Sora Kanosue and Sonika Vuyyuru
EECS151 Project Lab Report
Team 23

Project Functional Description and Design Requirements

We aimed to implement a CPU with a 3-stage pipeline structure, with a single level memory hierarchy, and a branch predictor. The 3-stages ended up being fetch, decode/execute, and memory/writeback. Our branch predictor began with the version required for Checkpoint 3, and we later added a global branch history to improve its prediction rate, ultimately topping out at 96.1% on mmult.

High-level organization

Our datapath was implemented in cpu.v, and contained a number of smaller modules instantiated within it. The control signals were generated by a separate control logic module, with inputs as necessary. In the block diagram, control signals inputs are represented by arrows pointing downward coming out of blocks, and control signal outputs are represented by arrows pointing upward into blocks and muxes. The muxes were all implemented directly within cpu.v using combinational logic. Other than the pipeline registers, which were implemented as always posedge clk blocks, as we found this improved the frequency over the staff-provided register modules, all the blocks in the diagram were implemented as separate modules.

In the fetch stage of our pipeline, we have three inputs, the PC coming from the PC register, the d_out of IMEM, and the d_out of BIOS. The d_out's of IMEM and BIOS are muxed to choose a single instruction, before this instruction and a NOP constant are muxed in case the pipeline needs to be flushed. The branch target calculator is used to calculate the targets of branches when they are predicted as taken, so that the branch target can be used as the next PC. For input, it takes in the PC and the instruction. Using the instruction it calculates the B-type immediate, and adds it to the PC to get the branch target. As part of our JAL optimization, it also calculates the jump targets for JAL's so that JAL's can complete in the fetch stage. The branch predictor takes in the PC and a control signal indicating whether the instruction is a branch in the fetch stage. The lower bits of the PC are used to index into the branch predictor's cache to get the prediction, while the upper bits are used as a tag to determine whether the cache was hit. If a branch is predicted as taken, the next PC is chosen to be the output of the branch target calculator.

The decode/execute stage of our pipeline has three inputs, the PC, the instruction, and the output of the branch predictor in the F stage, used to determine if there was a mispredict. The corresponding bits of the instruction are used as the read addresses for the register file, which then outputs the relevant data due to its asynchronous read property. Meanwhile, the instruction is also fed into the immediate generator, which parses the instruction and outputs the immediate which might be encoded in it.

Input A of the ALU is muxed between the pc, the value found in rs1, or the result of the instruction currently in the writeback stage. Input B of the ALU is muxed between the generated immediate, the value found in rs2, or the result of the instruction currently in the writeback stage. The ALU then uses these two inputs to compute a result, with ALUSel choosing the operation to perform on the two operands. The result is then used as the address input to IMEM, DMEM, and BIOS, and also pipelined onto the writeback stage.

The input of the CSR is muxed between the value found in rs1, the generated immediate, or the result of the instruction currently in the writeback stage.

The inputs of the branch comparator are both muxed between the data found in the corresponding register, or the result of the instruction currently in the writeback stage. Depending on the BrUn signal, the branch comparator then performs either a signed or unsigned comparison of its inputs, outputting the results through the BrEq and BrLT signals.

The branch predictor then takes these outputs, as well as the PC, to update the saturating counter stored in its cache. Meanwhile, the control logic also uses the branch predictor outputs as well as the branch predictor guess from the previous stage to determine whether or not there was a mispredict. If necessary, it flushes the instruction currently in the fetch stage, and ensures that the next PC is either the D/EX PC + 4, or the ALU result, the branch target.

The store shifter (to the bottom right of the ALU), is used to align data correctly for store operations. For its input, it muxes between the data found in rs2, or the result of the instruction currently in the writeback stage. Its output is taken to the input of the UART transmitter, and the data in ports of IMEM and DMEM.

Due to their synchronous read natures, IMEM, DMEM, and BIOS are used as pipeline registers separating the decode/execute stage and the memory/writeback stage.

The UART transmitter takes the store shifter's output for its data in port, while its data in ready signal is pipelined to the next stage. The UART receiver's data out port and data our valid signal are both pipelined to the next stage as well.

For inputs, the memory/writeback stage takes in the UART control signals, the UART transmitter's data out, the PC, the ALU result, the data outs of DMEM and BIOS, and the instruction. The UART control signals are packaged together, and the transmitter data out padded with zeros for the relevant I/O reads.

In this cycle, we also have the instruction, cycle, branch instruction, and branch prediction counters, also available to read for the relevant loads.

As a result, the input to the load extender is muxed between the packaged UART control signals, the padded receiver data out, the various counters, and the data outs of DMEM and BIOS. The load extender then selects and sign or zero extends the chosen value as necessary. Then, the writeback mux chooses between the ALU result, PC + 4, or the load extender out.

If necessary, the register file is also written back to at the end of this stage, using the value chosen by the writeback mux and the rd parsed from the instruction in the writeback stage.

Also as a part of the writeback stage, the potential values for the next PC are muxed between. These options are the RESET_PC if reset is asserted, the branch target calculator if a branch is predicted as taken or a JAL is seen, the ALU result from the execute stage for JALRs and branch not taken mispredicts, the execute PC + 4 for branch taken mispredicts, and the fetch PC + 4, the standard option. This chosen PC value is also fed into the addresses ports of IMEM and BIOS so that they can issue instructions for the fetch stage.

Detailed Description of Sub-pieces

Describe how your circuits work. Concentrate here on novel or non-standard circuits. Also, focus your attention on the parts of the design that were not supplied to you by the teaching staff. (≈ 2 pages).

Instruction Counter

The next value of the instruction counter was chosen every cycle depending on the value of the instruction in the memory/writeback stage. If a store to the reset value occurred, a 0 was chosen. If the M/WB instruction was not a NOP, the previous value was incremented. If the M/WB instruction was a NOP, then the previous value was held constant. While looking at the dumps produced by the 151 gcc toolchain, we realized that NOPs generated during compilation used ADDI instructions writing to x0, so for our own NOPs injected when flushing, we used ADD instructions writing to x0. As a result, we were able to distinguish between compiler generated NOPs, and our own injected NOPs to get an accurate instruction count.

Cycle Counter

The cycle counter simply incremented its previous value every cycle unless a store to the counter reset address occurred.

Branch Instruction Counter

The next value of the branch instruction counter was chosen every cycle depending on the value of the instruction in the memory/writeback stage. If a store to the reset value occurred, a 0 was chosen. If the M/WB instruction's opcode indicated that it was a branch, the previous value was incremented, while in every other case, it was held constant.

Correct Branch Prediction Counter

The next value of the correct branch prediction counter was chosen every cycle depending on the output of the branch comparator in the D/X stage and the branch predictor guess forwarded from the fetch stage. If a store to the reset value occurred, a 0 was chosen. If the result of the branch comparator matched the guess, then the previous value was incremented, while in every other case, it was held constant.

Branch Predictor

Our branch predictor modified the given implementation by utilizing a global branch history (GBH) and multiple caches storing different saturating counter bits. The GBH was implemented as a shift register h bits long, along with 2^h caches. Each time a branch instruction enters the fetch stage, the GBH is used to index into the correct cache and access the guess. In the execute stage, the GBH is used again to index into and update the value in the correct cache, before being updated with the result of the branch in the execute stage.

Immediate Generator

The immediate generator took in a control signal signaling the immediate type and the instruction to parse the immediate from. It then selected and concatenated the relevant bits from the instruction, shifting or extending the immediate as needed.

Branch Target Calculator

Our branch target calculator took in the instruction currently in the fetch stage, and calculated two offsets, using the same logic as the immediate generator, one assuming that the instruction was a branch instruction, and the other assuming that it was a JAL. Then, the opcode was used to select the correct offset, and added to the PC to get the branch/jump target.

Branch Comparator

The branch comparator took in two values to compare, and a signal indicating whether the comparison should be signed or not. The BrEq output was not dependent on this, and just returned whether the values were equal or not. However, the BrLT output did depend on BrUn.

ALU

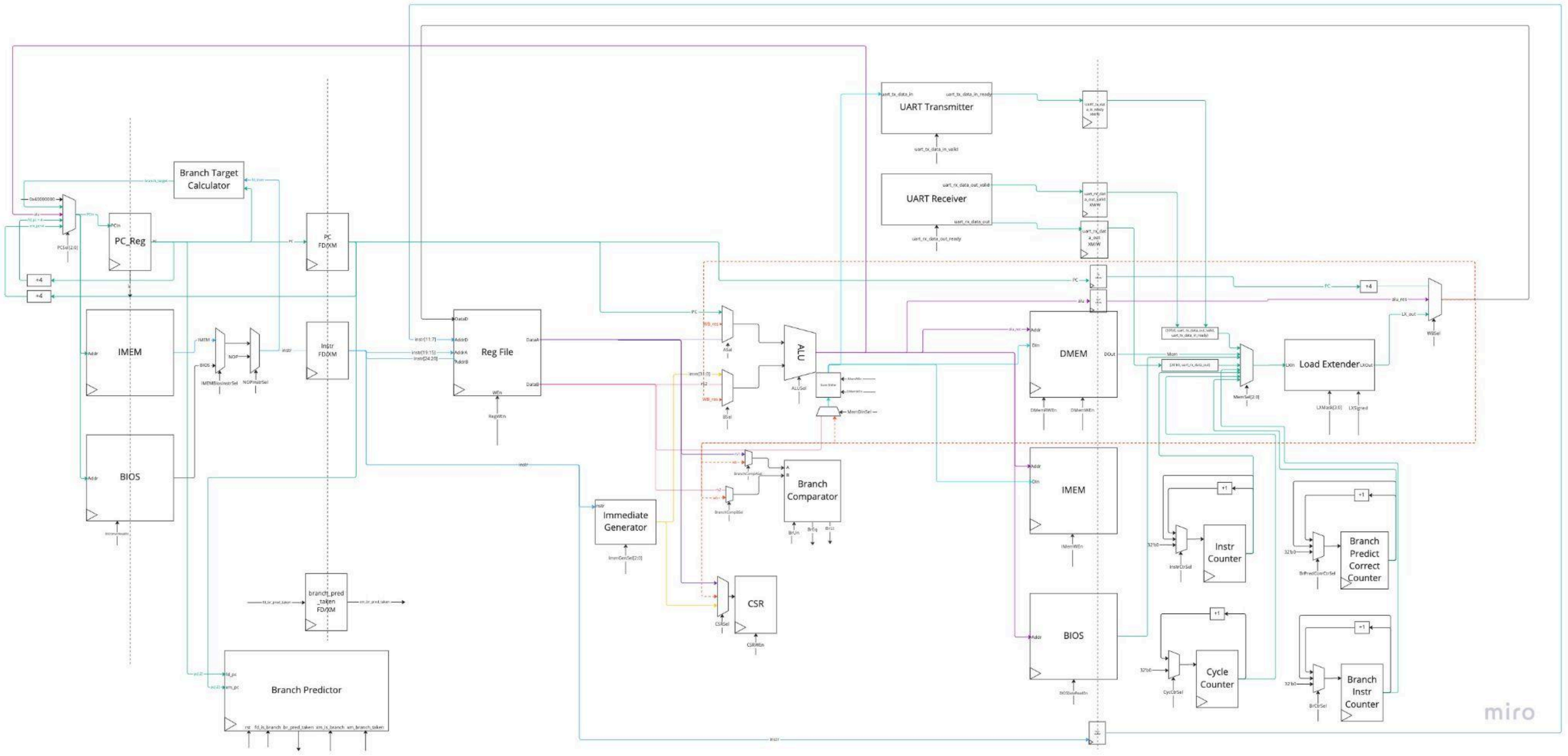
The ALU took in two values and computed a value using the operation specified by the ALUSel control signal.

Load Extender

For instructions which loaded less than a word, the output from the memory being read had to be either zero or sign extended by the load extender. Also, since the memory read addresses were word-aligned, the load extender also handled shifting the correct bits of the memory output to the lower bits and extending as appropriate.

Store Shifter

The store shifter essentially did the inverse operation of the load extender. For instructions which stores less than a word, it shifted the lowest byte or halfword as appropriate to the correct position in the data word which eventually made its way to the data in port of the memory being written to.

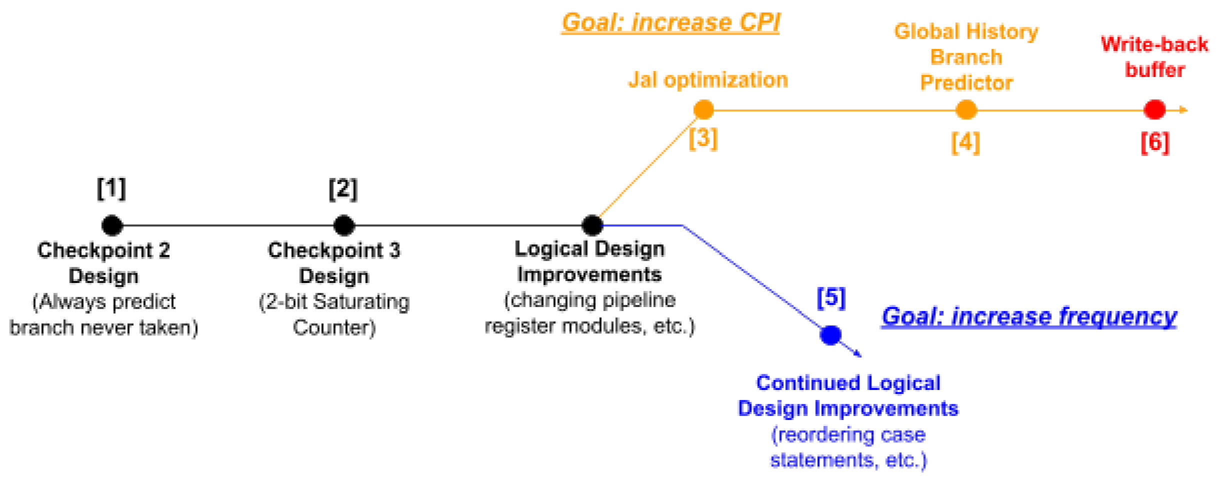


Branch Target Calculator
PC_Reg
PCSel[2:0]
+4
+4
IMEM
BIOS
IMEMBiosInstrSel
NOP
NOPInstrSel
PC FD/XM
Instr FD/XM
branch_pred_taken FD/XM
Branch Predictor
Reg File
DataD
AddrD
AddrA
AddrB
DataA
DataB
WEn
RegWEn
instr[11:7]
instr[19:15]
instr[24:20]
Immediate Generator
ImmGenSel[2:0]
ASel
BSel
ALU
ALUSel
MemDInSel
Branch Comparator
BrUn
BrEq
BrLt
CSR
CSRSel
CSRWEn
UART Transmitter
uart_tx_data_in
uart_tx_data_in_ready
uart_tx_data_in_valid
UART Receiver
uart_rx_data_out_valid
uart_rx_data_out
uart_rx_data_out_ready
DMEM
Addr
DIn
DOut
DMemRWEn
DMemWEn
IMEM
IMemWEn
BIOS
BIOSDataReadEn
MemSel[2:0]
Load Extender
LXIn
LXOut
LXMask[2:0]
LXSigned
WBSel
alu_res
LX_out
PC
+4
Instr Counter
InstrCtrSel
Cycle Counter
CycCtrSel
Branch Predict Correct Counter
BrPredCorrCtrSel
Branch Instr Counter
BrCtrSel
+1
32'b0
miro

Status and Results

By the end of the project, we had a fully functioning 3-stage pipelined RISC-V CPU. The highest frequency that our design clocked at was **75 MHz**. However, this was our design without the advanced global history branch predictor. The more complicated global history branch predictor increased the critical path of our design and ran at a maximum frequency of 60 MHz.

Below is a figure depicting the evolution of our design, including an abridged list of our optimizations and design iterations.


Goal: increase CPI
Jal optimization
[3]
Global History Branch Predictor
[4]
Write-back buffer
[6]
[1]
Checkpoint 2 Design
(Always predict branch never taken)
[2]
Checkpoint 3 Design
(2-bit Saturating Counter)
Logical Design Improvements
(changing pipeline register modules, etc.)
[5]
Goal: increase frequency
Continued Logical Design Improvements
(reordering case statements, etc.)

Below are the results of optimization attempts for the different design iterations. We will mainly be comparing the resource area utilization between design [3] with jal optimization, design [4] with jal optimization and global history branch prediction, and the design [5] with logical design improvements that allowed it to reach a clock frequency of 75 MHz. We will be comparing the mmult CPI, clock period, and branch prediction accuracy results between all 6 design iterations.

[1] “Vanilla” Checkpoint 2 Design (Always predict branch never taken)

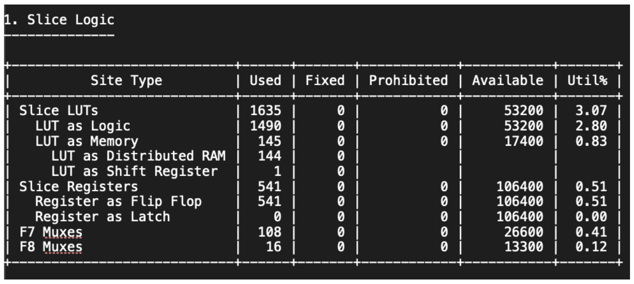
Frequency: 50 Mhz
Minimum Clock Period: 20 ns
CPI: 1.184
Branch Prediction Accuracy: 49%

[2] Checkpoint 3 Design with standard 2-bit saturating counter branch predictor

Frequency: 50 Mhz
Minimum Clock Period: 20 ns
CPI: 1.102
Branch Prediction Accuracy: 78.454%

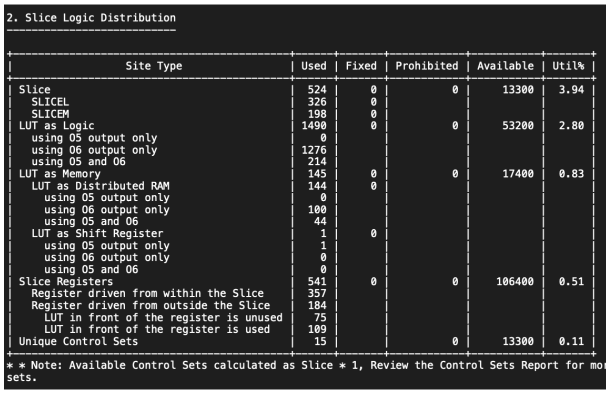
[3] Design with jal optimization (goal of decreasing CPI)

This design included the jal optimization, which ended up increasing the critical path.
Frequency: 60 MHz
Minimum Clock Period: 16.67 ns
CPI: 1.082
Branch Prediction Accuracy: 78.454%
Resource Utilization Details:



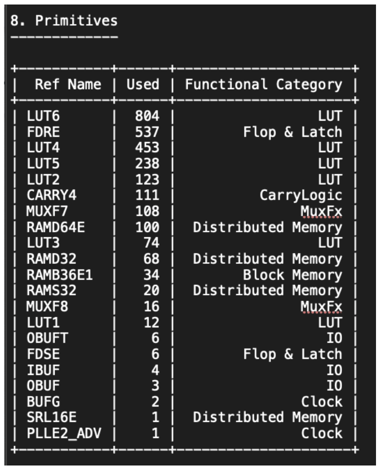
```
1. Slice Logic
--------------

+---------------------------+------+-------+------------+-----------+-------+
|         Site Type         | Used | Fixed | Prohibited | Available | Util% |
+---------------------------+------+-------+------------+-----------+-------+
| Slice LUTs                | 1635 |     0 |          0 |     53200 |  3.07 |
|   LUT as Logic            | 1490 |     0 |          0 |     53200 |  2.80 |
|   LUT as Memory           |  145 |     0 |          0 |     17400 |  0.83 |
|     LUT as Distributed RAM|  144 |     0 |            |           |       |
|     LUT as Shift Register |    1 |     0 |            |           |       |
| Slice Registers           |  541 |     0 |          0 |    106400 |  0.51 |
|   Register as Flip Flop   |  541 |     0 |          0 |    106400 |  0.51 |
|   Register as Latch       |    0 |     0 |          0 |    106400 |  0.00 |
| F7 Muxes                  |  108 |     0 |          0 |     26600 |  0.41 |
| F8 Muxes                  |   16 |     0 |          0 |     13300 |  0.12 |
+---------------------------+------+-------+------------+-----------+-------+
```



```
2. Slice Logic Distribution
---------------------------

+-------------------------------------------+------+-------+------------+-----------+-------+
|                 Site Type                 | Used | Fixed | Prohibited | Available | Util% |
+-------------------------------------------+------+-------+------------+-----------+-------+
| Slice                                     |  524 |     0 |          0 |     13300 |  3.94 |
|   SLICEL                                  |  326 |     0 |            |           |       |
|   SLICEM                                  |  198 |     0 |            |           |       |
| LUT as Logic                              | 1490 |     0 |          0 |     53200 |  2.80 |
|   using O5 output only                    |    0 |       |            |           |       |
|   using O6 output only                    | 1276 |       |            |           |       |
|   using O5 and O6                         |  214 |       |            |           |       |
| LUT as Memory                             |  145 |     0 |          0 |     17400 |  0.83 |
|   LUT as Distributed RAM                  |  144 |     0 |            |           |       |
|     using O5 output only                  |    0 |       |            |           |       |
|     using O6 output only                  |  100 |       |            |           |       |
|     using O5 and O6                       |   44 |       |            |           |       |
|   LUT as Shift Register                   |    1 |     0 |            |           |       |
|     using O5 output only                  |    1 |       |            |           |       |
|     using O6 output only                  |    0 |       |            |           |       |
|     using O5 and O6                       |    0 |       |            |           |       |
| Slice Registers                           |  541 |     0 |          0 |    106400 |  0.51 |
|   Register driven from within the Slice   |  357 |       |            |           |       |
|   Register driven from outside the Slice  |  184 |       |            |           |       |
|     LUT in front of the register is unused|   75 |       |            |           |       |
|     LUT in front of the register is used  |  109 |       |            |           |       |
| Unique Control Sets                       |   15 |       |          0 |     13300 |  0.11 |
+-------------------------------------------+------+-------+------------+-----------+-------+
* * Note: Available Control Sets calculated as Slice * 1, Review the Control Sets Report for mor
sets.
```



```
8. Primitives
-------------

+-----------+------+---------------------+
|  Ref Name | Used | Functional Category |
+-----------+------+---------------------+
| LUT6      |  804 |                 LUT |
| FDRE      |  537 |        Flop & Latch |
| LUT4      |  453 |                 LUT |
| LUT5      |  238 |                 LUT |
| LUT2      |  123 |                 LUT |
| CARRY4    |  111 |          CarryLogic |
| MUXF7     |  108 |               MuxFx |
| RAMD64E   |  100 |  Distributed Memory |
| LUT3      |   74 |                 LUT |
| RAMD32    |   68 |  Distributed Memory |
| RAMB36E1  |   34 |        Block Memory |
| RAMS32    |   20 |  Distributed Memory |
| MUXF8     |   16 |               MuxFx |
| LUT1      |   12 |                 LUT |
| OBUFT     |    6 |                  IO |
| FDSE      |    6 |        Flop & Latch |
| IBUF      |    4 |                  IO |
| OBUF      |    3 |                  IO |
| BUFG      |    2 |               Clock |
| SRL16E    |    1 |  Distributed Memory |
| PLLE2_ADV |    1 |               Clock |
+-----------+------+---------------------+
```

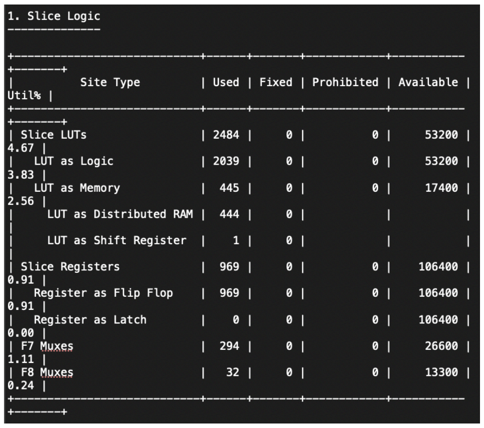
This design uses fewer LUTs than design [4] with the global branch predictor.

[4] Design with jal optimization and 5-bit global history branch predictor (goal of decreasing CPI)

This design included two improvements: 1) jal optimization, 2) 5-bit global branch history predictor. Both of these improvements were attempts at improving the CPI, but both ended up increasing the critical path.

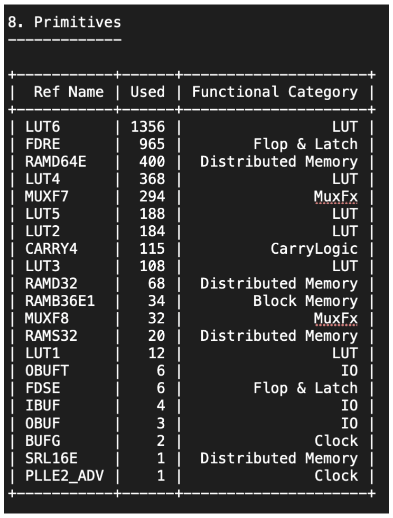
In this design, we first started with a 2-bit history register. However, we continued to see improvements as we increased the number of bits, with our final design having a 5-bit history register.

Frequency: 60 Mhz
Minimum Clock Period: 16.67 ns
CPI: 1.051
Branch Prediction Accuracy: 96.3%
Resource Utilization Details:



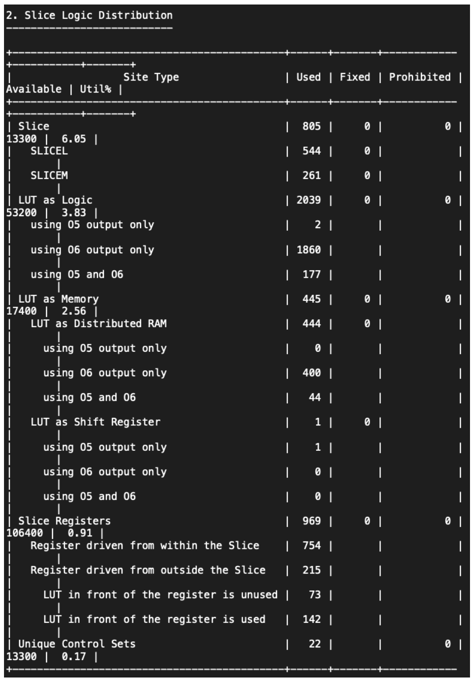
```
1. Slice Logic
--------------

+----------------------------+------+-------+------------+-----------
+-------+
|          Site Type         | Used | Fixed | Prohibited | Available |
Util% |
+----------------------------+------+-------+------------+-----------
+-------+
| Slice LUTs                 | 2484 |     0 |          0 |     53200 |
4.67 |
|   LUT as Logic             | 2039 |     0 |          0 |     53200 |
3.83 |
|   LUT as Memory            |  445 |     0 |          0 |     17400 |
2.56 |
|     LUT as Distributed RAM |  444 |     0 |            |           |
|
|     LUT as Shift Register  |    1 |     0 |            |           |
|
| Slice Registers            |  969 |     0 |          0 |    106400 |
0.91 |
|   Register as Flip Flop    |  969 |     0 |          0 |    106400 |
0.91 |
|   Register as Latch        |    0 |     0 |          0 |    106400 |
0.00 |
| F7 Muxes                   |  294 |     0 |          0 |     26600 |
1.11 |
| F8 Muxes                   |   32 |     0 |          0 |     13300 |
0.24 |
+----------------------------+------+-------+------------+-----------
+-------+
```



```
8. Primitives
-------------

+------------+------+---------------------+
|  Ref Name  | Used | Functional Category |
+------------+------+---------------------+
| LUT6       | 1356 |                 LUT |
| FDRE       |  965 |        Flop & Latch |
| RAMD64E    |  400 |  Distributed Memory |
| LUT4       |  368 |                 LUT |
| MUXF7      |  294 |               MuxFx |
| LUT5       |  188 |                 LUT |
| LUT2       |  184 |                 LUT |
| CARRY4     |  115 |          CarryLogic |
| LUT3       |  108 |                 LUT |
| RAMD32     |   68 |  Distributed Memory |
| RAMB36E1   |   34 |        Block Memory |
| MUXF8      |   32 |               MuxFx |
| RAMS32     |   20 |  Distributed Memory |
| LUT1       |   12 |                 LUT |
| OBUFT      |    6 |                  IO |
| FDSE       |    6 |        Flop & Latch |
| IBUF       |    4 |                  IO |
| OBUF       |    3 |                  IO |
| BUFG       |    2 |               Clock |
| SRL16E     |    1 |  Distributed Memory |
| PLLE2_ADV  |    1 |               Clock |
+------------+------+---------------------+
```



```
2. Slice Logic Distribution
---------------------------

+-------------------------------------------+------+-------+-----------
+-----------+-------+
|                 Site Type                 | Used | Fixed | Prohibited |
Available | Util% |
+-------------------------------------------+------+-------+-----------
+-----------+-------+
| Slice                                     |  805 |     0 |          0 |
13300 |  6.05 |
|   SLICEL                                  |  544 |     0 |            |
|       |
|   SLICEM                                  |  261 |     0 |            |
|       |
| LUT as Logic                              | 2039 |     0 |          0 |
53200 |  3.83 |
|   using O5 output only                    |    2 |       |            |
|       |
|   using O6 output only                    | 1860 |       |            |
|       |
|   using O5 and O6                         |  177 |       |            |
|       |
| LUT as Memory                             |  445 |     0 |          0 |
17400 |  2.56 |
|   LUT as Distributed RAM                  |  444 |     0 |            |
|       |
|     using O5 output only                  |    0 |       |            |
|       |
|     using O6 output only                  |  400 |       |            |
|       |
|     using O5 and O6                       |   44 |       |            |
|       |
|   LUT as Shift Register                   |    1 |     0 |            |
|       |
|     using O5 output only                  |    1 |       |            |
|       |
|     using O6 output only                  |    0 |       |            |
|       |
|     using O5 and O6                       |    0 |       |            |
|       |
| Slice Registers                           |  969 |     0 |          0 |
106400 |  0.91 |
|   Register driven from within the Slice   |  754 |       |            |
|       |
|   Register driven from outside the Slice  |  215 |       |            |
|       |
|     LUT in front of the register is unused |   73 |       |            |
|       |
|     LUT in front of the register is used  |  142 |       |            |
|       |
| Unique Control Sets                       |   22 |       |          0 |
13300 |  0.17 |
+-------------------------------------------+------+-------+-----------
```

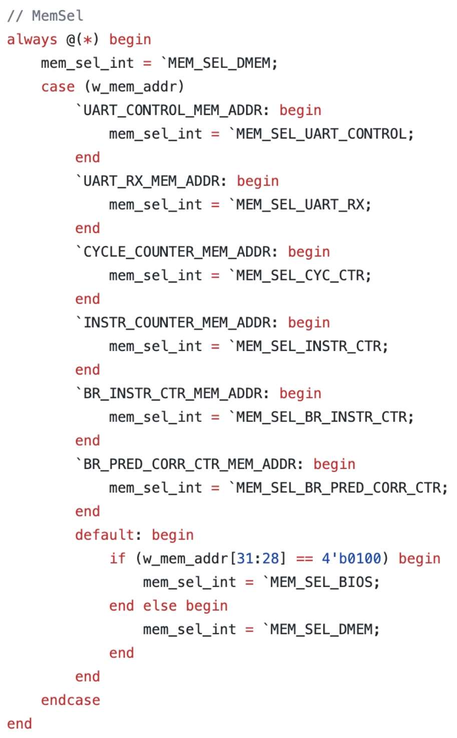
This design uses significantly more LUTs and Distributed Memory than the previous design [3]. This is because the global history branch predictor has separate caches to keep track of the history of each branch, making it use up significantly more resources.

[5] Design with continued logic design improvements (goal of increasing frequency)

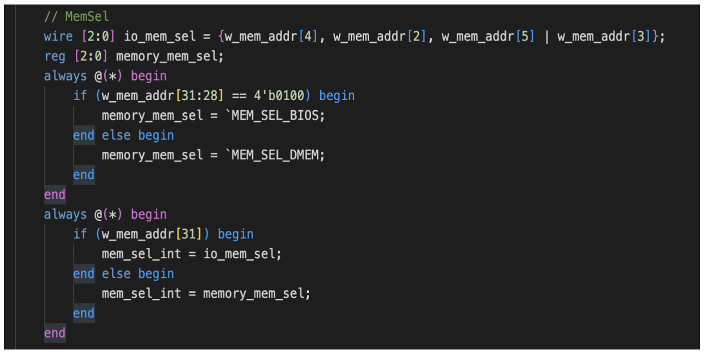
The goal of this iteration of the design was to take the iteration of the design that had the smallest critical path (before adding in the jal optimizaton and global history BP) and continue to make logic design improvements to increase the clock rate of our design. We took a two-pronged approach to this continual optimization process: 1) identifying the critical path and attempting to move components around or precompute when possible, and 2) modifying our method for calculation of signals that may be waiting on another signal or doing an unnecessary comparison multiple times.

One example of a way that we modified our data path and reduced the critical path is that we moved our immediate generator to the F stage, since it was in our critical path. By precomputing the immediate, we were able to reduce the time of our critical path. Furthermore, by moving it to the F stage, it was no longer in the critical path because the other steps in the XDM stage still took longer.

An example of a way that we modified the calculation of signals to limit delay is the way that we reorganized the structure of computing the MemSel control signal. Originally, we just had a case statement that determined the mem_sel value based on the mem_addr. However, this large case statement resulted in a large delay in our critical path. We identified a pattern in the mem_addr using boolean logic and Karnaugh maps to compute the signal directly from the mem_addr bits and further split up the signal into an io_mem_sel and memory_mem_sel computation. This parallelization greatly improved the time to compute this signal.

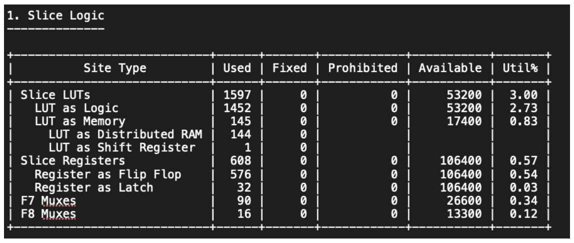


```
// MemSel
always @(*) begin
    mem_sel_int = `MEM_SEL_DMEM;
    case (w_mem_addr)
        `UART_CONTROL_MEM_ADDR: begin
            mem_sel_int = `MEM_SEL_UART_CONTROL;
        end
        `UART_RX_MEM_ADDR: begin
            mem_sel_int = `MEM_SEL_UART_RX;
        end
        `CYCLE_COUNTER_MEM_ADDR: begin
            mem_sel_int = `MEM_SEL_CYC_CTR;
        end
        `INSTR_COUNTER_MEM_ADDR: begin
            mem_sel_int = `MEM_SEL_INSTR_CTR;
        end
        `BR_INSTR_CTR_MEM_ADDR: begin
            mem_sel_int = `MEM_SEL_BR_INSTR_CTR;
        end
        `BR_PRED_CORR_CTR_MEM_ADDR: begin
            mem_sel_int = `MEM_SEL_BR_PRED_CORR_CTR;
        end
        default: begin
            if (w_mem_addr[31:28] == 4'b0100) begin
                mem_sel_int = `MEM_SEL_BIOS;
            end else begin
                mem_sel_int = `MEM_SEL_DMEM;
            end
        end
    endcase
end
```



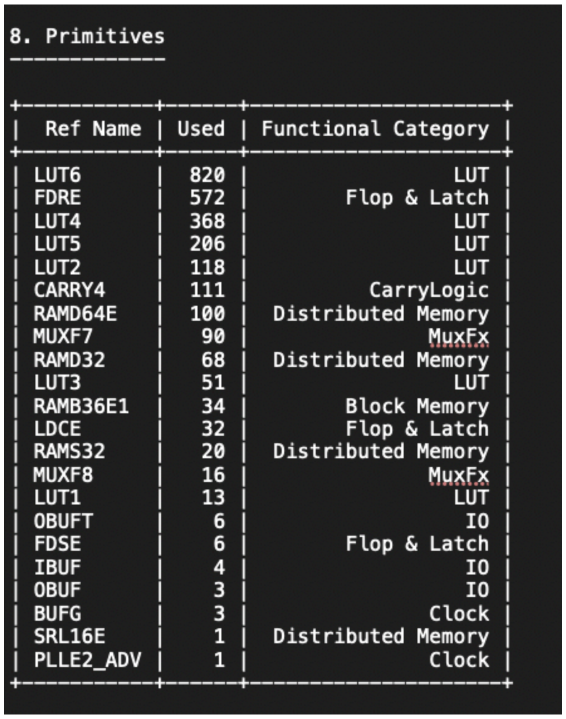
```
// MemSel
wire [2:0] io_mem_sel = {w_mem_addr[4], w_mem_addr[2], w_mem_addr[5] | w_mem_addr[3]};
reg [2:0] memory_mem_sel;
always @(*) begin
    if (w_mem_addr[31:28] == 4'b0100) begin
        memory_mem_sel = `MEM_SEL_BIOS;
    end else begin
        memory_mem_sel = `MEM_SEL_DMEM;
    end
end
always @(*) begin
    if (w_mem_addr[31]) begin
        mem_sel_int = io_mem_sel;
    end else begin
        mem_sel_int = memory_mem_sel;
    end
end
```

Frequency: 75 Mhz
Minimum Clock Period: 13.33 ns
CPI: 1.102
Branch Prediction Accuracy: 78.454%
Resource Utilization Details:



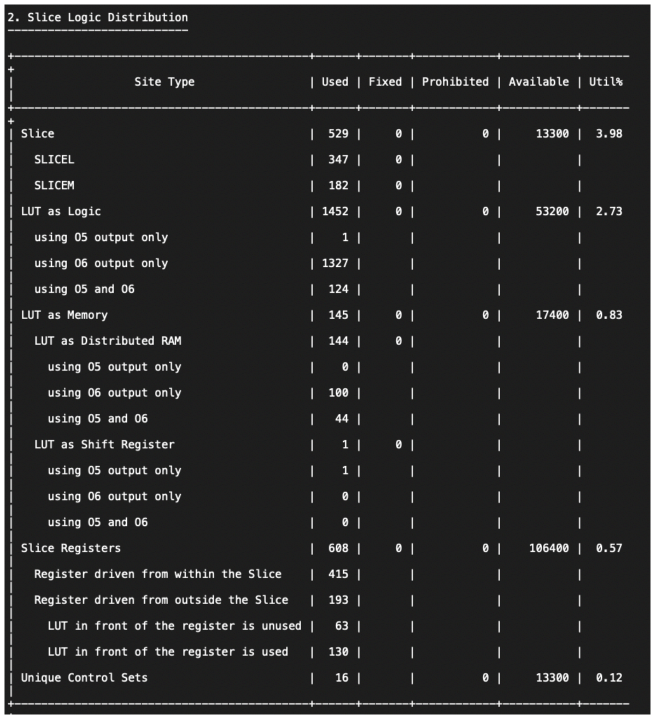
```
1. Slice Logic
--------------

+----------------------------+------+-------+------------+-----------+-------+
|          Site Type         | Used | Fixed | Prohibited | Available | Util% |
+----------------------------+------+-------+------------+-----------+-------+
| Slice LUTs                 | 1597 |     0 |          0 |     53200 |  3.00 |
|   LUT as Logic             | 1452 |     0 |          0 |     53200 |  2.73 |
|   LUT as Memory            |  145 |     0 |          0 |     17400 |  0.83 |
|     LUT as Distributed RAM |  144 |     0 |            |           |       |
|     LUT as Shift Register  |    1 |     0 |            |           |       |
| Slice Registers            |  608 |     0 |          0 |    106400 |  0.57 |
|   Register as Flip Flop    |  576 |     0 |          0 |    106400 |  0.54 |
|   Register as Latch        |   32 |     0 |          0 |    106400 |  0.03 |
| F7 Muxes                   |   90 |     0 |          0 |     26600 |  0.34 |
| F8 Muxes                   |   16 |     0 |          0 |     13300 |  0.12 |
+----------------------------+------+-------+------------+-----------+-------+
```



```
8. Primitives
-------------

+-----------+------+---------------------+
|  Ref Name | Used | Functional Category |
+-----------+------+---------------------+
| LUT6      |  820 |                 LUT |
| FDRE      |  572 |        Flop & Latch |
| LUT4      |  368 |                 LUT |
| LUT5      |  206 |                 LUT |
| LUT2      |  118 |                 LUT |
| CARRY4    |  111 |          CarryLogic |
| RAMD64E   |  100 |  Distributed Memory |
| MUXF7     |   90 |               MuxFx |
| RAMD32    |   68 |  Distributed Memory |
| LUT3      |   51 |                 LUT |
| RAMB36E1  |   34 |        Block Memory |
| LDCE      |   32 |        Flop & Latch |
| RAMS32    |   20 |  Distributed Memory |
| MUXF8     |   16 |               MuxFx |
| LUT1      |   13 |                 LUT |
| OBUFT     |    6 |                  IO |
| FDSE      |    6 |        Flop & Latch |
| IBUF      |    4 |                  IO |
| OBUF      |    3 |                  IO |
| BUFG      |    3 |               Clock |
| SRL16E    |    1 |  Distributed Memory |
| PLLE2_ADV |    1 |               Clock |
+-----------+------+---------------------+
```



```
2. Slice Logic Distribution
---------------------------

+-------------------------------------------+------+-------+------------+-----------+-------
+
|                 Site Type                 | Used | Fixed | Prohibited | Available | Util%
|
+-------------------------------------------+------+-------+------------+-----------+-------
+
| Slice                                     |  529 |     0 |          0 |     13300 |  3.98
|
|   SLICEL                                  |  347 |     0 |            |           |
|
|   SLICEM                                  |  182 |     0 |            |           |
|
| LUT as Logic                              | 1452 |     0 |          0 |     53200 |  2.73
|
|   using O5 output only                    |    1 |       |            |           |
|
|   using O6 output only                    | 1327 |       |            |           |
|
|   using O5 and O6                         |  124 |       |            |           |
|
| LUT as Memory                             |  145 |     0 |          0 |     17400 |  0.83
|
|   LUT as Distributed RAM                  |  144 |     0 |            |           |
|
|     using O5 output only                  |    0 |       |            |           |
|
|     using O6 output only                  |  100 |       |            |           |
|
|     using O5 and O6                       |   44 |       |            |           |
|
|   LUT as Shift Register                   |    1 |     0 |            |           |
|
|     using O5 output only                  |    1 |       |            |           |
|
|     using O6 output only                  |    0 |       |            |           |
|
|     using O5 and O6                       |    0 |       |            |           |
|
| Slice Registers                           |  608 |     0 |          0 |    106400 |  0.57
|
|   Register driven from within the Slice   |  415 |       |            |           |
|
|   Register driven from outside the Slice  |  193 |       |            |           |
|
|     LUT in front of the register is unused |   63 |       |            |           |
|
|     LUT in front of the register is used   |  130 |       |            |           |
|
| Unique Control Sets                       |   16 |       |          0 |     13300 |  0.12
|
+-------------------------------------------+------+-------+------------+-----------+-------
```

This design had a similar amount of LUTs used as design [3], since it does not have the global history buffer that takes up a lot of resources.

[6] Write-back buffer design (no improvement to CPI)

Although this design is functional, it did not have the anticipated result of a lower CPI. The reason for this is that unless the last instruction is a load, no cycles will be saved since the fetch of the next instruction cannot happen any sooner. After implementation, we realized that this design of a write-back buffer is another method of forwarding, which we had already implemented, so it did not result in any lower CPI.

However, another tradeoff we wanted to explore with this design is to consider if utilizing the write-back buffer design resulted in lower resource utilization.

Frequency: 60 Mhz
Minimum Clock Period: 16.67 ns
CPI: 1.051

Conclusions

We learned a lot from the experience of building this final project. Some of the most important things we learned from this project were the skills of proper planning and design, and how to go through the process of debugging when implementing our design. Throughout the project, we made sure to keep our initial block diagram up to date with any modifications to our design, and made different versions of the diagram for big changes or features, like when attempting the WB buffer feature. Doing this was helpful in our version control efforts, and made it easier to refer back to the diagram and know that the signals and components were all up to date and correct, making following expected behavior and debugging easier.

The other thing that we learned a lot from this experience is improving our debugging skills, and how to approach thinking about debugging as well. Because this project included integrating many different modules and components, like the UART, BIOS memory, and the actual CPU, figuring out how to debug early on was extremely important. There was a point in our implementation process where we were trying to integrate our UART and BIOS with our CPU to run the mmult program, but it was extremely difficult to identify where in the pipeline our bug was. While our design was passing the simulation, it seemed like there was no way to identify where on the actual FPGA board the issue was occurring. To resolve this issue, we used Paul's suggestion to utilize the LEDs on the board to encode which where in the design flow the bug was happening. We created checks for each step, like receiving the PC from the UART, receiving the instruction from the UART, etc. By being able to see if each of these individual steps was successful, one by one, we were able to identify the location of our bug. In addition to this, thinking about what was different between the simulation process and actual deployment on the board helped us narrow our scope of potential bugs.

Overall, one thing that we would do differently next time is we would try to complete checkpoint 2 earlier so that we could have more time to optimize. Ideally, we would’ve been able to integrate both the global branch history predictor with the logical modifications that allowed for a higher clock frequency. Debugging was the most time-consuming part of checkpoint 2, so knowing about the different debugging techniques we learned earlier on would have helped us save time and reach this goal.

Division of Labor

Written separately.