'b0;
    tri_en           <= 1'b0;
    //done_int <= 0;
end
//done <= 1'b0;
end

else // If memory is not enabled, make sure that it is not
enabling anything else.
begin
    ROM_en           <= 1'b0;
    RAM_en           <= 1'b0;
    LED_en           <= 1'b0;
    dpsw_en          <= 4'b0;
    bpsw_en          <= 5'b0;
    LCD_en           <= 1'b0;
    data_select <= 2'b0;
    tri_en           <= 1'b0;
end
endmodule

```

19.9.3. ROM.v Template

```
`timescale 1ns / 1ps
///////////////////////////////
/////
// Company: Harvey Mudd College E158: VLSI Design
// Engineer: Systems Cluster: Howard Chen and Matt McKnett
//
// Design Name: External Memory and IO System
// Module Name: ROM
// Project Name:
// Target Devices: XV2PRO XC2VP30 for use in conjunction with a 32-bit MIPS
Processor
// Description:
//
// Dependencies: extmem.v, memcon.v, RAM.v
///////////////////////////////
/////

module ROM(clock, en, address, instr);
    input clock;
    input en;
    input [16:0] address;
    output reg [31:0] instr;

    always @ (posedge clock)
        if(en)
            begin
                case(address)
                    {1'b0, 16'h0}: instr <= 32'h0bf00002;
                    {1'b0, 16'h1}: instr <= 32'h00000000;
                    {1'b0, 16'h2}: instr <= 32'h0000000d;
                    {1'b0, 16'h3}: instr <= 32'h3c089fc0;
                    {1'b0, 16'h4}: instr <= 32'h35081000;
                    {1'b0, 16'h6}: instr <= 32'h00000000;
                    {1'b0, 16'h7}: instr <= 32'h00000000;
                    {1'b0, 16'h8}: instr <= 32'h00000000;
                    {1'b0, 16'h9}: instr <= 32'h00000000;
                    {1'b0, 16'ha}: instr <= 32'h00000000;
                    .
                    .
                    .
                    default: instr <= 32'h00000000;
                endcase
            end
endmodule
```

19.9.4. RAM (RAM.v)

```
module RAM(clock, en, rwb, address, byte_mask, data_in, data_out);
    input clock;
    input en;
    input rwb;
    input [16:0] address;
    input [3:0] byte_mask;
    input [31:0] data_in;
    output reg [31:0] data_out;

    reg [7:0] RAM1[69631:23200];           // data[7:0]
    reg [7:0] RAM2[69631:23200];           // data[15:8]
    reg [7:0] RAM3[69631:23200];           // data[23:16]
    reg [7:0] RAM4[69631:23200];           // data[31:24]

    integer i;

    initial begin
        for (i = 23200; i <= 69631; i = i +1)
            begin
                RAM1[i] <= 0;
                RAM2[i] <= 0;
                RAM3[i] <= 0;
                RAM4[i] <= 0;
            end
    end

    always @(*( posedge clock ))
        if (en)
            begin
                if (~rwb)
                    begin
                        if (byte_mask[0]) RAM1[address] <= data_in[7:0];
                        if (byte_mask[1]) RAM2[address] <= data_in[15:8];
                        if (byte_mask[2]) RAM3[address] <= data_in[23:16];
                        if (byte_mask[3]) RAM4[address] <= data_in[31:24];
                    end
                data_out <= {RAM4[address], RAM3[address], RAM2[address],
                RAM1[address]};
            end
    endmodule
```

19.10. Test Files

Included is the test file written specifically for the external memory system to be tested. All others can be found in the code repository.

```
/* extmem_test.c
 *
 * Created by Howard Chen, HMC-MIPS Project, VLSI Spring 2007
 * $Author: hnchen $, $Date: 2007-04-18
 *
 * The purpose of this test is to confirm that the external memory works
 * at the corner cases.
 *
 * Systems tested:
 *     - external memory RAM corner cases
 *     - global variables and stack
 */
#include "muddCLib/muddCLib.h"

char test[4];

int main() {

    while(1)
    {

        char* first =      ((char*)0xA0043FFC); // Last address in RAM
        char* second =     ((char*)0xA0016A80); // First address in RAM

        *first = (char)0x1;
        *second = (char)0x2;

        *test = (char)(*first + *second); // Stores value into global variable
        setLED(* (char*)test); // Sets the LEDs

    }

    return 0;
}
```

19.11. Lessons Learned and Experience Gained

A lesson that I learned throughout the process of design is to carefully inspect specifications that are given to you. Even though I have asked multiple people about the signals that are coming into and going out of the memory system, the finer details were not discovered until closer inspection of the other components that are related to the external memory system. What I mean is, before designing, do not assume anything. You have to be extremely clear how each signal works and why they work the way they do or don't.

Another lessoned that I learned was the Xilinx requires a lot of system resources and when working with a large project, you have to be careful of crashes and fatal errors which happens quite a bit. This leads to the easy corruption of project files that are created. Initially, this fact was unknown to us so we were copying project files from computer to computer to avoid having to recreate a new project file which then entails new Package Pin Assignments and the whole thing. But if you recopy the project files enough, they will also get corrupted easily. Therefore, we now only copy the Verilog files and recreate the project files as much as possible. This has two main benefits: it saves space and prevents file corruption.

One final lesson comes with the cluster management. Even though things may seem to be going fine, always expect the unexpected. Initially, we would have our own cluster design reviews to make sure that tasks are going fine, but when things seemed to be getting on track we stopped having them which lead to disorganization and unexpected road blocks. After bringing back our design reviews, it seemed to have made things much easier.

20. Compiler, Benchmarks, and Demo Software

Owner: Matt McKnett

Revision: 4

Date: April 17, 2007

The source for the systems cluster's toolchain can be found on Google code at <http://code.google.com/p/hmc-mips/source> and has been placed in the E158/proj07/compiler directory on Charlie.

20.1. Compiler User Instructions

20.1.1. Introduction

The compiler used for this project is a build of GCC 4.1.1 (using binutils 2.16.1) by Professor James Stine of Oklahoma State University called "yoda_warrior." The compiler was built for Cygwin to run on Windows, and the tarball containing yoda_warrior can be found in the 'proj07/compiler' directory of the E158 folder. It is called 'yoda_warrior.tar.gz'.

20.1.2. Toolchain Setup Instructions

Install Cygwin on your machine:

1. Download cygwin from this site: <http://www.cygwin.com/>.
2. Run 'setup.exe'.
3. click 'next' twice.
4. Set to install to a directory on your main drive (e.g. 'C:\cygwin'), and click next.
5. Set where download files will be placed, and click next twice.
6. Select an ftp mirror near you geographically. I usually use <ftp://mirrors.xmission.com>.
7. Select the packages you want to install. Please include the following packages (by clicking where it says "skip" next to each entry):
 - a. Everything in Base
 - b. Devel: automake, binutils, bison, gcc-core, make, subversion
 - c. Editors: Choose your favorite editor (e.g. vim or emacs)
 - d. Interpreters: python
 - e. Shells: make sure bash (or your favorite shell) will install
8. Click 'next' to start installation.
9. Click 'finish'.

Next, load yoda_warrior onto your new Cygwin installation:

1. Open the Cygwin shell from the Windows start menu.
2. Copy yoda_warrior.tar.gz to your Cygwin home directory. If you installed Cygwin to the default location, this will be at 'C:\cygwin\home\YourUserName'.
3. Create a new directory in your home directory called yoda_warrior. In Cygwin, this is done using the command

```
mkdir yoda_warrior
```

- Having created the directory, unpack the yoda_warrior tarball into the directory by entering it and untarring it using the following commands:

```
cd yoda_warrior
tar xvf ../yoda_warrior.tar.gz
```

Finally, create a local copy of the toolchain scripts and sources in Cygwin:

- Either create the directory ‘hmc-mips/systems’ in your home directory in Cygwin and copy the files from the ‘compiler/toolchain’ folder in the ‘E158/proj07’ class folder, or else go to your Cygwin folder and use subversion to check out the project files from the repository with the following command:

```
svn checkout http://hmc-mips.googlecode.com/svn/trunk/ hmc-mips
```

- Once you have a local copy of the toolchain, test out the compiler to make sure it is working using the command

```
make lightsOut.v
```

If you have opted to build your own compiler, then you will have to modify the file ‘Makefile’ in the ‘hmc-mips/systems/src’ directory so that the variables CC, LD, AS, and DUMP reflect the commands or executable files associated with the compiler, linker, assembler, and objdump programs of your compiler, respectively.

When the build completes successfully, you will see a message from the Boot and Program Verilog script that tells you how many lines were written to the file lightsOut.v; if there was an error, make will notify you and give an error return code.

20.1.3. Writing Programs for the HMC-MIPS

Any C program can be written and compiled for the HMC-MIPS processor. Here are some guidelines for writing and compiling your code, followed by a checklist to follow when writing the code:

- Since yoda_warrior was not built to include any C standard libraries (like glibc or uclibc), many standard library headers like string.h and math.h are unavailable in our toolchain. You will get compiler errors if you attempt to include C standard library methods. I/O to the LCD and some limited string functions have been built into MuddCLib, which is included in the project source, but full basic standard library functionality will either have to be written manually or their functionality will have to be gotten by building GCC for yourself with library files.
- The processor has no floating-point unit (coprocessor 1), and the lack of floating-point libraries in yoda_warrior mean that floating-point numbers in programs for this chip are infeasible.
- String literals will not be recognized when your program is loaded onto the system. When the toolchain creates the Verilog file for FPGA synthesis, and when it creates a dat file for simulation, it uses the objdump tool to disassemble the contents of a program’s

binary, read the machine code (in ASCII) into a .v or .rom file and link in .dat machine code for the boot loader. Because of the nature of the disassembly, the memory locations where string literals are kept do not disassemble, so their data is not recorded in the final dat or Verilog file.

If you need strings in your program, you can work around this problem by using the makeCharArrayFromString.py python script. This script reads strings from standard in and outputs C code that declares a character array. So if your string is “hi!”, the script generates “char string1[3] = {‘h’, ‘i’, ‘!’};” and you can declare the output strings either as global variables outside of functions or inside functions as stack variables. NOTE: this script has its own glitches, so the resulting code may require some minor tweaking to make it perfect. Please see the section on toolchain scripts (p. ???) for more information.

- It is a good idea to look at your ‘myProgram.dump’ file immediately after you compile ‘myProgram’ to a .dat, .dump, .rom, or .v file and make sure that your program’s ‘main’ routine appears as the first symbol in the file (at address 0x9fc0 1000). If it does not, then the scripts that generate .v and .rom files will not link the main routine into the proper location and the program will behave incorrectly.

If you do find that your main routine is not the first symbol in the .dump file, you can work around this by splitting your source file into two C files, one with only your main function and the other with only the other functions. Then, when you invoke the linker in your make file (the line that uses \$(LD)), you can supply both of the object files associated with your newly split sources to the linker. Order matters! List the object file with the main function first. For an example of what this should look like, see ‘dhystone.c,’ ‘dhryFuncs.c,’ and ‘dhystone.mk’ for an example of this workaround in action.

- When the microarchitecture for the HMC-MIPS was created, there were four patented instructions that could not be included in the instruction set for legal reasons. In MIPS assembly, these instructions are ‘lwl’, ‘lwr’, ‘swl’, and ‘swr’, the unaligned load and store instructions. The compiler will still emit these instructions, so it is important to design your code such that the compiler will not use them. Luckily, they are rare. From what we have found, they can show up in cases where an odd struct such as a ‘union’ is used in the C code, or when a string is constructed as described above but the array is not dimensioned large enough to hold all of the characters. These are probably not the only cases when unsupported instructions can crop up, but if the toolchain complains about invalid instructions, these are probably good places to start looking.
- If you #include “muddCLib.h” in your C programs, then you will have access to I/O device methods, clock-based delays, and implementations of strcpy, strcmp, and memset from the C standard library. See the section on MuddCLib (p.230) for more information.
- When you use MuddCLib, you will need to update the muddCLib.h file to reflect the clock frequency that you are using. If you do not, the LCD display instructions may not be spaced enough in time.
- When you create a program for the HMC-MIPS that includes other libraries (for example, if your program uses MuddCLib), then in order to build it properly it is best to create a

makefile for them. Copy ‘template.mk’ to ‘myProgram.mk’ (where “myProgram” is the name of your program’s C file) like this:

```
cp template.mk myProgram.mk
```

You also need to modify the makefile a little bit so that the dependencies lists match those for your program. If you only have one .c file and one .h file, then you can modify the makefile variables **MYPROGRAM** and **MYPROGRAMHEADER** to match the names of those files. If you have more than one .c file to compile your program, you will need to decide on a name for the overall program that will be used for the .out dependency and create dependencies for the .c, .asm, and .o files for each of your C source files. For a good example of this, check out ‘dhystone.mk’. If you are writing a program in assembly, you do not need to include the dependency for .c files.

If you create your own makefile, you will have to force make to include it either by modifying the variable INCLUDEFILES in ‘Makefile’ to reference your makefile or by passing the INCLUDEFILES variable to make on the command prompt like this:

```
make myProgram.v INCLUDEFILES+=myProgram.mk
```

To automatically remove only intermediate files (like .o and .out) use

```
make clean
```

To delete all generated files, including .v and .rom use

```
make clean-all INCLUDEFILES+=myProgram.mk
```

HMC-MIPS programming checklist:

- No references to C standard library files
- No floating point variables (float or double)
- No ASCII string literals
- Generated .dump file has main at the top address
- Included muddCLib/muddCLib.h
- muddCLib.h has updated clock frequency (important for using LCD!)
- Template makefile copied and customized for your program

20.1.4. Compiling Programs for the HMC-MIPS

The process for building simulatable or synthesizable .rom/.v files is somewhat odd due to the nature of the chip and boot loader. All program files are compiled to assembly first, and their instructions are sifted for invalid instructions. After .out binary files have been built, instead of loading the binaries directly onto memory as might be expected, they are instead disassembled to .dump files using objdump, and then the machine code is extracted from the .dump file by a grep command that creates a .dat file – the ASCII representation of the machine

code in hex. Then a python script munges the .dat files for the boot loader and the main program, spitting out either a .rom file (readable by ModelSim for simulating purposes) or a .v file (synthesizable by Xilinx to put onto an FPGA) that represent what amounts to a ROM. The Makefile will also build a .coe file for Coregen, but we have not kept the script up to date because we found that Coregen memories were far too cumbersome to synthesize. At the end of the build process, a named .v file will be generated (e.g. ‘myProgram.v’) and it will be copied to the file ‘ROM.v’. Therefore ROM.v reflects the most recently generated Verilog file.

If you have used the template makefile, then the process for creating your program ROM in Verilog should be simply:

```
make myProgram.v INCLUDEFILES+=myProgram.mk
```

An example illustrating the build process for the HMC-MIPS is shown in Figure 108.



Figure 108. An example illustrating the HMC-MIPS compiler build process

20.1.5. The Boot Loader

The boot loader has been custom written for our system. Its basic job is to invalidate the chip’s cache memory, and to initialize the Status register and global and stack pointers to a usable configuration.

When building a program, you have the option to leave out the boot loader code. If you do this, then your code will be responsible for establishing the stack and global pointers, properly setting up the status register, and making no reads from cached addresses (in the range 0x8000

0000 – 0x9FFF FFFF). Programs that do not set up the stack and global pointers and then try to use stack or global memory will almost certainly break. Using the cache without invalidating it is unsafe, because when the cache powers up its valid bits are in a random state, so bad data may masquerade as good data. It is a good idea to use the boot loader that is included.

To generate Verilog without the boot loader, pass the BOOTLOAD variable to make

```
make myProgram.v BOOTLOAD=false
```

Actually, any value besides “true” will cause the toolchain to compile without the boot loader.

Additionally, there is a DEBUG flag that you can pass to the makefile like this:

```
make myProgram.rom DEBUG=true
```

It only has an effect on .rom output, in which case it adds line numbers at the beginning of each line.

20.1.6. Modifying the Memory Map of Generated ROMs

How .dat files are mapped to regions of ROM is specified by the ‘toolchain_memory_specs.txt’ file. There are three spaces for .dat files to go: The bootstrapper region at the beginning, the exception handler in the middle, and the program at the end. The following parameters can be specified:

- reset_loc: The address of the bootstrapper
- reset_name: The name of the bootstrapper .dat file (usually boot_start.dat)
- except_loc: The address of the exception handler
- except_name: The name of the exception handler .dat file (usually boot_loader.dat)
- program_loc: The address of the first program instruction
- mem_size: The last address that can be used for ROM
- Verilog_template: Specifies the file that serves as the Verilog template (usually Verilog_template.txt)

Here is an example of a memory specification:

```
reset_loc:0x00000000
reset_name:boot_start.dat
except_loc:0x00000100
except_name:boot_loader.dat
program_loc:0x00000200
mem_size:0x00016A7C
Verilog_template:Verilog_template.txt
```

Note that you cannot put spaces between a parameter and its value.

20.1.7. Modifying Program Entry Location

Three files have to be modified in order to change where a program is placed in memory. As it stands, programs are placed at 0x9fc00200 – to change this, you must modify ‘toolchain_memory_specs.txt’ so that the program_loc entry reflects the new program location; ‘Makefile’ so that PROG_LOC reflects the new program location; and ‘boot_start.asm’ so that the jal instruction that jumps to the first program location knows what instruction to go to.

20.1.8. Simulating and Synthesizing Compiled Programs

Synthesizing the compiled programs is easy. You need only replace the ROM.v file that currently exists in the external memory system source files with the generated ROM.v file. Then, you can re-synthesize the external memory system onto the external memory FPGA.

The simulation process is fairly simple. In Xilinx, you should be able to switch contexts from Synthesis/Implementation to Behavioral Modeling and run the Verilog test bench for simulating in ModelSim.

Note: We use ModelSim SE for our simulation. A version of ModelSim comes with Xilinx called ModelSim XE, and it is not recommended that you attempt to simulate the chip on the Xilinx Edition because it does not have the same optimization ability as ModelSim SE and it does not use system resources as well. Using XE is a painful endeavor.

20.2. Using Supplied Programs

20.2.1. Test Programs

There are four test programs given along with the toolchain:

- test_simple.c
- test_buttons.c
- test_leds.c
- test_lcd.c

All of the tests use the muddCLib functions for device (LCD, LED, DIP switch, and button) I/O. The simple test loops infinitely, counting up on a stack variable and displaying the value on the LEDs with a delay after the display.

The button test lights up an LED if the left, up, or down buttons are pressed, and toggles an LED on or off if the right button is pressed (but only if all of the other buttons are not pressed when it is pressed.)

The LED test stores a global variable where the values of DIP switches are stored after they are read. Then, the values are printed on the LEDs. The effect is that the DIP switches on the Virtex board turn on and off corresponding LEDs.

The LCD test initializes the LCD screen, prints two stars in the first and second rows on the LCD, and then displays the message “Hello, World!” on the first line and “HMC VLSI 07” on the second line in two ways: first by manually displaying each character, then by using the muddCLib ‘dispMessage()’ function.

Running the test programs is like running any other program. Compile the source to a Verilog ROM. For example:

```
make test_lcd.v
```

Then replace the ROM file in the external memory's Verilog source with the freshly generated ROM. Synthesize and program the board, and it should begin running.

20.2.2. Demo Program

The demo program that was created for the HMC-MIPS is the game "Lights Out!" The object of the game is to turn all light (blank) squares into dark (filled) squares. Here is how the game works:

1. The game prints a welcome message and waits for the user to press a button on the Virtex board. Note that the "enter" (middle) button is mapped to reset, so nothing will happen if you press that button.
2. When a button is pressed, the game begins. The "board" is the top row of either completely blank or completely filled characters on the LCD display. Filled characters (the darker ones) mean that light is off.

A carat is displayed in the second row to show you what location you are currently at. To move the carat, press the "left" and "right" buttons on the Virtex board. In order to turn lights on and off, you press the "up" button.

The trick in lights out is that whenever you toggle one light on or off, you also toggle the lights to its left and right on or off. The object of the game is to turn all of the lights off, so toggle lights and their neighbors until you finally get them all off. NOTE: Some starting configurations of lights out are unwinnable (about one out of six)¹ – you cannot turn off all the lights. If you want to get a new board in the middle of the game, press the "down" button.

3. When the board is completely blank, the program displays a message that informs you of the win and asks if you would like to play again. If you want to play again, press the "up", "right", or "left" buttons. If you press "down", then the message "Game Over" is displayed and the program stops until reset. As always, reset is the "enter" button.

To build the demo program, run the following command:

```
make lightsOut.v
```

20.2.3. Dhrystone Benchmark

In order to determine how the HMC-MIPS chip measures up to other MIPS processors, we customized the popular Dhrystone benchmark (taken from the MIPS-SDE compiler package examples, which had already been customized for MIPS) to run with our own tools. Since we do

¹ Dodds, Zach. "Intro Programming with Java: Homework 8." Harvey Mudd College Computer Science Website, CS 5 Fall 2004. Accessed 4/16/07 <http://www.cs.hmc.edu/courses/2003/fall/cs5/week_08/homework.html>.

not have a way of measuring time in hardware and we do not have floating point support, calculating the running time of Dhrystone in software is infeasible. Therefore, we have implemented Dhrystone with the following data-taking procedure:

1. When dhrystone starts running, you will see the four LEDs counting down to starting time. Just before the tests begin, all four LEDs will flash quickly and turn off – this is when you should begin timing the test.
2. While the test runs, the LEDs remain off. If the LEDs remain off for a very long time, this is probably an indication that NUMBER_OF_RUNS is set too high.
3. When the test has completed, all four LEDs will flash on and off slowly ten times. After the test is over, Dhrystone will check the results of its calculations to make sure that all of the results turned out correct. If they did, then the LEDs will light up in a “fireworks” pattern ($0100 \rightarrow 1010 \rightarrow 0001 \rightarrow 0000 \rightarrow 0010 \rightarrow 0101 \rightarrow 1000 \rightarrow 0000$) and repeat infinitely. If, however, the results were not correct, then all four LEDs will flash rapidly (about twice as fast as the “end of test” flashing) to indicate the failure.

To set the number of Dhrystone runs that the program should perform, open the file ‘dhrystone.c’ and change the definition of “NUMBER_OF_RUNS” to whatever you would like. Keep in mind that the execution time of Dhrystone ought to remain above two seconds. Our tests on the FPGA put it around 20 seconds with 10,000 runs at 2 MHz, and close to 3 seconds with the same number of runs at 18.5 MHz, and this may change when Dhrystone is run on the fabricated hardware.

In order for Dhrystone to work properly with our toolchain, it was necessary to make a modification that bends the rules of its use: originally, the benchmark was split into two files in order to tax the compiler more, but we found that if the compiler did not make ‘main’ the first symbol entry in memory, the toolchain’s assumption that the program starts at a specific location was broken. It was therefore necessary to move some functions into the source file that contains the rest of the C functions such that one file contains only the ‘main’ function and the other contains only the helper functions that it calls. This might constitute a violation of the rules at their strictest, but it does not violate their spirit.²

It would be possible to implement the timing for Dhrystone in hardware. In muddCLib, we have supplied a function to get the cycle count from a memory address. This leaves room, in the future, for making a cycle counter in the external memory system.

Using a Java stopwatch program from <http://www.pccl.demon.co.uk/java/stopwatch.html>, we timed the execution time of Dhrystone running on a single FPGA for 15 runs, with the number of cycles set to 10,000 and a clock speed of 2 MHz. Our results are shown in Figure 109. The single FPGA at 2 MHz ran at 0.13 DMIPS/MHz.

We report DMIPS per MHz as an indication of processor performance. A DMIPS is a measurement exclusive to the Dhrystone benchmark and is designed to generalize the concept of a MIPS (Million Instructions per Second – not to be confused with the MIPS Technologies,

² Weiss, Alan R. “Dhrystone Benchmark: History, Analysis, ‘Scores,’ and Recommendations White Paper”. Nov. 1, 2002, ECL, LLC. Accessed 4/19/07 <<http://www.synchromeshcomputing.com/pdf/dhrystoneWhitePaper.pdf>>.

where our chip's architecture comes from). DMIPS, unlike MIPS, are comparable across instruction set architectures (ISAs), and are therefore preferable for comparing processors with different ISAs. DMIPS is a reported Dhystone/second score divided by that score for a VAX 11/780, a purportedly Million-Instruction-per-Second machine, which scores 1757 Dhystones/second.³ Reporting the figure as DMIPS/MHz gives us a way to compare our chip to any chip regardless of clock speed.

In addition to running Dhystone on a single FPGA at 2 MHz, we also ran 15 runs on our Dual-FPGA chip/board emulation setup at 8.5 MHz. Those results are shown in Figure 110. The dual FPGA reported 0.15 DMIPS/MHz.

Run #	time (sec)	comments	Number of Cycles:	Clock speed (MHz):
1	21.187	Sources of error probably include my reaction time, and my leading/lagging the start time.	10000	2
2	20.938			
3	21.313			
4	21.11			
5	21.094			
6	21.281			
7	21.032			
8	21.14			
9	21.047			
10	21.079			
11	21.11			
12	21.078			
13	21.234			
14	21.078			
15	21.109			
mean	21.122		μs / cycle	Dhystones / sec Vax score (DMIPS)
stdev	0.0976137		2112.20	473.44 0.27
				DMIPS/MHz 0.13

Figure 109 -- Sample Dhystone runs at 2 MHz on a single FPGA. These timing data were gathered using a Java stopwatch program and the LED signals during Dhystone's execution.

³ Glover, Paul. "Running the Dhystone 2.1 Benchmark on a Virtex-II Pro PowerPC Processor." July 11, 2005, Xilinx. Accessed 4/24/07 <<http://direct.xilinx.com/bvdocs/appnotes/xapp507.pdf>>.

Run #	time (sec)	comments	Number of Cycles:	Clock speed (MHz):
1	4.5	Sources of error probably include my reaction time, and my leading/lagging the start time.	10000	8.5
2	4.609			
3	4.609			
4	4.407			
5	4.453			
6	4.625			
7	4.64			
8	4.672			
9	4.5			
10	4.609			
11	4.609			
12	4.454			
13	4.469			
14	4.5			
15	4.547			
mean	4.5468667		μs / cycle	Dhrystones / sec Vax score (DMIPS)
stdev	0.0826048		454.69	2199.32 1.25
				DMIPS/MHz 0.15

Figure 110 -- Sample Dhrystone runs at 8.5 MHz on the dual FPGA. These timing data were gathered as in Figure 109.

The results shown in Figure 109 and Figure 110 are for demonstration only. They do not reflect the performance of the actual chip, since they were taken while simulating the chip on an FPGA.

To calculate the values for microseconds per cycle, Dhrystones per second, and the Vax score, we use the following formulae:

$$\text{Microseconds per cycle} = \frac{\text{Mean run time} \times 10^6}{\text{Number of Cycles}} \quad (1)$$

$$\text{Dhrystones per second} = \frac{\text{Number of Cycles}}{\text{Mean run time}} \quad (2)$$

$$\text{DMIPS} = \text{Vax score} = \frac{\text{Microseconds per cycle}}{1757.0} \quad (3)$$

These equations are taken from the calculations done by the original dhrystone source code, which was written for MIPS processors that had floating point units. Because our processor does not have an FP unit and cannot take timing data, we have simply applied these equations in an Excel spreadsheet.

Table 15 is a chart comparing various MIPS R2000 and R3000 processors to our HMC-MIPS processor.

Processor Name	Vax Score (DMIPS)	Clock Speed (MHz)	DMIPS/MHz
HMC-MIPS	1.25	8.5	0.147058824
DECstation 2100 R2000	11.193	12	0.93275
SGI Personal Iris 4D/20 R2000	9.812	12.5	0.78496

SGI Personal Iris 4D/20 R2000	9.799	12.5	0.78392
SGI PowerSeries 4D/420VGX R3000	33.224	40	0.8306
SGI Personal Iris 4D/35S R3K	32.954	36	0.915388889
DEC Personal DECstation 5000/33 R3000	29.821	33	0.903666667
SGI Indigo R3000	29.694	33	0.899818182
DECstation 5000/125 R3000	23.305	25	0.9322
SGI PowerSeries 4D/240S R3000	22.12	25	0.8848
DECstation 5000/200 R3000	20.459	25	0.81836
Sumistation S-P300 R3000	18.884	25	0.75536
SGI Personal Iris 4D25TG R3000	15.799	20	0.78995

Source: <http://sites.inka.de/pcde/dbp/dhrystone.html>

Table 15 -- Comparison of various R2000 and R3000 processors to the HMC-MIPS.

20.3. Using Supplied Libraries

20.3.1. muddCLib

The muddCLib library contains functions and definitions that are useful for dealing with the hardware attached to the Xilinx board. It also defines strcpy, strcmp, and memset, which are from the C standard library and are necessary to run the Dhrystone benchmarks.

Include muddCLib in your programs with:

```
#include "muddCLib/muddCLib.h"
```

Here is how you use the hardware when you have included muddCLib:

- LEDs can be set using the function ‘setLED(char value)’. The lower four bits of ‘value’ are displayed on the LED. So, for example, ‘setLED(0x9)’ would turn on LED0 and LED3 and turn off LED1 and LED2. ‘setLED(0xF)’ would turn on all of the LEDs, and ‘setLED(0x0)’ would turn them all off.
- DIP switches and buttons are read using the method ‘readSwitch(char *switch)’. You can give this function any of the following macros: SWITCH0, SWITCH1, SWITCH2, SWITCH3, BUTTON_UP, BUTTON_DOWN, BUTTON_LEFT, BUTTON_RIGHT, BUTTON_ENTER. The function will return BUTTON_PRESSED or BUTTON_RELEASED for buttons, and SWITCH_ON or SWITCH_OFF for switches. If you are using a char* variable to store, for example, what button was last pressed, a macro NOSWITCH is defined as a null value for convenience.
- The LCD display has to be initialized using initLCD() before anything can be written to it. Once that is done, you can show characters using ‘dispChar(char character)’ or ‘dispMessage(char* str1, char* str2)’. Example:

```
initLCD();
dispChar('a');
dispMessage(myString, anotherString);
```

Be careful to remember that string literals cannot currently be used in our toolchain, so it would not work to write:

```
dispMessage("Hello, World!", "Nice day?");
```

You can move to any location on the LCD screen by issuing the move command. Valid move positions are 0x0 through 0x13 (the first line) and 0x40 through 0x53 (the second line). A move command looks like this:

```
move(0x40);
```

which would result in the cursor moving to the first character on the second line.

Additionally, you can send instructions to the LCD screen using the ‘sendInst(unsigned char instruction)’ method. This method is exactly the same as the ‘dispChar’ method except that it includes delays that wait the necessary 1.5ms for a L_clear or L_moveHome command to finish. A list of macros for commands can be found in the LCD MANIPULATION METHODS section in muddLibC.h. Here is an example of the sendInst method:

```
sendInst(L_clear);
sendInst(L_disp | L_curs);
sendInst(L_moveRight);
```

These commands cause the LCD screen to clear, then turns off the blinking cursor and moves the cursor one square to the right.

There is also a ‘checkLoc’ method that is supposed to return a valid location on the screen given any number. The ‘checkLoc’ method has not been tested for validity, but remains for future work.

20.3.2. mtRand

The file mtRand is an implementation of the Mersenne Twister random number generator based entirely off of pseudo code found in the Wikipedia article on this algorithm⁴. It is used by the Lights Out! demo program to generate a random board (based on the number of times it has polled buttons for input) each time the game is played.

The random number generator works by taking a seed, running the algorithm, and generating an array of 624 pseudorandom numbers. Initialize the generator like this:

⁴ "Mersenne twister." Wikipedia, The Free Encyclopedia. 17 Apr 2007, 04:47 UTC. Wikimedia Foundation, Inc. 17 Apr 2007 <http://en.wikipedia.org/w/index.php?title=Mersenne_twister&oldid=123450074>.

```

initializeGenerator(seed);
generateNumbers();

```

and then access a random number using index 0 to 623:

```

int myVariable = extractNumber(42);

```

20.4. Listing of Compiler Files

20.4.1. Source Files

Below is a listing of the source files and a description.

Source File Name	Notes
boot_loader.asm	The assembly code that goes in the exception region and does the heavy lifting of the boot loading process. It invalidates the instruction and data caches, and sets up the status register and \$sp and \$gp.
boot_start.asm	The initial part of the bootstrapper. This performs a jump to make sure everything is working, and then causes an exception, executing the boot loader. When that returns, this jumps to the program. If that returns, it loops infinitely to avoid undefined behavior.
cache_test_000.asm	A test that was once used by the cache team to help them check that the ca
dhry.h	This header file defines the functions for Dhrystone and contains a long narrative regarding the function of Dhrystone and distribution of instructions.
dhryFuncs.c	Contains all of the functions used by Dhrystone except for ‘main()’. This separation was necessary so that the main function appeared first in the program space on ROM.
dhrystone.c	Contains the ‘main’ function of the Dhrystone program and all variable declarations. If you need to modify the number of Dhrystone runs that the code performs, do it here.
lightsOut.h	The header file for Lights Out!
lightsOut.c	The Lights Out! demo program.
muddCLib/muddCLib.h	The header file defining the functions and macros used in muddCLib
muddCLib/muddCLib.c	The source file for our library of functions that operate the I/O devices and supply any necessary C standard library calls that are not available for want of compiler libraries.
muddCLib/mtRand.h	Defines the mtRand functions
muddCLib/mtRand.c	An implementation of the Mersenne Twist pseudorandom number generator
mips_corner_test.c	This file is currently incomplete, but was intended to be a test bench for running the corner-case-checking assembly that was written by the Microarchitecture team.
test_buttons.c	This program is designed to test button functionality by lighting up LEDs corresponding to button presses.
test_lcd.c	This program is designed to test the LCD screen by writing “Hello, World!”
test_leds.c	This program is designed to test the LEDs on the board by lighting them up according to what DIP switches are switched on and off.
test_simple.c	This program tests memory I/O and LEDs by counting up on a variable and

Source File Name	Notes
	displaying its value in binary on the LEDs.

20.4.2. Makefiles

In addition to these source files, every program that uses muddCLib has a corresponding makefile (.mk) that automates the building of that program. The following are all the makefiles:

- boot.mk
- dhryston.mk
- lightsOut.mk
- mips_corner_test.mk
- template.mk
- test_buttons.mk
- test_lcd.mk
- test_led.mk
- test_simple.mk

20.4.3. Text Files

Two text files are important for the toolchain to work. They are:

- toolchain_memory_specs.txt
- Verilog_template.txt

The second of these is a mostly empty Verilog file that serves as a template for the script that generates Verilog ROMs. The first is used by the scripts to characterize how the ROM should be created. For usage of this file, see “Modifying the Memory Map of Generated ROMs.”

20.4.4. Toolchain Scripts

Below is a listing of the toolchain scripts and their purpose.

Script Name	Notes
generateVerilog.py	This is responsible for taking the ‘boot_start.dat,’ ‘boot_loader.dat,’ and your program’s .dat file and dumping them into their proper memory locations in a Verilog case statement. Essentially, this script makes a Verilog file that pretends to be a ROM. Appropriately, the makefile outputs ROM.v after this script is invoked.
generateROM.py	This script generates a memory image in ASCII that looks essentially the same as a .dat file, but in this case the script links in the boot_start.dat and boot_loader.dat code. It outputs a file full of machine code (in ASCII) where the first line represents address 0x0 and each line after is one word address higher.
checkInstructions.py	This script sifts through assembly code looking for instructions that are in its list of invalid instructions. Its purpose is to find ‘lwl’, ‘lwr’, ‘swl’, and ‘swr’ functions, and any obvious floating point instructions. The mentioned

	instructions are patented unaligned loads and stores, and as such could not be implemented on our processor. For sake of time, we chose to look for these bad instructions with a script instead of attempting to patch yoda_warrior.
coeBootAndProgram.py	This script does the same thing as generateROM, only it formats the output for reading by Coregen. We did not keep this file up to date because simulating memory in Coregen was cumbersome and slow.
makeCharArrayFromString.py	This script outputs C code that represents a constant-initialized char array based on string literal inputs. It has the following issues: The output strings are not null-terminated, but space for '\0' is given in the array size parameter. The output gives "" instead of \" because of how Python works.

20.5. Printouts of Source Files

20.5.1. Boot Loader

```
# boot_loader.asm
# Created 2/27/07 by Matt McKnott
# Harvey Mudd College, CMOS VLSI Design Spring '07, MIPS project
#
# This is the segment of the boot loader that will do the actual heavy
lifting.
# It must be implemented as exception code so that interrupts will be
disabled
# while it is running.
#
# This boot loader is in no way general purpose!! It is specific to the
# idiosyncrasies of the Harvey Mudd MIPS implementation from Spring 2007.
# It has been written to conform to the special situations and
unimplemented
# features of that chip and that chip only.

# Don't let the assembler reorder or fill branch delay slots.
.set noreorder

# Step 1: Set the Status Register to something that makes sense
# Bits to set:
#   29 (CU1) *not implemented*
#   22 (BEV) = 0 for now, even though we haven't initialized the cache yet
#             (according to See MIPS run p. 76, this is desirable)
#   20 (PE) *not implemented*
#   18 (PZ) *not implemented*
#   17 (SwC) = 1 start with the I-cache
#   16 (IsC) *Cache isolation has not been implemented, but the bit is
settable
#           in the status register.*
#   8-15 (IM) = 1 for each, enabling all interrupts.
#   5 (KUo) = 1 so that we don't rfe into a bad user mode
#   4 (IEo) = 0 for similar reasons
#   3 (KUp) = 1
#   2 (IEp) = 0
#   1 (KUc) = 1 because we must always run our CPU in kernel mode.
## 0 (IEc) *can only be set on interrupt*
#
# SR words: OR with 0x0002FF2A, AND with 0xFF4FFFEB

# Instructions for setting SR (register $12 in cp0) on See MIPS Run p. 105

.text

nop
```

```

.globl      reset
.ent   reset

reset:
    mfc0  $8, $12
set_srl1:
    nop
    nop
    lui   $9, 0x0002      # set the specified bits
    or    $8, $8, $9
    ori   $8, $8, 0xFF2A
    lui   $9, 0xFF4F      # clear the specified bits
    ori    $9, $9, 0xFFEB
    and   $8, $8, $9
set_sr2:
    mtc0  $8, $12
    nop
    nop

# Step 2: Invalidate the cache
# With the I-cache in isolation, we can find out the size of the cache,
and
# then write that many invalid bits.

# Now that we know the cache ignores the isolate bit, we must
# write bytes to 0x8000 8000 so that we do not clobber what is in
# the ROM region and so that we don't write into the device I/O.
# This means we only support up to 32K cache sizes. (Our cache is
# 512 B).

# Originally, my intention was to generalize the bootloader for any
# cache size, but it is far easier to assume the 512 B cache that
# is on our HMC-MIPS chip.

    addi  $9, $0, 128      # The cache size is 128 words.
    li    $10, 0x400 # Load $10 with an address above the inst's
inval_i_loop:
    sb    $0, 0($10) # Invalidate the word at address $10
    addi $10, $10, 4 # Move to next word address
    addi $9, $9, -1 # Use cache size to count down loop iterations
    bnez $9, inval_i_loop
                                # Keep writing bytes until we run out of space.
    nop

# Now that we have invalidated the I-cache, we move on to the D-cache
swap_i_for_d:
    mfc0  $8, $12
    nop
    nop
    lui   $9, 0xFFFFD
    ori   $9, $9, 0xFFFF

```

```

and $8, $8, $9      # clear the SwC bit
mtc0 $8, $12
nop
nop

# Having swapped the caches, we can repeat the same operation on the D-
cache
# that we did on the I-cache.

addi $9, $0, 128 # The cache size is again 128 words
li   $10, 0x400 # Load $10 with the address 0x400
inval_d_loop:
    sb   $0, 0($10) # Invalidate the word at address $10
    addi $10, $10, 4 # Move to next word address
    addi $9, $9, -1 # Use cache size to count down loop iterations
    bnez $9, inval_d_loop
                    # Keep writing bytes until we run out of space.
    nop

# Step 3: Initialize $sp ($29) and $gp ($28) and reset $12 to proper
values.
    lui   $29, 0x8004
    ori   $29, $29, 0x3FFC # We want the stack pointer at 0x80043FFC

    lui   $28, 0x8001
    ori   $28, $28, 0xEA80 # We want the global pointer at 0x8001EA80

# Set register bits:
# 22 (BEV) = 0 now that we have initialized the cache.
# 16 (ISC) = 0 to disable isolation
## 0 (IEC) = 1 now that we are done with the cache we can re-enable
interrupts

    mfc0 $8, $12
    nop
    nop
    ori   $8, $8, 0x1 # set IEC
    lui   $9, 0xFFBE
    ori   $9, $9, 0xFFFF
    and   $8, $8, $9   # clear BEV and ISC.
    mtc0 $8, $12
    nop
    nop

# Step 4: Return from the exception handler, where this code should
reside.
    mfc0 $8, $14      # Get EPC (cp0 reg. 14)
    nop
    nop
    addi $8, $8, 4    # We don't want to execute the break instruction
    jr   $8           # Jump back to the instruction that called this
code.

```

```
rfe          # Don't forget to return from the exception in the  
BD slot.
```

20.5.2. Bootstrapper

```
# Harvey Mudd College VLSI MIPS Project  
# Matt McKnett  
# Spring, 2007  
#  
# Bootloader bootstrap code  
#  
# This code is step 1 in the bootloading process. It is designed to jump  
# to the main boot code that is stored in ROM. This keeps the bootloader  
code  
# from running into exception entry points and gives us a simple test of  
CPU  
# function (according to See MIPS Run page 110).  
  
# We don't want the assembler to reorder or fill branch delay slots.  
.set noreorder  
.text  
  
reset:  
    j      load  
    nop  
  
load:  
    # Boot-loading code must be placed in exception-handling. We'll use  
the  
    # breakpoint features to cause an exception to get there. The  
exception  
    # handler appears to start at 0xBFC00100 on our chip, which is  
contrary to  
    # See MIPS Run.  
    break 0x0  
  
    # Once the cache setup is done, jump to the first instruction (in  
the  
    # uncached region). This ought to be at 0x9FC00200  
    lui    $8, 0x9FC0  
    ori    $8, $8, 0x0200  
    jalr   $8  
    nop  
final:  
    j      final  
    nop  
  
# Originally, we were going to jump to a location to prove that the  
processor is  
# configured enough to handle it, but it is probably best to simply jump  
to the  
# next instruction (which will be the loader) as above.
```

```

#boot:      lui $8, 0xBFC0          # Our main bootloader is located at
0xBFC00E00
#      ori $8, $8, 0x0E00
#      jr  $8
#      nop

#      b      reset      # For now, we will try getting the linker
#                          # to use the boot_loader tag __reset

```

20.5.3. Dhrystone Header

```

/*
*****Benchmark Program*****
*
*          "DHRYSTONE" Benchmark Program
*          -----
* This is a modified version of the Dhrystone benchmark modified
* specifically for the Harvey Mudd College HMC-MIPS project, CMOS VLSI
* Design, Spring 2007.
* Major modifications began Apr 10, 2007 by Matt McKnett
*
* Changes from the original are marked with a "CHANGED" comment in code.
*
* $Author: whiterook2004 $, $Date: 2007-04-15 15:47:26 -0700 (Sun, 15
Apr 2007) $ -- $Revision: 569 $
*
*
*
* Version:      C, Version 2.1
*
* File:         dhry.h (part 1 of 3)
*
* Date:         May 25, 1988
*
* Author:        Reinhold P. Weicker

```

[COMMENTS OMITTED FOR BREVITY – SEE SOURCE CODE FOR DETAILS]

```

*****Source Code*****  

*/

```

```

#ifndef DHRY_INCLUDED
#define DHRY_INCLUDED 1

/* Compiler and system dependent definitions: */
#define Mic_secs_Per_Second    1000000.0
                           /* Berkeley UNIX C returns process times in seconds/HZ */

#endif  NOSTRUCTASSIGN
#define structassign(d, s)      memcpy(&(d), &(s), sizeof(d))
#endif

```

```

#define structassign(d, s)      d = s
#endif

#ifndef NOENUM
#define Ident_1 0
#define Ident_2 1
#define Ident_3 2
#define Ident_4 3
#define Ident_5 4
typedef int Enumeration;
#else
typedef enum {Ident_1, Ident_2, Ident_3, Ident_4, Ident_5}
    Enumeration;
#endif
/* for boolean and enumeration types in Ada, Pascal */

/* General definitions: */

/* CHANGE: I'll write strcpy and strcmp myself.
#include <stdio.h>
        //for strcpy, strcmp
*/

#define Null 0
        /* Value of a Null pointer */
#define true 1
#define false 0

typedef int One_Thirty;
typedef int One_Fifty;
typedef char Capital_Letter;
typedef int Boolean;
typedef char Str_30 [31];
typedef int Arr_1_Dim [50];
typedef int Arr_2_Dim [50] [50];

typedef struct record
{
    struct record *Ptr_Comp;
    Enumeration Discr;

    struct {
        Enumeration Enum_Comp;
        int Int_Comp;
        char Str_Comp [31];
    } variant;

    /* CHANGE:
       This union is screwing up our output because
       it makes the compiler emit invalid instructions.
       We can get rid of it because it only uses the
       var_1 type of struct anyway.
       union {

```

```

        struct {
            Enumeration Enum_Comp;
            int         Int_Comp;
            char        Str_Comp [31];
        } var_1;
    struct {
        Enumeration E_Comp_2;
        char        Str_2_Comp [31];
    } var_2;
    struct {
        char        Ch_1_Comp;
        char        Ch_2_Comp;
    } var_3;
} variant;

/*
} Rec_Type;

typedef Rec_Type *Rec_Pointer;

/* DHRYSTONE FUNCTION DECLARATIONS */
/* "Back in my day, there were no stinkin' declarations!" */

void Proc_1 (Rec_Pointer Ptr_Val_Par);
void Proc_2 (One_Fifty *Int_Par_Ref);
void Proc_3 (Rec_Pointer *Ptr_Ref_Par);
void Proc_4 (void);
void Proc_5 (void);
void Proc_6 (Enumeration Enum_Val_Par, Enumeration *Enum_Ref_Par);
void Proc_7 (One_Fifty Int_1_Par_Val, One_Fifty Int_2_Par_Val, One_Fifty
*Int_Par_Ref);
void Proc_8 (Arr_1_Dim Arr_1_Par_Ref, Arr_2_Dim Arr_2_Par_Ref, int
Int_1_Par_Val, int Int_2_Par_Val);
Enumeration Func_1 (Capital_Letter Ch_1_Par_Val, Capital_Letter
Ch_2_Par_Val);
Boolean Func_2 (Str_30 Str_1_Par_Ref, Str_30 Str_2_Par_Ref);
Boolean Func_3 (Enumeration Enum_Par_Val);

#endif NOSTRUCTASSIGN
    memcpy (register char *d, register char *s, register int l);
#endif

#endif // DHRY_INCLDED

```

20.5.4. Dhystone

```

/*
*****"DHRYSTONE" Benchmark Program
-----
```

```

* This is a modified version of the Dhrystone benchmark modified
* specifically for the Harvey Mudd College HMC-MIPS project, CMOS VLSI
* Design, Spring 2007.
* Major modifications began Apr 10, 2007 by Matt McKnott
*
* Changes from the original are marked with a "CHANGED" comment in code.
*
* $Author: whiterook2004 $, $Date: 2007-04-17 02:00:22 -0700 (Tue, 17
Apr 2007) $ -- $Revision: 594 $
*
*
* Version:      C, Version 2.1
*
* File:         dhry_1.c (part 2 of 3)
*
* Date:         May 25, 1988
*
* Author:        Reinhold P. Weicker
*
***** */
*/
/* CHANGE:
   These includes were originally used, but cannot happen with
Yoda_warrior */
/*#include <stdio.h>
#include <stdlib.h>
#include <string.h> */
#include "dhry.h"
#include "muddCLib/muddCLib.h"

#define NUMBER_OF_RUNS 10000

/* Global Variables: */

Rec_Type      Rec_Glob,
               Next_Rec_Glob;
Rec_Pointer   Ptr_Glob,
               Next_Ptr_Glob;
int           Int_Glob;
Boolean       Bool_Glob;
char          Ch_1_Glob,
               Ch_2_Glob;
int           Arr_1_Glob [50];
int           Arr_2_Glob [50] [50];

/* CHANGE: No longer need this string.
   char Reg_Define[] = "Register option selected."; */

/* CHANGE:
   You go away because you declare the function wrong!!
Enumeration   Func_1 ();
//forward declaration necessary since Enumeration may not simply be int

```

```

*/
/* CHANGE:
   I don't know where it's supposed to get ROPT, but I don't
   see it anywhere, so it's going away.
#ifndef ROPT
#define REG
    // REG becomes defined as empty
    // i.e. no register variables
#else
#define REG register
#endif
*/
#define REG      // Make REG empty.

/* CHANGE:
   We'll be measuring our timing by hand!

// variables for time measurement:

#define Too_Small_Time 2
    // Measurements should last at least 2 seconds

double      Begin_Time,
            End_Time,
            User_Time;

double      Microseconds,
            Dhrystones_Per_Second,
            Vax_Mips;

*/
/* end of variables for time measurement */

int main ()
*****
{
    /* CHANGE:
       We no longer use the code itself for timing.
       double dtime(); */

    /* CHANGE:
       We need a strcpy function in MuddCLib. Since we don't pull
       string literal data out of object files, we have to put them
       on the stack. */
    /* "DHRYSTONE PROGRAM, SOME STRING" */
}

```

```

char some_string[32] = {'D', 'H', 'R', 'Y', 'S', 'T', 'O', 'N', 'E', ' '
, 'P', 'R', 'O', 'G', 'R', 'A', 'M', ',', ' ', 'S', 'O', 'M', 'E', ' '
, 'S', 'T', 'R', 'I', 'N', 'G', '\0'};

/* "DHRYSTONE PROGRAM, 1ST STRING" */
char first_string[32] = {'D', 'H', 'R', 'Y', 'S', 'T', 'O', 'N', 'E', ' '
, 'P', 'R', 'O', 'G', 'R', 'A', 'M', ',', ' ', '1', '\'', 'S', 'T',
, 'S', 'T', 'R', 'I', 'N', 'G', '\0'};

/* Before we count down, we need to define the strings that will
   be used in the tests. "DHRYSTONE PROGRAM, 2ND STRING" */
char second_string[32] = {'D', 'H', 'R', 'Y', 'S', 'T', 'O', 'N', 'E', ' '
, 'P', 'R', 'O', 'G', 'R', 'A', 'M', ',', ' ', '2', '\'', 'N', 'D',
, 'S', 'T', 'R', 'I', 'N', 'G', '\0'};

/* "DHRYSTONE PROGRAM, 3RD STRING" */
char third_string[32] = {'D', 'H', 'R', 'Y', 'S', 'T', 'O', 'N', 'E', ' '
, 'P', 'R', 'O', 'G', 'R', 'A', 'M', ',', ' ', '3', '\'', 'R', 'D',
, 'S', 'T', 'R', 'I', 'N', 'G', '\0'};

      One_Fifty      Int_1_Loc;
REG    One_Fifty      Int_2_Loc;
      One_Fifty      Int_3_Loc;
REG    char           Ch_Index;
      Enumeration   Enum_Loc;
      Str_30         Str_1_Loc;
      Str_30         Str_2_Loc;
REG    int            Run_Index;
REG int           Number_Of_Runs = NUMBER_OF_RUNS; // Number
of runs is 10,000
      int             i;
      int             testsSucceeded;
/* We'll hard-code 100,000 runs in for now and see how that does. */

/* Initializations */

/* CHANGE: We're never using sde!!
#if !#system(sde)
FILE          *Ap;
if ((Ap = fopen("dhry.res","a+")) == NULL)
{
    printf("Can not open dhry.res\n\n");
    exit(1);
}
#endif

```

```

*/



/* CHANGE: We don't have a malloc function.
Next_Ptr_Glob = (Rec_Pointer) malloc (sizeof (Rec_Type));
Ptr_Glob = (Rec_Pointer) malloc (sizeof (Rec_Type));      */
Next_Ptr_Glob = &Next_Rec_Glob;
Ptr_Glob = &Rec_Glob;

Ptr_Glob->Ptr_Comp          = Next_Ptr_Glob;
Ptr_Glob->Discr            = Ident_1;
Ptr_Glob->variant.Enum_Comp = Ident_3;
Ptr_Glob->variant.Int_Comp = 40;

strcpy (Ptr_Glob->variant.Str_Comp, some_string);
strcpy (Str_1_Loc, first_string);

Arr_2_Glob [8][7] = 10;
/* Was missing in published program. Without this statement,      */
/* Arr_2_Glob [8][7] would have an undefined value.           */
/* Warning: With 16-Bit processors and Number_Of_Runs > 32000, */
/* overflow may occur for this array element.                  */

/* CHANGE:
   No printing necessary!
printf ("\n");
printf ("Dhrystone Benchmark, Version 2.1 (Language: C)\n");
printf ("\n");*/



/*
if (Reg)
{
    printf ("Program compiled with 'register' attribute\n");
    printf ("\n");
}
else
{
    printf ("Program compiled without 'register' attribute\n");
    printf ("\n");
}
*/
/* CHANGE: We'll never run this on sde and we set the number of runs
   at the beginning of main, so we don't need to read it!
#endif
// Call a non-inlineable function to prevent loop unrolling
// from knowing the number of loops.
Number_Of_Runs = number_of_runs ();
#else
printf ("Please give the number of runs through the benchmark: ");
{
    int n;
    scanf ("%d", &n);

```

```

        Number_Of_Runs = n;
    }
    printf ("\n");
#endif */

/* CHANGE:
   Instead of printing that execution starts here, we'll
   flash some LEDs.
printf ("Execution starts, %d runs through Dhrystone\n",Number_Of_Runs);
*/

/* The LEDs count down one at a time, then flash to indicate the
   beginning of tests. */
setLED(0x8);
delay1000clock(200);
setLED(0x4);
delay1000clock(200);
setLED(0x2);
delay1000clock(200);
setLED(0x1);
delay1000clock(200);
setLED(0xF);
delay1000clock(50);
setLED(0x0);

/*****************/
/* Start timer */
/*****************/

/* CHANGE:
   No more internal timing.
Begin_Time = dtime();
*/
for (Run_Index = 1; Run_Index <= Number_Of_Runs; ++Run_Index)
{
    Proc_5();
    Proc_4();
    /* Ch_1_Glob == 'A', Ch_2_Glob == 'B', Bool_Glob == true */
    Int_1_Loc = 2;
    Int_2_Loc = 3;
    strcpy (Str_2_Loc, second_string);
    Enum_Loc = Ident_2;
    Bool_Glob = ! Func_2 (Str_1_Loc, Str_2_Loc);

    /* Bool_Glob == 1 */
    while (Int_1_Loc < Int_2_Loc) /* loop body executed once */
    {
        Int_3_Loc = 5 * Int_1_Loc - Int_2_Loc;
        /* Int_3_Loc == 7 */
        Proc_7 (Int_1_Loc, Int_2_Loc, &Int_3_Loc);
        /* Int_3_Loc == 7 */
    }
}

```

```

        Int_1_Loc += 1;
    } /* while */

    /* Int_1_Loc == 3, Int_2_Loc == 3, Int_3_Loc == 7 */
    Proc_8 (Arr_1_Glob, Arr_2_Glob, Int_1_Loc, Int_3_Loc);
    /* Int_Glob == 5 */
    Proc_1 (Ptr_Glob);

    for (Ch_Index = 'A'; Ch_Index <= Ch_2_Glob; ++Ch_Index)
        /* loop body executed twice */
    {
        if (Enum_Loc == Func_1 (Ch_Index, 'C'))
            /* then, not executed */
        {
            Proc_6 (Ident_1, &Enum_Loc);
            strcpy (Str_2_Loc, third_string);
            Int_2_Loc = Run_Index;
            Int_Glob = Run_Index;
        }
    }
    /* Int_1_Loc == 3, Int_2_Loc == 3, Int_3_Loc == 7 */
    Int_2_Loc = Int_2_Loc * Int_1_Loc;
    Int_1_Loc = Int_2_Loc / Int_3_Loc;
    Int_2_Loc = 7 * (Int_2_Loc - Int_3_Loc) - Int_1_Loc;
    /* Int_1_Loc == 1, Int_2_Loc == 13, Int_3_Loc == 7 */
    Proc_2 (&Int_1_Loc);
    /* Int_1_Loc == 5 */

} /* loop "for Run_Index" */

/*****************/
/* Stop timer */
/*****************/

/* Flash the LEDs to indicate the end of the tests. */
for(i = 0; i < 10; ++i)
{
    setLED(0xF);
    delay1000clock(100);
    setLED(0x0);
    delay1000clock(100);
}

/* Check all of the results, and display pass or fail on the LEDs. */
/* Assume tests succeeded and challenge. */
testsSucceeded = (Int_Glob == 5) &&
    (Bool_Glob == 1) &&
    (Ch_1_Glob == 'A') &&
    (Ch_2_Glob == 'B') &&
    (Arr_1_Glob[8] == 7) &&
    (Arr_2_Glob[8][7] == Number_Of_Runs + 10) &&
    ((int)Ptr_Glob->Ptr_Comp == (int)Next_Ptr_Glob->Ptr_Comp) &&
    (Ptr_Glob->Discr == 0) &&

```

```

        (Ptr_Glob->variant.Enum_Comp == 2) &&
        (Ptr_Glob->variant.Int_Comp == 17) &&
        !strcmp(Ptr_Glob->variant.Str_Comp, some_string) &&           // 
strcmp returns 0 if they're the same!
        (Next_Ptr_Glob->Discr == 0) &&
        (Next_Ptr_Glob->variant.Enum_Comp == 1) &&
        (Next_Ptr_Glob->variant.Int_Comp == 18) &&
        !strcmp(Next_Ptr_Glob->variant.Str_Comp, some_string) &&
        (Int_1_Loc == 5) &&
        (Int_2_Loc == 13) &&
        (Int_3_Loc == 7) &&
        (Enum_Loc == 1) &&
        !strcmp(Str_1_Loc, first_string) &&
        !strcmp(Str_2_Loc, second_string);

if(testsSucceeded)
{
    /* LEDs show fireworks */
    while(1)
    {
        setLED(0x2);
        delay1000clock(100);
        setLED(0x5);
        delay1000clock(100);
        setLED(0x8);
        delay1000clock(100);
        setLED(0x0);
        delay1000clock(500);
        setLED(0x4);
        delay1000clock(100);
        setLED(0xA);
        delay1000clock(100);
        setLED(0x1);
        delay1000clock(100);
        setLED(0x0);
        delay1000clock(700);
    }
}
else
{
    /* LEDs flash quickly */
    while(1)
    {
        setLED(0xF);
        delay1000clock(50);
        setLED(0x0);
        delay1000clock(50);
    }
}

/* CHANGE:
   We don't print any of the data, but we'll validate it and flash

```

the LEDs accordingly.

[COMMENTS OMITTED FOR BREVITY – SEE ORIGINAL SOURCE FOR DETAILS]

```
    return (0);  
}  
/* End of main() */
```

20.5.5. Dhystone Functions

```
/*  
 * dhryFuncs.c  
 *  
 * These are all of the functions used by Dhystone besides the  
 * main function.  
 */  
  
#include "dhry.h"  
#include "muddCLib/muddCLib.h"  
  
#ifndef REG  
#define REG  
    /* REG becomes defined as empty */  
    /* i.e. no register variables */  
#else  
#define REG register  
#endif  
  
extern Rec_Type      Rec_Glob,  
                    Next_Rec_Glob;  
extern Rec_Pointer   Ptr_Glob,  
                    Next_Ptr_Glob;  
extern int           Int_Glob;  
extern Boolean       Bool_Glob;  
extern char          Ch_1_Glob,  
                    Ch_2_Glob;  
extern int           Arr_1_Glob [50];  
extern int           Arr_2_Glob [50] [50];  
  
/* CHANGE:  
 * This code didn't seem to work properly!  
Proc_1 (Ptr_Val_Par)  
// *****  
  
REG Rec_Pointer Ptr_Val_Par;  
    //executed once  
*/  
void Proc_1 (Rec_Pointer Ptr_Val_Par)  
{  
    REG Rec_Pointer Next_Record = Ptr_Val_Par->Ptr_Comp;  
        /* == Ptr_Glob_Next */
```

```

/* Local variable, initialized with Ptr_Val_Par->Ptr_Comp,
/* corresponds to "rename" in Ada, "with" in Pascal */
```

```

structassign (*Ptr_Val_Par->Ptr_Comp, *Ptr_Glob);
Ptr_Val_Par->variant.Int_Comp = 5;
Next_Record->variant.Int_Comp
    = Ptr_Val_Par->variant.Int_Comp;
Next_Record->Ptr_Comp = Ptr_Val_Par->Ptr_Comp;
Proc_3 (&Next_Record->Ptr_Comp);
/* Ptr_Val_Par->Ptr_Comp->Ptr_Comp
   == Ptr_Glob->Ptr_Comp */
if (Next_Record->Discr == Ident_1)
    /* then, executed */
{
    Next_Record->variant.Int_Comp = 6;
    Proc_6 (Ptr_Val_Par->variant.Enum_Comp,
            &Next_Record->variant.Enum_Comp);
    Next_Record->Ptr_Comp = Ptr_Glob->Ptr_Comp;
    Proc_7 (Next_Record->variant.Int_Comp, 10,
            &Next_Record->variant.Int_Comp);
}
else /* not executed */
    structassign (*Ptr_Val_Par, *Ptr_Val_Par->Ptr_Comp);
} /* Proc_1 */

/* CHANGE:
   This doesn't appear to be valid C!
Proc_2 (Int_Par_Ref)
// *****
// executed once
// *Int_Par_Ref == 1, becomes 4

One_Fifty *Int_Par_Ref;
*/
void Proc_2 (One_Fifty *Int_Par_Ref)
{
    One_Fifty Int_Loc;
    Enumeration Enum_Loc;

    Int_Loc = *Int_Par_Ref + 10;
    do /* executed once */
        if (Ch_1_Glob == 'A')
            /* then, executed */
        {
            Int_Loc -= 1;
            *Int_Par_Ref = Int_Loc - Int_Glob;
            Enum_Loc = Ident_1;
        } /* if */
        while (Enum_Loc != Ident_1); /* true */
} /* Proc_2 */

```

```

/* CHANGE:
   This doesn't appear to be valid C!
Proc_3 (Ptr_Ref_Par)
// *****
   // executed once
   // Ptr_Ref_Par becomes Ptr_Glob

Rec_Pointer *Ptr_Ref_Par;
*/
void Proc_3 (Rec_Pointer *Ptr_Ref_Par)
{
    if (Ptr_Glob != Null)
        /* then, executed */
        *Ptr_Ref_Par = Ptr_Glob->Ptr_Comp;
    Proc_7 (10, Int_Glob, &Ptr_Glob->variant.Int_Comp);
} /* Proc_3 */

void Proc_4 (void) /* without parameters */
/*
   /* executed once */
{
    Boolean Bool_Loc;

    Bool_Loc = Ch_1_Glob == 'A';
    Bool_Glob = Bool_Loc | Bool_Glob;
    Ch_2_Glob = 'B';
} /* Proc_4 */

void Proc_5 (void) /* without parameters */
/*
   /* executed once */
{
    Ch_1_Glob = 'A';
    Bool_Glob = false;
} /* Proc_5 */

      /* Procedure for the assignment of structures,
      /* if the C compiler doesn't support this feature */
#endif NOSTRUCTASSIGN
memcpy (register char *d, register char *s, register int l)
//register char    *d;
//register char    *s;
//register int     l;
{
    while (l--) *d++ = *s++;
}
#endif

void Proc_6 (Enumeration Enum_Val_Par, Enumeration *Enum_Ref_Par)
// *****

```

```

/* executed once */
/* Enum_Val_Par == Ident_3, Enum_Ref_Par becomes Ident_2 */

//Enumeration  Enum_Val_Par;
//Enumeration *Enum_Ref_Par;
{
    *Enum_Ref_Par = Enum_Val_Par;
    if (! Func_3 (Enum_Val_Par))
        /* then, not executed */
        *Enum_Ref_Par = Ident_4;
    switch (Enum_Val_Par)
    {
        case Ident_1:
            *Enum_Ref_Par = Ident_1;
            break;
        case Ident_2:
            if (Int_Glob > 100)
                /* then */
                *Enum_Ref_Par = Ident_1;
            else *Enum_Ref_Par = Ident_4;
            break;
        case Ident_3: /* executed */
            *Enum_Ref_Par = Ident_2;
            break;
        case Ident_4: break;
        case Ident_5:
            *Enum_Ref_Par = Ident_3;
            break;
    } /* switch */
} /* Proc_6 */

```

```

void Proc_7 (One_Fifty Int_1_Par_Val,
              One_Fifty Int_2_Par_Val,
              One_Fifty *Int_Par_Ref)
/*****************************************/
/* executed three times */
/* first call:      Int_1_Par_Val == 2, Int_2_Par_Val == 3, */
/*                  Int_Par_Ref becomes 7 */
/* second call:     Int_1_Par_Val == 10, Int_2_Par_Val == 5, */
/*                  Int_Par_Ref becomes 17 */
/* third call:      Int_1_Par_Val == 6, Int_2_Par_Val == 10, */
/*                  Int_Par_Ref becomes 18 */
//One_Fifty      Int_1_Par_Val;
//One_Fifty      Int_2_Par_Val;
//One_Fifty      *Int_Par_Ref;
{
    One_Fifty Int_Loc;

    Int_Loc = Int_1_Par_Val + 2;
    *Int_Par_Ref = Int_2_Par_Val + Int_Loc;
} /* Proc_7 */

```

```

void Proc_8 (Arr_1_Dim Arr_1_Par_Ref,
             Arr_2_Dim Arr_2_Par_Ref,
             int Int_1_Par_Val,
             int Int_2_Par_Val)
/*********************************************
/* executed once      */
/* Int_Par_Val_1 == 3 */
/* Int_Par_Val_2 == 7 */
//Arr_1_Dim      Arr_1_Par_Ref;
//Arr_2_Dim      Arr_2_Par_Ref;
//int            Int_1_Par_Val;
//int            Int_2_Par_Val;
{
    REG One_Fifty Int_Index;
    REG One_Fifty Int_Loc;

    Int_Loc = Int_1_Par_Val + 5;
    Arr_1_Par_Ref [Int_Loc] = Int_2_Par_Val;
    Arr_1_Par_Ref [Int_Loc+1] = Arr_1_Par_Ref [Int_Loc];
    Arr_1_Par_Ref [Int_Loc+30] = Int_Loc;
    for (Int_Index = Int_Loc; Int_Index <= Int_Loc+1; ++Int_Index)
        Arr_2_Par_Ref [Int_Loc] [Int_Index] = Int_Loc;
    Arr_2_Par_Ref [Int_Loc] [Int_Loc-1] += 1;
    Arr_2_Par_Ref [Int_Loc+20] [Int_Loc] = Arr_1_Par_Ref [Int_Loc];
    Int_Glob = 5;
} /* Proc_8 */

```

```

Enumeration Func_1 (Capital_Letter Ch_1_Par_Val,
                     Capital_Letter Ch_2_Par_Val)
/*********************************************
/* executed three times
/* first call:      Ch_1_Par_Val == 'H', Ch_2_Par_Val == 'R' */
/* second call:     Ch_1_Par_Val == 'A', Ch_2_Par_Val == 'C' */
/* third call:      Ch_1_Par_Val == 'B', Ch_2_Par_Val == 'C' */

//Capital_Letter   Ch_1_Par_Val;
//Capital_Letter   Ch_2_Par_Val;
{
    Capital_Letter       Ch_1_Loc;
    Capital_Letter       Ch_2_Loc;

    Ch_1_Loc = Ch_1_Par_Val;
    Ch_2_Loc = Ch_1_Loc;
    if (Ch_2_Loc != Ch_2_Par_Val)
        /* then, executed */
        return (Ident_1);
    else /* not executed */
    {
        Ch_1_Glob = Ch_1_Loc;
        return (Ident_2);
    }
}

```

```

} /* Func_1 */

Boolean Func_2 (Str_30 Str_1_Par_Ref, Str_30 Str_2_Par_Ref)
/*****************/
/* executed once */
/* Str_1_Par_Ref == "DHRYSTONE PROGRAM, 1'ST STRING" */
/* Str_2_Par_Ref == "DHRYSTONE PROGRAM, 2'ND STRING" */

// Str_30 Str_1_Par_Ref;
// Str_30 Str_2_Par_Ref;
{
    REG One_Thirty      Int_Loc;
    Capital_Letter     Ch_Loc;

    Int_Loc = 2;
    while (Int_Loc <= 2) /* loop body executed once */
        if (Func_1 (Str_1_Par_Ref[Int_Loc],
                     Str_2_Par_Ref[Int_Loc+1]) == Ident_1)
            /* then, executed */
        {
            Ch_Loc = 'A';
            Int_Loc += 1;
        } /* if, while */
    if (Ch_Loc >= 'W' && Ch_Loc < 'Z')
        /* then, not executed */
        Int_Loc = 7;
    if (Ch_Loc == 'R')
        /* then, not executed */
        return (true);
    else /* executed */
    {
        if (strcmp (Str_1_Par_Ref, Str_2_Par_Ref) > 0)
            /* then, not executed */
        {
            Int_Loc += 7;
            Int_Glob = Int_Loc;
            return (true);
        }
        else /* executed */
            return (false);
    } /* if Ch_Loc */
} /* Func_2 */

Boolean Func_3 (Enumeration Enum_Par_Val)
/*****************/
/* executed once */
/* Enum_Par_Val == Ident_3 */
//Enumeration Enum_Par_Val;
{
    Enumeration Enum_Loc;

```

```

Enum_Loc = Enum_Par_Val;
if (Enum_Loc == Ident_3)
    /* then, executed */
    return (true);
else /* not executed */
    return (false);
} /* Func_3 */

```

20.5.6. Lights Out! Header

```

/*
 * lightsOut.h
 *
 * Created by Matt McKnott on 4/1/07
 * VLSI Spring 2007
 *
 * $Author: whiterook2004 $, $Date: 2007-04-15 07:11:05 -0700 (Sun, 15
Apr 2007) $ -- $Revision: 568 $
*
* This is a demo program for the hmc-mips project. It is a
* "lights out" game playable on the LCD display onboard the
* PCB board.
*/

```

```

#ifndef LIGHTSOUT_INCLUDED
#define LIGHTSOUT_INCLUDED 1

#include "muddCLib/muddCLib.h"

/* Comment this line to use the LCD instead of LEDs */
// #define USE_LEDS 1

#endif USE_LEDS
    #define NUM_LIGHTS 4
    int lastOn;
#else
    #define NUM_LIGHTS 20
#endif

#define LIGHT_OFF 1
#define LIGHT_ON 0

#define PRINT_LIGHT_ON 0x20
#define PRINT_LIGHT_OFF 0xFF
#define PRINT_CARAT '^'

int lights[NUM_LIGHTS];
int lightsOut;
int lightPosition;

/* readInput shall return the button that was pressed, or 0 if no

```

```

        button is being pressed. */
char* readInput();

/* update shall shift the lightPosition carat if it receives a
   left or right button, or else it shall flip the corresponding
   lights if it sees an up button. */
void update(char* input);

/* areLightsOut returns true if all of the lights are out. */
int areLightsOut();

/* print the lights and carat out in an understandable way. */
void printLights();

#endif /* LIGHTSOUT_INCLUDED. */

```

20.5.7. Lights Out! Source

```

/*
 *  lightsOut.c
 *
 *  Created by Matt McKnott on 4/1/07
 *  VLSI Spring 2007
 *
 * $Author: whiterook2004 $, $Date: 2007-04-15 07:11:05 -0700 (Sun, 15
Apr 2007) $ -- $Revision: 568 $
*
*  This demo program is a "lights out" game playable on the
*  hmc-mips system.
*/

```

```

//#define DEBUG_SIMULATOR 1

#ifndef DEBUG_SIMULATOR
#include <stdio.h>
#endif

#include "lightsOut.h"
#include "muddCLib/muddCLib.h"
#include "muddCLib/mtRand.h"

char *lastPressed;
int numLoops;

int main()
{
    /* Declare variables */
    int i;
    char* buttonPressed;
    int numberExtracted;
    int done = 0;

```

```

/* Initialize variables */
lastPressed = NOSWITCH;
numLoops = 0;

#ifndef USE_LEDS
    /* Construct all of the strings used in the game. */
    char lightsOutMsg[LCD_WIDTH] = {' ', ' ', ' ', ' ', ' ', 'L', 'i',
'g', 'h', 't', 's', 'O', 'u', 't', '!'}; 
    char hmcMipsMsg[LCD_WIDTH] = {'H', 'M', 'C', '-', 'M', 'I', 'P',
'S', ' ', 'V', 'L', 'S', 'I', ' ', '0', '7'}; 
    char youWinMsg[LCD_WIDTH] = {'Y', 'o', 'u', ' ', 'w', 'i', 'n',
!'!'}; 
    char playAgainMsg[LCD_WIDTH] = {'P', 'l', 'a', 'y', ' ', 'a', 'g',
'a', 'i', 'n', '?'}; 
    char gameOverMsg[LCD_WIDTH] = {' ', ' ', ' ', 'G', 'a', 'm', 'e',
', 'O', 'v', 'e', 'r'}; 
    char blankMsg[LCD_WIDTH] = {'\0'}; 
#endif

#ifdef DEBUG_SIMULATOR
    printf("LightsOut!\n");
#else
    #ifdef USE_LEDS
        lastOn = 0;
    #else // USE_LEDS not defined
        /* Initialize the LCD display. */ 
        initLCD(); 
        sendInst(L_disp); 
        dispMessage(lightsOutMsg, hmcMipsMsg); 
    #endif //USE_LEDS

        /* Wait for a button press. */ 
        while(readInput() == NOSWITCH);
#endif

    while(!done) /* The main program loop */ 
    { 

        /* Initialize the random number generator. 
           Seed it with the cycle count. */ 
        initializeGenerator(numLoops); 
        generateNumbers(); 
        numberExtracted = extractNumber(NUM_LIGHTS); 

        /* Initialize variables, array, and LCD. */ 
        for(i = 0; i < NUM_LIGHTS; ++i) 
        { 
            /* We'll place the lights randomly. */ 
            lights[i] = numberExtracted & 0x1; 
            numberExtracted = numberExtracted >> 1; 
        } 
        lightsOut = 0; 
        lightPosition = 0; 
    }

```

```

printLights();
buttonPressed = NOSWITCH;

// Here's the game loop where all of the light selection
// happens.
while(!lightsOut)
{
    /* Read input from the buttons and update the game state. */
    buttonPressed = readInput();
    update(buttonPressed);
    lightsOut = areLightsOut();
    if(buttonPressed == BUTTON_DOWN)
    {
        break;
    }
    if(buttonPressed != NOSWITCH)
    {
        printLights();
    }
}

// If we exited the main game loop without the winning
// condition satisfied, the user wanted to start over,
// so just give them a new board.
if(!lightsOut)
    continue;

#ifndef DEBUG_SIMULATOR
    /* The lights are out, the game is over */
#endif USE_LEDS
    lastOn = 0;
#else
    dispMessage(youWinMsg, playAgainMsg);
#endif
#endif

buttonPressed = NOSWITCH;

/* Any button press continues except down, which ends. */
while(buttonPressed == NOSWITCH)
{
#endif DEBUG_SIMULATOR
    printLights();
    printf("You win!\nPlay again?\n");
#else
    #ifdef USE_LEDS
        /* Finite-state fireworks animation */
        switch(lastOn)
        {
            case 0: setLED(0x2);
            break;
            case 1: setLED(0x5);

```

```

        break;
    case 2: setLED(0x8);
    break;
    case 10: setLED(0x4);
    break;
    case 11: setLED(0xA);
    break;
    case 12: setLED(0x1);
    break;
    default: setLED(0x0);
}

++lastOn;

#endif
#endif

buttonPressed = readInput();
if(buttonPressed == BUTTON_DOWN)
{
    done = 1;
}
}

} /* END of the main program loop. */

#ifndef DEBUG_SIMULATOR
#ifndef USE_LEDS
    dispMessage(gameOverMsg, blankMsg);
#endif
while(1); /* Loop forever instead of letting the program
            counter run up. */
#endif

return 0;
} /* END of the main function. */

/* This will find out if a button was just pressed and return the
pointer to its address. */
char* readInput()
{
    // Increment the number of loops at readInput.
    ++numLoops;

#ifdef DEBUG_SIMULATOR /* Are we simulating? */

    char c;
    scanf(" %c", &c);

    if(c == 'a')
        return BUTTON_LEFT;
    if(c == 'd')
        return BUTTON_RIGHT;

```

```

        if(c == 'w')
            return BUTTON_UP;
        if(c == 's')
            return BUTTON_DOWN;

        return NOSWITCH;

#else /* This would mean we aren't simulating. */

char leftVal;
char rightVal;
char upVal;
char downVal;

leftVal = readSwitch(BUTTON_LEFT);
rightVal = readSwitch(BUTTON_RIGHT);
upVal = readSwitch(BUTTON_UP);
downVal = readSwitch(BUTTON_DOWN);

/* Check to see if any button was just pressed. */
if(lastPressed == NOSWITCH)
{
    if(leftVal == BUTTON_PRESSED)
    {
        lastPressed = BUTTON_LEFT;
        return BUTTON_LEFT;
    }
    else if(rightVal == BUTTON_PRESSED)
    {
        lastPressed = BUTTON_RIGHT;
        return BUTTON_RIGHT;
    }
    else if(upVal == BUTTON_PRESSED)
    {
        lastPressed = BUTTON_UP;
        return BUTTON_UP;
    }
    else if(downVal == BUTTON_PRESSED)
    {
        lastPressed = BUTTON_LEFT;
        return BUTTON_DOWN;
    }
}

/* If no button was just pressed, check to see if all buttons are
   released, and if they are then show the last value as having
   been released. */
if( leftVal == BUTTON_RELEASED
    && rightVal == BUTTON_RELEASED
    && upVal == BUTTON_RELEASED
    && downVal == BUTTON_RELEASED )
{
    lastPressed = NOSWITCH;
}

```

```

    }

    return NOSWITCH;

#endif /* We determined whether we want the simulator version of readInput
        or the real version. */
}

/* Update the array and the carat according to the button that
   was pressed. */
void update(char* input)
{
    /* If the button was the right or left button, we want to move the
       carat in the appropriate direction. */
    if(input == BUTTON_LEFT)
    {
        if(lightPosition != 0)
            lightPosition -= 1;
    }
    else if(input == BUTTON_RIGHT)
    {
        if(lightPosition != (NUM_LIGHTS - 1) )
            lightPosition += 1;
    }
    /* If the button was the up button, we want to flip the values
       in the square pointed to by the carat, and also its neighbors. */
    else if(input == BUTTON_UP)
    {
        /* Consider the corners first, and if we aren't at a corner,
           then just flip the one pointed at and its neighbors. */
        if(lightPosition == 0)
        {
            lights[0] = (~lights[0]) & 0x1;
            lights[1] = (~lights[1]) & 0x1;
        }
        else if(lightPosition == NUM_LIGHTS - 1)
        {
            lights[NUM_LIGHTS - 1] = (~lights[NUM_LIGHTS - 1]) & 0x1;
            lights[NUM_LIGHTS - 2] = (~lights[NUM_LIGHTS - 2]) & 0x1;
        }
        else
        {
            lights[lightPosition - 1] = (~lights[lightPosition - 1])
& 0x1;
            lights[lightPosition] = (~lights[lightPosition]) & 0x1;
            lights[lightPosition + 1] = (~lights[lightPosition + 1])
& 0x1;
        }
    }
}

/* Return 1 if the lights are out, 0 otherwise */
int areLightsOut()

```

```

{
    int i;
    for(i = 0; i < NUM_LIGHTS; ++i)
    {
        if(lights[i] == LIGHT_ON)
            return 0;
    }

    /* We didn't find any lights that were on. */
    return 1;
}

/* Display the lights in some form the user can interact with. */
void printLights()
{
#ifdef DEBUG_SIMULATOR
    int i;

    printf("--\n");

    for(i = 0; i < NUM_LIGHTS; ++i)
    {
        if(lights[i] == LIGHT_ON)
            printf("_");
        else
            printf("*");
    }
    printf("\n");
    for(i = 0; i < NUM_LIGHTS; ++i)
    {
        if(i == lightPosition)
            printf("^");
        else
            printf(" ");
    }
    printf("\n\n");
#endif
}
#endif
#endif

```

```

        output = output & ~(1 << lightPosition); // Turn off cursor
position.
    else if(lastOn >= 12 && lastOn < 15)
        output = output | (1 << lightPosition); // Turn on cursor
position.

    // Print the output on the LEDs.
    setLED(output);

    lastOn++;
    if(lastOn == 15)
        lastOn = 0; // Reset the finite state machine if we're at
a certain point.

#else
    char str1[LCD_WIDTH]; /* The upper LCD bar */
    char str2[LCD_WIDTH]; /* The lower LCD bar */
    int i;

    for(i = 0; i < NUM_LIGHTS; ++i)
    {
        if(lights[i] == LIGHT_ON)
            str1[i] = PRINT_LIGHT_ON;
        else
            str1[i] = PRINT_LIGHT_OFF;
    }

    for(i = 0; i < NUM_LIGHTS; ++i)
    {
        if(i == lightPosition)
            str2[i] = PRINT_CARAT;
        else
            str2[i] = ' ';
    }

    dispMessage(str1, str2);
#endif
#endif
}

```

20.5.8. Corner-Case Test

```

/* mips_corner_test.c
 *
 * Bart Oegema    boegema@hmc.edu
 * 4/16/2007
 *
 * Written as part of the HMC/Adelaide MIPS processor implementation
project.
 *
 * Combines the corner cases defined by the µArch team into a single C
* test file, outputting status to the LCD display. Intended to aid in

```

```

* debugging of any problems that arise in the hardware implementation
* of the MIPS processor after fabrication.
*
* Test cases adopted from earlier implementations. Memory reallocated
according
* to defined memory map.
*/
#include "muddCLib/muddCLib.h"

#define NUM_TESTS 1
#define TEST0_ADDRESS ((int*)0x20000) // I don't know if this address
works. Also, line 58 uses a magic
// number version
of this.
#define TEST0_RESULT 21

void test0();

int main()
{
    int test_num = 0;
    char passStr[LCD_WIDTH] = {'T', 'e', 's', 't', ' ', ' ', '0', ':', ' ',
    'P', 'a', 's', 's', '\0'};
    char nothing[LCD_WIDTH] = {'\0'};
    char failStr[LCD_WIDTH] = {'T', 'e', 's', 't', ' ', ' ', '0', ':', ' ',
    'F', 'a', 'i', 'l', '\0'};
    char halted[LCD_WIDTH] = {'H', 'a', 'l', 't', 'e', 'd', '.', '\0'};
    int strDigHi = 5;
    int strDigLo = 6;

    initLCD();

    while(test_num < NUM_TESTS)
    {

        switch(test_num)
        {
            /* Test 000
             * Commands tested:
             *      addi, add, beq, sw
             *
             * Expected Behavior:
             *      Fibonacci Sequence.
             */
            case 0:
            {
                test0();

                if (*TEST0_ADDRESS == TEST0_RESULT) // suffix 'u'
makes an unsigned long
                {
                    passStr[strDigHi] = ' ';

```

```

                passStr[strDigLo] = '0';
                dispMessage(passStr, nothing);
            }
        else
        {
            failStr[strDigHi] = ' ';
            failStr[strDigLo] = '0';
            dispMessage(failStr, halted);
            return 0;
        }
    } // case 0

} // test_num switch statement

++test_num; // Go through each test.

} // end of program while loop

return 0;
}

void test0()
{
    asm volatile (
        ".set noreorder\n\t"
        "main0: addi $2, $0, 0\n\t"           // initialize $2 = 0
            "addi $3, $0, 1\n\t"           // initialize $3 = 1
            "addi $5, $0, 21\n\t"          // initialize $5 = 21
(stopping point)
        "loop0: add $4, $2, $3\n\t"          // $4 <= $2 + $3
            "add $2, $3, $0\n\t"          // $2 <= $3
            "add $3, $4, $0\n\t"          // $3 <= $4
            "beq $4, $5, write0\n\t"       // when sum is 21,
jump to write
            "nop\n\t"
            "beq $0, $0, loop0\n\t"        // loop (beq is
easier to assemble than jump)
            "nop\n\t"
        "write0: sw $4, 0x20000($2)\n\t"     // should write
21 @ 7 + 13 = 20 = 0x14
        "end0: beq $0, $0, end0\n\t"       // loop forever
            "nop" );
}

```

20.5.9. Button Test

```
/*
 * test_buttons.c
 *
 * Created by Matt McKnott, HMC-MIPS Project, VLSI Spring 2007
 */
```

```

* $Author: whiterook2004 $, $Date: 2007-04-15 23:55:16 -0700 (Sun, 15
Apr 2007) $ -- $Revision: 584 $
*
* The purpose of this test is to read the buttons on the board
* and indicate by LEDs whether they are being read properly.
* It also tests a small version of the code used by lightsOut
* to read a single button press.
*
* - The left button causes LED0 to be turned on when pressed, and
*   off when released.
* - The right button causes LED1 to toggle on and off.
*
* Some recommended additions to this program:
* - The top button causes LED2 and LED3 to blink.
* - The bottom button causes LED2 and LED3 to blink faster.
*
* Pressing the left, right, and top or bottom buttons together
* will cause all of their functionalities to happen at once;
* however, pressing buttons top and bottom together causes
* only the bottom's functionality.
*
* Note: We don't use BUTTON_ENTER because it is used for
*       reset in our implementation.
*
* Systems tested:
*   - memory I/O -- buttons
*   - memory I/O -- LEDs
*/

```

```

#include "muddCLib/muddCLib.h"

// Uncomment this line if you want to try making the right button toggle
LED3.
#define TRY_TOGGLE 1

int main()
{
    // Initially, all LEDs off.
    char ledVal = 0x0;
    setLED(ledVal);

    // These variables let us make a button press only happen
    // when the button is first read as pressed.
    char *lastPressed = NOSWITCH;

    /* Read each button and light up LEDs accordingly. */
    while(1)
    {
        // Left button press causes the LED0 to light up
        // when pressed.
        if(readSwitch(BUTTON_LEFT) == BUTTON_PRESSED)
        {
            ledVal = ledVal | 0x1; // binary 0001

```

```

        lastPressed = BUTTON_LEFT;
    }
else
{
    ledVal = ledVal & ~0x1; // binary 1110
    if(lastPressed == BUTTON_LEFT)
        lastPressed = NOSWITCH;
}

// Top button press causes the LED1 to light up
// when pressed.
if(readSwitch(BUTTON_UP) == BUTTON_PRESSED)
{
    ledVal = ledVal | 0x2; // binary 0010
    lastPressed = BUTTON_UP;
}
else
{
    ledVal = ledVal & ~0x2; // binary 1101
    if(lastPressed == BUTTON_UP)
        lastPressed = NOSWITCH;
}

// Bottom button press causes the LED2 to light up
// when pressed.
if(readSwitch(BUTTON_DOWN) == BUTTON_PRESSED)
{
    ledVal = ledVal | 0x4; // binary 0100
    lastPressed = BUTTON_DOWN;
}
else
{
    ledVal = ledVal & ~0x4; // binary 1011
    if(lastPressed == BUTTON_DOWN)
        lastPressed = NOSWITCH;
}

#ifndef TRY_TOGGLE
    // Right button press turns on LED3 for as long as it is held.
    if(readSwitch(BUTTON_RIGHT) == BUTTON_PRESSED)
    {
        ledVal = ledVal | 0x8; // binary 1000
        lastPressed = BUTTON_RIGHT;
    }
    else
    {
        ledVal = ledVal & ~0x8; // binary 0111
        if(lastPressed == BUTTON_RIGHT)
            lastVal = NOSWITCH;
    }
#else
    // Right button press toggles LED3
    if(readSwitch(BUTTON_RIGHT) == BUTTON_PRESSED)

```

```

    {
        if(lastPressed == NOSWITCH)
        {
            // If LED3 is not off
            if(ledVal & 0x8)
                ledVal = ledVal & ~0x8;      // Turn LED3 off
            else
                ledVal = ledVal | 0x8; // Turn LED3 on

            lastPressed = BUTTON_RIGHT;
        }
    }
    else
    {
        if(lastPressed == BUTTON_RIGHT)
            lastPressed = NOSWITCH;
    }
#endif

        // After figuring out what buttons were pressed, show the
LEDs.
        setLED(ledVal);
    }

    return 0;
}

```

20.5.10. LCD Test

```

/* test_lcd.c
 *
 * Created by Matt McNett, HMC-MIPS Project, VLSI Spring 2007
 * $Author: whiterook2004 $, $Date: 2007-04-15 23:53:45 -0700 (Sun, 15
Apr 2007) $ -- $Revision: 583 $
 *
 * The purpose of this test is to test the function of the LCD screen
 * that is built into the PCB board. It does the following:
 *   - Initializes the LCD screen and initiates an infinite loop
 *   - Causes the LEDs to blink before each iteration through the loop
 *   - Flashes two stars on the LCD, and then prints
 *     "Hello, World!
 *       HMC VLSI 07"
 *     on the LCD.
 *   - Leaves the message on the screen for a short time, then clears
 *     the screen and loops again.
 *
 * Systems tested:
 *   - memory I/O -- LCD
 */

```

```
#include "muddCLib/muddCLib.h"
```

```

int main()
{
    char str1[16] = {'H', 'e', 'l', 'l', 'o', ',', ' ', 'w', 'o', 'r',
'l', 'd', '!'};
    char str2[16] = { ' ', ' ', ' ', 'H', 'M', 'C', ' ', 'V', 'L', 'S',
'I', ' ', '0', '7'};
    int i;

    initLCD();

    while(1)
    {
        // Make some LEDs blink so we know we're getting
        // some response from the board.
        setLED(0x1);
        delay1000clock(300);
        setLED(0x2);
        delay1000clock(300);
        setLED(0x4);
        delay1000clock(300);
        setLED(0x8);
        delay1000clock(300);
        setLED(0x9);

        // Move test
        move(L_secondRow);
        dispChar('*');

        delay1000clock(200); // Flash a * for a short bit

        setLED(0x00);

        sendInst(L_clear);
        move(0x08);
        dispChar('*');

        delay1000clock(200);

        setLED(0x00);

        // Print "Hello, Word!"
        //         " HMC VLSI 07"
        // manually, without using dispMessage...
        sendInst(L_clear);
        move(0x00);

        for(i = 0; i < LCD_WIDTH; ++i)
        {
            if(str1[i] == '\0')
                break;
            dispChar(str1[i]);
        }
    }
}

```

```

move(L_secondRow);

for(i = 0; i < LCD_WIDTH; ++i)
{
    if(str2[i] == '\0')
        break;
    dispChar(str2[i]);
}

delay1000clock(3000);

// Clear the screen.
sendInst(L_clear);

delay1000clock(500);

// And now print "Hello World!" using dispMessage...
dispMessage(str1, str2);

delay1000clock(3000);

sendInst(L_clear);
}

return 0;
}

```

20.5.11. LED Test

```

/* test_leds.c
 *
 * Created by Matt McNett, HMC-MIPS Project, VLSI Spring 2007
 * $Author: whiterook2004 $, $Date: 2007-04-12 17:24:38 -0700 (Thu, 12
Apr 2007) $ -- $Revision: 541 $
 *
 * The purpose of this test is to constantly read the dip switches
 * on the Xilinx board and make the LEDs reflect the value shown
 * by the switches. It also tests the system's ability to use
 * global data by storing the values of the dip switches in memory.
 *
 * Systems tested:
 *      - memory I/O -- dip switches
 *      - memory I/O -- LEDs
 *      - memory -- global pointer/global data
 */

#include "muddCLib/muddCLib.h"

char dip_switch[4];

int main() {

```

```

        while(1)
        {

            dip_switch[0] = 0x1;
            dip_switch[1] = 0x2;
            dip_switch[2] = 0x4;
            dip_switch[3] = 0x8;

            if(readSwitch(SWITCH0) == SWITCH_ON)
                dip_switch[0] = 0;
            if(readSwitch(SWITCH1) == SWITCH_ON)
                dip_switch[1] = 0;
            if(readSwitch(SWITCH2) == SWITCH_ON)
                dip_switch[2] = 0;
            if(readSwitch(SWITCH3) == SWITCH_ON)
                dip_switch[3] = 0;
            setLED((char)(dip_switch[0] + dip_switch[1] + dip_switch[2] +
dip_switch[3]));

        }
        return 0;
}

```

20.5.12. Simple Test

```

/*
 * test_simple.c
 *
 * Created by Matt McNett, HMC-MIPS Project, VLSI Spring 2007
 * $Author: whiterook2004 $, $Date: 2007-04-15 23:55:16 -0700 (Sun, 15
Apr 2007) $ -- $Revision: 584 $
*
* This test just counts up on the LEDs. Its purpose is to give the
testers something to run
* while they are determining what clock frequency is best for the chip.
*
* Systems tested:
*     - memory I/O -- LEDs
*/

```

```

#include "muddCLib/muddCLib.h"

int main() {
    char count = 0;

    while(1)
    {
        setLED(count);
        delay1000clock(50);
        ++count;
    }

    return 0;
}

```

```
}
```

20.5.13. LED Assembly Test

```
# test_led_asm.asm
#
# Created by Matt McKnett on 4/19/2007
#
# This file is designed for debugging the dual FPGA.
# It turns on LED0 and LED3, and turns off LED1 and LED2.
#
.set noreorder

main:
    lui    $8, 0xA004      # Load the LED address 0xA0044000
    ori    $8, $8, 0x4000
    addi   $9, $0, 0x9       # Print 1001 on the LEDs
    sw    $9, 0($8)
loop:
    j     loop
    nop
```

20.5.14. Instruction Checker Script

```
"""
checkInstructions.py

Created by Matt McKnett
on 2/26/07

This should be able to parse an assembly output file and find MIPS
instructions
that are not supported by our processor, causing python to return a bad
value
if it found bad instructions.
"""

import sys

# The list of invalid instructions for our MIPS processor.
# This includes the patented instructions and the floating point
instructions
# most likely to be used when other FP instructions are used.
#
# This list could be made more comprehensive.
instructs = ["lwl", "swl", "lwr", "swr", "lwcl", "swcl", "mtcl", "mfcl"]

"""
Define the checkInst method to check instructions for words in instructs
"""
def checkInst(filename):
```

```

# Open the file to be read and initialize the line counter.
myFile = open(filename)
line_num = 0
found_bad = 0

# Look through each line in the file for the instructions.
for line in myFile:
    line_num += 1
    # Look for each of the bad instructions in the line
    for op in instructs:
        # If we find it, tell the user.
        try:
            if line.split()[0] == op:
                print " Illegal MIPS assembly instruction %s " % op \
                      + "found on line %d:" % line_num
                print line.replace("\n","")
                found_bad += 1
        except IndexError:
            continue      # I don't know a better way to say "do
nothing"

    # Close the file.
myFile.close()

print " Found %d bad instructions in %s" % (found_bad, filename)

if found_bad:
    return 1
else:
    return 0

"""
Read the command line arguments as files and run checkInst on each.
"""
def readCmdLine():
    args = sys.argv[1:]
    if len(args) == 0:
        raise SystemExit("checkInstructions requires at least one file
name \
                           as input.")
    return args

# Read from the command line and initialize counters
args = readCmdLine()
fileCount = 0
badFileCount = 0

# Go through each argument and read the file.
for filename in args:
    fileCount += 1
    print "%d: Checking %s for bad instructions" % (fileCount, filename)
    badFileCount += checkInst(filename)

```

```

print "Instruction Check: %d file(s) of %d passed instruction check."\
    % (fileCount - badFileCount, fileCount)

if badFileCount == 0:
    sys.exit(0)
else:
    sys.exit(1)

```

20.5.15. Verilog Generator Script

```
"""
generateVerilog.py
```

Created 2/27/07 by Matt McKnett
HMC Spring 2007 CMOS VLSI, MIPS project

This script is designed to accept three dat files (each file composed of MIPS instructions in hex, one instruction per line) and convert them to a Verilog file that can be synthesized for the chip.

It would definitely be nice to generalize the execution of this function so that code isn't copied and pasted, but since it works, I won't touch it.

```
"""


```

```

import sys

#
# The function that takes the three files and puts them together into
output
#
def VerilogBootAndProgram(params):

    # Get pertinent values out of the parameters
    try:
        # We bit-shift the addresses by 2 because the memory controller
        treats
            # memory as word-addressable, and masks to provide byte-
        addressing.
        reset_name = params['reset_name']
        reset_loc = int(params['reset_loc'], 16) >> 2
        except_name = params['except_name']
        except_loc = int(params['except_loc'], 16) >> 2
        program_name = params['program_name']
        program_loc = int(params['program_loc'], 16) >> 2
        mem_size = int(params['mem_size'], 16) >> 2
        output_name = params['output_name']
        debug = params['debug']
        boot = params['use_boot_loader']
        Verilog_template = params['Verilog_template']
    except KeyError:
        print "generateVerilog: A needed parameter was not defined!"
```

```

# Initialize the current location and the output file.
current_loc = reset_loc

# We will construct our output in a string.
outputString = ""
caseStmtTemplate = "{1'b0, 16'h(address)]: instr <= 32'h(data);"

if boot:
    #First open the bootstrapper start file and output the lines.
    reset_file = open(reset_name, 'rU')
    for line in reset_file:
        line_data = line.replace("\n", "")
        caseStmt = caseStmtTemplate
        caseStmt = caseStmt.replace("(address)", "%x" % current_loc)
        caseStmt = caseStmt.replace("(data)", line_data)
        outputString += caseStmt

        outputString += "\n"
        current_loc += 1
    reset_file.close()

    # Make sure the reset code didn't overrun the exception code.
    offset = except_loc - current_loc
    if offset < 0:
        print " Verilog Generation: boot code exceeded available " \
              "memory region. Read %d lines from %s." %
(current_loc, reset_name)
        sys.exit(1)

    # We don't need to write any 0's in between memory blocks.
    current_loc = except_loc

    #print " Diagnostic: current_loc = %d, and boot_loc = %d (should
match)" % (current_loc, except_loc)

    # Next open the boot_loader and output its lines.
    except_file = open(except_name, 'rU')
    for line in except_file:
        line_data = line.replace("\n", "")
        caseStmt = caseStmtTemplate
        caseStmt = caseStmt.replace("(address)", "%x" % current_loc)
        caseStmt = caseStmt.replace("(data)", line_data)
        outputString += caseStmt

        outputString += "\n"
        current_loc += 1
    except_file.close()

    # Make sure the boot_loader didn't overrun the program code.
    offset = program_loc - current_loc
    if offset < 0:

```

```

        print " Verilog Generation: exception code exceeded
available memory region.\"
                " Read %d lines from %s" % (current_loc -
except_loc, except_name)

        # Again, no 0's needed between memory blocks.
        current_loc = program_loc

        #print " Diagnostic: current_loc = %d, and prog_loc = %d (should
match)" % (current_loc, program_loc)

        # Last, write the program out to the memory.
        program_file = open(program_name, 'rU')
        for line in program_file:
            line_data = line.replace("\n", "")
            caseStmt = caseStmtTemplate
            caseStmt = caseStmt.replace("(address)", "%x" % current_loc)
            caseStmt = caseStmt.replace("(data)", line_data)
            outputString += caseStmt

            outputString += "\n"
            current_loc += 1
        program_file.close()

        # Make sure the program didn't overrun the memory (leave room for a
final word of 0's)
        offset = mem_size - current_loc
        if offset < 0:
            print " Verilog Generation: The program exceeded available
memory region." \
                    " Read %d lines from %s" % (current_loc - program_loc,
program_file)

        # Now that we have constructed the replacement string, we will replace
the token
        # in the template file with that string.
        Verilog_file = open(Verilog_template, 'rU')
        Verilog_output = Verilog_file.read()
        Verilog_file.close()

        Verilog_output = Verilog_output.replace("(case_statements)",

outputString)

        output_file = open(output_name, 'wb')
        output_file.write(Verilog_output)
        output_file.close()

        print " Verilog Generation: output %d words to file %s" %
(current_loc / 4, output_name)
# End of VerilogBootAndProgram()

```

```

# Here is our main functionality

# parse the arguments
args = sys.argv[1:]

if "--help" in args or not("-params" in args and "-output" in args):
    print "Usage: python VerilogBootAndProgram.py -params (file with
parameters) \
           -program (program filename) -output (output filename) [-
debug]"
    sys.exit(0)

if "-debug" in args:
    params = {'debug': True}
else:
    params = {'debug': False}

if "-noboot" in args:
    params['use_boot_loader'] = False
else:
    params['use_boot_loader'] = True

# Continue parsing
try:
    i = args.index("-params")
    params_filename = args[i+1]
except IndexError:
    print "No parameter file was specified."
    sys.exit(1)
except ValueError:
    print "-params requires a parameter file name."
    sys.exit(1)

try:
    i = args.index("-output")
    params['output_name'] = args[i+1]
except IndexError:
    print "No output file was specified."
    sys.exit(1)
except ValueError:
    print "-output requires a file name."
    sys.exit(1)

try:
    i = args.index("-program")
    params['program_name'] = args[i+1]
except IndexError:
    print "No program file was specified."
    sys.exit(1)
except ValueError:
    print "-program requires a file name."
    sys.exit(1)

```

```

# Parse the parameter file.
params_file = open(params_filename, 'rU')
lineNum = 0
for line in params_file:
    lineNum += 1
    # Try to read the parameters from the file.
    try:
        paramsNameVal = line.replace("\n", "").split(":")
        params[paramsNameVal[0]] = paramsNameVal[1]
    except IndexError:
        print "The parameter defined in %s on line %d is invalid." %
        (params_filename, lineNum)
    params_file.close()

VerilogBootAndProgram(params)
sys.exit(0)

```

20.5.16. ROM Generator Script

```

"""
dumpBootAndProgram.py

Created 2/27/07 by Matt McKnett
HMC Spring 2007 CMOS VLSI, MIPS project

This script is designed to accept three dat files (each file composed of
MIPS instructions in hex, one instruction per line) and convert them to a
vector file simulatable in Modelsim.

In debug mode, line numbers are included.
"""

import sys

#
# The function that takes the three files and puts them together into
output
#
def dumpBootAndProgram(params):

    # Get pertinent values out of the parameters
    try:
        print params
        reset_name = params['reset_name']
        reset_loc = int(params['reset_loc'], 16)
        except_name = params['except_name']
        except_loc = int(params['except_loc'], 16)
        program_name = params['program_name']
        program_loc = int(params['program_loc'], 16)
        mem_size = int(params['mem_size'], 16)
        output_name = params['output_name']
        debug = params['debug']
    
```

```

boot = params['use_boot_loader']
except KeyError:
    print "generateDat: A needed parameter was not defined!""

# Initialize the current location and the output file.
current_loc = reset_loc
output_file = open(output_name, 'wb')

if boot:
    #First open the bootstrapper start file and output the lines.
    reset_file = open(reset_name, 'rU')
    for line in reset_file:
        if debug:
            output_file.write("%X: " % current_loc)
            output_file.write(line)
            current_loc += 4
    reset_file.close()

    # Make sure the reset code didn't overrun the exception code.
    offset = except_loc - current_loc
    if offset < 0:
        print "  Dat Generation: initial boot code exceeded available
" \
              "memory region.  Read %d lines from %s." %
(current_loc, reset_name)
        sys.exit(1)

    # Write 0's as a buffer between reset and exception.
    while current_loc < except_loc :
        if debug:
            output_file.write("%X: " % current_loc)
            output_file.write("00000000\n")
            current_loc += 4

    print "  Diagnostic: current_loc = %d, and boot_loc = %d (should
match)" % (current_loc, except_loc)

    # Next open the boot_loader and output its lines.
    except_file = open(except_name, 'rU')
    for line in except_file:
        if debug:
            output_file.write("%X: " % current_loc)
            output_file.write(line)
            current_loc += 4
    except_file.close()

    # Make sure the boot_loader didn't overrun the program code.
    offset = program_loc - current_loc
    if offset < 0:
        print "  Dat Generation: boot loader exceeded available
memory region."\
              "  Read %d lines from %s" % (current_loc -
except_loc, except_name)

```

```

# Write 0's as a buffer between the boot_loader and the program
while current_loc < program_loc :
    if debug:
        output_file.write("%X: " % current_loc)
    output_file.write("00000000\n")
    current_loc += 4

    print " Diagnostic: current_loc = %d, and prog_loc = %d (should
match)" % (current_loc, program_loc)

# Last, write the program out to the memory.
program_file = open(program_name, 'rU')
for line in program_file:
    if debug:
        output_file.write("%X: " % current_loc)
    output_file.write(line)
    current_loc += 4
program_file.close()

# Make sure the program didn't overrun the memory (leave room for a
final word of 0's)
offset = mem_size - current_loc
if offset < 0:
    print " Dat Generation: The program exceeded available memory
region.\\" \
          " Read %d lines from %s" % (current_loc - program_loc,
program_file)

# Write 0's as a buffer between the program and the end of memory
while current_loc < mem_size :
    if debug:
        output_file.write("%X: " % current_loc)
    output_file.write("00000000\n")
    current_loc += 4

output_file.close()

print " Dat Generation: output %d words to file %s" % (current_loc /
4, output_name)
# End of dumpBootAndProgram()

# Here is our main functionality

# parse the arguments
args = sys.argv[1:]

if "--help" in args or not("-params" in args and "-output" in args):
    print "Usage: python dumpBootAndProgram.py -params (file with
parameters) \

```

```

        -program (program filename) -output (output filename) [-
debug]"
    sys.exit(0)

if "-debug" in args:
    params = {'debug': True}
else:
    params = {'debug': False}

if "-noboot" in args:
    params['use_boot_loader'] = False
else:
    params['use_boot_loader'] = True

# Continue parsing
try:
    i = args.index("-params")
    params_filename = args[i+1]
except IndexError:
    print "No parameter file was specified."
    sys.exit(1)
except ValueError:
    print "-params requires a parameter file name."
    sys.exit(1)

try:
    i = args.index("-output")
    params['output_name'] = args[i+1]
except IndexError:
    print "No output file was specified."
    sys.exit(1)
except ValueError:
    print "-output requires a file name."
    sys.exit(1)

try:
    i = args.index("-program")
    params['program_name'] = args[i+1]
except IndexError:
    print "No program file was specified."
    sys.exit(1)
except ValueError:
    print "-program requires a file name."
    sys.exit(1)

# Parse the parameter file.
params_file = open(params_filename, 'rU')
lineNum = 0
for line in params_file:
    lineNum += 1
    # Try to read the parameters from the file.
    try:
        paramsNameVal = line.replace("\n", "").split(':')

```

```

        params[paramsNameVal[0]] = paramsNameVal[1]
    except IndexError:
        print "The parameter defined in %s on line %d is invalid." %
    (params_filename, lineNumber)
params_file.close()

dumpBootAndProgram(params)
sys.exit(0)

```

20.5.17. String to Charater Array Script

```

# makeCharArrayFromString.py
#
# Created by Matt McKnott
# 4/10/07 for VLSI Spring '07
#
# Turns a string literal into an array of chars for C programs.
#

import sys

def strToList(stringin):
    charList = []
    for c in stringin:
        charList.append(c)
    return charList

strCnt = 0

print "Don't forget to add null terminators to strings! \
      An extra char is included in the array for this purpose.\n\n"

for param in sys.stdin:
    # Don't interpret newlines.
    outputString = param.replace("\n","").replace("\r","");
    # Make the length include null terminators we don't use, and don't
    # forget
    # that we must declare one greater than the length.
    strLength = len(outputString) + 2

    # Turn the string into a string list.
    outputList = strToList(outputString)

    # Add the null terminator to the list. Python likes to print \\0 and
    # can't make it stop.
    #outputList.append("\x5C" + "0")

    outputString = str(outputList)

    # Change [ to {, ] to }
    outputString = outputString.replace('[','{').replace(']','}')

```

```

    outputString = "char str%d[%d] = " % (strCnt, strLength) +
outputString
    outputString += ";""
print outputString
strCnt += 1

```

20.5.18. Toolchain Memory Specifications

```

reset_loc:0x00000000
reset_name:boot_start.dat
except_loc:0x00000100
except_name:boot_loader.dat
program_loc:0x00000200
mem_size:0x00016A7C
Verilog_template:Verilog_template.txt

```

20.5.19. Verilog Template

```

`timescale 1ns / 1ps
///////////////////////////////
/////
// HMC-MIPS Auto-generated code
////
///////////////////////////////
module ROM(clock, en, address, instr);
    input clock;
    input en;
    input [16:0] address;
    output reg [31:0] instr;

    always @(posedge clock)
        if(en)
            begin
                case(address)
                    (case_statements)
                    default: instr <= 32'h00000000;
                endcase
            end
endmodule

```

20.5.20. Makefile

```

# Tell where various toolchain utilities exist.
CC = ~/yoda_warrior/bin/mips-cygwin-elf-gcc-4.1.1
AS = ~/yoda_warrior/bin/mips-cygwin-elf-as.exe
LD = ~/yoda_warrior/bin/mips-cygwin-elf-ld.exe
DUMP = ~/yoda_warrior/bin/mips-cygwin-elf-objdump.exe

```

```

# Where the source directories are
BLD = ./build
SRC = ./
INCLUDEFILES = boot.mk test_leds.mk test_lcd.mk lightsOut.mk dhrystone.mk
test_buttons.mk test_simple.mk

# These variables control the build process.
CFLAGS = -Wall -msoft-float -march=r3000
LDFLAGS = -Tdata 0x80016A80
DEBUG = -nodebug
BOOTLOAD = true
MEM_SPECS = toolchain_memory_specs.txt
RESET_LOC = 0xbfc00000
BOOT_LOC = 0xbfc00100
PROG_LOC = 0x9fc00200

# Build all of the main programs in the src folder
all: lightsOut.v dhrystone.v test_simple.v test_leds.v test_lcd.v
test_buttons.v

# This line prevents make from automatically deleting these files as
temporary
.PRECIOUS: %.dat %.dump %.o %.s %.asm

include $(INCLUDEFILES)

%.asm: %.c
    $(CC) $(CFLAGS) -S $< -o $@

%.o: %.asm
    python checkInstructions.py $<
    $(AS) -o $@ $<

%.out: %.o
    $(LD) $(LDFLAGS) -Ttext=$(PROG_LOC) -o $@ $<

%.dump: %.out
    $(DUMP) -d --disassemble-zeroes $< > $@

# This magic grep command strips out the instruction hex from the objdump
# output.

%.dat: %.dump
    cat $< | grep --only-matching "^\ * [0-9a-fA-F] \+: [^0-9a-fA-F]* [0-9a-
fA-F] \+" | grep --only-matching ":[^0-9a-fA-F]* [0-9a-fA-F] \+" | tr -d " "
    | grep --only-matching "[0-9a-fA-F] \{8\}" > $@

# If we want to generate a Verilog file using the bootloader, we have
different
# options than if we want to generate it without.

```

```

%.v: %.dat boot_loader.dat boot_start.dat
ifeq ($(BOOTLOAD),true)
    python generateVerilog.py $(DEBUG) -params $(MEM_SPECS) -program $<
-output $@
    cp $@ ROM.v
else
    python generateVerilog.py $(DEBUG) -noboot -params $(MEM_SPECS) -
program $< -output $@
    cp $@ ROM.v
endif
@echo ""
@echo "    Verilog file generated successfully!"
@echo "    It is a good idea to inspect $*.dump before continuing
further to ensure that the program's main routine is the first symbol
(address 0x9fc01000)."

%.rom: %.dat boot_loader.dat boot_start.dat
ifeq ($(BOOTLOAD), true)
    python generateROM.py $(DEBUG) -params $(MEM_SPECS) -program $< -
output $@
else
    python generateROM.py $(DEBUG) -noboot -params $(MEM_SPECS) -program
$< -output $@
endif

clean:
    rm -f *.o *.out *.dump *.coe *.dat
    rm -f muddCLib/*.o

# Assuming that clean- is the prefix for any cleaning rule defined in the
# included makefiles, this rule will use it.
clean-all: $(patsubst %.mk, clean-%, $(INCLUDEFILES))
    rm -f *.rom *.v
    make clean

```

20.5.21. Template Makefile

```

# template.mk
#
# The makefile responsible for making my program

# Just replace 'myProgram' with the name of your program's c file. So, if
you
# have 'helloWorld.c', make this be 'helloWorld'.
MYPROGRAM=myProgram

# This might be nothing, so delete 'myProgram.h' if you don't have a
header file
MYPROGRAMHEADER=myProgram.h

```

```

MYPROGFILES = $(MYPROGRAMHEADER) $(SRC)/muddCLib/muddCLib.h boot_start.dat
boot_loader.dat

$(MYPROGRAM).asm: $(MYPROGRAM).c $(MYPROGFILES)
    $(CC) $(CFLAGS) -O2 -S $< -o $@

$(MYPROGRAM).o: $(MYPROGRAM).asm
    python $(SRC)/checkInstructions.py $<
    $(AS) -o $@ $<

$(MYPROGRAM).out: $(MYPROGRAM).o $(SRC)/muddCLib/muddCLib.o
    $(LD) $(LDFLAGS) -Ttext=$(PROG_LOC) -o $@ $^

clean-$(MYPROGRAM):
    rm -f $(MYPROGRAM).asm

```

20.5.22. Boot Makefile

```

# boot.mk
#
# This defines the dependencies for the boot_start and boot_load files.
#

boot_start.out: boot_start.o
    $(LD) $(LDFLAGS) -Ttext=$(RESET_LOC) -o $@ $<

boot_loader.out: boot_loader.o
    $(LD) $(LDFLAGS) -Ttext=$(BOOT_LOC) -o $@ $<

clean-boot:

```

20.5.23. Dhrystone Makefile

```

# dhrystone.mk
#
# The makefile responsible for making dhrystone.

DHRYFILES = dhry.h $(SRC)/muddCLib/muddCLib.h boot_start.dat
boot_loader.dat

dhystone.asm: dhystone.c $(DHRYFILES)
    $(CC) $(CFLAGS) -O2 -S $< -o $@

dhryFuncs.asm: dhryFuncs.c $(DHRYFILES)
    $(CC) $(CFLAGS) -O2 -S $< -o $@

dhystone.o: dhystone.asm
    python $(SRC)/checkInstructions.py $<
    $(AS) -o $@ $<

```

```

dhryFuncs.o: dhryFuncs.asm
    python $(SRC)/checkInstructions.py $<
    $(AS) -o $@ $<

dhystone.out: dhystone.o dhryFuncs.o $(SRC)/muddCLib/muddCLib.o
    $(LD) $(LDFLAGS) -Ttext=$(PROG_LOC) -o $@ $^

clean-dhystone:
    rm -f dhryFuncs.asm dhystone.asm

```

20.5.24. Lights Out! Makefile

```

# lightsOut.mk
#
# Defines the dependencies for lightsOut

lightsOut.asm: lightsOut.c lightsOut.h muddCLib/muddCLib.h
muddCLib/mtRand.h
    $(CC) $(CFLAGS) -S lightsOut.c -o lightsOut.asm

lightsOut.o: lightsOut.asm
    python checkInstructions.py $<
    $(AS) -o $@ $<

lightsOut.out: lightsOut.o muddCLib/muddCLib.o muddCLib/mtRand.o
    $(LD) $(LDFLAGS) -Ttext=$(PROG_LOC) -o lightsOut.out lightsOut.o
muddCLib/muddCLib.o muddCLib/mtRand.o

lightsOut.exe: lightsOut.c lightsOut.h muddCLib/muddCLib.h
muddCLib/mtRand.h
    gcc -o lightsOut lightsOut.c muddCLib/muddCLib.c muddCLib/mtRand.c

clean-lightsOut:
    rm -f lightsOut.asm lightsOut.exe

```

20.5.25. Corner-Case Test Makefile

```

# mips_corner_test.mk
#
# The makefile responsible for making mips_corner_test

# Just replace 'myProgram' with the name of your program's c file. So, if
you
# have 'helloWorld.c', make this be 'helloWorld'.
MYPROGRAM=mips_corner_test

# This might be nothing, so delete 'myProgram.h' if you don't have a
header file
MYPROGRAMHEADER=

```

```

MYPROGFILES = $(MYPROGRAMHEADER) $(SRC)/muddCLib/muddCLib.h boot_start.dat
boot_loader.dat

mips_corner_test.asm: mips_corner_test.c $(MYPROGFILES)
    $(CC) $(CFLAGS) -O2 -S $< -o $@

mips_corner_test.o: mips_corner_test.asm
    python $(SRC)/checkInstructions.py $<
    $(AS) -o $@ $<

mips_corner_test.out: mips_corner_test.o $(SRC)/muddCLib/muddCLib.o
    $(LD) $(LDFLAGS) -Ttext=$(PROG_LOC) -o $@ $^

clean-mips_corner_test:
    rm -f mips_corner_test.asm

```

20.5.26. Button Test Makefile

```

# test_buttons.mk
#
# Defines the dependencies for making test_buttons

test_buttons.asm: test_buttons.c muddCLib/muddCLib.h boot_start.dat
boot_loader.dat
    $(CC) $(CFLAGS) -O2 -S $< -o $@

test_buttons.o: test_buttons.asm
    python checkInstructions.py $<
    $(AS) -o $@ $<

test_buttons.out: test_buttons.o muddCLib/muddCLib.o
    $(LD) $(LDFLAGS) -Ttext=$(PROG_LOC) -o $@ $^

clean-test_buttons:
    rm -f test_buttons.asm

```

20.5.27. LCD Test Makefile

```

# test_lcd.mk
#
# Defines the dependencies for making test_lcd

test_lcd.asm: test_lcd.c $(SRC)/muddCLib/muddCLib.h boot_start.dat
boot_loader.dat
    $(CC) $(CFLAGS) -O2 -S $< -o $@

test_lcd.o: test_lcd.asm
    python $(SRC)/checkInstructions.py $<
    $(AS) -o $@ $<

test_lcd.out: test_lcd.o $(SRC)/muddCLib/muddCLib.o
    $(LD) $(LDFLAGS) -Ttext=$(PROG_LOC) -o $@ $^

```

```
clean-test_lcd:
    rm -f test_lcd.asm
```

20.5.28. LED Test Makefile

```
# test_leds.mk
#
# Gives the dependencies for building test_leds

test_leds.asm: test_leds.c muddCLib/muddCLib.h boot_start.dat
boot_loader.dat
    $(CC) $(CFLAGS) -S test_leds.c -o test_leds.asm

test_eds.o: test_leds.asm
    python checkInstructions.py $<
    $(AS) -o $@ $<

test_leds.out: test_leds.o muddCLib/muddCLib.o
    $(LD) $(LDFLAGS) -Ttext=$(PROG_LOC) -o test_leds.out test_leds.o
muddCLib/muddCLib.o

clean-test_leds:
    rm -f test_leds.asm
```

20.5.29. Simple Test Makefile

```
# test_simple.mk
#
# The makefile responsible for making my program

MYPROGFILES = $(SRC)/muddCLib/muddCLib.h boot_start.dat boot_loader.dat

test_simple.asm: test_simple.c $(MYPROGFILES)
    $(CC) $(CFLAGS) -O2 -S $< -o $@

test_simple.o: test_simple.asm
    python $(SRC)/checkInstructions.py $<
    $(AS) -o $@ $<

test_simple.out: test_simple.o $(SRC)/muddCLib/muddCLib.o
    $(LD) $(LDFLAGS) -Ttext=$(PROG_LOC) -o $@ $^

clean-test_simple:
    rm -f test_simple.asm
```

20.6. Lessons Learned and Experience Gained

The compiler and benchmarks portion of the Systems Cluster deliverables were plagued by software deficiencies. My initial attempts to put together a working compiler myself were fruitless, and when I received the compiler we use now (yoda_warrior), I discovered it had no library functions. The combination of lacking support in hardware for instructions and in software for library files paired with my own lack of knowledge in the field of compilers contributed to the piecemeal, hacky nature of our compiler toolchain. If I had the skills and opportunity, I would build GCC for this project with some proper library files, probably ucLibc, and I would patch the compiler so that it did not try to use the unaligned load and store instructions. I would also try to write a linker script and modify the boot loader to be more general, not requiring that we have the main function of the program being loaded start at a specific address. I would also make sure we got ASCII data into the ROM so that string literals could be used.

As much as the build process is ugly, though, it works. We can compile substantial programs that run on the board as desired. The compiler toolchain is probably not ready to crunch on Linux or something like that yet, but we can run the programs that we specified we needed to. More personally, I found from this work that the most important thing to do in group projects is communicate. Tell people about your status often. If something is going to be late, or if you are having trouble, or if it seems infeasible, talk to the team leaders and project manager. Deadlines on these projects are different from most regular school deadlines – not harder or softer, but of a different nature. We were not working according to the times that the professor laid out and set in stone, but we were working with and for each other. I had a hard time grasping that.

21. FPGA EMULATION

Owner: Eddy Chavarria

Revision: 2

Date: April 25, 2007

21.1. Introduction

In order to verify the Verilog generated by the μ Architecture team, the Systems Cluster was responsible for synthesizing and emulating the chip onto the Xilinx-II Pro XC2VP30 FPGA. There are two approaches to emulation, one involves synthesizing both the μ Processor and external memory system onto a single FPGA. Another option is to use two boards, and emulate the processor and memory system separately, having them communicate over the two low speed expansion headers available on the boards. The second option saves time because you will not be synthesizing both the chip and the memory system every time you make a change in the code.

21.2. External Clock Modification

Before you can run emulation in any mode, you need to make one modification to the Virtex II Pro FPGA. An external clock needs to be connected to the FPGA in order to allow for a quick and easy way to adjust the clock that the processor is running at. If you choose not to perform this modification, the processor will still run but only on a 32MHz internal clock. Currently, the external memory module takes in a clock and divides the clock frequency by four to produce a two phase non-overlapping clock.

To perform this modification, simply solder two pins (clock and ground) according to the Figure shown below.

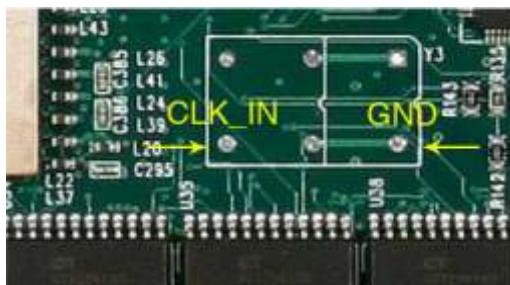


Figure 111. External clock modification on the Virtex II Pro.

21.3. Single FPGA Emulation

In Xilinx, start a new project, name it something like MIPS_Emulation. Next, set the device properties as follows:

Product Category	All
Family	Virtex2P
Device	XC2VP30
Package	FF896
Speed	-7
Top-Level Source Type	HDL Synplify Pro
Synthesis Tool	(VHDL/Verilog)
Simulator	Modelsim-SE Verilog

Next, Xilinx will ask you if you wish to create new source files, do not create any source files, simply click next. The next screen will ask you to add existing files, when prompted too, add the following source files to this project:

- clkgen8x.v
- extmem.v
- memcon.v
- multdiv.v
- ROM.v
- components.v
- mem.v
- mipspipelined.v
- RAM.v
- top.v

All the files required for single FPGA emulation are available on the included project DVD under Systems\Single_FPGA.

Once the files are added, click next and then finish. Hit ok on the next screen.

The next step is to assign package pins, there is a pre-generated user constraints file available on the DVD folder, but the following steps need to be taken for it to work.

First, make sure the top file is highlighted in the sources tab in Xilinx. Under processes, expand the User Constraints tab and double click assign package pins. Without actually assigning anything, click save and ok to the default settings. Close the assign package pins window. Next, browse to your project folder, a .ucf file has been created for your project. Replace this file with the one from the DVD, this has all the pins already properly assigned. Verify this by going back to Xilinx and going to Assign Package Pins again, all the pins should now be properly assigned. The ucf file should look as follows.

```

#PACE: Start of Constraints generated by PACE

#PACE: Start of PACE I/O Pin Assignments
NET "bp_data_in[1]" LOC = "AH4" | IOSTANDARD = LVTTL ;
NET "bp_data_in[2]" LOC = "AG3" | IOSTANDARD = LVTTL ;
NET "bp_data_in[3]" LOC = "AH1" | IOSTANDARD = LVTTL ;
NET "bp_data_in[4]" LOC = "AH2" | IOSTANDARD = LVTTL ;
NET "clk" LOC = "AH16" | IOSTANDARD = LVCMOS25 ;
NET "dp_data[0]" LOC = "AC11" | IOSTANDARD = LVCMOS25 ;
NET "dp_data[1]" LOC = "AD11" | IOSTANDARD = LVCMOS25 ;
NET "dp_data[2]" LOC = "AF8" | IOSTANDARD = LVCMOS25 ;
NET "dp_data[3]" LOC = "AF9" | IOSTANDARD = LVCMOS25 ;
NET "LCD_en" LOC = "U2" | IOSTANDARD = LVTTL ;
NET "LED_data_out[0]" LOC = "AC4" | IOSTANDARD = LVTTL ;
NET "LED_data_out[1]" LOC = "AC3" | IOSTANDARD = LVTTL ;
NET "LED_data_out[2]" LOC = "AA6" | IOSTANDARD = LVTTL ;
NET "LED_data_out[3]" LOC = "AA5" | IOSTANDARD = LVTTL ;
NET "resetb" LOC = "AG5" | IOSTANDARD = LVTTL ;
NET "writedata[0]" LOC = "N6" | IOSTANDARD = LVTTL ;
NET "writedata[1]" LOC = "N5" | IOSTANDARD = LVTTL ;
NET "writedata[2]" LOC = "L5" | IOSTANDARD = LVTTL ;
NET "writedata[3]" LOC = "L4" | IOSTANDARD = LVTTL ;
NET "writedata[4]" LOC = "M2" | IOSTANDARD = LVTTL ;
NET "writedata[5]" LOC = "N2" | IOSTANDARD = LVTTL ;
NET "writedata[6]" LOC = "P9" | IOSTANDARD = LVTTL ;
NET "writedata[7]" LOC = "R9" | IOSTANDARD = LVTTL ;
NET "writedata[8]" LOC = "M4" | IOSTANDARD = LVTTL ;

#PACE: Start of PACE Area Constraints

#PACE: Start of PACE Prohibit Constraints

#PACE: End of Constraints generated by PACE

```

Once that is completed, you are ready to load the external memory onto the board. Run Implement Design followed by Generate Programming File. Once completed, double click on Generate PROM, ACE, or JTAG File. On the first screen, make sure “Configure devices using Boundary-Scan (JTAG)” is selected and click finish. The program will try to find the board but will not be able too. Click ok on the following two error messages. Now right click on the main window. On the menu that pops up, select “Cable Set Up.” Select “Platform USB cable from the Communication Mode menu, leave every thing else defaulted, and click ok. Right click the main window again, this time select “Initialize Chain.” The main screen should read Identify Completed and a menu asking you to assign a configuration file will pop up. Select bypass. Another menu will pop up, select bypass again. On the third menu, you should see a .bit file with the name of your top module in the menu. Select it and click open. Click ok on the following two pop ups. Now you should see 3 Xilinx chips on the main screen, with the first two reading bypass under them and the last one “xc2vp30” with your .bit file name under that, right click that icon and select program. On the next menu make sure verify is not selected, and select ok, this will now download the .bit file to the board. You should see the done bit go high on the board and the main menu read “Programming Succeeded.” You can probe some of the outputs to make sure everything is running properly. The clock source for the board should be coming from an

external, variable clock. The clock that is input to the system will go into the clkgen module. This module divides the clock into four states, in order to generate the appropriate Ph1 and Ph2 waveforms. Depending on what ROM.v file is being used, you can test the pushbuttons and LED's available on the board.

In order to change the program that is loaded in ROM, one must copy and paste all the new ROM code and replace the existing one in the project. This requires that everything be re-synthesized, which can be very tedious. Replacing the ROM should be the only change needed to load a new program. This step is much faster when the external memory is separated from the chip, which we will cover in the next section.

21.4. Dual FPGA Emulation

In order to prepare the Xilinx Virtex-II Pro FPGA for dual Emulation, be sure to have the following files in the project directory.

- clkgen8x.v
- extmem.v
- memcon.v
- multdiv.v
- ROM.v
- components.v
- mem.v
- mipspipelined.v
- RAM.v
- top.v

These files can be found on the DVD under Systems/Dual_FPGA

The boards should be connected across the two 40 pin expansion headers. The cable utilized during the project is available from Digikey, part number H3KKH-4006G-ND. This is a 40 pin flat IDC cable, with pins on a 2.5mm pitch. The only issue here is to insure that the power pins on each of the corresponding boards are not connected. These are pins 2 and 3 on both the J5 and J6 expansion headers. The last pin, 40, on the J5 expansion is not connected. The final pins on the J6 header, 36-40, should not be connected. These pins are reserved for the FPGA and are not used for board to board communication. Be sure that all unused pins are disconnected. *This step is very important in ensuring the safety and proper functionality of the two boards. Be sure that the boards share a common ground between each other, across the 1st pin on both headers.*

The external memory system and processor must be split. In Xilinx, start a new project, name it something like EXTMEM. Next, set the device properties as follows:

Product Category	All
Family	Virtex2P
Device	XC2VP30
Package	FF896
Speed	

Top-Level Source	
Type	HDL
Synthesis Tool	Synplify Pro (VHDL/Verilog)
Simulator	Modelsim-SE Verilog

Next, Xilinx will ask you if you wish to create new source files, do not create any source files, simply click next. The next screen will ask you to add existing files, when prompted too, add the following source files to this project: Clkgen8x.v, extmem.v, memcon.v, ROM.v, and RAM.v. Once the files are added, click next and then finish. Hit ok on the next screen.

Extmem.v should be the top file in the project. It is responsible for communicating with the processor. The port list should look like the following:

```
module extmem_top(clk,resetloc,ph1, ph2, en, rwb, reset, address, byteen,
data, dp_data, bp_data, done, LED_data, LCD_en,interrupts);

    input clk;
    input resetloc;
    output ph1;
    output ph2;
    input en;
    input rwb;
    output reset;
    input [16:0] address;
    input [3:0] byteen;
    inout [31:0] data;
    input [3:0] dp_data;
    input [3:0] bp_data;
    output done;
    output [3:0] LED_data;
    output LCD_en;
    output [7:0] interrupts;
```

Ensure that this the port list in your version of extmem.v, or else you might be using the wrong version. The next step, assigning package pins, is crucial.

There is a pre-generated user constraints file available on the DVD in the Dual FPGA emulation folder, one for the external memory and one for the processor. The following steps need to be taken for it to work.

First, make sure the top file is highlighted in the sources tab in Xilinx. Under processes, expand the User Constraints tab and double click assign package pins. Without actually assigning anything, click save and ok to the default settings. Close the assign package pins window. Next, browse to your project folder, a .ucf file has been created for your project. Replace this file with the one from the DVD, this has all the pins already properly assigned. Verify this by going back to Xilinx and going to Assign Package Pins again, all the pins should now be properly assigned. The pin outs are as follows:

address[0]	Input	U4	LVTTL
address[1]	Input	T8	LVTTL
address[2]	Input	V2	LVTTL
address[3]	Input	U5	LVTTL
address[4]	Input	T9	LVTTL
address[5]	Input	W2	LVTTL
address[6]	Input	V3	LVTTL
address[7]	Input	U9	LVTTL
address[8]	Input	W1	LVTTL
address[9]	Input	V4	LVTTL
address[10]	Input	U7	LVTTL
address[11]	Input	Y1	LVTTL
address[12]	Input	V5	LVTTL
address[13]	Input	U8	LVTTL
address[14]	Input	Y2	LVTTL
address[15]	Input	V6	LVTTL
address[16]	Input	V7	LVTTL
bp_data[0]	Input	AH4	LVTTL
bp_data[1]	Input	AG3	LVTTL
bp_data[2]	Input	AH1	LVTTL
bp_data[3]	Input	AH2	LVTTL
byteen[0]	Input	AA3	LVTTL
byteen[1]	Input	W6	LVTTL
byteen[2]	Input	W7	LVTTL
byteen[3]	Input	Y5	LVTTL
clk	Input	AH15	LVCMOS25
data[0]	InOut	N6	LVTTL
data[1]	InOut	N5	LVTTL
data[2]	InOut	L5	LVTTL
data[3]	InOut	L4	LVTTL
data[4]	InOut	M2	LVTTL
data[5]	InOut	N2	LVTTL
data[6]	InOut	P9	LVTTL
data[7]	InOut	R9	LVTTL
data[8]	InOut	M4	LVTTL
data[9]	InOut	M3	LVTTL
data[10]	InOut	N1	LVTTL
data[11]	InOut	P1	LVTTL
data[12]	InOut	P8	LVTTL
data[13]	InOut	P7	LVTTL
data[14]	InOut	N4	LVTTL
data[15]	InOut	N3	LVTTL
data[16]	InOut	P3	LVTTL
data[17]	InOut	P2	LVTTL
data[18]	InOut	R8	LVTTL
data[19]	InOut	R7	LVTTL
data[20]	InOut	P5	LVTTL
data[21]	InOut	P4	LVTTL
data[22]	InOut	R2	LVTTL
data[23]	InOut	T2	LVTTL
data[24]	InOut	R6	LVTTL
data[25]	InOut	R5	LVTTL
data[26]	InOut	R4	LVTTL
data[27]	InOut	R3	LVTTL
data[28]	InOut	U1	LVTTL
data[29]	InOut	V1	LVTTL
data[30]	InOut	T5	LVTTL
data[31]	InOut	T6	LVTTL
done	Output	T4	LVTTL
dp_data[0]	Input	AC11	LVCMOS25
dp_data[1]	Input	AD11	LVCMOS25
dp_data[2]	Input	AF8	LVCMOS25
dp_data[3]	Input	AF9	LVCMOS25
en	Input	AA4	LVTTL
interrupts[0]	Output	W3	LVTTL
interrupts[1]	Output	AA2	LVTTL
interrupts[2]	Output	AA1	LVTTL
interrupts[3]	Output	V8	LVTTL
interrupts[4]	Output	W5	LVTTL
interrupts[5]	Output	W4	LVTTL
interrupts[6]	Output	Y4	LVTTL
interrupts[7]	Output	AB1	LVTTL
LCD_en	Output	U2	LVTTL
LED_data[0]	Output	AC4	LVTTL
LED_data[1]	Output	AC3	LVTTL
LED_data[2]	Output	AA6	LVTTL
LED_data[3]	Output	AA5	LVTTL
ph1	Output	U3	LVTTL
ph2	Output	AB3	LVTTL
reset	Output	T3	LVTTL
resetloc	Input	AG5	LVTTL
rwb	Input	W8	LVTTL

Table 16. External Memory Pin Out for Dual FPGA Emulation

Once that is completed, you are ready to load the external memory onto the board. Run Implement Design followed by Generate Programming File. Once completed, double click on Generate PROM, ACE, or JTAG File. On the first screen, make sure “Configure devices using Boundary-Scan (JTAG)” is selected and click finish. The program will try to find the board but

will not be able too. Click ok on the following two error messages. Now right click on the main window. On the menu that pops up, select “Cable Set Up.” Select “Platform USB cable from the Communication Mode menu, leave every thing else defaulted, and click ok. Right click the main window again, this time select “Initialize Chain.” The main screen should read Identify Completed and a menu asking you to assign a configuration file will pop up. Select bypass. Another menu will pop up, select bypass again. On the third menu, you should see a .bit file with the name of your top module in the menu. Select it and click open. Click ok on the following two pop ups. Now you should see 3 Xilinx chips on the main screen, with the first two reading bypass under them and the last one “xc2vp30” with your .bit file name under that, right click that icon and select program. On the next menu make sure verify is not selected, and select ok, this will now download the .bit file to the board. You should see the done bit go high on the board and the main menu read “Programming Succeeded.” Probe some of the outputs, such as the clock outputs, to make sure that the board is operating correctly.

Now create a new project, this time calling it something like MIPS or Processor. Again, do not create any source files. When prompted, add the following files to the project: multdiv.v, components.v, mem.v, mipspipelined.v, top.v. Top should be the top file in the project. The port list on top should look like the following:

```
module top(ph1, ph2, memen, memrwb, reset, memadr, membyteen, memdata,
           memdone,interrupts,MLEDS);

  input ph1,ph2,reset,memdone;
  input [7:0] interrupts;
  inout [31:0] memdata;
  output [16:0] memadr;
  output memrwb, memen;
  output [3:0] membyteen;
  output [3:0] MLEDS;
```

There should not be a need to change any Verilog. Now assign package pins. Follow the same procedure used in assigning to the external memory. Make sure to grab the right .ucf file from the DVD under dual emulation. The pin outs should look like the following when the constraints are applied:

interrupts[0]	Input	W3	LVTTL
interrupts[1]	Input	AA2	LVTTL
interrupts[2]	Input	AA1	LVTTL
interrupts[3]	Input	V8	LVTTL
interrupts[4]	Input	W5	LVTTL
interrupts[5]	Input	W4	LVTTL
interrupts[6]	Input	Y4	LVTTL
interrupts[7]	Input	AB1	LVTTL
memadr[0]	Output	U4	LVTTL
memadr[1]	Output	T8	LVTTL
memadr[2]	Output	V2	LVTTL
memadr[3]	Output	U5	LVTTL
memadr[4]	Output	T9	LVTTL
memadr[5]	Output	W2	LVTTL
memadr[6]	Output	V3	LVTTL
memadr[7]	Output	U9	LVTTL
memadr[8]	Output	W1	LVTTL
memadr[9]	Output	V4	LVTTL
memadr[10]	Output	U7	LVTTL
memadr[11]	Output	Y1	LVTTL
memadr[12]	Output	V5	LVTTL
memadr[13]	Output	U8	LVTTL
memadr[14]	Output	Y2	LVTTL
memadr[15]	Output	V6	LVTTL
memadr[16]	Output	V7	LVTTL
membyteen[0]	Output	AA3	LVTTL
membyteen[1]	Output	W6	LVTTL
membyteen[2]	Output	W7	LVTTL
membyteen[3]	Output	Y5	LVTTL
memdata[0]	InOut	N6	LVTTL
memdata[1]	InOut	N5	LVTTL
memdata[2]	InOut	L5	LVTTL
memdata[3]	InOut	L4	LVTTL
memdata[4]	InOut	M2	LVTTL
memdata[5]	InOut	N2	LVTTL
memdata[6]	InOut	P9	LVTTL
memdata[7]	InOut	R9	LVTTL
memdata[8]	InOut	M4	LVTTL
memdata[9]	InOut	M3	LVTTL
memdata[10]	InOut	N1	LVTTL
memdata[11]	InOut	P1	LVTTL
memdata[12]	InOut	P8	LVTTL
memdata[13]	InOut	P7	LVTTL
memdata[14]	InOut	N4	LVTTL
memdata[15]	InOut	N3	LVTTL
memdata[16]	InOut	P3	LVTTL
memdata[17]	InOut	P2	LVTTL
memdata[18]	InOut	R8	LVTTL
memdata[19]	InOut	R7	LVTTL
memdata[20]	InOut	P5	LVTTL
memdata[21]	InOut	P4	LVTTL
memdata[22]	InOut	R2	LVTTL
memdata[23]	InOut	T2	LVTTL
memdata[24]	InOut	R6	LVTTL
memdata[25]	InOut	R5	LVTTL
memdata[26]	InOut	R4	LVTTL
memdata[27]	InOut	R3	LVTTL
memdata[28]	InOut	U1	LVTTL
memdata[29]	InOut	V1	LVTTL
memdata[30]	InOut	T5	LVTTL
memdata[31]	InOut	T6	LVTTL
memdone	Input	T4	LVTTL
menen	Output	AA4	LVTTL
memrwb	Output	W8	LVTTL
ph1	Input	U3	LVTTL
ph2	Input	AB3	LVTTL
reset	Input	T3	LVTTL

Table 17. MIPs Pin Outs for Dual FPGA Emulation.

After assigning the pins, run Synthesize, Implement Design, and Generate Programming File. Once completed, click on Generate PROM, ACE, or JTAG File and follow the same instructions used in programming the external memory.

21.5. Test Results

Dual emulation was operational using the test_leds ROM.v. During emulation, a user was able to move the dipswitches to select which corresponding LED would turn on or off. The program was resettable and stable for some time. There was a very important issue that came up in our Dual emulation set up. The ground pin going from board to board had been disconnected when attempting to disconnect the power pins on the expansion headers. This meant that the boards were not sharing a common ground to each other. This made the system very susceptible to

noise. One observation that was made was that when many of the signals were changing, other signals were influenced by it. Long programs were never able to execute because the noise would disturb the clocks or sometimes pull the reset high, causing the program to malfunction. The noise event was observed using an analog probe. We verified this was the root of the problem by running a ground wire directly across the two boards, after which they were more stable. Noise aside, dual emulation further verified the functionality of the external memory and processor. Dhystone behaved the same way under dual emulation as in single FPGA emulation.

21.6. PCB Testing

The emulation system was also used to verify that the PCB was designed correctly. The first test was to verify the LCD. This was done by connecting the PCB to the Virtex board, and outputting the data and control signals that go to the LCD. The ucf file provided for single emulation is already set up to output the correct signals to the LCD. The finished processor will eventually go on the PCB, which means that it could be used in lieu of the processor FPGA used in dual emulation. The pins on the PCB were tested by using the PCB as a bridge between the two boards. Noise became an issue and only short programs were able to complete, which still verified that the connections were correct. One thing to keep in mind once the chip is ready is that the reset on the system currently comes from the external memory system. In a real implementation, the reset will be coming from the PCB, so the pin out on the external memory should be changed accordingly.

21.7. Lessons Learned

I felt like working on the systems cluster provided some insight into what one might expect in industry. We were constantly on our toes, and time management became a huge issue. At the beginning of the class I was familiar with the idea of a memory mapped I/O system, but never thought about how to implement it. Using the built in Ram and Rom generating capabilities of Xilinx became tedious, so I had to look into how to generate those two modules in Verilog, which was kind of straight forward. The next step was to design the controller for the external memory system, which relied on the processor specs provided to us by the μ Architecture team. We went through many iterations of our memory system in order to get it right. I learned how to use the tools available to us, such as the chip RTL and waveforms, to debug a complicated system like the one we had. I spent a lot of time trying to figure out why the dual emulation system worked one day and stopped working the next. This was real world insight and made me see that things are not always ideal and that the way things are set up have a huge influence on the performance of this type of system. This course also helped me further develop my team dynamic skills. We all had to find the best time to meet, and often compromises had to be made. I felt like we supported each other rather well when someone got stuck. I learned to be more patient and accept that some things are out of my control. In general I learned a great deal in this class, even if I did not get a chance to do much VLSI, I had the opportunity to work with a large team of talented individual taking on a real world project.

22. Printed Circuit Board

Owner: Bart Oegema

Revision: 2

Date: April 26, 2007

The Systems cluster has designed a Printed Circuit Board (PCB) on which to mount the MIPS microprocessor. The PCB provides an interface between the PGA package of the manufactured microprocessor chip and its necessary power supplies, as well as providing connection between the I/O pins of the microprocessor and the interface provided on the Xilinx Virtex-II Pro Development Board. Details of the designed PCB are as follows. CAD files of the design are also provided in the software portion of this documentation.

22.1. PCB Features

The PCB is approximately 3.125 in. by 4.625 in in dimension. It contains four copper layers, with the inner two layers reserved as power planes (a common ground and 3.3VDC). Below is a brief description of the features of the PCB, touching on the main subsections of its design.

22.1.1. PGA Socket

The PCB is centered around the MIPS processor, and provides the necessary inputs and outputs from the chip to/from system peripherals, most notably the Virtex-II FPGA operating as system memory subsystem. As previously specified, the MIPS processor will be encased in a 108-pin pin grid array (PGA) package, as provided by MOSIS. Accordingly, the PCB subsystem will include a 108-pin PGA socket to mount the processor. The socket initially selected is a molded through-hole PGA socket, with gold contact plating to interface with the PGA IC package, and tin/lead pin plating for the through-hole pins.

22.1.2. Power

Power signals is provided to the PCB via a 2.1 mm male DC jack. A power switch gates current immediately after the DC power jack. The supplied power is regulated down to 5VDC and 3.3VDC with on-board voltage regulators, which are rated up to a 15VDC input. It is recommended that input power be driven at 9VDC, to provide sufficient driving voltage for the regulators but keep a sufficient safety margin from the maximum voltage rating of said regulators. The entire PCB shares a common ground network as one of the internal power planes, which is also bridged to/from the XUPV2P Development Board through the connective headers to maximize signal integrity. The remaining internal power plane runs 3.3VDC to the entire board. 5VDC is used solely to power the LCD display, and as such is not provided on a power plane.

Two green status LEDs will display power is supplied to the board through both power nets, as well as to drain residual charge on the power networks. Bypass capacitors of 0.1 and 1, 0.1, and 0.01 μ F will be onboard before and after, respectively, each voltage regulator to stabilize the power signals.

22.1.3. Miscellaneous Features

A Reset signal is connected to the MIPS processor, with a pushbutton switch to drive the appropriate IC pin. Reset also is connected to the XUPV2P Development Board through the connecting headers.

Both the FPGA and MIPS processor require two clock signals, PH1 and PH2. These clock signals are generated in the FPGA and output to the MIPS processor on the PCB. This enables flexibility in clock timing, a particularly useful thing when the team is not sure of exact capabilities of the MIPS processor after fabrication.

22.1.4. Liquid Crystal Display

The MIPS PCB will contain a 2x20 LCD display. The chip enable for the LCD (LCD_E) is included on pin 38 of header J1. LCD data will be included on bits 0-7 of the MemData bus (MemData[7:0]), and the LCD read select (RS) bit is connected to MemData[8]. A dedicated 16-bit 2x8 header has been included on the PCB for the LCD display, and will be discussed further in the following section.

22.1.5. Connections

The main function of the PCB is to provide an interface between the MIPS microprocessor and the XUPV2P Development board, on which the memory and many of the system peripherals are contained. Connections between the MIPS and Virtex-II boards are contained on two 2x20 right angle male header pins, designed to interface cleanly with the XUPV2P board, without use of ribbon cables. Connection information can be found in both the XUPV2P User Guide, as well as the included connection information in the Section 23. These headers contain all 32 bits of the MemoryData bus, 17 bits of the MemoryAddress bus, the Reset signal, four ByteEn bits, eight Interrupt bits, both clock signals (PH1 and PH2), a Done status bit, a RWB bit, and an EN bit. Although not connected to the MIPS processor, an LCD_EN bit also is contained on the interface with the Virtex-II FPGA.

The remaining ten MemoryAddress bits are available on a separate 2x5 header J4, should their use be desired in the future using more complicated connective wiring. Exact wiring specifications are available in the Appendix.

A separate header has been included for the LCD, which includes all necessary power for the LCD logic (5VDC) as well as connections to pertinent pins defined in the memory-mapped I/O specification. Header J3 has been designed with a female 2x8 header in mind, which should interface neatly if a male 2x8 header is soldered into the Crystalfontz LCD as designed, providing a clean and versatile interface.

22.2. PCB Post-Fab Test Plan

After receipt of the PCB by the team, verification of functionality was tested. First, using a digital multimeter, proper connectivity was verified (especially that 3.3VDC and 5VDC do not short to ground). After basic mechanical testing, the team connected one of the two XUPV2P

Development boards through the PGA socket to another XUPV2P, thereby verifying proper connectivity through the PCB, and verifying that all subsystems worked as expected.

22.3. PCB Setup and Corrections

In the testing and verification process, several errors were found in the original PCB. These changes have been made, and two modified PCBs assembled, one for HMC and one for Adelaide. When Adelaide receives their assembled board from HMC, parts have been included for the assembly of another MIPS PCB. Parts are listed on the MIPS PCB bill of materials, and are included on a parts list enclosed with the parts sent to Adelaide. Two 9VDC AC transformer is also included to power the board through the 2.1mm DC jack. Two Crystalfontz 2x20 character LCDs are also included, which conveniently mount on the PCB through J3. Assembly of the PCB should be straightforward with labeled footprints on the PCB and the included schematic, with the addition of the following modifications to the PCB.

Changes to be made

Due to encountered errors in the original PCB design, alterations must be made to v1.0 of the PCB to obtain an operational system. A modified PCB has been assembled, and should be ready for use as is. This can also be used as a reference for assembly of the second PCB, if necessary. All errors known to date have been rectified and included in a newer revision of the PCB library – hmc_mips_v1_1.

1. The 2.1mm DC jack footprint was mirrored. Soldering the jack onto the bottom of the board maintains correct connections with no additional soldering.
2. The voltage regulators (both 5VDC and 3.3VDC) had pads associated with incorrect wire nets. Fixes involve breaking existing nets and creating correct nets by way of soldered wires. Traces on the PCB can be cut by scraping them with tweezers or pliers to break the copper. Check that the trace circuit is open using a multimeter. Then, new wires can be soldered on.

3.3VDC Regulator:

- break the trace between the top pad (Vin, pad 1) and the via next to it (part of the 3.3V power plane network)
- break the trace between the large pad (Vout, pad 2) and the via next to it (part of the GND plane network)
- break the trace between the bottom pad (GND, pad 3) and pin 1 of C4.
- join the top pad (Vin, pad 1) to the center pin of the power switch
- join the center pad (Vout, pad 2) with the 3.3V power network. The assembled board uses the via directly next to pad 1.
- join the bottom pad (GND, pad 3) with the GND power plane network. The most convenient pad is the via centered in the footprint. A lead clipped off of a soldered component is convenient.

5VDC Regulator:

- break the trace between the top pad (Vin, pad 1) and C2 (part of the 5V power network)

- break the trace between the top pad and the via directly above it (part of the 5V power network)
- break the trace between the large pad (Vout, pad 2) and the via next to it (part of the GND plane network)
- break the trace between the bottom pad (GND, pad 3) and pin 2 of the power switch.
- join the top pad (Vin, pad 1) to the center pin of the power switch
- join the center pad (Vout, pad 2) with the 5V power network. The assembled board uses the via directly above pad 1.
- join this via to pin 1 of C2
- join the bottom pad (GND, pad 3) with the GND power plane network. The most convenient pad is the via centered in the footprint. A lead clipped off of a soldered component is convenient.

3. The $20k\Omega$ potentiometer (POT1) for adjustment of LCD contrast has an incorrect footprint. Fixes are similar in nature to the changes made to the regulators, with cut traces and soldered wires.

POT1:

- break the trace between the top pin (2) and pin 1 of C3 (part of the 5V power network)
- on the bottom of the board, break the trace between the top pin (2) and pin 2 of R4.
- on the bottom of the board, break the trace between the bottom right pin (3) and pin 2 of R5.
- join the top pin (2) with pin 2 of R5.
- join the bottom right pin (3) with pin 2 of R4.
- join the bottom right pin (3) with pin 1 of C3.

In addition, it should be noted that the anode of the power indicator LEDs is on the left side of the footprint. Polarity was not marked on the first revision of the PCB.

22.4. Concluding Comments

The PCB has been ordered from Advanced Circuits (4pcb.com). Five boards were ordered at a price of \$135 each, with a turn-around time of four days.

In order to minimize unnecessary effort, portions of this design are taken from the Harrisboard 2.0, a known working system.

Data sheets for components are included in the software portion of the documentation.

Lessons Learned

The entire process of working on the MIPS project was a great learning experience, and as many great learning experiences tend to do, most of the lessons learned are learned in retrospect. One of the things that was reinforced for me was getting things done early, if at all possible. It may make your life a bit more difficult at that time, but things will be better later on. In addition, the extra leeway gained by diligence can be greatly leveraged for extra review of work completed. This leads me into the next thing: spend a lot of time reviewing designs, especially ones that

have only been worked on by one individual. Bother other people to spend some time looking over your work, and mistakes might be caught that you hadn't seen, just because you had been looking at them for such a long time. Thirdly, I would recommend having clear communication with those you are working with, especially those you are working most closely with. A lot of grief can be saved by asking someone for help or clarifying exactly what it is they want from you.

23. Post-Silicon Test Plan

23.1. Introduction

The post-silicon test plan is designed so that the Adelaide team, upon receiving a package from HMC containing PCB boards, parts to solder onto those boards, and a DVD containing all of the files they need to get the FPGA and board operational, can set up the PCB, program the Virtex board, and run the programs described for testing purposes.

23.2. Contents

23.3. Quick Start Guide

This guide will provide references to information in the Chip Report on how to set up the PCB board, FPGA, and software, and get you to the point where you can compile, program, and run software. Follow the steps described here to get the FPGA-PCB package to the point where a program can be compiled and programmed into the FPGA memory.

23.3.1. Unpack the box

Unpack the box and look for the checklist of included items. Check off every item that is supposed to be included.

23.3.2. Set up the single-FPGA emulation

Set up the Virtex II board for Single-FPGA emulation (chip and memory on the FPGA) as described in Section 21.2. Solder the clock pins to the Virtex II board as described in the Section 21.2. You will need an 80 MHz signal generator hooked up to the clock pins so that you can properly set the clock rate. For now, leave the clock at something low like 2 MHz meaning you have to set the waveform generator to a square wave running at 8MHz since the input clock is divided by 4 to generate the two phase clock.

Create a project file based on the Verilog.zip file in the "Systems/Single_FPGA" folder on the DVD. Note: follow the project file creation guide found in Section 21.3 and use the zipped package that contains project files only for reference.

A very simple ROM should be included in the Single-FPGA emulation package that sets a value on the Virtex board LEDs. Once the package pins have been assigned, you should be able to synthesize the provided Verilog and program the Virtex board and see the LEDs light up in a 1001 pattern (LEDs 0 and 3 on, LEDs 1 and 2 off). Hardware reset is the ENTER button.

23.3.3. Set up the FPGA-PCB package

Set up the PCB according to the schematic in the Chip Report Sectoion 24.4. Set up the Virtex II board in the "FPGA and PCB" configuration (memory on FPGA only, with the headers on the Virtex II plugged into the PCB) as described in the Chip Report under the section for Dual-FPGA Emulation found in Section

21.4. You will use half of the Dual-FPGA setup (the half that has the external memory) to program the FPGA to work with the PCB. Create a project file based on the code taken from the extmem folder in Verilog.zip file that is in the “Systems\Dual_FPGA” folder on the DVD.

The same simple ROM is included in the FPGA-PCB package from single-FPGA emulation. Once the package pins are set up, you can synthesize the Verilog source and program the Virtex board to see this program run again. Be sure to check that hardware reset is mapped to the button on the PCB.

23.3.4. Set up the Compiler and Toolchain

Now follow the instructions in the Chip Report under Compiler User Instructions found in Section 20.1 to set up the toolchain. This will have you install Cygwin and unpack yoda_warrior (the compiler) and the toolchain the Systems team created. The yoda_warrior.tar.gz tarball and the Toolchain.zip file containing all the compiler files can be found in the “Systems\Compiler” folder on the DVD. Once you have set up the toolchain, go into the compiler source folder (hmc-mips\systems\src) in Cygwin and run

```
make test_simple.v
```

Replace ROM.v with the output Verilog file (test_simple.v) in the Xilinx project folder and re-synthesize the project. Then, program the FPGA. Now you should see the LED counting up in binary.

Now the hardware and toolchain are properly set up for you to compile programs and load them onto the FPGA for the chip to run them.

23.4. Test Plan

After the fabricated chip is received from MOSIS, it will be necessary to verify full functionality. It is preferable that the test plan include sequential increase of complexity or individual test cases, in order to identify specific errors, rather than a general assessment of working or non-working. With this in mind, the Systems cluster recommends the following process for post-silicon functionality verification:

1. Clock Frequency Verification
2. Device-specific Test Programs
3. Demo Program
4. Random Tests
5. Benchmark

These different steps are covered in depth below. Different stages are often combined together in order to minimize the effort necessary to reprogram the FPGA and to simplify the testing process.

23.4.1. Clock Frequency Verification

Due to unknowns of fabrication, the final maximum operating frequency of the HMC MIPS processor is unknown. Because of the imperative nature of the clock for operation of the MIPS processor, it is necessary to determine the clock frequency at which the MIPS processor will run. Using a signal generator attached to the clock pins that have been set up on the Virtex II (as per the Quick Start Guide) allows the frequency to be tuned for optimal performance. The Systems team recommends tuning the processor by running a non-trivial program such as Dhrystone or

LightsOut!, and then finding the clock speed at which the program begins to fail. This gives an upper bound on the operating range. (It is important that the program be non-trivial. We have seen simple programs operate normally at clock speeds where more complicated programs fail.) A good strategy for finding the optimal clock would be to time Dhrystone and get a feel for the performance around the high end of the operating range. Information on what output to expect from Dhrystone is in the Chip Report under *Compiler, Benchmarks, and Demo Software* in the subsection *Using Supplied Programs* in Section 20.

23.4.2. Device-Specific Test Programs

The Systems team recommends running the three device-specific test programs after setting up the board and getting a working clock. These devices target the LCD, LEDs, DIP switches, and buttons. There are three source files: test_leds.c, test_lcd.c, and test_buttons.c. Information on building and using these programs can be found in the Chip Report under *Using Supplied Programs* in the section on the compiler, benchmarks, and demo software.

23.4.3. Demo Program

The Systems team then recommends running the prepared demo software, “Lights Out!”, the source for which is called lightsOut.c. Detailed documentation on the demo program is available in the Chip Report under *Using Supplied Programs*.

23.4.4. Random Tests

If no problems occur, the vectors generated by the random test generator can then be run on hardware, providing both a thermal stress test and a thorough gamut of instructions to be run in various sequences. It is likely that any problems with the hardware will be caught in this final testing round, as random instructions should manifest any situation the chip may encounter without spending the time necessary to do an absolutely thorough evaluation of every possible code situation, a prohibitively costly method of verification. If the chip passes all tests, the team can be quite confident of proper functionality in any circumstance.

To build the random tests for the FPGA- PCB configuration, consult the Chip Report section from the MicroArchitecture team that explains how to generate random tests found in Section 6.2. The Systems cluster was never able to test the finished random tests in hardware due to the lack of the fabricated chip, so it cannot be guaranteed that the test will work correctly the first time.

23.4.5. Benchmark

Once all of the tests have been completed, produce some benchmark scores for the chip. It might be a good idea to have a range of DMIPS/MHz scores for various clock speeds, allowing for profiling the characteristic performance of the chip and perhaps more accurately specifying the optimum clock speed. (In an ideal chip with no delay, DMIPS/MHz would remain constant.) Measuring Dhrystone requires a stopwatch, since the external memory system does not provide a system clock for software timing.

The procedure for running Dhrystone and for calculating scores based on raw timing is recorded in the *Compiler, Benchmarks, and Demo Software* section of the Chip Report under *Using Supplied Programs*.

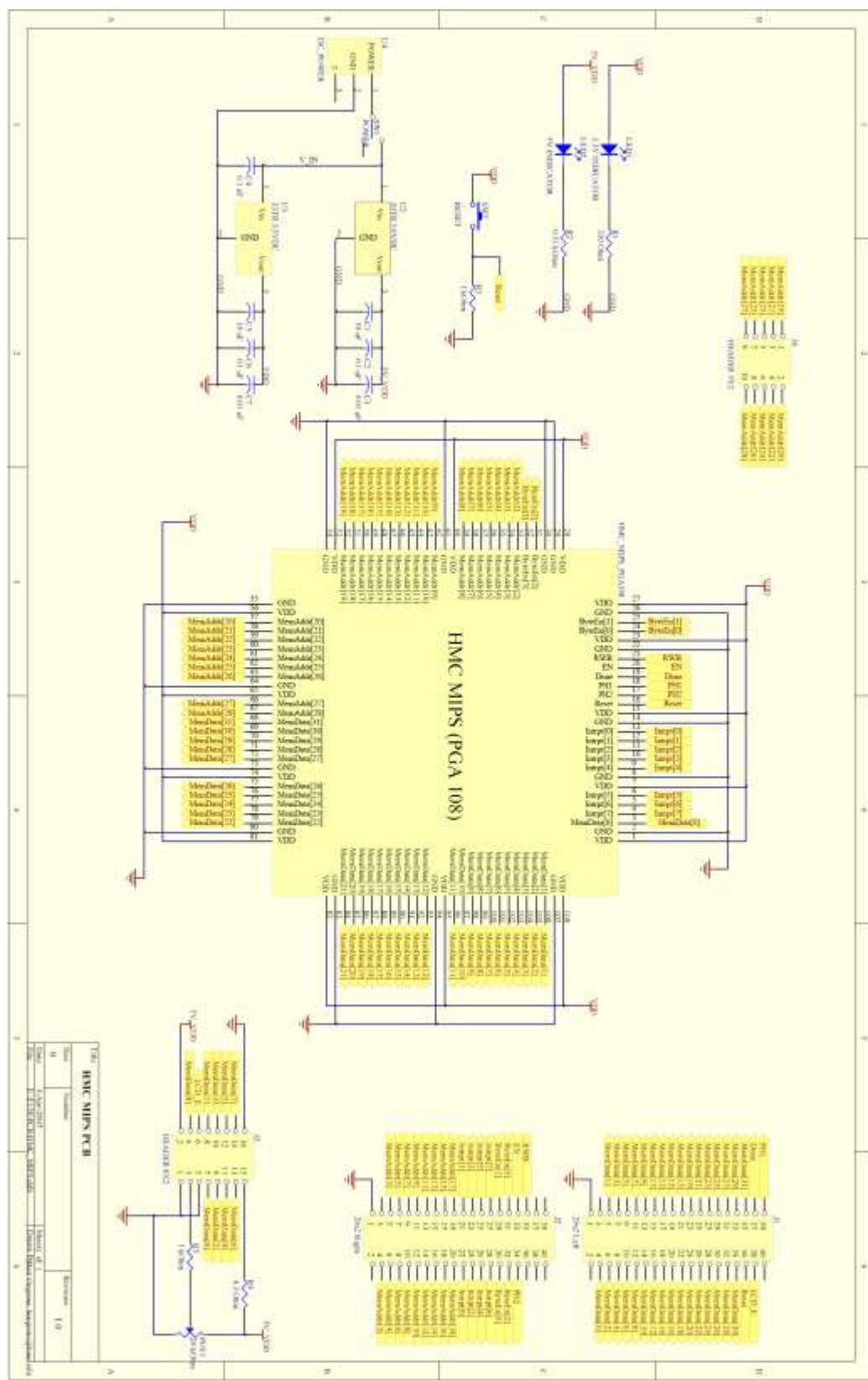
24. Systems Cluster Appendix

24.1. Pin Specifications

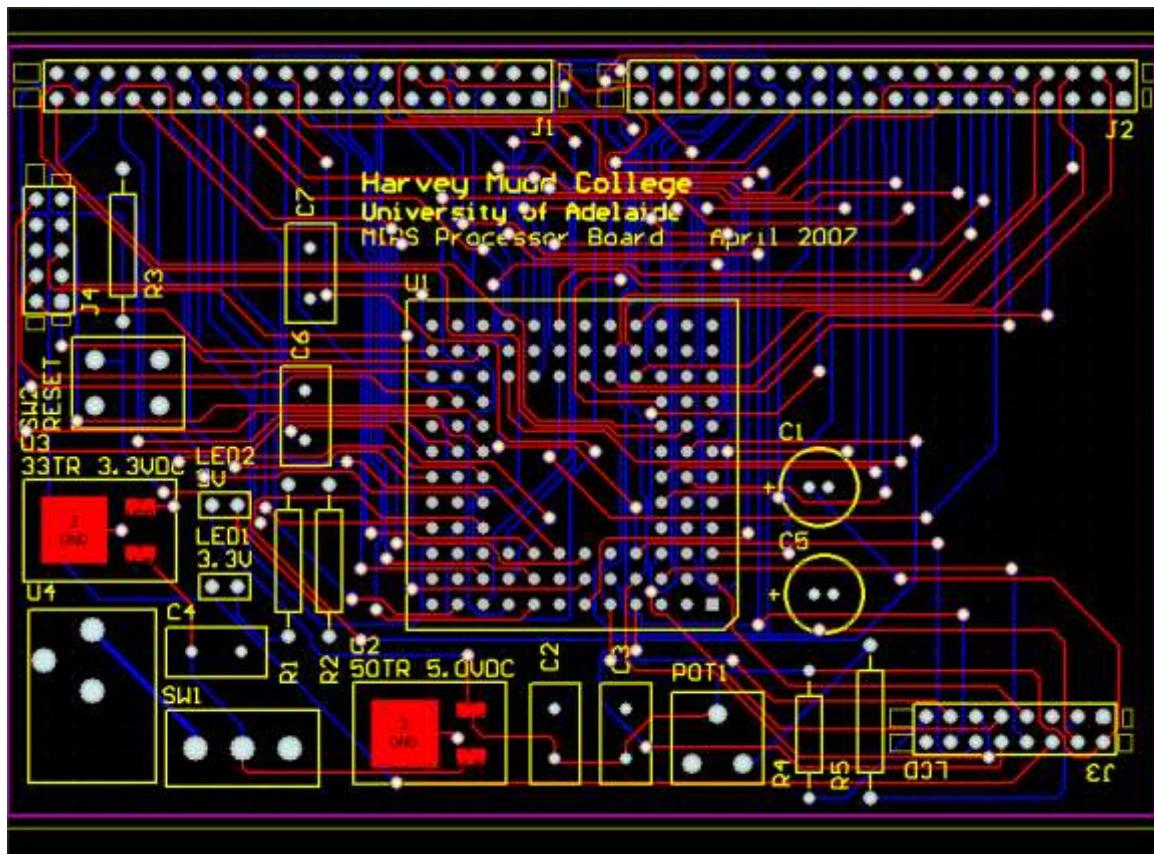
MIPS Signal	Pin Name	Pin #	PCB	XUPV2P Pin	Signal	Virtex II Pin	LCD Pin	LCD Pin #
MemData1	L10	106	J1.5	J5.5	EXP_IO_9	N5	DB1	8
MemData3	M9	104	J1.7	J5.7	EXP_IO_11	L4	DB3	10
MemData5	K9	102	J1.9	J5.9	EXP_IO_13	N2	DB5	12
MemData7	L8	100	J1.11	J5.11	EXP_IO_15	R9	DB7	14
MemData9	M7	98	J1.13	J5.13	EXP_IO_17	M3		
MemData11	K7	96	J1.15	J5.15	EXP_IO_19	P1		
MemData13	M5	92	J1.17	J5.17	EXP_IO_21	P7		
MemData15	K5	90	J1.19	J5.19	EXP_IO_23	N3		
MemData17	L4	88	J1.21	J5.21	EXP_IO_25	P2		
MemData19	M3	86	J1.23	J5.23	EXP_IO_27	R7		
MemData21	M2	84	J1.25	J5.25	EXP_IO_29	P4		
MemData23	K3	78	J1.27	J5.27	EXP_IO_31	T2		
MemData25	J2	76	J1.29	J5.29	EXP_IO_33	R5		
MemData27	H3	72	J1.31	J5.31	EXP_IO_35	R3		
MemData29	G2	70	J1.33	J5.33	EXP_IO_37	V1		
MemData31	F1	68	J1.35	J5.35	EXP_IO_39	T6		
Done	E11	19	J1.37	J5.37	EXP_IO_41	T4		
PH1	E10	18	J1.39	J5.39	EXP_IO_43	U3		
MemData0	L12	3	J1.4	J5.4	EXP_IO_8	N6	DB0	7
MemData2	K10	105	J1.6	J5.6	EXP_IO_10	L5	DB2	9
MemData4	L9	103	J1.8	J5.8	EXP_IO_12	M2	DB4	11
MemData6	M8	101	J1.10	J5.10	EXP_IO_14	P9	DB6	13
MemData8	K8	99	J1.12	J5.12	EXP_IO_16	M4	RS	4
MemData10	L7	97	J1.14	J5.14	EXP_IO_18	N1		
MemData12	K6	93	J1.16	J5.16	EXP_IO_20	P8		
MemData14	L5	91	J1.18	J5.18	EXP_IO_22	N4		
MemData16	M4	89	J1.20	J5.20	EXP_IO_24	P3		
MemData18	K4	87	J1.22	J5.22	EXP_IO_26	R8		
MemData20	L3	85	J1.24	J5.24	EXP_IO_28	P5		
MemData22	K2	79	J1.26	J5.26	EXP_IO_30	R2		
MemData24	J1	77	J1.28	J5.28	EXP_IO_32	R6		
MemData26	J3	75	J1.30	J5.30	EXP_IO_34	R4		
MemData28	G1	71	J1.32	J5.32	EXP_IO_36	U1		
MemData30	G3	69	J1.34	J5.34	EXP_IO_38	T5		
Reset	F11	16	J1.36	J5.36	EXP_IO_40	T3		
LCD_E			J1.38	J5.38	EXP_IO_42	U2	E	6
PH2	F12	17	J1.40	J5.40	EXP_IO_44	U7		
MemAddr3	B9	34	J2.5	J6.5	EXP_IO_45	T8		
MemAddr5	C8	36	J2.7	J6.7	EXP_IO_47	U5		
MemAddr7	A8	38	J2.9	J6.9	EXP_IO_49	W2		
MemAddr9	C6	42	J2.11	J6.11	EXP_IO_51	U9		
MemAddr11	A6	44	J2.13	J6.13	EXP_IO_53	V4		
MemAddr13	B5	46	J2.15	J6.15	EXP_IO_55	Y1		
MemAddr15	C4	48	J2.17	J6.17	EXP_IO_57	U8		
MemAddr17	A4	50	J2.19	J6.19	EXP_IO_59	V6		
MemAddr19	B3	52	J4.1					
MemAddr21	C2	58	J4.3					
MemAddr23	D3	60	J4.5					
MemAddr25	D1	62	J4.7					
MemAddr27	F3	66	J4.9					
Intrpt1	G10	12	J2.21	J6.21	EXP_IO_61	AA2		

Intrpt3	H11	10	J2.23	J6.23	EXP_IO_63	V8
Intrpt5	J10	6	J2.25	J6.25	EXP_IO_65	W4
Intrpt7	K11	4	J2.27	J6.27	EXP_IO_67	AB1
ByteEn1	C11	25	J2.29	J6.29	EXP_IO_69	W6
ByteEn3	A10	32	J2.31	J6.31	EXP_IO_71	Y5
EN	E12	20	J2.33	J6.33	EXP_IO_73	AA4
RWB	D10	21	J2.35	J6.35	EXP_IO_75	W8
MemAddr2	C9	33	J2.4	J6.4	EXP_IO_46	U4
MemAddr4	A9	35	J2.6	J6.6	EXP_IO_48	V2
MemAddr6	B8	37	J2.8	J6.8	EXP_IO_50	T9
MemAddr8	C7	39	J2.10	J6.10	EXP_IO_52	V3
MemAddr10	B6	43	J2.12	J6.12	EXP_IO_54	W1
MemAddr12	C5	45	J2.14	J6.14	EXP_IO_56	U7
MemAddr14	A5	47	J2.16	J6.16	EXP_IO_58	V5
MemAddr16	B4	49	J2.18	J6.18	EXP_IO_60	Y2
MemAddr18	C3	51	J2.20	J6.20	EXP_IO_62	V7
MemAddr20	B1	57	J4.2			
MemAddr22	C1	59	J4.4			
MemAddr24	D2	61	J4.6			
MemAddr26	E3	63	J4.8			
MemAddr28	F2	67	J4.10			
Intrpt0	G11	13	J2.22	J6.22	EXP_IO_64	W3
Intrpt2	H12	11	J2.24	J6.24	EXP_IO_66	AA1
Intrpt4	H10	9	J2.26	J6.26	EXP_IO_68	W5
Intrpt6	K12	5	J2.28	J6.28	EXP_IO_70	Y4
ByteEn0	C10	24	J2.30	J6.30	EXP_IO_72	AA3
ByteEn2	B10	31	J2.32	J6.32	EXP_IO_74	W7
VDD	M12	1				
VDD	J11	7				
VDD	F10	15				
VDD	D12	23				
VDD	B12	27				
VDD	A12	28				
VDD	B7	40				
VDD	A3	53				
VDD	B2	56				
VDD	E1	65				
VDD	H1	74				
VDD	L1	81				
VDD	M1	82				
VDD	M6	95				
VDD	M11	108				
GND	L11	2				
GND	J12	8				
GND	G12	14				
GND	D11	22				
GND	C12	26				
GND	B11	29				
GND	A11	30				
GND	A7	41				
GND	A2	54				
GND	A1	55				
GND	E2	64				
GND	H2	73				
GND	K1	80				
GND	L2	83				
GND	L6	94				
GND	M10	107				

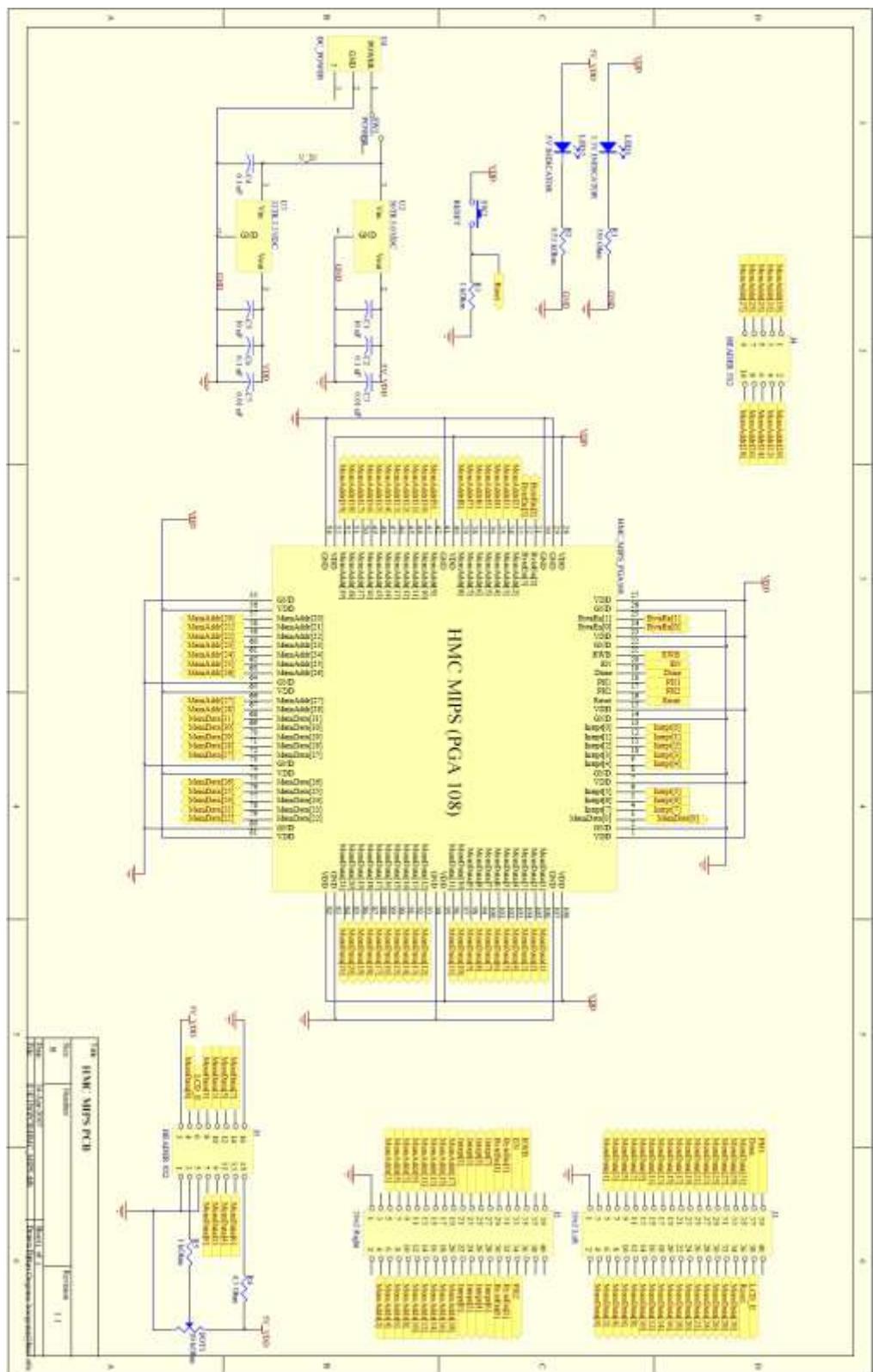
24.2. V1.0 Schematic



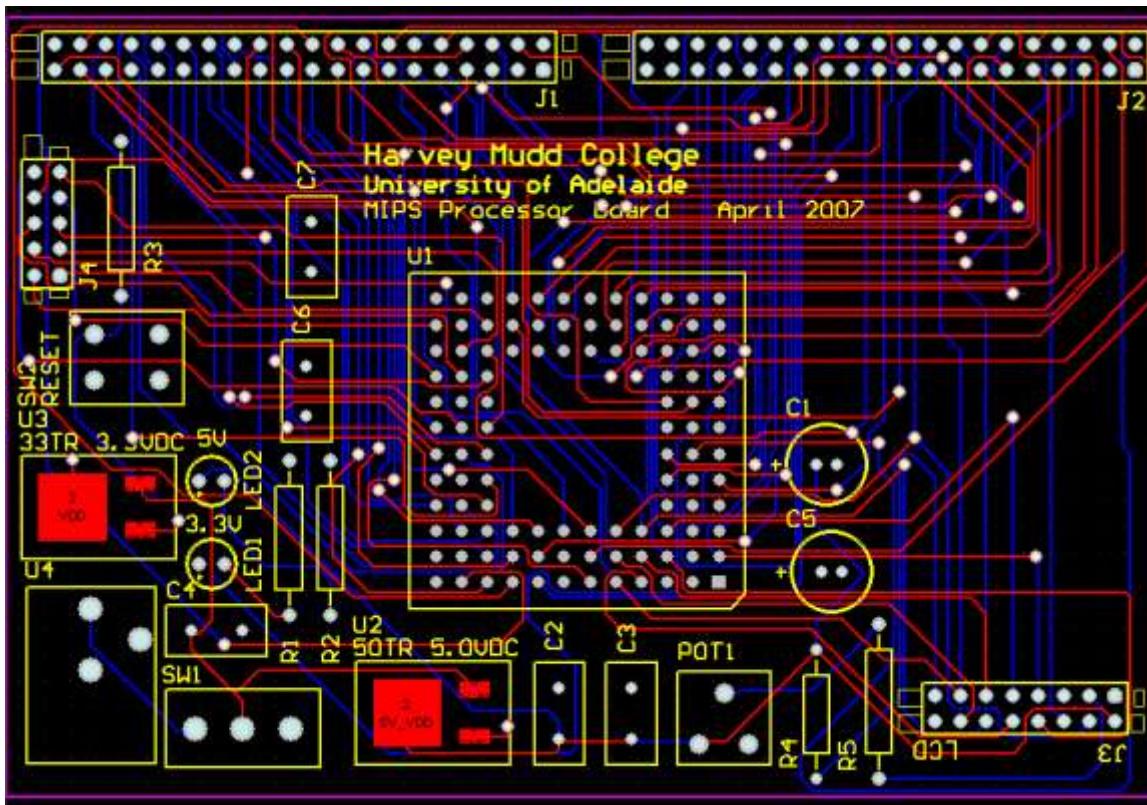
24.3. V1.0 Layout



24.4. V1.1 Schematic



24.5. V1.1 Layout



24.6. Bill of Materials

Component	Part	Manufacturer	Manufacturer Part #	Supplier	Supplier Part #	Quantity	Price (ea.)	Total
108 pin PGA socket	U1	Advanced Interconnection	CIS108-01TG			1		
0.01 μ F capacitor	C3, C7	Panasonic - ECG	ECQ-V1H103JL	Digikey	P4513-ND	2	0.095	\$0.19
0.1 μ F (100 nF) capacitor	C2, C4, C6	Panasonic - ECG	ECQ-V1H104JL	Digikey	P4525-ND	3	0.108	\$0.32
10 μ F capacitor	C1, C5	Panasonic - ECG	ECA-1CM100	Digikey	P5134-ND	2	0.15	\$0.30
Pushbutton switch	SW2	Panasonic	EVQ-PAE04M	Digikey	P80085-ND	1	0.29	\$0.29
Power switch	SW1	NKK Switches	CS12ANW03	Digikey	360-2131-ND	1	2.56	\$2.56
2.1 mm DC jack	U4	CUI Inc.	PJ-102A	Digikey	CP-102A-ND	1	0.38	\$0.38
5V regulator	U2	STMicroelectronics	LD1117DT50TR	Digikey	497-1238-1-ND	1	0.77	\$0.77
3.3V regulator	U3	STMicroelectronics	LD1117DT33TR	Digikey	497-1236-1-ND	1	0.77	\$0.77
Discrete radial LED, green	LED1, LED2	LITE-ON	LTL-10233W	Digikey	160-1089-ND	2	0.073	\$0.15
4.3 Ω axial resistor	R4	Yageo	CFR-12JB-4R3	Digikey	4.3EBK-ND	1	0.052	\$0.05
0.33 k Ω axial resistor	R1	Yageo	CFR-25JB-330R	Digikey	330QBK-ND	1	0.054	\$0.05
0.51 k Ω axial resistor	R2	Yageo	CFR-25JB-510R	Digikey	510QBK-ND	1	0.054	\$0.05
1.0 k Ω axial resistor	R3, R5	Yageo	CFR-25JB-1K0	Digikey	1.0KQBK-ND	2	0.054	\$0.11
20 k Ω potentiometer	POT1	Panasonic - ECG	EVN-D8AA03B24	Digikey	D4AA24-ND	1	0.36	\$0.36
2x16 LCD display		Crystalfontz	CFAH 1602J-YYH-JP	Crystalfontz		1	20.76	\$20.76
2x20 male right angle header	J1, J2	Tyco/AMP	5103310-8	Digikey	A33187-ND	2	2.85	\$5.70
2x40 straight male header	J4, LCD	Tyco/AMP	9-146256-0	Digikey	A34269-40-ND	1	3.68	\$3.68
2x8 female header	J3	Sullins	PPPC082LFBN-RC	Digikey	S7111-ND	1	1.49	\$1.49
							Est. Total:	\$37.99
Last revised 4/16/2007 Bart Oegema boegema@hmc.edu								

Library Cluster

Principal Team Members:

Justin Gries

Danny LaValle

25. Library (muddLib07.jelib) Creation.

Owner: Justin Gries

Milestone Date: 20 February, 2007

25.1. Personnel:

Justin Gries: **Team leader; co-editor**

- Directed the editing process and verified task completion.
- Provided initial critique/grading and feedback for editing on the following cells: a22o2i, o2a1i, invtri_dp, and2, and3, or2, or3, a22o2, buftri_c, xor3, mux4i_dp, mux4_dp, mux3_c.
- Principal architect and/or editor on the following cells: or3, or4, a2o1, a22o2, o2a1, o22a2, buftri_dp, buftri_c, xor2, xor3, xnor2, mux2i_dp, mux3i_dp, mux4i_dp, mux2_dp, mux3_dp, mux4_dp, mux2_c, mux3_c, mux4_c.
- Performed a quality control check on all cells edited by other team members.

Danny LaValle: **Team member; co-editor**

- Provided initial critique/grading and feedback for editing on the following cells: o22a2i, o2a1, buftri_dp, xor2, xnor2, mux3_dp, mux4_c.
- Principal architect and/or editor on the following cells: inv, nand2, nand3, nand4, nor2, nor3, nor4, a2o1i, a22o2i, o2a1i, o22a2i, invtri_dp, invtri_c, buf, and2, and3, and4, or2, xnor3.
- Performed a quality control check on all cells edited by other team members.

Nate Schlossberg: **Team member; assistant editor**

- Provided cell generation and principal editing on all flop-cells, including: flop_dp, flopr_dp, flops_dp, fopen_dp, flopnr_dp, flopens_dp, flop_c, flopr_c, flops_c, fopen_c, flopnr_c, flopens_c.

Creation of the cell Library took place in three distinct phases:

1. Preliminary Critique and Editing: Upon receipt of the cells, which were generated as part of a homework assignment for the course, the team evaluated each cell to ensure that all cell guidelines were followed, and all required components were present. The cells were split up, with Danny and Justin both taking roughly half of the cells. The original cell owner/creator was then notified of any corrections that were necessary. This initial review process took approximately 30-40 man-hours to complete, and lasted for roughly one week.
2. Final Editing: Upon receipt of the now revised cells, the cells were once again split between the team members, and each cell was gone over again; this time including some (supposedly good) cells that had been generated prior to the main batch discussed above. Some extensive editing still proved to be necessary on many cells, and rather than repeating the lengthy task of writing out detailed editing instructions to the original creators, most editing was done by

the team members themselves at this point. This editing once again took between 40-50 man hours, spread over about 2 weeks

- a. While this editing was going on, Nate Schlossburg helped out greatly by generating the flop cells used in the library. These were quite large and time consuming to generate.
3. Error Scrubbing: Upon finishing their assigned cells, each team member submitted those cells to the other for a final overview, to hopefully catch any remaining oversights. The flop cells were also reviewed by Justin and Danny at this point. The cells were fairly clean by this point, so this process took about 5-7 man-hours total, over about a two-day period.

Library creation consisted of design and optimization of the layouts of the cells, based off of the schematics that were provided. Though there were a few situations where minor editing of the schematics was necessary, in general the schematic-files were not altered, and were not checked for logical accuracy. With the exception of some changes made to the mux4i_dp cell, the library team claims no responsibility for errors generated due to incorrect logic/schematic design.

All cells were checked against the following criteria:

- Passing of all fundamental tests, including DRC, ERC, and NCC.
- Consistency of export locations between cells of the same type but different size. Exceptions to this rule were made only in cases where cell size could be optimized by moving exports.
- All exports located “on-column”.
- All non-power/gnd exports located either on metal-2 pins or on metal-1/metal-2 vias.
- 90- λ cell height.
- Optimization of the layout (number of columns) to minimize cell size.
- Minimization of intra-transistor spacing in order to diminish capacitance.
- All layout components situated “on-grid”.
- Elimination of extraneous material (metal, or polysilicon).
- Elimination of unnecessary pins.

Table 18, in Appendix A shows a complete list of the cells that were included in the muddlib07.jelib file.

25.2. Lessons learned:

- Though every effort was made to eliminate errors prior to library publication, some problems were still uncovered that had to be addressed. There is only so much that can be done to eliminate human error. Resources must be allocated to handle debugging for a significant time period after a resource like this “goes live”.
- There is a diminishing return on time spent “outsourcing” work. When major overhauls are necessary this it may be worthwhile, but as editing becomes more specific it may be just as quick to edit things directly instead of writing out another set of detailed instructions to coworkers regarding the changes and handling the logistics associated with such management.
- “More bodies” does not always equate to better performance. The increased logistics involved necessitates more management. A good skill to have would be the ability to look at a given project and intuit the optimal number of people to perform the task. In the case of

this project, one or two more people on the Library Cluster may have made things a bit easier, but any more than that would have been a waste of personnel resources.

- When setting schedules, budgeting a significant “safety buffer” is a very good idea. In our case, budgeting an extra week more than we expected to need actually allowed us to exactly meet the milestone date.

26. The PLA generator

Owner: Justin Gries

Milestone Date: 6 March, 2007

26.1. Personnel:

Justin Gries: **Principal code architect; Programmer**

- Outlined basic class structure for the application
- Coded all classes dealing with layout generation

Danny LaValle: **Programmer; Code designer**

- Coded all classes dealing with schematic generation

The PLA generator is modeled after the PLA design illustrated in CMOS VLSI Design⁵. Its purpose is to take in a case statement, in Verilog format, and return a text file (.jelib format) that is readable by Electric, which contains the schematic and layout views of the PLA. The basic concept of the code is as follows:

1. The user is prompted to choose two files: Both must be plaintext files. The first contains a list of parameters corresponding to the PLA, such as desired transistor or arc sizes; The second is a Verilog-format case statement.
2. The program calls classes that parse these files and make the parsed data available for reference.
3. The LayCell class constructs the layout, and returns that layout's text representation to the main application.
 - i. The PLA might be considered as a collection of zones, which are joined together into a single entity. Those zones are the And-Plane, Or-Plane, Input-Inverters, Output-Inverters, And-Pulldowns, Or-Pulldowns, Ground-line, and Power Line.
 - The zones are further divided into individual cells, which are created in their own classes, and joined together to make the zones. Each level of this heirarchy generates a set of nodes and/or arcs, which are compiled and handed off to the level above.
 - ii. The LayCell class constructs a skeleton of sorts, determining the boundary between zones, and creating linking nodes that will join these areas together. LayCell then calls classes representing the zones, which return their nodes and arcs to be added to the growing list of such objects that is kept by the LayCell class. The “skeleton” nodes are passed into the zones to give them attachment points to link up with.

⁵ Weste, Neil H., and David M. Harris. CMOS VLSI Design. 3rd ed. New York: Pearson, 1993. 754

- iii. The LayCell also constructs the N-Diffusion and P-Diffusion wells, as well as all exports.
 - iv. Once all the zones have been created, all of the nodes, arcs, and exports are concatenated into a single string object, which is referenced by the main (PlaGen) class.
- 4. The SchCell class constructs the schematic, and returns its text representation to the main application.
 - i. The schematic cell is a bit more direct in its approach. The SchCell class calls various methods in the SchematicManager class to create the various zones of the PLA, and the wire them together.
 - The SchematicManager does the vast majority of the work here, generating all of the zones, keeping track of interfaces between the zones as they are generated, and using those interfaces to link up with subsequent zones. It also stores all node-, arc-, and export-objects as they are generated, and compiles their text-interpretations to return the text version of the schematic when it is called for.
- 5. The main class concatenates the layout and schematic strings into a single string, and outputs it as a file, using the name that is specified in the parameters file.

To use the application, the user should perform the following:

1. Double-click on the PLAGenerator.jar file to start the application
2. As prompted, choose the parameter.txt file, and then the case file that will be used for the PLA.
 - a. Both files must be in plaintext format. Any hidden encoding will break the parsing.
 - b. The only items in the parameters file that the user should be changing are the “cellName” entry, which will be the filename that the .jelib file will be output as, and “nMosInvWidth” which is the width of the inverter nmos transistors. (The inverter pmos transistors are considered to be 2x the nmos width.)
 - c. The case statement should be in a .v file, and in Verilog coding format. No defaults other than all zeros are permitted, as addressing other defaults would have made the PLA much larger
3. The file should appear in the same folder as the PLAGenerator.jar file that is run. Processing time for large PLAs can take up to a minute.
4. Although the PLA that is generated should pass all tests in Electric without any editing, it is highly recommended that a “pin cleanup” be run to remove the many unnecessary pins that are a product of the way that the layout and schematic are formed. (From the Electric menu-bar: Edit > Cleanup Cell > Cleanup Pins Everywhere). This can take a couple of minutes to finish for larger PLAs.

Coding took approximately 200+ man-hours, over the course of a three-week period. The source-code has been uploaded to the course server. It consists 33 classes, which put together contain approximately 10000 lines of Java code.

26.2. Lessons Learned

- Originally, the compiled code took a prohibitively long period of time to run for larger case statements. Originally, both the layout and the schematic generation code collected all of the Node, Arc, and Export objects, and then once all of the objects were collected, their String interpretations were joined together all at once via concatenation. Testing showed that this concatenation process was the bottleneck of the layout generation, and a different method of forming the layout's text-file was used. The Schematic generator appears to have a similar issue, but because the number of nodes and arcs in the schematic are far less than there are in the layout, the slowdown is not as pronounced and the issue was not addressed for the schematic. Altering the code for the layout increased the speed of the application by over an order of magnitude (~60x).
- Managing a software project provides unique challenges, not the least of which is the division of labor.
- Coding takes a very significant amount of time. The rate of code generation was about 45 lines of code per man-hour, where anywhere from 20 to 30% of that code was commenting or copy/paste repetitious material. Working out the proper algorithms to cover all conceivable corner cases can slow code generation down to a crawl in some places.
- Breaking down code into blocks and fully utilizing the power of object oriented programming can appear somewhat cumbersome at first, but can make troubleshooting much easier.
- Learning and using a development environment (Eclipse was used on this project) will not make up for a total lack of programming knowledge, but it can certainly provide a huge augmentation to existing skills, allowing the programmer to focus more on the code itself rather than syntax. Eclipse also allows quick autoformatting, which was a great help.
- It is most likely better to comment code as it is written, as opposed to going back through and writing it in later. A bit of time is shaved off of the initial release by skipping commenting initially, but the price paid later is quite steep, especially if a sizeable body of code must be reviewed.

27. Chip Logo

Owner: Danny LaValle

Milestone Date: 17 March, 2007

27.1. Personnel:

Danny LaValle: Designer

The logo is a single metal-2 layer. It was created by transforming a grayscale bitmap into an Electric layout library file using a Matlab script. The script treats each pixel of the image as a $1-\lambda$ cell and fills in metal for each colored pixel.

The logo image and chip name were created to incorporate Harvey Mudd related themes. Additionally, the names of those individuals contributing to the chip's design are also displayed.

27.2. Lessons Learned

- Graphical details in a logo should be several λ in size. If created smaller, the detail is lost during fabrication. Additionally, when creating names, it is often desired that they be readable on a 3 ft by 3 ft poster printout of the entire chip. It may be necessary to use only initials if large enough names cannot be fit in the logo's allotted space.

28. Chip Report

Owner: Justin Gries

Milestone Date: 10 April, 2007

28.1. Personnel:

Danny LaValle: **Editor; Content source**
Danny LaValle: **Content source**

Documentation on all milestones has been compiled and submitted for inclusion in the final chip report. A succinct description, along with a review of lessons learned was included for each task.

28.2. Lessons Learned

- Documentation is not something that should be put off until the last minute. It's far easier to generate it while the tasks are still fresh in the mind, rather than attempting to remember things about tasks that were completed some time ago, especially in a fast paced environment where details might fade quickly from memory as other tasks are undertaken.

29. Library Cluster, Appendix A: Library Cells

Table 18: Full list of library cells generated.
Along with transistor sizes for various strength cells.

MuddLib Standard Cell Library								
	Sizes are in the form "p, n"							
Cell	_lp	_1x	_1_5x	_2x	_3x	_4x	_6x	_8x
inv	4, 4	10, 7	15, 11	20, 14	30, 21	37, 27	60, 42	74, 54
nand2	4, 4	12, 12	18, 18	24, 24		48, 48		
nand3	4, 4	13, 16	20, 24					
nand4	4, 4	14, 20	21, 27					
nor2	4, 4	16, 8	24, 12	32, 16				
nor3	4, 4	21, 9	32, 14					
nor4	4, 4	25, 9						
a2o1i	4, 4	20, 14/7	30, 21/10	37, 27/14				
a22o2i	4, 4	20, 14	30, 21	37, 27				
o2a1i	4, 4	20/10, 14	30/15, 21	37/19, 27				
o22a2i	4, 4	20, 14	30, 21	37, 27				
invtri_dp	4, 4	20, 14	30, 21	37, 27				
invtri_c	4, 4	20,14 + 9,6	30, 21 + 9,6					
buf	4, 4	9,6 + 10, 7	9,6 + 15,11	9,6 + 20,14	9,6+ 30,21	9,6 + 37,27	14,9 + 60,42	18,12 + 74,54
and2	4, 4	6,6 + 10,7	6,6 + 15,11	6,6 + 20,14	9,9 + 30,21	12,12 + 37,27		
and3	4, 4	6,8 + 10,7	6,8 + 15,11	6,8 + 20,14	9,12 + 30,21	12,16 + 37,27		
and4	4, 4	6,9 + 10,7	6,9 + 15,11	6,9 + 20,14	9,13 + 30,21	12,18 + 37,27		
or2	4, 4	12,6 + 10,7	12,6 + 15,11	12,6 + 20,14	18,9 + 30,21	24,12 + 37,27		
or3	4, 4	14,6 + 10,7	14,6 + 15,11	14,6 + 20,14	21,9 + 30,21	28,12 + 37,27		
or4	4, 4	16,6 + 10,7	16,6 + 15,11	16,6 + 20,14	24,9 + 30,21	32,12 + 37,27		
a2o1	4, 4	9,6 + 10,7		9,6 + 20,14		18,12 + 37,27		
a22o2	4, 4	9,6 + 10,7		9,6 + 20,14		18,12 + 37,27		

Cell	_lp	_1x	_1_5x	_2x	_3x	_4x	_6x	_8x
o2a1	4, 4	9,6 + 10,7		9,6 + 20,14		18,12 + 37,27		
o22a2	4, 4	9,6 + 10,7		9,6 + 20,14		18,12 + 37,27		
buftri_dp	4, 4	9,6 + 20,14	9,6 + 30,21	9,6 + 37,27				
buftri_c	4, 4	9,6 + 20,14 + 9,6	9,6 + 30,21 + 9,6	9,6 + 37,27 + 9,6				
xor2	4, 4	20, 14 + 9,6	30, 21 + 9,6	9,6 + 9,6 + 20,14		9,6 + 18,12 + 37,27		
xor3	4, 4	9,6 + 9,6 + 10,7		9,6 + 9,6 + 20,14		9,6 + 18,12 + 37,27		
xnor2	4, 4	20, 14 + 9,6	30, 21 + 9,6	9,6 + 9,6 + 20,14		9,6 + 18,12 + 37,27		
xnor3	4, 4	9,6 + 9,6 + 10,7		9,6 + 9,6 + 20,14				
mux2i_dp	4, 4	20,14	30,21					
mux3i_dp	4, 4	30,21						
mux4i_dp	4, 4	30,21						
mux2_dp	4, 4	9,6 + 10,7	9,6 + 15,11	9,6 + 20,14	14,9 + 30,21	18,12 + 37,27		
mux3_dp	4, 4	9,6 + 10,7	9,6 + 15,11	9,6 + 20,14	14,9 + 30,21	18,12 + 37,27		
mux4_dp	4, 4	9,6 + 10,7	9,6 + 15,11	9,6 + 20,14	14,9 + 30,21	18,12 + 37,27		
mux2_c	4, 4	9,6 + 9,6 + 10,7	9,6 + 9,6 + 15,11	9,6 + 9,6 + 20,14	9,6 + 14,9 + 30,21	9,6 + 18,12 + 37,27		
mux3_c	4, 4	9,6 + 9,6 + 10,7	9,6 + 9,6 + 15,11	9,6 + 9,6 + 20,14	9,6 + 14,9 + 30,21	9,6 + 18,12 + 37,27		
mux4_c	4, 4	9,6 + 9,6 + 10,7	9,6 + 9,6 + 15,11	9,6 + 9,6 + 20,14	9,6 + 14,9 + 30,21			
flop_dp	4,4	output 10,7	out 15, 11	out 20,14	out 30,21	out 37,27		
flopr_dp	4,4	output 10,7	out 15, 11	out 20,14	out 30,21	out 37,27		

Cell	_lp	_1x	_1_5x	_2x	_3x	_4x	_6x	_8x
flops_dp	4,4	output 10,7	out 15, 11	out 20,14	out 30,21	out 37,27		
flopen_dp	4,4	output 10,7	out 15, 11	out 20,14	out 30,21	out 37,27		
flopenr_dp	4,4	output 10,7	out 15, 11	out 20,14	out 30,21	out 37,27		
flopsens_dp	4,4	output 10,7	out 15, 11	out 20,14	out 30,21	out 37,27		
flop_c	4,4	output 10,7	out 15, 11	out 20,14	out 30,21	out 37,27		
flopr_c	4,4	output 10,7	out 15, 11	out 20,14	out 30,21	out 37,27		
flops_c	4,4	output 10,7	out 15, 11	out 20,14	out 30,21	out 37,27		
flopen_c	4,4	output 10,7	out 15, 11	out 20,14	out 30,21	out 37,27		
flopenr_c	4,4	output 10,7	out 15, 11	out 20,14	out 30,21	out 37,27		
flopsens_c	4,4	output 10,7	out 15, 11	out 20,14	out 30,21	out 37,27		
fulladder	These three non-standard cells cannot be "sized" per the convention that is used in the rest of the cells in this chart							
srambit								
cambit								
invbuf						37,27		
clkinvbuf						18,9+36 ,18		
clkinvbufdual						(Paired 4x clkinvbuf cells...)		