CAPSTONE DESIGN REPORT

JACOB KOBZA
BRUNO PEYNETTI

EECS 362
COMPUTER ARCHITECTURE PROJECTS
WINTER 2016

Table of Contents

LIST OF FIGURES, TABLES

EXECUTIVE SUMMARY

Brief summary of the design problem. Brief statement of design approach, benefits of design and limitations

In this quarter-long project, we were asked to design and evaluate a pipelined processor that implements the DLX ISA, as well as design and test all individual components of the design of the processor. The implementation of all design elements used the Verilog Hardware Description Language. We were asked to evaluate the design for correctness and performance using benchmark programs. Specifically, we designed a 5-stage pipelined processor that was able to perform a subset of DLX instructions related to integer operations. Because of the added complexity of performing operations with floating point numbers in hardware, most of the floating point section of the DLX ISA was not implemented in our design.

Our approach to the design was to build modules that were first proven to work individually and within our constraints, and then to assemble each of the modules into a working design that eventually became a working pipelined processor. The entirety of our design was built using structural Verilog code because of the high control and extreme modularity of structural code in a hardware description language. With highly structural code and modularity, the design of the processor is easily modifiable. Each module is essentially a 'plug and play' module that acts as expected given standard inputs and outputs, and the implementation of an internal module does not interact with the implementation of its external module.

We first implemented the Arithmetic and Logic Unit as well as any and all components that were used by the unit. We tested each design as it was built and slowly assembled them into a fully functional ALU, testing the overall unit as necessary. We were able to quickly come up with a working ALU that performed all of the necessary operations for a basic DLX processor. To ensure reliability and accuracy, we built a Python-based testing module that compared tens of thousands of random operations in the ALU with the expected output.

The next step in our approach was to expand the ALU into a fully-operational Single-Cycle processor. This process consisted of first planning and creating a datapath for the Single-Cycle DLX processor, followed by carefully implementing and testing each of the components in our design. The design of the Single-Cycle processor implemented all the stages of processing in one cycle, as well as all the operations listed in the design requirements. It should be noted that the multiplication operation was left as 32-bit multiplication in behavioral code in Verilog for this step.

The final step in the development was to build the 5-stage pipelined processor. This design was based heavily on the design of the Single-Cycle processor, but the actual code was almost completely overhauled. We focused on strict modularity and invested time in creating clear code that could be easily understood and modified by any of our peers without the need of having an understanding of the entire architecture. Because of this, debugging was relatively straight-forward

and features were independently implemented and tested before adding them to the larger processor. The main additions required for this step were to strictly separate each pipeline stage in our design, add the pipeline registers and implement structural and data hazard detection.

The final design of the pipelined processor exceeds expectations in terms of the requirements. It correctly performs all expected DLX instructions, and it meets the timing requirement of a 800MHz clock frequency. The processor was built and tested for a 1GHz clock frequency in order to be sure that any new modules would easily surpass the 800MHz barrier. In terms of area, our design does not meet the area requirements since it has an area of approximately 40,000 um^2. Nevertheless, this area is considering both the integer register and floating point register. The floating point register is used only to implement the movfp2i and movi2fp, as well as the mult and multu signals.

BODY

Introduction

The goal of this project was to design and evaluate a fully functional 5-stage pipelined processor that implements the DLX ISA. Additionally, we were asked to design and test all the components of the processor, as well as the datapath and the control of the processor. We then were asked to evaluate this design for correctness and performance using benchmark programs that used the DLX ISA.

The physical constraints of the architecture were a maximum area of 35,000 um^2, a minimum clock frequency of 800 MHz, 32-bits, and a 5-stage design. Further constraints were set up directly by the DLX ISA, which we followed closely as we built the capabilities for each of the subset of DLX instruction set we were expected to implement. A complete outline of the instructions implemented by our processor can be seen in Appendix B at the back of this report.

Most of the knowledge to implement this design came either directly from this class or from EECS 361, Computer Architecture. In order to understand the DLX architecture and have a basis for our design, we used slides from the EECS362 class [1] , from EECS 361 [2], and 2 websites [3], and [4]. Any questions regarding a specific implementation of DLX ISA was consulted in either of the two websites ([3] and [4]). Any concerns regarding the pipeline implementation, data hazards, and all other specific computer architecture questions were solved using slides from EECS361 and EECS362.

Our approach is similar to the standard approach to building a pipelined processor except for a few design choices.

- Modularity : We used stage modules in order to build our processor (**see design**)
- Hazard detection and action : We individually tested assembly code for known hazards and created modules to either forward data or stall the pipeline for as long as needed
- Next instruction logic - We developed a component, **check_branch**, while implementing the Single-Cycle processor, that served as the main module for calculating the next PC value (address of the next instruction to fetch).
- Multiplication : We implemented a fast multiplication solution learned in EECS 336 Algorithms class that, if necessary, could be expanded to an even smaller multiplier at the expense of increasing the number of stall cycles. We are able to multiply 2 32-bit numbers by using 3 16-bit operations and a combination of additions and subtractions, as well as smart shifting logic.
- Stalling and Flushing : Additionally, we implemented a simple way to pause, or stall, the pipeline registers while the multiplication is being executed or a stall needs to be inserted to avoid a hazard.

Design Constraints and Requirements

There were two main constraints that we had to meet with our design: area and timing. We were expected to create a 5-stage pipelined processor with an area no greater than 35,000 um^2 and a clock frequency no lower than 800 MHz. Additionally, we were expected to run a subset of the DLX instructions. We had to restrict our code to Verilog code. Because of the nature of the DLX ISA, we had to implement a big endian architecture.

We had no components available for our design except for the basic gate implementation of AND,OR,XOR,NOT that is built-in in Verilog by typing the following characters: &,|,^,~. We built basic components out of these 4 operations and then we built larger components based on these basic components. As an example, we built a 1-bit 2:1 mux, which we then expanded to capability to build an n-bit 2:1 mux and also various other ratios for muxes (4:1, 8:1, 16:1, and 32:1). We also built a small register (flip flop) and from that we built all the necessary registers for our design.

Our final design met the timing constraint that was required, as all timing in the synthesis fit within the 800MHz frequency. This can be confirmed by looking in the appendix at the synthesis results, specifically by looking at the register to register timing analysis. Our execution stage originally had one timing violation, because too many arithmetic operations occurred simultaneously in one clock period. So, as is demonstrated in the performance testing section, we moved a small amount of logic out of the execution stage and eliminated the timing violations. The area constraint, meanwhile, is met without considering the floating point register file. A more detailed breakdown of the area of our processor can be seen in the performance testing section, but we are well within the goal of 35,000 um^2 without considering the floating point registers, which take up an additional 11,000 um^2.

Broader Considerations

The design of this pipelined processor is of high quality and easy adaptability. The way in which the datapath was designed and developed allows for modularity and therefore allows for possible improvements. Even though it is not in itself a commercial product and there are many commercial processors with highly optimized logic and better timing and area properties, this implementation complies with the design requirements for a DLX processor that implements the subset of DLX we were asked to implement. Further research into each component could help students discover the trade-offs of structural vs behavioral verilog, the effects of Verilog code on the final synthesis, among other possible improvements.

Design Description

As mentioned previously, our design is highly modular and therefore it is easy to adapt, understand, and modify. The top level of the architecture consists of mappings of the data and instruction memory modules, as well as the pipelined processor in itself. The reasoning for this implementation is to be able to synthesize the pipelined processor without using the data or instruction memory cache in the synthesis. We will now provide a detailed description of the system and sub-systems that form the pipelined processor.

The following figure provides a general layout of the architecture. Each module is explained in detail in this section, and its functionality and performance metrics explained later in the report.

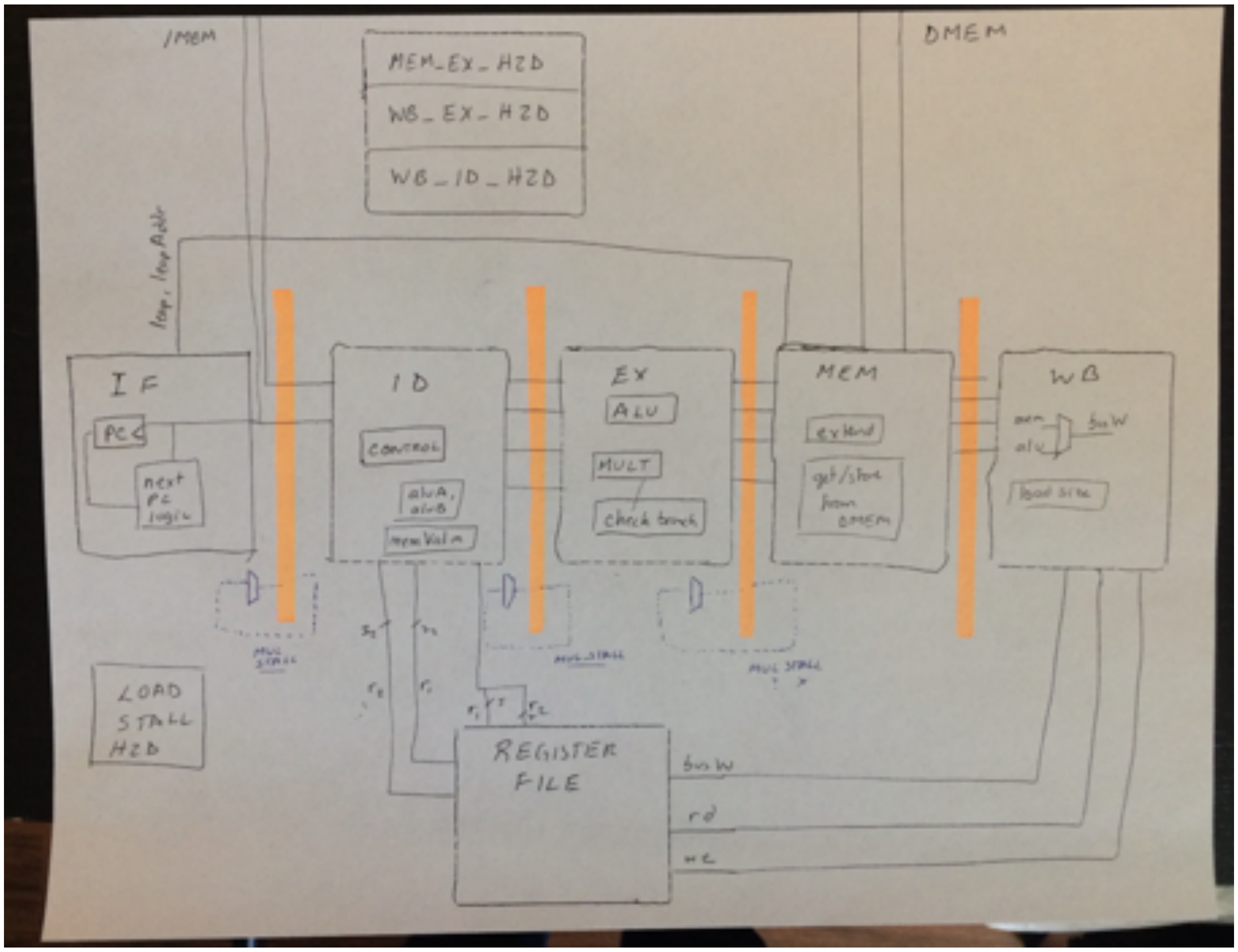


Figure 1 - Pipeline Architecture Design

Next-Instruction Logic

We mention the next-instruction logic first in order to justify the use of some particular signals in our design. We used the name **leap** for an operation that modifies the PC in order to move to a different instruction other than the standard PC+4. A leap operation is therefore a jump operation or a branch operation that meets its branching condition. We then created logic that determined if a leap would happen and we created logic to determine what the next address in the PC should be (**leapAddr**). This logic is used throughout our entire implementation, and the control signals (leap and leapAddr) are computed in the control module in the instruction decode stage and passed through to each pipeline stage for each instruction.

Top Level

This module holds the data and instruction memory, as well as the pipelined processor module. Its only inputs are a reset and a clock wire, which are set for basic operation and for resetting all values in all memory elements within the architecture. The data memory and instruction memory are the same data and instruction cache memory modules given to us, and we wired inputs and outputs according to the specifications given to us. The data and instruction memory are initialized when a program is first loaded, after which all data read from instruction or data memory is passed into the processor via an input, while any data written to the data memory arrives as an output of the processor.

Pipelined Processor

The pipelined processor holds the entire architectural design of our implementation. It contains clock and reset inputs, as well as a variety of data buses to interact with the instruction and data memories. There is one outgoing and one incoming bus for each type of data (data and instruction). Therefore, there are a total of 4 buses out of the pipelined processor. The pipelined processor itself holds virtually no logic. It does, however, contain each pipeline stage and the pipeline registers, as well as hazard detection logic. It assigns inputs into each pipeline stage, and it assigns the outputs of each stage to the following pipeline register. Afterwards, the processor extracts the output from the pipeline register and wires it back into the next stage module.

Pipeline Register

The pipeline registers are variable-width arrays of registers (flip flops) that take as inputs the output of the previous stage, and wire outputs directly to the following pipeline stage. Consequently, each pipeline register is named by the previous stage and the following stage (for example, the EX/MEM register). We implemented these registers using the module PipelineCtlRegN1, which is a standard N-bit register with a modification that adds a write-enable function. We made changes from the given PipelineCtlReg implementation in order to allow for variable width-registers and use the write-enable function to stall the pipeline when necessary.

Instruction Fetch Stage

The Instruction Fetch Stage is responsible for managing the PC logic that retrieves the next instruction and the next PC to be put into the registers. As inputs, we take leap and leap_addr (which come from the memory stage), as well as the write-enable for the PC (only 0 if a stall is necessary). Its components include a 32-bit PC register, a 32-bit adder that adds +4 to the value of the PC, and a mux that decides if the PC register's input is the PC+4 value or the leap_addr value. The processor continues to move to the next instruction (either PC+4 or the leap address) each clock cycle unless a stall is detected, in which case the PC stays the same until a stall is no longer needed and the processor can move to the next instruction.

Instruction Decode Stage

The Instruction Decode Stage is responsible for reading the specified location in instruction memory (specified by the PC passed from the instruction fetch stage) to retrieve an instruction. With this instruction, we are able to determine the control signals needed for later in the pipeline. Using mainly the function field and opcode field, we generate control signals in the control module, and these are passed through to the ID/EX pipeline register, and further through the pipeline. Additionally, the instruction decode stage is responsible for splitting up the instruction into its different parts. After this is done, the source registers (rs1 and rs2) are outputs of the instruction decode stage, so the values in these registers can be retrieved from the register file.

Execution Stage

The execution stage carries out all arithmetic and logical operations necessary to complete the current instruction in the pipeline. For most instructions, this arithmetic or logical operation is directly dictated by what the instruction is (for example, an add instruction does an add in the ALU and a shift instruction does a shift in the ALU). However, in some instructions, addition must be done in the execution stage to compute an immediate or offset value. This occurs mainly in load and store instructions, because the given address must be added to an immediate value to determine the address to write to/read from in data memory. The execution stage also contains the multiplier module. This module cannot be included in the ALU, because it takes multiple cycles and requires the rest of the processor to stall while it finishes computation. As a result, the execution module contains a mux that chooses between the ALU output and the output of the multiplier unit, depending on the type of instruction (determined by a control signal). An additional mux is used to determined whether the instruction is a "movfp2i", in which case the result of the ALU execution is replaced by the value in the floating point register needed. In the end, the required computational outputs, as well as the inputs from the previous stage, as passed as outputs and into the EX/MEM register to be used by the memory stage.

Memory Stage

The Memory Stage of the pipeline serves one main purpose: to retrieve data from, and write data to, the data memory module. Since the data memory is separate from the pipeline processor and located all the way up in the top level module, this cannot be done directly inside the memory stage. Consequently, the memory stage itself doesn't contain any logic; rather, its main purpose is to send the proper inputs to the data memory as outputs, and receive the data output from the data memory as inputs. Additionally, the memory stage is simply made up of many wires, passing the other needed signals through to the write-back stage through the MEM/WB pipeline register.

Write-Back Stage

The main responsibility of the write-back stage is to write the data that has been computed back into the register file after the computation has been done in the execution stage. This applies to a very large range of instructions, such as R-Type and I-Types (add, addi, etc) that write the result to a register, as well as loads, and some jump instructions in which the current PC is written to a register (jal and jalr). Since the write-back stage is the only stage not followed by a pipeline register, the only outputs are the data needed to write to the appropriate register in the register files (both integer register file and floating point register file). Additionally, the write-back stage also contains a component, called outData, that chooses the correct portion of the data when a load occurs, since a byte, halfword or word can all be loaded from data memory to the register file.

Register File

The register file is made up of 32 general purpose registers, each 32 bits long, that can be written to or read from using the rd (destination register, to write to register file), ra and rb (source registers, to read from register file) identifiers. Using these identifier (5 bit numbers) we implemented a 5:32 decoder so only the correct write-enable in the register file is enabled for register writes. To extract the correct data from the specified source registers, meanwhile, two 32:1 muxes are used to choose the data from the correct registers.

FP Register-File

The floating point register file is very similar to the register file outlined above, with a few small but important additions. The floating point register file still contains 32 registers, each 32 bits long. The main difference from the register file above, however, is that busW, the data being written to the floating point register file, is 64 bits, because the result of a multiply is 64 bits and must be stored in two consecutive registers. In order to account for this, the write enable has the option of having two enabled registers rather than just one. In the case when only one register is being written to, the decoder is used to generate the write enables as above. However, if two registers are being written to, only the top four bits of rd number (register being written to) are taken, which is equivalent to integer division by 2. This 4-bit signal is then sent to a 4:16 decoder, which gives us a write enable that is enabled for only 1 of 16 registers, each representing two registers in the register file. Each one of these write enables is then fed into a pair of consecutive registers, allowing two consecutive registers to be written to. The other change in the floating point register file is that the 64 bit busW signal needs to be split into its upper and lower parts, and they need to be put into the correct registers. To do this, we label the two halves the "even" half (even numbered registers) and the "odd" half (odd numbered registers), and put the upper half of busW into the "even" half and the other half into the "odd" half. This only occurs when two registers are being written to, so a mux is used to decide whether busW is split into these two halves.

Hazards

In order to properly implement any pipelined processor, you need to account for the situations in which consecutive instructions use the same data, and it hasn't been changed in time to use in the following instruction. These situations are called "hazards", and accounting for them is an important part of the design of any pipelined processor. These hazards can be accounted for by using a combination of data forwarding (passing data between pipeline stages to allow it to be used before it is written back to the register file) and stalling the processor when necessary to allow required computations to run to completion. In all, there are five different kinds of hazards we had to account for: multiplication stalls, stalling from a load hazard, MEM-EX data forwarding, WB-EX data forwarding and WB-ID data forwarding. Each of the forwarding cases have the same structure: we built a component for each, which sits in the main pipeline processor module, that outputs a control signal determining whether the given hazard occurs in the current clock cycle. This control signal is fed into the selector bit of a mux that goes into the stage the data is forwarded to (execution or instruction decode depending on the hazard type) in order to select the current value of the data in question or the forwarded value. For stalling, we have to make sure two things occur: the PC does not move to the next instruction, and the pipeline registers do not take in new values. In order to do this, we created a stall control signal, and fed it to both the write enable of the PC and the write enables of the pipeline registers, allowing the processor to wait for the execution (of either the current instruction or a multiply) to finish before continuing.

Trap detection (end of execution)

The trap instruction on .asm (**trap 0x300**) was used in order to indicate the end of execution (end of simulation). In order to implement this, we used a similar approach to the single cycle, where we stopped execution as soon as this instruction was reached. Nevertheless, read/write operations can be taking place in further pipeline stages, and therefore we implemented logic in the control (ID Stage) that detected a trap. This trap signal was passed until it reached the memory stage, point at which no more read/writes could happen in the program. It is not until this point that the simulation test file stops execution.

Multiplication

The multiplication operation was implemented as a 4-cycle operation using fast multiplication as learned in the EECS 336 - Algorithms class. The operation is based off a finite state machine with the following steps. Given 32-bit A and B operands, A*B =

- H = A_H * B_H
- L = B_L * B_H
- P = (A_H + A_L) * (B_H + B_L)
- Z = P - H - L
- Result = A*B = (H<<32)+(Z<<16)+(L);

The implementation was distributed over all 4 cycles in order to use only one multiplier and one adder each cycle. This avoided creating any critical paths and eliminated the possibility of a synthesis in which 2 or more adders, and 2 or more multipliers were synthesized within the multiplier.

Performance Testing

In order to test the design and assess its performance, we tested exhaustively all modules until we knew each stage worked. Afterwards, we started combining modules and testing until all modules worked correctly. When the pipeline processor was assembled, we tested the processor without data hazard, and incrementally added new instructions and tested them individually, debugging where necessary and making incremental changes. A description of our approach to testing and performance evaluation follows:

In order to run each test file, a Makefile was generated. The command **make <file>** will compile the dlx code for the file **file.asm** into instr.hex and data.hex. Afterwards, it will compile the design using IVERILOG. Execution of the resulting binary can be achieved by typing **./bin/test_pipe_<file>**. As an example, to test Fibonacci, first **make fibonacci** and then **./bin/test_fibonacci**

Debugging of individual components

We debugged every component starting from the ALU. We created test files that covered most modules. We successfully finished successfully testing all individual modules before we started building the overall design. This way, we maintained our approach to modularity and efficiently built the overall design.

Debugging of overall design

We implemented simple test programs and used the **dlxasm** script to render hex files of these implementations. Given that we knew that each element worked, it was first essential to test the simple operations: writing and reading from a register. This tested both the instruction fetch, the correct passing of pipeline signals, and the correct wiring of the register file and ALU. We then tested input/output from memory. It should be noted that we added **nop** instructions whenever a hazard was detected in our code, since this implementation had no hazard detection or data forwarding. We then moved on to test the **fib.asm** file adding **nop** whenever necessary to eliminate all data hazards. Only after the successful test of **fib.asm** did we implement hazard detection and forwarding.

Hazard detection and implementation

We incrementally added hazard detection and forwarding to our design. We started by adding specific hazards in an **asm** file. After successfully implementing the hazard detection, we implemented the corresponding data forwarding. This allowed a very modular detection of hazards and an incremental implementation of the end system.

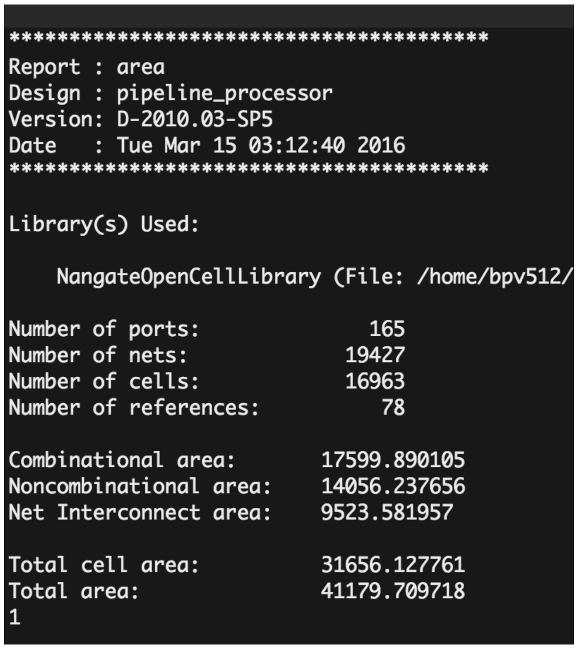
Multiplication

Just as any other module, the multiplication was tested independently. In order to correctly implement the multiplication, we had to stall the pipeline registers and the PC register. We opted to create write-enable signals on all of these registers. We then tested the multiplication successfully after adding the floating point register file and the necessary logic.

Synthesis results

Area

After testing exhaustively the pipelined processor with both test files (fib.asm and unsigned_sum.asm) and more test files to include all modules, the output of the synthesis was as follows:



```
****************************************
Report : area
Design : pipeline_processor
Version: D-2010.03-SP5
Date   : Tue Mar 15 03:12:40 2016
****************************************

Library(s) Used:

    NangateOpenCellLibrary (File: /home/bpv512/

Number of ports:              165
Number of nets:             19427
Number of cells:            16963
Number of references:          78

Combinational area:       17599.890105
Noncombinational area:    14056.237656
Net Interconnect area:    9523.581957

Total cell area:          31656.127761
Total area:               41179.709718
1
```

Figure 2 - Initial Area Synthesis

As observed in the synthesis output, the cell area is 31,656 um^2 and the total area of the module is 41,179 um^2. This number is roughly 17% over the specified constraint for area. In order to identify the bottleneck for area, we synthesized individual modules and obtained the following information:

Module	Area (um^2)
Floating Point Register File	11,400
Register File	11,040
Execution Stage	9,188
Multiplier	6,500
Pipeline Registers (all)	5,794
ALU	1200
Control	151
Hazard Detection (all)	166

Looking at the largest modules and attempting to reduce the area of these could go furthest in attempting to reduce total area. Nevertheless, both register files are optimized almost to a maximum, since the decoders and registers are already optimized. Having 1024 Flip-Flops generates almost 5000 um^2 in area, and when adding all the interconnections, the area increases quickly.

Timing

When testing the timing of the system at 800MHz, the timing was violated multiple times. After tracing the source of the slack violation, the critical path was found to be the path from the output of the ID_EX register to the input of the EX_MEM register regarding the leap address and leap value. A simplistic design of the critical path is shown, noting the path that bits from the opA bus takes in the calculation of next address. The input to the EX_MEM address represents the leap_addr instruction.

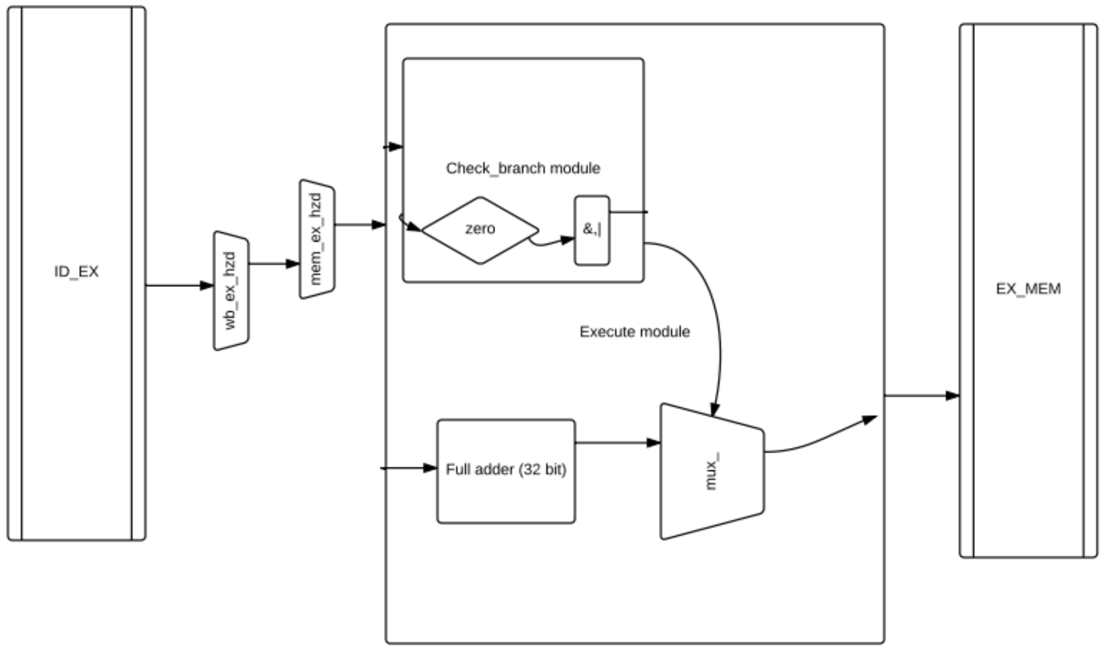


Figure 3 - Identified Critical Path

Improvements and results

We implemented a few improvements in order to reduce the area and the timing of various modules. These are:

- Muxes/Stalling
    - Initially, stalling was performed by using muxes for the pipeline registers. To save space, we built this capability into the pipeline registers by modifying the logic and adding a write enable bit.
- Multiplier:
    - In the 3rd stage, the operation (a_h+a_l)*(b_h*b_l) was being executed. We offloaded the two additions to the 1st and second states of the multiplier. This eliminated any critical paths and reduced area by synthesizing only one full adder rather than two parallel ones. The only negative of this implementation is the addition of 2 more 32-bit registers
- Next Address Logic
    - We modified the **zero** module which was originally implemented as a cascading 32-bit or + a NOT gate into behavioral logic, where the synthesis optimizes such operation to probably a logarithmic approach using parallel gates, thus greatly decreasing the delay through this module.
    - We offloaded the full adder and final mux from the execution stage to the memory stage. This modification eliminated the critical path of the next address logic.

The results of this operation were a successful synthesis regarding timing. The resulting area report for synthesis is shown below:

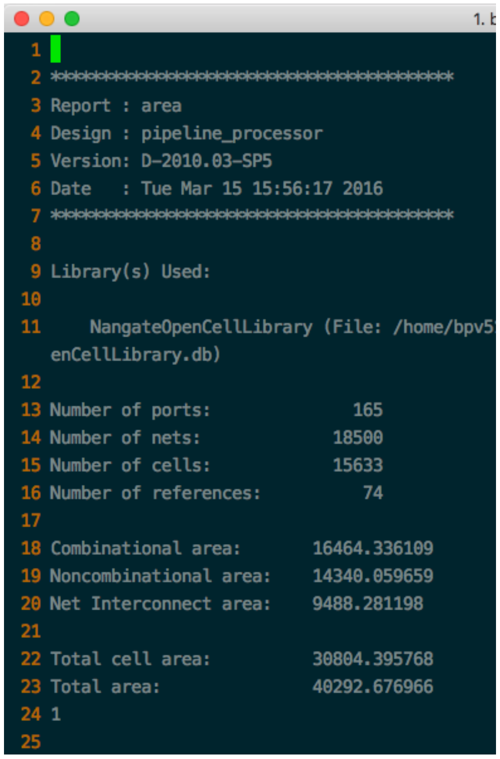

```
1. b

*****************************************
Report : area
Design : pipeline_processor
Version: D-2010.03-SP5
Date   : Tue Mar 15 15:56:17 2016
*****************************************

Library(s) Used:

    NangateOpenCellLibrary (File: /home/bpv5
enCellLibrary.db)

Number of ports:               165
Number of nets:              18500
Number of cells:             15633
Number of references:           74

Combinational area:       16464.336109
Noncombinational area:    14340.059659
Net Interconnect area:    9488.281198

Total cell area:          30804.395768
Total area:               40292.676966
1

```

Figure 4 - Improved Area Synthesis

The new area shows a decrease of **2%** which is almost negligible, and still **14%** above the area constraints. Nevertheless, it should be noted that without using the floating point register file and using a modified, smaller register file for floating point registers in order to satisfy the requirements of the project could have had a large effect on the resulting area. Additionally, optimizing the multiplier area could provide a significant decrease in area, but it is likely that the total area would still be slightly above the constraint.

Additionally, the resulting report regarding timing is attached in the project. After modifying the critical path for both the multiplication and the next address logic, the resulting design satisfied the timing requirement of 800MHz.

Test results with given assembly instructions

The final design was tested with the test assembly files given to us: **fib.asm, unsigned_sum.asm,** and **quicksort.asm.** It should be noted that we were not able to determine the correct output of the quicksort file in either the pipelined process or or the single cycle processor we implemented before implementing this design. The resulting memory file is close to being correctly sorted but it is not fully sorted. We suspect that there is a small error in the assembly file which leads to an early termination of the code.

In order to test correctness of the implementation, we created 2 simple Python scripts (shown in appendices) that calculate the nth fibonacci number as well as the unsigned sum of the numbers in an array, and display the result in both decimal and hexadecimal. We then proceeded to look at the memory writes to identify when and where exactly data was being written, and compared the outputs. The output of the fib.asm and unsigned_sum.asm simulations are also attached in appendices C and D.

Test of features not implemented in test files (multiplication, jal)

To test other requirements not shown in the unsigned sum or Fibonacci test programs, we implemented our own programs. We show a simple multiplication file, **mult.asm** in Appendix E, which performs a simple multiplication. We did not implement data hazard detection and data forwarding for any operation making use of floating point registers. Thus, we simply inserted **nop** instructions to eliminate all possible hazards.

Limitations, weaknesses

The design of the 5-stage pipelined processor is completely functional and up to the standards and expectations of the class. We did not do any gate-level synthesis, which means that the area value is just an estimate, and it is within 15% of the goal. We believe that a few improvements could easily reduce the size to within the constraints.

A weakness of the design is the extreme modularity of the overall processor. We considered this modularity an advantage when building the processor, but the high degree of modularity made it confusing at times to implement small changes to the design. In addition, data hazard detection should be done in the Instruction Decode Stage, but we have individual modules in the top level pipeline processor to identify these. Even though this design works correctly, it is against standard practice and therefore could be difficult to upgrade.

Conclusions

Summarize your design and how well it met the requirements. Suggest potential future improvements.

REFERENCES

- [1] -EECS 361 Class Slides – G. Memik. Northwestern University, Fall 2014
- [2] -EECS 362 Class Slides – Y. Fan and C. Yao, Northwestern University, Winter 2016
- [3] – G. Prabhu, DLX Instruction Set. Iowa State University, http://web.cs.iastate.edu/~prabhu/Tutorial/PIPELINE/DLXinstrSet.html
- [4] – K. Pingali, The DLX Instruction Set Architecture, University of Texas, http://www.cs.utexas.edu/~pingali/CS378/2011sp/lectures/DLX.pdf
- [5] Github Repository – http://github.com/bpeynetti/362Project

Appendix A: Layout of Github Repository

The basic layout of our Github repository is that each section of our design (ALU, Single Cycle and Pipeline) is self-contained and each contains four main folders: bin, which contains all the binary files from the compilations of our designs; py, which contains any python scripts we used for testing; src, which contains all verilog source files used for this part of the design; and tests, which contains all the testbenches we used to confirm the correct functionality of our components. In addition, the main folder for each section of the design contains a Makefile for that section of the design, so the entire design can be compiled with one simple command. The detailed layout of the repository can be seen below:

- Pipeline_project
    - ALU_files
        - bin
        - py
        - src
        - tests
    - Examples
        - Contains example verilog files we were given as references to help us figure out our designs for various components
    - Icarus
        - Contains the binaries for running Icarus Verilog simulation and compilation. We put this in our repository because we did most of our development on cloud9, an online code-sharing platform
    - Pipeline_files
        - bin
        - py
        - src
            - lib
                - Contains the "library" files that are not unique to the pipeline but have already been implemented elsewhere in the repository
        - synthesis
            - Contains the output of the synthesis for the pipeline processor
        - tests
    - Single_Cycle_files
        - bin
        - src
        - Tests

Appendix B: List of DLX instructions implemented

Appendix C: Fibonacci

The memory output of fib.asm is shown below, side by side with the output of the Python script fib.py

```
Brunos-MacBook-Pro:Pipeline_files brunopeynetti$ ./bin/test_pipe
VCD info: dumpfile dump.vcd opened for output.
writing to mem at 00002008 val 00000001 size  3
writing to mem at 0000200c val 00000002 size  3
writing to mem at 00002010 val 00000003 size  3
writing to mem at 00002014 val 00000005 size  3
writing to mem at 00002018 val 00000008 size  3
writing to mem at 0000201c val 0000000d size  3
writing to mem at 00002020 val 00000015 size  3
writing to mem at 00002024 val 00000022 size  3
writing to mem at 00002028 val 00000037 size  3
writing to mem at 0000202c val 00000059 size  3
writing to mem at 00002030 val 00000090 size  3
writing to mem at 00002034 val 000000e9 size  3
writing to mem at 00002038 val 00000179 size  3
writing to mem at 0000203c val 00000262 size  3
writing to mem at 00002040 val 000003db size  3
writing to mem at 00002044 val 0000063d size  3
writing to mem at 00002048 val 00000a18 size  3
writing to mem at 0000204c val 00001055 size  3
writing to mem at 00002050 val 00001a6d size  3
writing to mem at 00002054 val 00002ac2 size  3
writing to mem at 00002058 val 0000452f size  3
writing to mem at 0000205c val 00006ff1 size  3
writing to mem at 00002060 val 0000b520 size  3
writing to mem at 00002064 val 00012511 size  3
writing to mem at 00002068 val 0001da31 size  3
writing to mem at 0000206c val 0002ff42 size  3
writing to mem at 00002070 val 0004d973 size  3
writing to mem at 00002074 val 0007d8b5 size  3
writing to mem at 00002078 val 000cb228 size  3
writing to mem at 0000207c val 00148add size  3
writing to mem at 00002080 val 00213d05 size  3
writing to mem at 00002084 val 0035c7e2 size  3
writing to mem at 00002088 val 005704e7 size  3
writing to mem at 0000208c val 008cccc9 size  3
writing to mem at 00002090 val 00e3d1b0 size  3
writing to mem at 00002094 val 01709e79 size  3
writing to mem at 00002098 val 02547029 size  3
writing to mem at 0000209c val 03c50ea2 size  3
writing to mem at 000020a0 val 06197ecb size  3
writing to mem at 000020a4 val 09de8d6d size  3
writing to mem at 000020a8 val 0ff80c38 size  3
writing to mem at 000020ac val 19d699a5 size  3
writing to mem at 000020b0 val 29cea5dd size  3
writing to mem at 000020b4 val 43a53f82 size  3
Hitting a trap, end, waiting 1 clock cycle and finishing
```

```
Brunos-MacBook-Pro:362Project  brunopeynetti$  python  fib.py
0xb4
0 - 0  -  0x0
1 - 1  -  0x1
2 - 1  -  0x1
3 - 2  -  0x2
4 - 3  -  0x3
5 - 5  -  0x5
6 - 8  -  0x8
7 - 13  -  0xd
8 - 21  -  0x15
9 - 34  -  0x22
10 - 55  -  0x37
11 - 89  -  0x59
12 - 144  -  0x90
13 - 233  -  0xe9
14 - 377  -  0x179
15 - 610  -  0x262
16 - 987  -  0x3db
17 - 1597  -  0x63d
18 - 2584  -  0xa18
19 - 4181  -  0x1055
20 - 6765  -  0x1a6d
21 - 10946  -  0x2ac2
22 - 17711  -  0x452f
23 - 28657  -  0x6ff1
24 - 46368  -  0xb520
25 - 75025  -  0x12511
26 - 121393  -  0x1da31
27 - 196418  -  0x2ff42
28 - 317811  -  0x4d973
29 - 514229  -  0x7d8b5
30 - 832040  -  0xcb228
31 - 1346269  -  0x148add
32 - 2178309  -  0x213d05
33 - 3524578  -  0x35c7e2
34 - 5702887  -  0x5704e7
35 - 9227465  -  0x8cccc9
36 - 14930352  -  0xe3d1b0
37 - 24157817  -  0x1709e79
38 - 39088169  -  0x2547029
39 - 63245986  -  0x3c50ea2
40 - 102334155  -  0x6197ecb
41 - 165580141  -  0x9de8d6d
42 - 267914296  -  0xff80c38
43 - 433494437  -  0x19d699a5
44 - 701408733  -  0x29cea5dd
45 - 1134903170  -  0x43a53f82
```

Appendix D: Unsigned sum

The assembly code for an unsigned sum should result in loads and one store for the value 0xfffffff0. The file unsigned_sum_output.txt within the tests directory contains a monitor signal for every clock cycle, as well as displays of values in a variety of register, floating point registers, instruction, PC, and clock.

```
Brunos-MacBook-Pro:Pipeline_files brunopeynetti$ make
pipeline
../dlxasm -C ./test/instr.hex -D ./test/data.hex ./test/test.asm
Starting pass 1.
Starting pass 2.
Last text address: 0x30
Last data address: 0x2028
iverilog -o ./bin/test_pipe -s testbench src/lib/*.v src/*.v
test/pipeline_test.v
Brunos-MacBook-Pro:Pipeline_files brunopeynetti$
./bin/test_pipe
VCD info: dumpfile dump.vcd opened for output.
writing to mem at 00002000 val fffffff0 size  3
Hitting a trap, end, waiting 1 clock cycle and finishing
```

```
Brunos-MacBook-Pro:assembly_example brunopeynetti$ python
unsigned_sum.py
r4,r5,r6:  3221225472 3221225472 32
r4,r5,r6:  536870912 3758096384 28
r4,r5,r6:  268435456 4026531840 24
r4,r5,r6:  251658240 4278190080 20
r4,r5,r6:  15728640 4293918720 16
r4,r5,r6:  983040 4294901760 12
r4,r5,r6:  61440 4294963200 8
r4,r5,r6:  3840 4294967040 4
r4,r5,r6:  240 4294967280 0
r6: 4294967280 0xfffffff0
```

Appendix E

Multiplication input asm file and output in writes to memory. Note that hazards for the floating point register file were not implemented, so simple **nop** instructions were inserted to eliminate the hazards

Multiply.asm

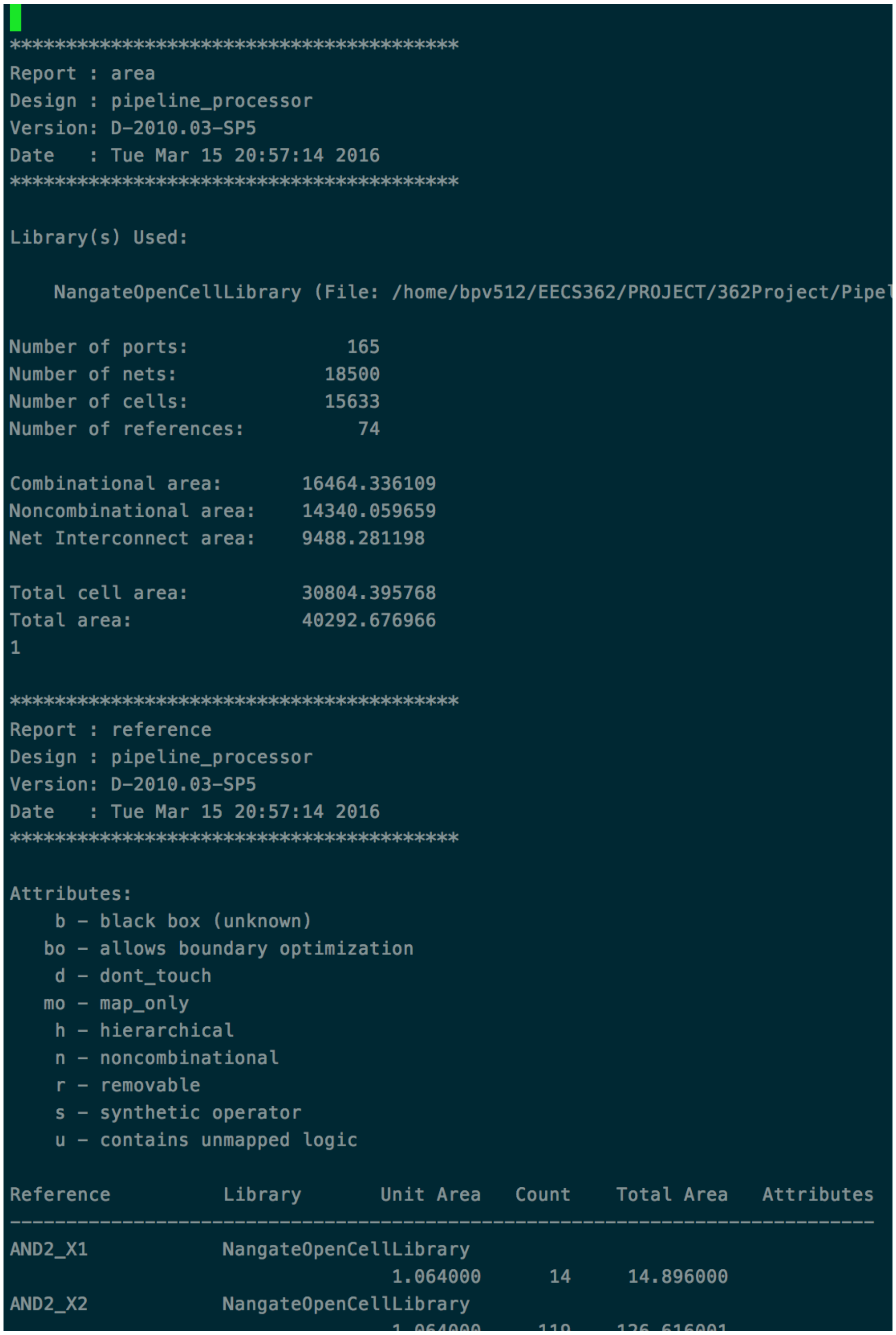
```
# MAIN
.text 0x0000
.data 0x2000
.global _dat
_dat:
.word 0

.text
.proc _usum
.global _usum
_usum:
   xor r0, r0, r0      #Remains zero
   xor r4, r4, r4
   xor r5,r5,r5
   addi r4, r4, 0x4
   addi r5, r5, 0x10
   nop
   nop
   nop
   movi2fp f1,r4
   movi2fp f2,r5
   nop ; data hazard
   nop ; data hazard
   nop ; data hazard
   nop
   nop
   mult f4,f1,f2 ; l f4 = x1234 * xAAAA =
   nop ; should be done with multiplication
   nop ; in wb stage
   nop ;
   nop
   movfp2i r5, f5 ; moving the lower 32 bits to r5
   nop
   nop
   nop
   nop
   sw _dat(r0), r5
   nop
   trap #0x300
.endproc _usum
```

Output of multiply.asm execution

```
Brunos-MacBook-Pro:Pipeline_files brunopeynetti$ make pipeline
../dlxasm -C ./test/instr.hex -D ./test/data.hex ./test/test.asm
Starting pass 1.
Starting pass 2.
Last text address: 0x74
Last data address: 0x2004
iverilog -o ./bin/test_pipe -s testbench src/lib/*.v src/*.v
test/pipeline_test.v

Brunos-MacBook-Pro:Pipeline_files brunopeynetti$ ./bin/test_pipe
VCD info: dumpfile dump.vcd opened for output.
MULTIPLICATION

MULTIPLICATION

MULTIPLICATION

MULTIPLICATION

MULTIPLICATION

writing to mem at 00002000 val 00000040 size  3
Hitting a trap, end, waiting 1 clock cycle and finishing
Brunos-MacBook-Pro:Pipeline_files brunopeynetti$
```

Appendix F - Area Synthesis Result

The result of the area synthesis is located in the project files, in the synthesis/log folder.

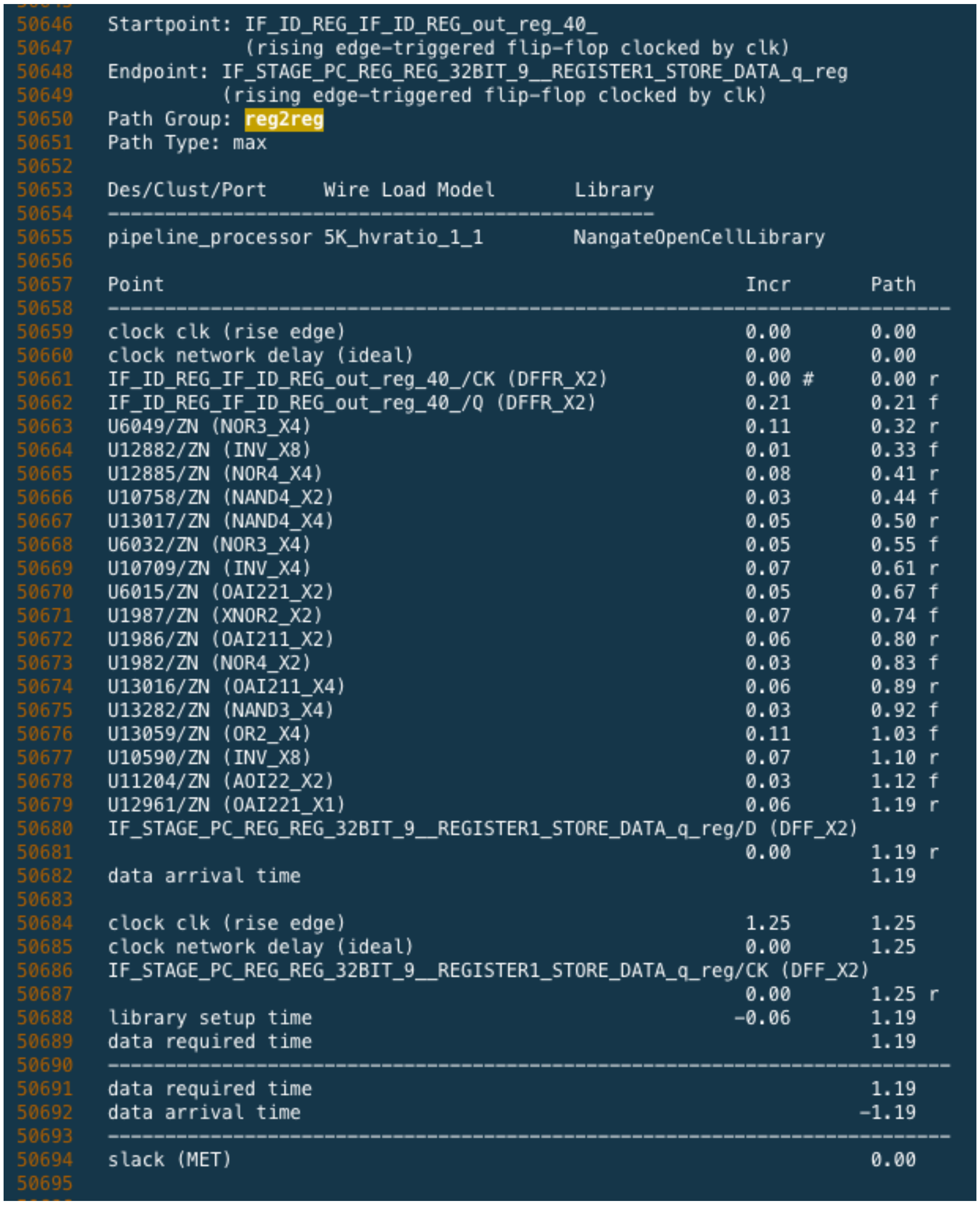


```
****************************************
Report : area
Design : pipeline_processor
Version: D-2010.03-SP5
Date   : Tue Mar 15 20:57:14 2016
****************************************

Library(s) Used:

    NangateOpenCellLibrary (File: /home/bpv512/EECS362/PROJECT/362Project/Pipel

Number of ports:              165
Number of nets:             18500
Number of cells:            15633
Number of references:          74

Combinational area:       16464.336109
Noncombinational area:    14340.059659
Net Interconnect area:    9488.281198

Total cell area:          30804.395768
Total area:               40292.676966
1

****************************************
Report : reference
Design : pipeline_processor
Version: D-2010.03-SP5
Date   : Tue Mar 15 20:57:14 2016
****************************************

Attributes:
    b - black box (unknown)
   bo - allows boundary optimization
    d - dont_touch
   mo - map_only
    h - hierarchical
    n - noncombinational
    r - removable
    s - synthetic operator
    u - contains unmapped logic

Reference          Library       Unit Area   Count    Total Area   Attributes
-----------------------------------------------------------------------------
AND2_X1            NangateOpenCellLibrary
                                  1.064000      14     14.896000
AND2_X2            NangateOpenCellLibrary
```

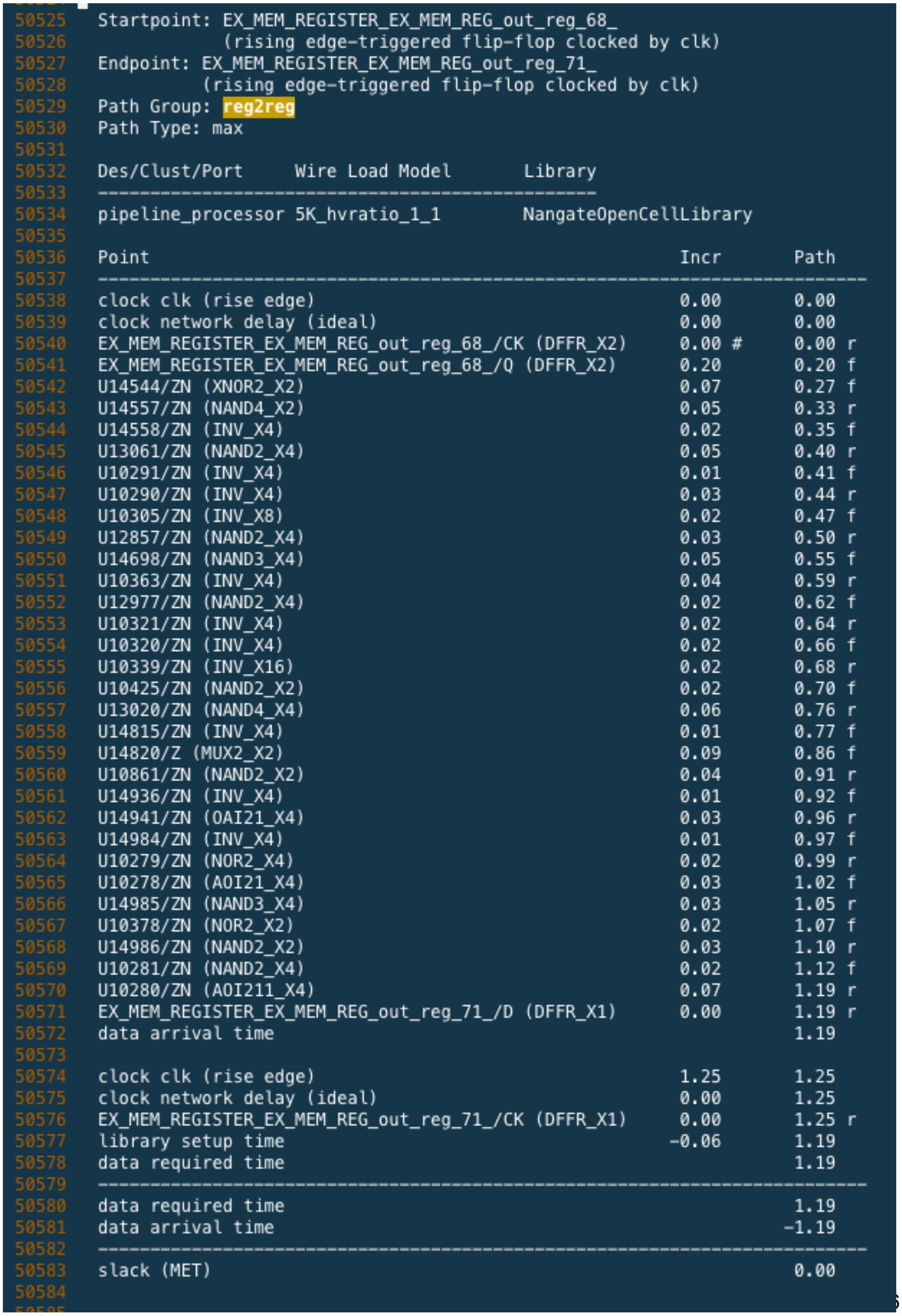
APPENDIX G - TIMING RESULTS

The most important timing constraints are the reg2reg delays. We will show a few of the largest delays, which correspond to the multiplier and the next address calculation.

The complete timing report is available in the synthesis folder on the github repository. 2 of the longest delays are seen in screenshots:



```
Startpoint: IF_ID_REG_IF_ID_REG_out_reg_40_
            (rising edge-triggered flip-flop clocked by clk)
Endpoint: IF_STAGE_PC_REG_REG_32BIT_9__REGISTER1_STORE_DATA_q_reg
          (rising edge-triggered flip-flop clocked by clk)
Path Group: reg2reg
Path Type: max

Des/Clust/Port     Wire Load Model       Library
------------------------------------------------
pipeline_processor 5K_hvratio_1_1        NangateOpenCellLibrary

Point                                                 Incr       Path
-----------------------------------------------------------------------
clock clk (rise edge)                                 0.00       0.00
clock network delay (ideal)                           0.00       0.00
IF_ID_REG_IF_ID_REG_out_reg_40_/CK (DFFR_X2)          0.00 #     0.00 r
IF_ID_REG_IF_ID_REG_out_reg_40_/Q (DFFR_X2)           0.21       0.21 f
U6049/ZN (NOR3_X4)                                    0.11       0.32 r
U12882/ZN (INV_X8)                                    0.01       0.33 f
U12885/ZN (NOR4_X4)                                   0.08       0.41 r
U10758/ZN (NAND4_X2)                                  0.03       0.44 f
U13017/ZN (NAND4_X4)                                  0.05       0.50 r
U6032/ZN (NOR3_X4)                                    0.05       0.55 f
U10709/ZN (INV_X4)                                    0.07       0.61 r
U6015/ZN (OAI221_X2)                                  0.05       0.67 f
U1987/ZN (XNOR2_X2)                                   0.07       0.74 f
U1986/ZN (OAI211_X2)                                  0.06       0.80 r
U1982/ZN (NOR4_X2)                                    0.03       0.83 f
U13016/ZN (OAI211_X4)                                 0.06       0.89 r
U13282/ZN (NAND3_X4)                                  0.03       0.92 f
U13059/ZN (OR2_X4)                                    0.11       1.03 f
U10590/ZN (INV_X8)                                    0.07       1.10 r
U11204/ZN (AOI22_X2)                                  0.03       1.12 f
U12961/ZN (OAI221_X1)                                 0.06       1.19 r
IF_STAGE_PC_REG_REG_32BIT_9__REGISTER1_STORE_DATA_q_reg/D (DFF_X2)
                                                      0.00       1.19 r
data arrival time                                                1.19

clock clk (rise edge)                                 1.25       1.25
clock network delay (ideal)                           0.00       1.25
IF_STAGE_PC_REG_REG_32BIT_9__REGISTER1_STORE_DATA_q_reg/CK (DFF_X2)
                                                      0.00       1.25 r
library setup time                                   -0.06       1.19
data required time                                               1.19
-----------------------------------------------------------------------
data required time                                               1.19
data arrival time                                               -1.19
-----------------------------------------------------------------------
slack (MET)                                                      0.00
```



```
Startpoint: EX_MEM_REGISTER_EX_MEM_REG_out_reg_68_
            (rising edge-triggered flip-flop clocked by clk)
Endpoint: EX_MEM_REGISTER_EX_MEM_REG_out_reg_71_
          (rising edge-triggered flip-flop clocked by clk)
Path Group: reg2reg
Path Type: max

Des/Clust/Port     Wire Load Model       Library
------------------------------------------------
pipeline_processor 5K_hvratio_1_1        NangateOpenCellLibrary

Point                                                   Incr       Path
--------------------------------------------------------------------------
clock clk (rise edge)                                   0.00       0.00
clock network delay (ideal)                             0.00       0.00
EX_MEM_REGISTER_EX_MEM_REG_out_reg_68_/CK (DFFR_X2)     0.00 #     0.00 r
EX_MEM_REGISTER_EX_MEM_REG_out_reg_68_/Q (DFFR_X2)      0.20       0.20 f
U14544/ZN (XNOR2_X2)                                    0.07       0.27 f
U14557/ZN (NAND4_X2)                                    0.05       0.33 r
U14558/ZN (INV_X4)                                      0.02       0.35 f
U13061/ZN (NAND2_X4)                                    0.05       0.40 r
U10291/ZN (INV_X4)                                      0.01       0.41 f
U10290/ZN (INV_X4)                                      0.03       0.44 r
U10305/ZN (INV_X8)                                      0.02       0.47 f
U12857/ZN (NAND2_X4)                                    0.03       0.50 r
U14698/ZN (NAND3_X4)                                    0.05       0.55 f
U10363/ZN (INV_X4)                                      0.04       0.59 r
U12977/ZN (NAND2_X4)                                    0.02       0.62 f
U10321/ZN (INV_X4)                                      0.02       0.64 r
U10320/ZN (INV_X4)                                      0.02       0.66 f
U10339/ZN (INV_X16)                                     0.02       0.68 r
U10425/ZN (NAND2_X2)                                    0.02       0.70 f
U13020/ZN (NAND4_X4)                                    0.06       0.76 r
U14815/ZN (INV_X4)                                      0.01       0.77 f
U14820/Z (MUX2_X2)                                      0.09       0.86 f
U10861/ZN (NAND2_X2)                                    0.04       0.91 r
U14936/ZN (INV_X4)                                      0.01       0.92 f
U14941/ZN (OAI21_X4)                                    0.03       0.96 r
U14984/ZN (INV_X4)                                      0.01       0.97 f
U10279/ZN (NOR2_X4)                                     0.02       0.99 r
U10278/ZN (AOI21_X4)                                    0.03       1.02 f
U14985/ZN (NAND3_X4)                                    0.03       1.05 r
U10378/ZN (NOR2_X2)                                     0.02       1.07 f
U14986/ZN (NAND2_X2)                                    0.03       1.10 r
U10281/ZN (NAND2_X4)                                    0.02       1.12 f
U10280/ZN (AOI211_X4)                                   0.07       1.19 r
EX_MEM_REGISTER_EX_MEM_REG_out_reg_71_/D (DFFR_X1)      0.00       1.19 r
data arrival time                                                  1.19

clock clk (rise edge)                                   1.25       1.25
clock network delay (ideal)                             0.00       1.25
EX_MEM_REGISTER_EX_MEM_REG_out_reg_71_/CK (DFFR_X1)     0.00       1.25 r
library setup time                                     -0.06       1.19
data required time                                                 1.19
--------------------------------------------------------------------------
data required time                                                 1.19
data arrival time                                                 -1.19
--------------------------------------------------------------------------
slack (MET)                                                        0.00
```