Super-pipelined Processor

Angela Zou(az292), Andrew Lin(yl656), Zoe Du(jd963)

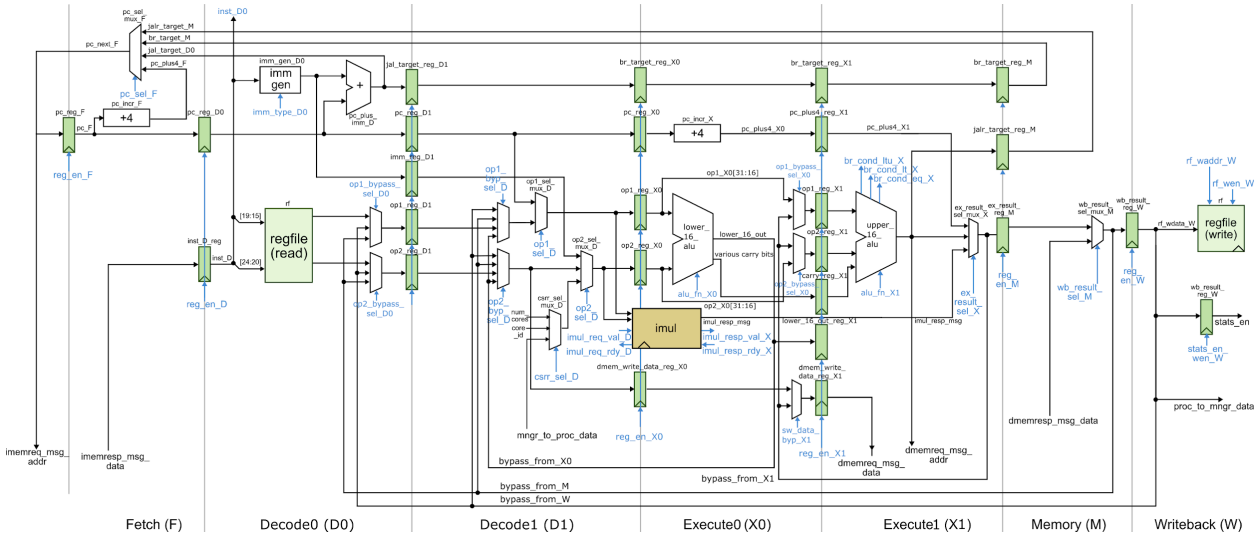


Table of Content

1. Introduction

One major topic of computer architecture is strategies to improve performance for general-purpose processors. The execution time of a program is determined by the following equation: $T = i \times cpi \times t$where *i* denotes the instructions in total, *cpi* denotes the cycles per instruction, and *t* denotes cycle time. Our goal is to improve the performance by reducing *t* and maintaining a *cpi* of approximately 1. The cycle time of the processor is determined by the critical path so one effective way to break the critical path into balanced stages is to insert registers in between, also known as pipelining. We intend to divide the well-known 5-stage processor further based on datapath characteristics and therefore achieve superpipelining. One rudimentary example discussed was splitting the M/memory stage into 2 stages, M0 and M1. Since memory operations take a long time due to the cache/memory delay, the critical path can be shortened and clock frequency can be raised, and throughput can be greatly improved as a result.

On the other hand, superscalar and VLIW processors exploit instruction-level parallelism and decrease *cpi* by running multiple instructions in parallel. In theory, a superpipelined design that doubles its pipeline stages and a superscalar design that issues two commands can both achieve a throughput two times higher than a normal pipelined processor under ideal conditions (no hazards/dependencies). However, superscalar and VLIW have their disadvantages. Superscalar processors need to either replicate their hardware units or limit how instructions can be combined to be processed in parallel. VLIW processors have multiple instructions in one word for different functional units but compilers have to decide that at compile time. If the program cannot be broken into parallel instructions that operate on different function units, the parallelism is rendered useless and much of the code size is wasted. Also, both suffer from dependencies between instructions as well as extra logic required for those dependencies and require extra help from smart compilers. They also lack extensibility and compatibility as increasing the level of parallelism means having to alter the machine code for the same program. Therefore, we believe that superpipelining is our best approach to improve performance.

After deciding on superpipelining, we start measuring how throughput is affected by the number of stages. Compared to the simple single-cycle processor, our baseline, a bypassing 5-stage pipelined processor, divides the main datapath into five stages: fetch, decode, execute, memory, and writeback, and thus reduce the critical path. However, as we increase the number of stages, we need to take care of more hazards caused by the dependencies among stages as they cause stalling or squashing, increasing the *cpi* and decreasing the processor performance. In response, we developed direct, unit and random tests to make sure each instruction works and then a sequence of mixed instructions to examine our processor under a variety of hazards.

Now we see the increase in the number of stages helps us reduce the cycle time by reducing the critical path and eventually improve program execution time, a question arises: what is the optimal number of stages for a pipelined processor? This question involves the exploration of the relationship between the number of stages and the number of hazards that

cannot be solved by bypassing. Under such motivation, our team decides to explore the super-pipelined processor capable of running the tinyRV2 instruction set as our alternative design. We apply the same tests from baseline design and develop specific tests to examine the newly emerged hazards.

As discussed in section II, we will use Fan-out of 4 (FO4) delay as a standard time unit to measure the propagation delay of our critical path so we can study our processors regardless of technology constraints. We will calculate FO4 delay for both of our designs and examine the difference. FO4 delay is also used to measure the processor cycle time trend across history so we can compare our designs with others by adopting this standardized unit.

1.

2. FO4 Spice Simulation

To measure the propagation delay of a path, we use the propagation delay of a minimum size inverter (3RC) as a relative delay unit. Similarly, to reduce the critical path length and build a high-frequency superpipelined processor, we need a unit that makes our processor cycle time comparable to processors built with different technologies. In this section, we introduce a technology-agnostic metric: Fan-out of 4 (FO4) delay.

For most technologies, the optimal fanout of buffers driving large loads is generally between 2.7 to 5.3, which makes FO4 a good delay measurement unit as tools try to select gates that fit into the fanout range. As a delay metric, one FO4 is the delay of an inverter, driven by an inverter four times smaller, and driving an inverter four times larger. Both conditions are necessary since the rise and fall time of the input signal affect the delay as well as the output load. The FO4 delays of various technologies are shown in *figure 1*.

Vendor		Orbit	HP	AMI	AMI	TSMC	TSMC	TSMC	IBM	IBM	IBM
Model		MOSIS	MOSIS	MOSIS	MOSIS	MOSIS	MOSIS	TSMC	IBM	IBM	IBM
Feature Size f	nm	2000	800	600	600	350	250	180	130	90	65
V_{DD}	V	5	5	5	3.3	3.3	2.5	1.8	1.2	1.0	1.0
Gates											
C_g (delay)	fF/μm	1.77	1.67	1.55	1.48	1.90	2.30	1.67	1.04	0.97	0.80
C_g (power)	fF/μm	2.24	1.70	1.83	1.76	2.20	2.92	2.06	1.34	1.23	1.07
FO4 Inv. Delay	ps	856	297	230	312	210	153	75.6	45.9	37.2	17.2

Figure 1: FO4 Delay Characteristics for a Variety of Processes (Weste and Harris p.312 Table 8.5)

To estimate the FO4 delay of the NanGate 45nm standard-cell library we are using, we used ngspice to simulate the inverter circuit. To gain more understanding of the characteristics of nMOS and pMOS, we first built our own inverters with nMOS and pMOS in the first spice deck. We created a scaling factor so transistor sizes can be expressed as multiples of the minimum width of an NMOS transistor. Since the nMOS of INV_X1 in the standard cell library has a width of 415nm and a length of 50nm, we used a scale factor such that a width of 1 is equal to 450nm and the length is approximately 0.1 unit. To get a fanout of 4, we used a parameter H=4 to increase the width of the inverter. In this model, we used 5 inverters, each 4 times larger than the inverter before it in the path. As seen in *figure 2*, the first two inverters shape the input waveform, the third one is used for the FO4 delay measurement, and the last two are loads of the path. Using transient analysis, we measured the rising and falling edge propagation delay and average these two to get an inverter FO4 delay of 16.74ps.

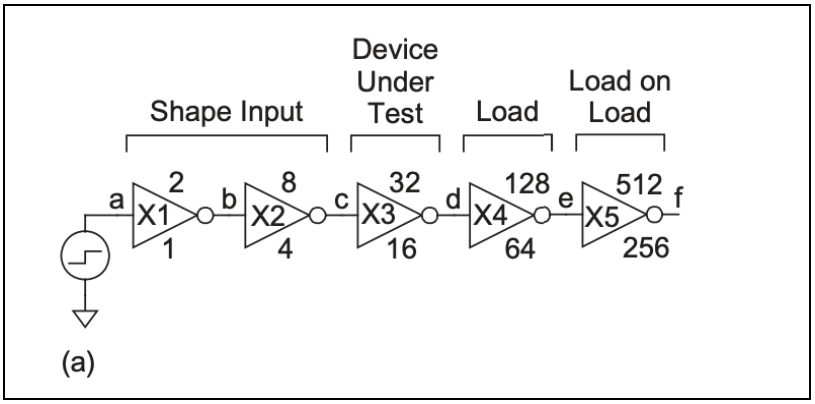


Figure 2: FO4 Simulation Model (Inverter built with nMOS and pMOS, Weste and Harris p.295 Fig. 8.9)

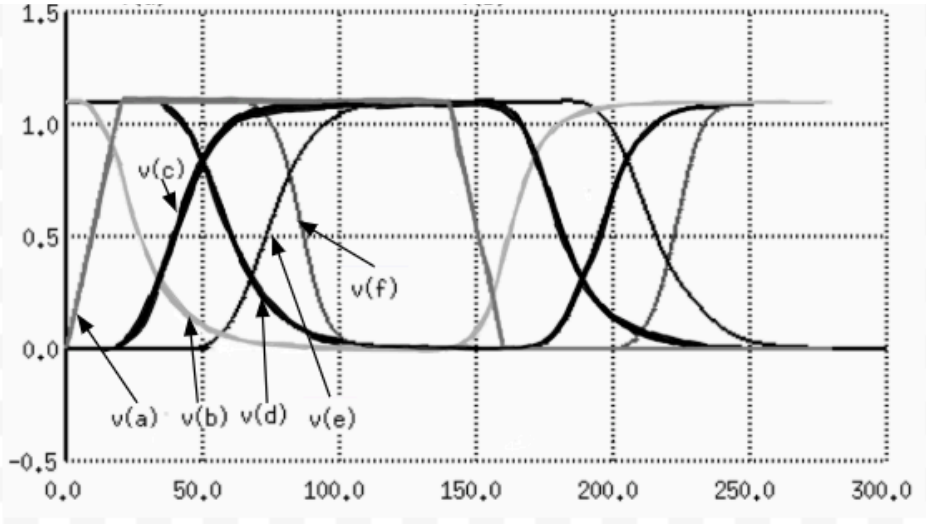


Figure 3: ngspice Waveform (Inverters built with nMOS and pMOS)

```
  Measurements for Transient Analysis

tpdr                =  1.532255e-11 targ=  1.958756e-10 trig=  1.805530e-10
tpdf                =  1.816119e-11 targ=  5.911459e-11 trig=  4.095339e-11
tpd                 =  1.67419e-11
```

Figure 4: ngspice Transient Analysis (Inverters built with nMOS and pMOS)

As we use cells from the NanGate 45nm library when pushing our designs through the ASIC flow, we also built another spice stack using standard inverter cells. Since there are only inverters of size X1, X2, X4, X8, X16, and X32, we used INV_X1, INV_X4, and INV_X16 to build a three-stage path with the first inverter shaping the input, the second as the device under test, and the third as the load. By inspecting the library we find both standard cells and standard cells with parasitic delays. Comparing results in *figure 4*, *figure 5*, and *figure 6*, we conclude that the standard cell library includes optimizations on the layout as well as the relative sizes of

nMOS and pMOS transistors so it has a smaller FO4 delay. By adding parasitic delay to the standard cell, the model more accurately reflected the delay which is longer than the ideal model. Our estimated FO4 is slightly below the FO4 delay of the IBM 65nm process, which is very reasonable.

```
  Measurements for Transient Analysis

tpdr                =  1.402418e-11 targ=  3.512440e-11 trig=  2.110022e-11
tpdf                =  1.280554e-11 targ=  1.750416e-10 trig=  1.622361e-10
tpd                 =  1.34149e-11
```

Figure 5: ngspice Transient Analysis (Inverters from stdcells.spi)

```
  Measurements for Transient Analysis

tpdr                =  1.524427e-11 targ=  3.692069e-11 trig=  2.167642e-11
tpdf                =  1.397153e-11 targ=  1.768417e-10 trig=  1.628701e-10
tpd                 =  1.46079e-11
```

Figure 6: ngspice Transient Analysis (Inverters from stdcells-lpe.spi)

The FO4 delay gives us a relatively accurate metric to assess how well our superpipelined processor divides the stages. According to Hrishikesh *et al.* [4], the optimal depth for each pipeline is 6-8 FO4 delays. Our baseline 5-stage bypassing processor has a cycle time of 1.2ns, which is around 82 FO4 delays. By comparing with other processors of different technologies and eras in *figure 7*, we find our baseline processor at a similar position as the Power/PowerPC processor in 1994, which also had about five stages. The FO4 delay provides a metric for processor cycle time and helps us set goals and evaluate performance of our superpipelined processor optimization.

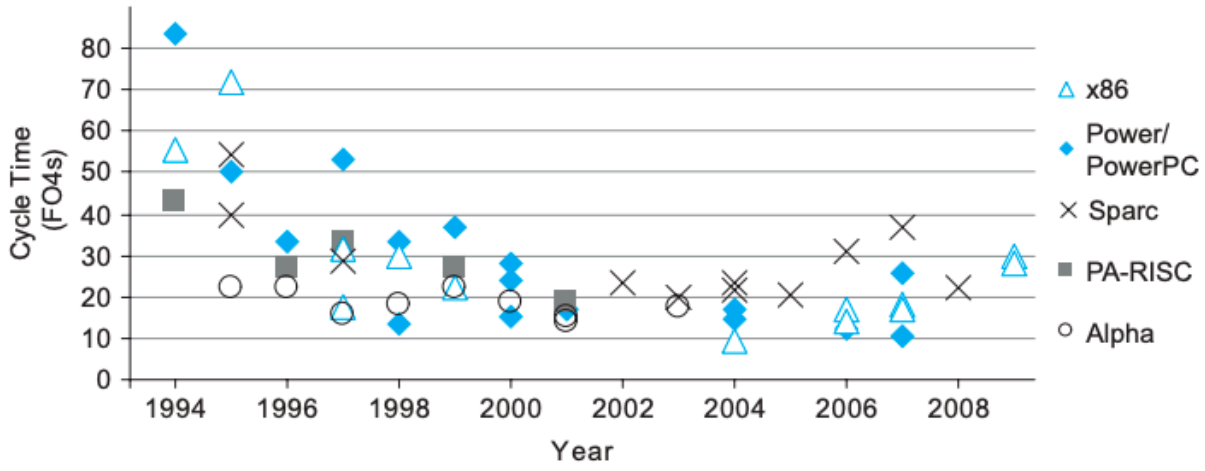


Figure 7: Microprocessor Cycle Time Trends (Weste and Harris p.175 Fig 4.38)

3. Baseline Design

The baseline design focuses on implementing a five-stage pipelined processor with hardware stall and bypass to handle data hazards. One of the most important features of the processor design is the separation of the datapath and control unit: such implementation corresponds to the actual separation between microarchitecture and ISA. Control unit translates the program into hardware language - signals and datapath understands the instruction semantics and performs the executions. We can easily change the instructions and control tables without changing the datapath design - a strategy creating an interface between software and hardware and ensures the safety of the design. This baseline design is a good start for alternative design since this five-stage pipelined processor divides the stages in a somewhat balanced manner, based on datapath characteristics and state functionality.

3.1 Datapath

As shown in *Figure 8*, the datapath is split into five stages: fetch, decode, execution, memory, and writeback. Right before the fetch stage, the program counter (PC) is sent to instruction memory. There is an imem drop unit if the requested instruction is not used, which happens due to a squash. Fetch stage increases the PC and reads the instruction pointed by the PC in the memory. A PC mux is used to select multiple PC targets, which can be generated from a branch or a jump. In the decode stage, instruction fetched in the previous cycle is sent to the control unit, which will interpret the 32-bit instruction into different control signals and send them back to the datapath; besides, the decode stage is also responsible for sending input data into the arithmetic logic unit (ALU) and the multiplier. The execution stage takes input from the decode stage, executes the instruction with ALU or multiplier, and outputs the data to the next stage; memory request is also sent if needed by the instruction. In the memory stage, memory response data is sent from the memory and the control unit sends the signal to decide whether to use memory data or use ALU data. The last stage, write back stage writes data into the register file, and whether the stage is useful depends on the instruction type.

Between pipeline stages there exist registers. They store the results of each stage and pass it onto the next stage if there are no stall signals. If there are they will hold the values until the stall is finished. These registers are critical as they are how we make a single-cycle processor into a pipelined processor.

3.2 Control Unit

As illustrated in the previous paragraph, the control unit determines all the signals that select mux inputs and controls pipeline register timing. The most important part, the decode stage, defines a control table that interprets instructions into signals and outputs them to the datapath. These signals include mux control signals, register enable signals, and data type signals. These different signals are passed to different stages accordingly, which maximizes the processor's performance. Another important function of the control unit is to issue stall and

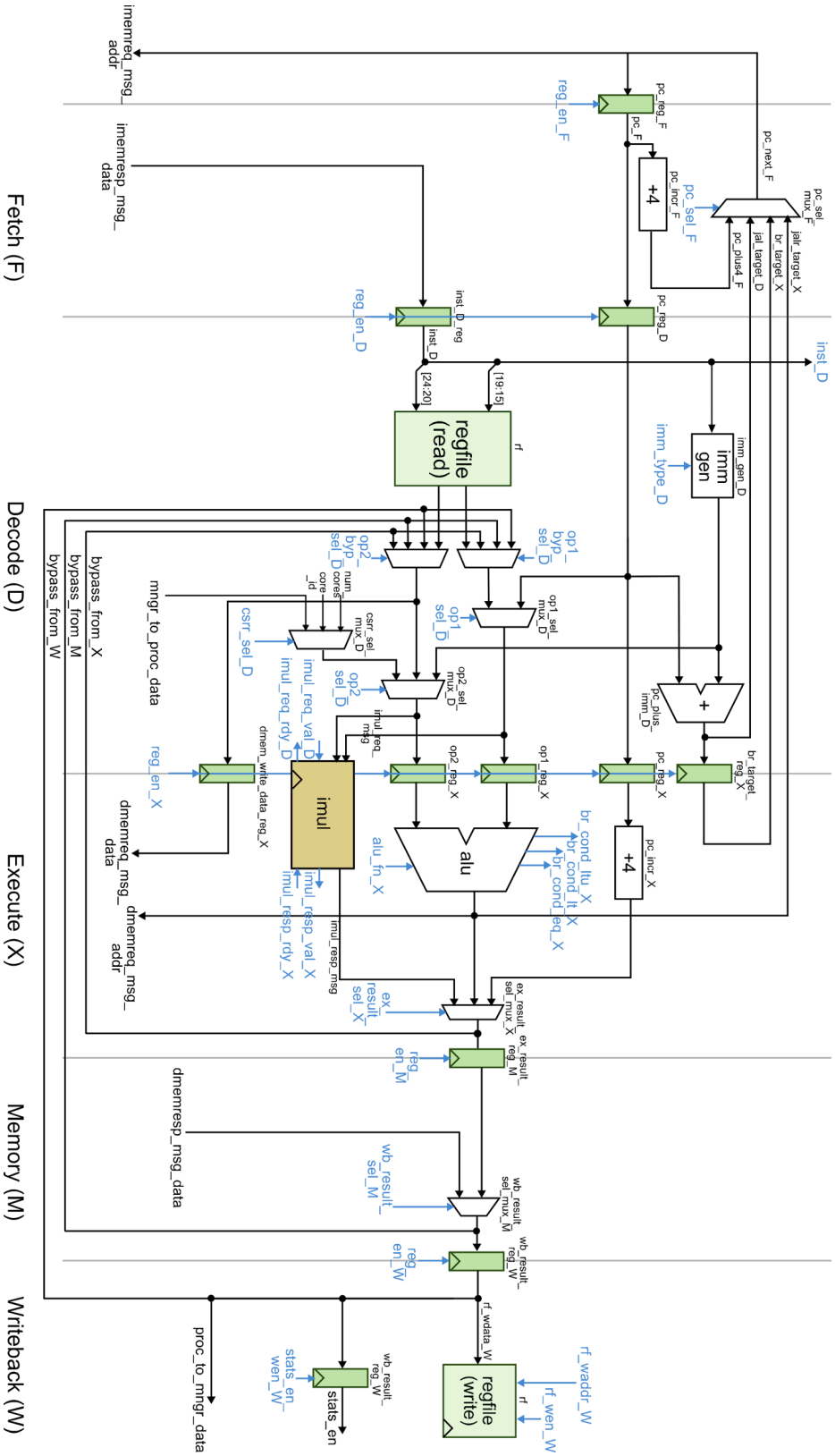


Figure 8: Baseline Processor Datapath
(https://www.csl.cornell.edu/courses/ece5745/handouts/ece5745-lab2-xcel.pdf)

squash signals to avoid data hazards. With all the signals interpreted by the control unit, it is easy to know what instructions are being executed in different stages so that we can implement stall and squash signals to prevent the pipeline from proceeding until data hazards are resolved.

3.3 Implementation Details

As we are following the Tiny RISC-V 2 processor ISA, we implemented 34 instructions that the processor supports by taking an incremental approach. The first set of instructions that we implemented is the register-register type instructions: `add`, `mul`, `and`, `or`, `xor`, `slt`, `sltu`, `sra`, `srl`, `sll`. For those instructions, we declared new ALU function types in the D stage and added them to the control table. The difference between those instructions is their ALU function type. However `mul` is very different. We had to add an execute result mux in the X stage. This mux can select from PC increment output, ALU output, or the single stage multiplier output. We also had to set the multiplier request and response in the control table for stall and squash signals. The second set of instructions we implemented is the register-immediate type instructions: `addi`, `andi`, `ori`, `xori`, `slti`, `sltiu`, `srai`, `srli`, `slli`, `auipc`. Those instructions are very similar to their corresponding register-register instructions except that they use immediate for operand 2 mux selection and have an immediate type select.

The third set of instructions we implemented is the memory instructions `lw` and `sw`. Load and store involves sending memory request messages and waiting for memory responses in the M stage. Memory access address is calculated in the X stage. If it is a `sw` instruction, data is read from the register and passed into the memory request message; otherwise if it is a `lw` instruction, we send a read type memory request and wait for the data from the memory response. The fourth set of instructions we implemented is the jump instructions `jal` and `jalr`. In the F stage, the PC selection logic is designed such that if a jump instruction is interpreted in D stage, we would use the PC from D stage. In the D stage, we first declare the new jump types, and then we check if the instruction is `jal` to redirect the PC if necessary. For `jalr` instructions, we calculate the target branch in the X stage with a new ALU function. If instruction is `jalr`, we need to use the jalr target as the next PC. The fifth and final set of instructions we implemented is the branch instructions: `bne`, `beq`, `blt`, `bltu`, `bge`, and `bgeu`. We first declare those branch types in D stage and fill the control table accordingly. Then we check if the branch is taken in the X stage and redirect the PC if it is taken.

Some efforts are put into creating correct stall, squash, and bypass logics to handle hazards that appear in a pipelined processor, including read-after-write data hazard and control hazard brought by branch and jump instructions. Bypassing allows us to avoid hazards without having to stall for every single instruction; as a result, the overhead for load-use data hazard becomes only one cycle. We only have to stall for load word instruction since the data needed to be bypassed is ready at M stage instead of D stage, so the load-use latency is two cycles.

As bypassing data is used to replace the original data output of the register file and we still want the immediate and the PC signals to be potential operands for the ALU unit, we choose to place the bypassing muxes between the register file and the select mux for its corresponding operand. Each bypass mux has four inputs, outputs from the bypassing paths of X, M and W stage and the output of the register file. The output of each bypass mux is connected to an input of its corresponding operand select mux in stage D. If bypassing is needed for resolving the hazard, the bypassing mux select signal will choose the bypassed data and the operand select mux will use the output from the bypassing mux.

As for the control units, we have six bypassing signals and two `ostall` signals for load use dependency. The bypassing signals indicate if a bypass is needed and, if needed, from which stage among X, M or W does the bypass path come from, and which operand, op1 or op2, will use the data. The load-use dependency is an exception because loaded data is available at the end of the M stage and we have to bypass data back to the D stage to resolve this hazard. To correctly handle stalls and squashes for branches and jump, we have additional status and control signals. `jal` needs to be resolved in the D stage and branches and `jalr` in the X stage. We change the PC target mux selection signal based on the instruction type and originate squashes from the stage if the PC is redirected.

The separation of the datapath and control unit is a clear example of modularity and encapsulation. It prevents other modules from being modified while changes are made. Modularity is also shown in the datapath design. As we instantiate registers, muxes, ALU, and the multiplier from existing modules, we create a hierarchy of modules with the processor design at the top, datapath, and control unit in the middle, and other basic components at the bottom. This adds simplicity to the design, makes it more extensible, and decreases the chance of corrupting other functionalities during implementation. As the stall strategy is purely hardware, bringing convenience to the programmers who don’t need to worry about adding extra operations to avoid data hazards.

4. Alternative Design

To implement a superpipelined processor based on the original baseline design, we decided to take an incremental approach, splitting the critical path of the previous design and then push the asic flow again to decide the constrainting path of the new design. The best timing we can achieve for the 5-stage bypassing processor is 1.2 ns, which is approximately 82 FO4 delays. The critical path starts from the register for operand 2 in the X stage, goes into the ALU, enters the control logic from ALU's branch prediction signal, and ends at the instruction memory request register. The gate level list for the critical path is shown in *figure 9* and *figure 10*. So the first step we take is to split the X stage.

Instance	Arc	Cell	Slew	Delay	Arrival Time	Required Time
	clk[0] ^		0.025		-0.134	-0.134
CTS_ccl_a_BUF_ideal_clock_G0_L1_1		CLKBUF_X2	0.025	0.001	-0.133	-0.133
CTS_ccl_a_BUF_ideal_clock_G0_L1_1	A ^ -> Z ^	CLKBUF_X2	0.036	0.064	-0.069	-0.069
proc/v/dpath/CTS_ccl_a_BUF_ideal_clock_G0_L2_3		CLKBUF_X2	0.036	0.002	-0.067	-0.067
proc/v/dpath/CTS_ccl_a_BUF_ideal_clock_G0_L2_3	A ^ -> Z ^	CLKBUF_X2	0.027	0.059	-0.009	-0.008
proc/v/dpath/op2_reg_X/clk_gate_q_reg/latch		CLKGATETST_X4	0.027	0.001	-0.008	-0.008
proc/v/dpath/op2_reg_X/clk_gate_q_reg/latch	CK ^ -> GCK ^	CLKGATETST_X4	0.027	0.054	0.046	0.046
proc/v/dpath/op2_reg_X/q_reg_11_		DFF_X1	0.027	0.002	0.048	0.048
proc/v/dpath/op2_reg_X/q_reg_11_	CK ^ -> Q ^	DFF_X1	0.078	0.174	0.221	0.222
proc/v/dpath/alu/lt_x_7/U689		NOR2_X2	0.078	0.002	0.223	0.224
proc/v/dpath/alu/lt_x_7/U689	A1 ^ -> ZN v	NOR2_X2	0.017	0.009	0.233	0.233
proc/v/dpath/alu/lt_x_7/U706		NOR2_X1	0.017	0.000	0.233	0.233
proc/v/dpath/alu/lt_x_7/U706	A1 v -> ZN ^	NOR2_X1	0.018	0.028	0.261	0.261
proc/v/dpath/alu/lt_x_7/U707		AOI22_X1	0.018	0.000	0.261	0.261
proc/v/dpath/alu/lt_x_7/U707	B2 ^ -> ZN v	AOI22_X1	0.027	0.026	0.287	0.287
proc/v/dpath/alu/lt_x_7/U710		AOI221_X1	0.027	0.000	0.287	0.287
proc/v/dpath/alu/lt_x_7/U710	B2 v -> ZN ^	AOI221_X1	0.043	0.087	0.373	0.374
proc/v/dpath/alu/lt_x_7/U714		AOI211_X1	0.043	0.000	0.374	0.374
proc/v/dpath/alu/lt_x_7/U714	A ^ -> ZN v	AOI211_X1	0.018	0.025	0.399	0.399
proc/v/dpath/alu/lt_x_7/FE_RC_840_0		OAI21_X2	0.018	0.000	0.399	0.400
proc/v/dpath/alu/lt_x_7/FE_RC_840_0	A v -> ZN ^	OAI21_X2	0.019	0.025	0.424	0.424
proc/v/dpath/alu/lt_x_7/U748		AOI21_X1	0.019	0.000	0.424	0.425
proc/v/dpath/alu/lt_x_7/U748	B1 ^ -> ZN v	AOI21_X1	0.011	0.020	0.444	0.444
proc/v/dpath/alu/lt_x_7/U781		OAI21_X2	0.011	0.000	0.444	0.444
proc/v/dpath/alu/lt_x_7/U781	B1 v -> ZN ^	OAI21_X2	0.041	0.057	0.501	0.501
proc/v/ctrl/FE_OFC368_br_cond_lt_X		INV_X1	0.041	0.001	0.502	0.502
proc/v/ctrl/FE_OFC368_br_cond_lt_X	A ^ -> ZN v	INV_X1	0.010	0.010	0.511	0.512
proc/v/ctrl/FE_RC_594_0		OAI33_X1	0.010	0.000	0.511	0.512
proc/v/ctrl/FE_RC_594_0	A1 v -> ZN ^	OAI33_X1	0.058	0.042	0.553	0.554
proc/v/ctrl/U148		NAND2_X1	0.058	0.000	0.553	0.554
proc/v/ctrl/U148	A1 ^ -> ZN v	NAND2_X1	0.020	0.026	0.579	0.580
proc/v/ctrl/U153		OAI211_X2	0.020	0.000	0.579	0.580
proc/v/ctrl/U153	A v -> ZN ^	OAI211_X2	0.025	0.029	0.608	0.609
proc/v/ctrl/U156		NAND2_X2	0.025	0.000	0.609	0.609
proc/v/ctrl/U156	A1 ^ -> ZN v	NAND2_X2	0.014	0.024	0.633	0.633
proc/v/ctrl/U158		AOI21_X2	0.014	0.000	0.633	0.634
proc/v/ctrl/U158	B2 v -> ZN ^	AOI21_X2	0.025	0.037	0.670	0.671
proc/v/imem_drop_unit/FE_RC_831_0		NOR3_X1	0.025	0.000	0.670	0.671
proc/v/imem_drop_unit/FE_RC_831_0	A1 ^ -> ZN v	NOR3_X1	0.009	0.015	0.686	0.686
proc/v/imemresp_q/genblk1_ctrl/U6		INV_X2	0.009	0.000	0.686	0.686
proc/v/imemresp_q/genblk1_ctrl/U6	A v -> ZN ^	INV_X2	0.007	0.012	0.698	0.699
proc/v/imemresp_q/genblk1_ctrl/U7		NAND2_X1	0.007	0.000	0.698	0.699
proc/v/imemresp_q/genblk1_ctrl/U7	A1 ^ -> ZN v	NAND2_X1	0.020	0.015	0.713	0.713
proc/v/ctrl/U160		NAND2_X2	0.020	0.000	0.713	0.713
proc/v/ctrl/U160	A1 v -> ZN ^	NAND2_X2	0.011	0.020	0.733	0.734
proc/v/ctrl/U162		NAND2_X1	0.011	0.000	0.733	0.734
proc/v/ctrl/U162	A1 ^ -> ZN v	NAND2_X1	0.010	0.016	0.750	0.750
proc/v/ctrl/FE_OFC485_imemresp_deq_en		INV_X2	0.010	0.000	0.750	0.750
proc/v/ctrl/FE_OFC485_imemresp_deq_en	A v -> ZN ^	INV_X2	0.011	0.019	0.768	0.769
proc/v/ctrl/U166		NOR2_X2	0.011	0.000	0.768	0.769
proc/v/ctrl/U166	A1 ^ -> ZN v	NOR2_X2	0.008	0.010	0.779	0.779
proc/v/imemreq_q/genblk1_ctrl/U5		INV_X2	0.008	0.000	0.779	0.779
proc/v/imemreq_q/genblk1_ctrl/U5	A v -> ZN ^	INV_X2	0.009	0.015	0.794	0.795
proc/v/imemreq_q/genblk1_ctrl/U9		NAND2_X1	0.009	0.000	0.794	0.795
proc/v/imemreq_q/genblk1_ctrl/U9	A2 ^ -> ZN v	NAND2_X1	0.024	0.014	0.809	0.809
proc/v/U4		AND2_X2	0.024	0.000	0.809	0.809
proc/v/U4	A2 v -> ZN v	AND2_X2	0.008	0.040	0.849	0.849
proc/v/imemreq_q/genblk1_ctrl/U6		AOI21_X1	0.008	0.000	0.849	0.850
proc/v/imemreq_q/genblk1_ctrl/U6	B1 v -> ZN ^	AOI21_X1	0.041	0.035	0.884	0.884
proc/v/imemreq_q/genblk1_dpath/qstore/FE_RC_850_0		AND2_X1	0.041	0.000	0.884	0.884
proc/v/imemreq_q/genblk1_dpath/qstore/FE_RC_850_0	A2 ^ -> ZN ^	AND2_X1	0.009	0.040	0.924	0.924
proc/v/imemreq_q/genblk1_dpath/qstore/clk_gate_rfile_reg_0_/latch		CLKGATETST_X4	0.009	0.000	0.924	0.924

Figure 9. Gate Level Critical Path for 5-stage processor

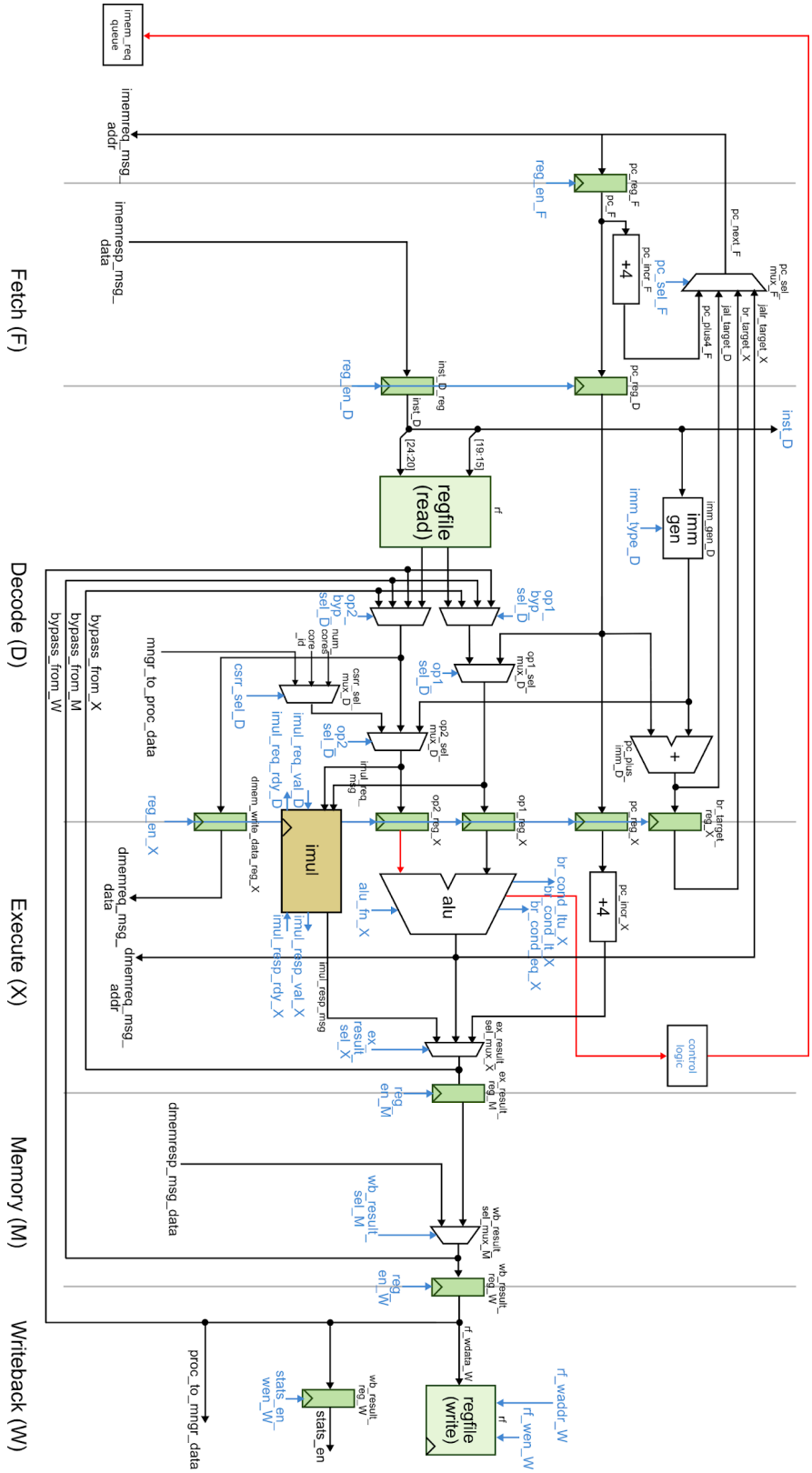


Figure 10. Critical Path for 5-stage Processor (Based on https://www.csl.cornell.edu/courses/ece5745/handouts/ece5745-lab2-xcel.pdf)

4.1 Splitting X Stage into X0/X1

As the Arithmetic Logic Unit (ALU) is the main component of the X stage, we decide to split the ALU into two different parts: the operation of the lower 16 bits of the operands are finished in X0 stage and the remaining necessary data are passed into X1 to finish the higher 16 bits computation. With additional logic described in 4.1.2, this design is possible to avoid stalling due to read-after-write dependencies between X0 and X1 by bypassing data from X0 back to D and from X1 back to X0. Instructions do not need to wait until the instructions they depend on to finish computing at X1 stage.

4.1.1 Stalling Without Additional Bypassing

4.1.1.a Datapath

There are 14 instructions we need to support: arithmetic operations (`add`, `sub`), logical operations (`and`, `or`, `xor`, `nor`), shift operation (`sll`, `srl`, `sra`), comparisons (`lt`, `ltu`), and special operations (`cp0`, `cp1`, and `adz`). To support arithmetic operation, we need to pass a `carry_out` bit between the lower-16-bit ALU and the upper-16-bit ALU. Like what we would do in a normal ripple carry adder, the `carrry_out` bit indicates whether we need to add/subtract 1 from the upper-16-bit addition/subtraction. The result of the lower-16-bit computation is also passed through a register into the X1 to generate the final result. The logical operation and the two copy instructions are the easiest to implement as they do not have any dependencies between the upper and lower 16 bits so we keep the original implementation and pass the output from X0 to X1 stage.

Even though shift operations take two 32-bit inputs, it only uses the lower 5 bits of operand 2 to indicate the shift amount, which means we need to pass the `shift_op1` value to the X1 stage so that the upper 16 bits can shift with the right parameters. Instruction `sll` performs a logical left shift, meaning there are 0 to 16 bits to be shifted into the upper 16 bit position. So we pass the `shift_out` bits into the upper-16-bit ALU together with the lower-16-bit ALU output. In X1 stage we shift the upper 16 bit of the original operand 0 and *or* the concatenated the results to generate the final output. Operation `srl` is a logical right shift, meaning the upper 16 bits will be shifted to the lower 16 bit position. So we set `shift_out` to 0, zero-fill the lower 16 bit position of operand 0 of the X1 stage (which is the upper 16 bits of the actual operand 0), shift, and *or* the result with the output of X0 stage. Operation `sra` is the arithmetic right shift and thus we use a `$signed` operation to decide whether to zero-fill or one-fill the vacant bit-positions based on the sign of the original operand 0. Comparisons are first determined by the comparison in the upper 16 bits. If they are of equal value, then the result of the lower-16-bit ALU is used to decide the output. Operation `adz` is a special operation for the `JALR` instruction. It performs an addition but masks the lowest bit to 0. So its implementation is exactly the same as the `add` operation, but with the lowest bit connected to 0 in the lower-16-ALU. Besides the 14 operations, ALU also calculates flags used for branch redirection.

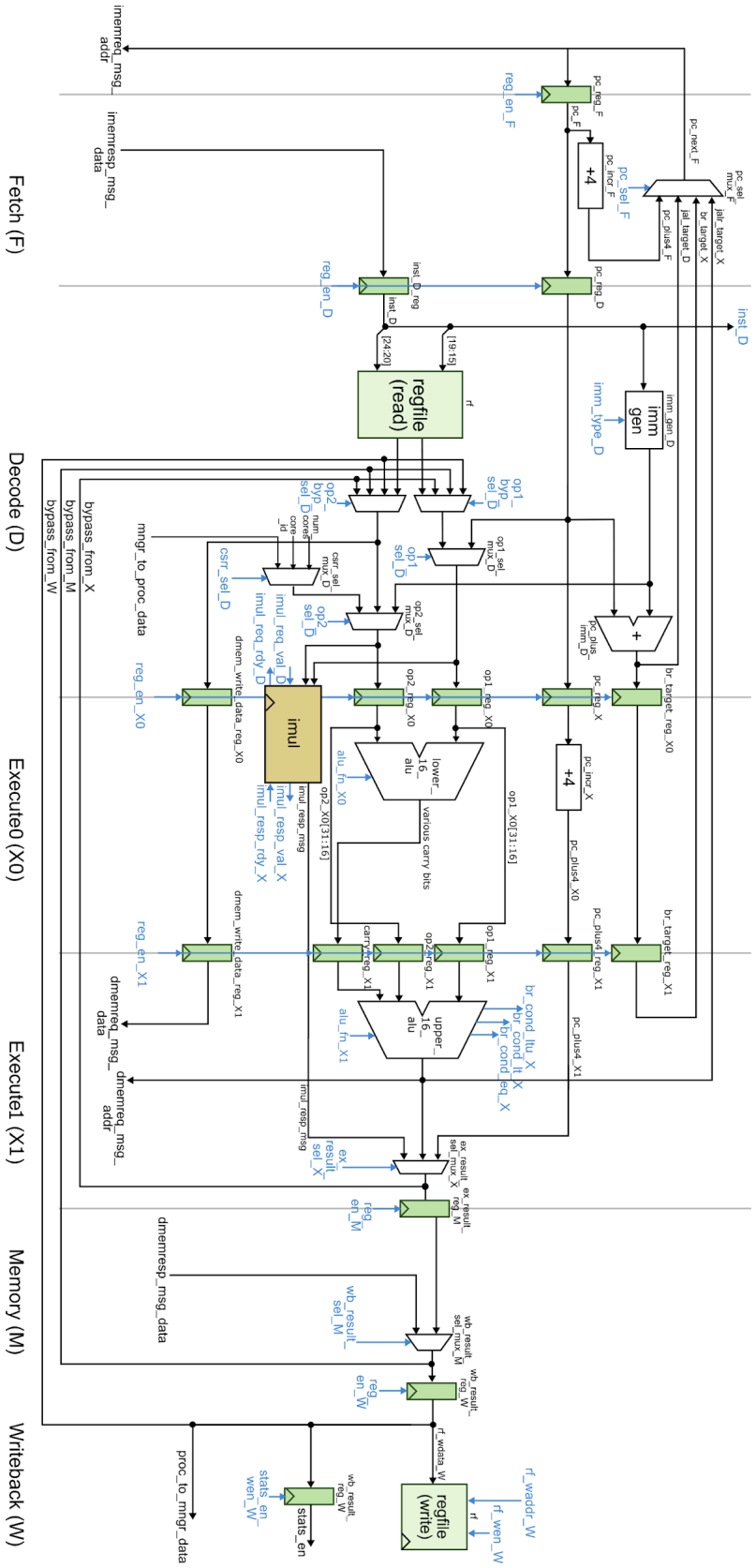


Figure 11. Datapath for X0/X1 Splitted Processor Without Additional Bypassing

Flag `ops_lt` and `ops_ltu` have exactly the same implementation as the `lt` and `ltu` operation, and `ops_eq` is implemented by doing comparison in both stages and output 1 if the two operands are the same in both X0 and X1.

After fully testing the two components, we started the integration of ALU and the processor. X0 stage is also responsible for sending the accelerator request message as they are ready at the beginning of X0 stage. In the X1 stage, after the upper-16-bit ALU finishes computation, its output is connected to the jump-and-link-register target and the data memory request message address port. There is a mux after the ALU which is used to select the actual output of the X1 stage among ALU results, multiplier response message, and PC+4 result for jump-and-link based on the instruction type.

4.1.1.b Control Logic

As we add one more stage to the processor, we need to carefully consider the stall, bypass, and squash logic for the processor. As we are implementing a basic 6-stage processor that supports all of the functionality the baseline processor has, we did not have to include the bypassing logic from X0 to D and from X1 to X0. Instead, in the Decode stage, when the processor realizes that the destination register of stage X0 matches with the operand registers in stage D, we stall for one cycle to avoid hazard caused by read-after-write dependency; if the instruction in X1 stage is a `lw`, and its destination register matches with the operand register in the D stage, we stall until we get the data back from the memory. Besides, as we implement the processor to support the accelerator, we also need to stall at D stage if there is a `csrrx` instruction in-flight at X0/X1 stage since its value will not be returned from the accelerator until M stage. X0 stage only originates stall when the accelerator request is not ready; X1 stage originates stalls if its instruction is multiplication and the multiplier is not ready to respond yet; it also stalls if its instruction needs to send a data memory request but the memory is not ready to take requests. X1 also originates a squash if it needs to handle a PC redirect for branch or jump instruction. Same to our design in the baseline, one stage will be stalled if itself or any state after it originates a stall, and will be squashed only when a stage after it originates a squash.

4.1.1.c Critical Path

By pushing the design through the ASIC flow, we get a minimum cycle time of 1.3 ns and the critical path is at the D stage (illustrated in *figure 12* and *figure 13*): it starts from the pipeline register for instruction, goes through the register file and the bypassing muxes, and ends at the operand register between D stage and X0 stage. This critical path shows that it might be necessary for us to insert pipeline registers in the D stage after the register file to reduce the cycle time.

Instance	Arc	Cell	Slew	Delay	Arrival Time	Required Time
	clk[0] ^		0.023		-0.134	-0.132
proc/v/CTS_ccl_a_BUF_ideal_clock_G0_L1_2		CLKBUF_X3	0.023	0.001	-0.133	-0.131
proc/v/CTS_ccl_a_BUF_ideal_clock_G0_L1_2	A ^ -> Z ^	CLKBUF_X3	0.032	0.062	-0.071	-0.069
proc/v/CTS_ccl_a_BUF_ideal_clock_G0_L2_13		CLKBUF_X2	0.032	0.004	-0.067	-0.065
proc/v/CTS_ccl_a_BUF_ideal_clock_G0_L2_13	A ^ -> Z ^	CLKBUF_X2	0.026	0.057	-0.010	-0.008
proc/v/dpath/inst_D_reg/clk_gate_q_reg/latch		CLKGATETST_X4	0.026	0.001	-0.009	-0.008
proc/v/dpath/inst_D_reg/clk_gate_q_reg/latch	CK ^ -> GCK ^	CLKGATETST_X4	0.026	0.053	0.044	0.045
proc/v/dpath/inst_D_reg/q_reg_15_		DFF_X1	0.026	0.001	0.045	0.047
proc/v/dpath/inst_D_reg/q_reg_15_	CK ^ -> Q ^	DFF_X1	0.069	0.156	0.201	0.202
proc/v/dpath/rf/rfile/U110		OR2_X1	0.069	0.003	0.204	0.205
proc/v/dpath/rf/rfile/U110	A2 ^ -> ZN ^	OR2_X1	0.083	0.116	0.320	0.321
proc/v/dpath/rf/rfile/U195		NOR2_X1	0.083	0.004	0.324	0.326
proc/v/dpath/rf/rfile/U195	A1 ^ -> ZN v	NOR2_X1	0.016	0.005	0.329	0.331
proc/v/dpath/rf/rfile/FE_OFC210_n645		CLKBUF_X1	0.016	0.000	0.329	0.331
proc/v/dpath/rf/rfile/FE_OFC210_n645	A v -> Z v	CLKBUF_X1	0.102	0.138	0.468	0.469
proc/v/dpath/rf/rfile/FE_OFC211_n645		CLKBUF_X1	0.102	0.002	0.470	0.472
proc/v/dpath/rf/rfile/FE_OFC211_n645	A v -> Z v	CLKBUF_X1	0.074	0.136	0.606	0.607
proc/v/dpath/rf/rfile/U828		AOI22_X1	0.074	0.003	0.609	0.611
proc/v/dpath/rf/rfile/U828	B1 v -> ZN ^	AOI22_X1	0.031	0.065	0.674	0.676
proc/v/dpath/rf/rfile/U829		NAND4_X1	0.031	0.000	0.674	0.676
proc/v/dpath/rf/rfile/U829	A4 ^ -> ZN v	NAND4_X1	0.045	0.083	0.757	0.759
proc/v/dpath/rf/rfile/U830		OR4_X1	0.045	0.001	0.759	0.760
proc/v/dpath/rf/rfile/U830	A4 v -> ZN v	OR4_X1	0.023	0.147	0.906	0.908
proc/v/dpath/rf/U48		AND2_X1	0.023	0.000	0.906	0.908
proc/v/dpath/rf/U48	A1 v -> ZN v	AND2_X1	0.007	0.036	0.942	0.944
proc/v/dpath/op1_byp_mux_D/U129		AOI22_X1	0.007	0.000	0.942	0.944
proc/v/dpath/op1_byp_mux_D/U129	A2 v -> ZN ^	AOI22_X1	0.027	0.038	0.980	0.982
proc/v/dpath/op1_byp_mux_D/U131		NAND2_X1	0.027	0.000	0.980	0.982
proc/v/dpath/op1_byp_mux_D/U131	A1 ^ -> ZN v	NAND2_X1	0.012	0.020	1.000	1.002
proc/v/dpath/op1_sel_mux_D/U54		MUX2_X1	0.012	0.000	1.000	1.002
proc/v/dpath/op1_sel_mux_D/U54	A v -> Z v	MUX2_X1	0.020	0.074	1.074	1.076
proc/v/dpath/op1_reg_X1/U33		AND2_X1	0.020	0.001	1.075	1.077
proc/v/dpath/op1_reg_X1/U33	A1 v -> ZN v	AND2_X1	0.007	0.034	1.109	1.110
proc/v/dpath/op1_reg_X1/q_reg_4_		DFF_X1	0.007	0.000	1.109	1.110

Figure 12. Gate Level Critical Path for X0/X1 Splitted Processor Without Additional Bypassing (X1 in figure corresponds to X0 in report)

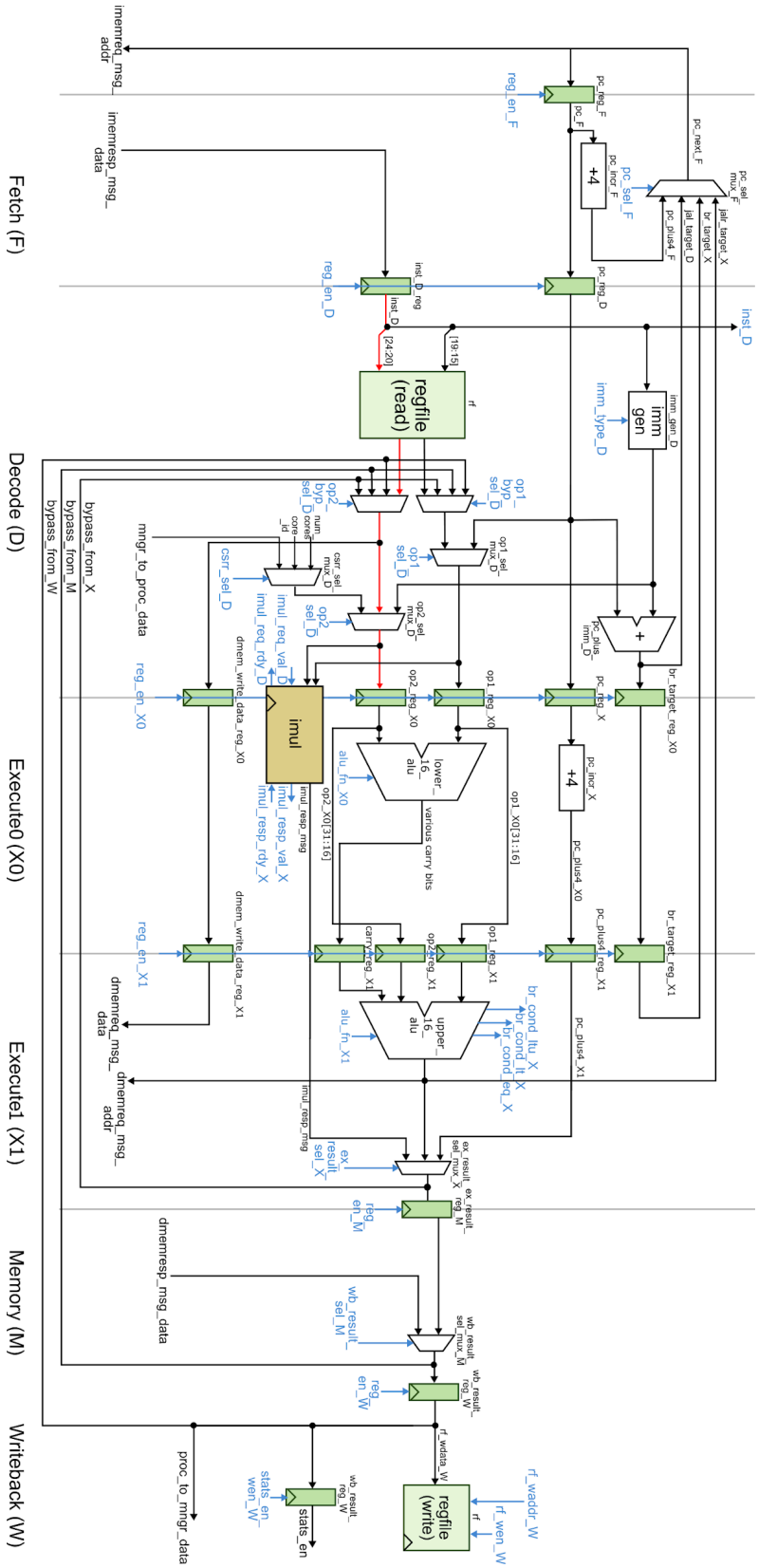


Figure 13. Critical Path for X0/X1 Splitted Processor Without Additional Bypassing

4.1.2 Bypassing in X0/X1 Stage

To avoid the extra bubble/stall introduced to the pipeline when executing back to back dependent arithmetic instructions, we plan to add extra bypass logic to our processor, namely from X0 to D and from X1 to X0. In the implementation described in *4.1*, the pipeline inserts a bubble whenever the destination register of the instruction in X0 matches one of the two input registers in D as the full 32-bit output will not be ready until the end of X1. To eliminate this bubble, we can forward the lower 16-bit result from the end of X0 to the input muxes at the D stage. We also need to add two additional input muxes at X0 so that the upper-16 bits can be forwarded from the end of X1. This will retrieve some of the CPI as back-to-back dependent instructions are extremely common in assembly. However, the extra delay slot from load-use dependency cannot be resolved as the data from memory does not reach datapath until M.

4.1.2.a Datapath

Based on the above illustration, we add a bypassing path from X0 to the end of the D stage: if the destination register of the instruction in X0 matches with any of the two operands in D stage, the lower 16 bits of the X0 ALU computation result will be bypassed into the D stage; similarly, two additional muxes are added at the end of X0 stage to conditionally bypass data from X1 to X0. To make sure data is correctly stored into memory when there is a dependency between data of the `sw` instruction and previous instruction destination, X1 result is also bypassed to the memory request data port in X0.

4.1.2.b Control Logic

We still insert a bubble for the load-use dependency scenario as data loaded from memory cannot be returned until the end of the M stage. If there is a match between accelerator request instruction (`csrw`) operands and previous instruction destination, `csrw` waits until the computation result is bypassed from X1 back to D stage as the accelerator request is sent in X0 stage. As there are dependencies between X0 and X1 stages, we add additional stall logic for multiplication, comparison, and right shift operation. As the multiplier is given two cycles to do the computation we wait until the end of the X1 stage to collect the result; comparison (`lt`, `ltu`) and right shift (`srl`, `sra`) result are not ready until X1 stage because the lower 16 bits of comparison operands are only useful when their higher 16 bits are the same and shifting right means higher 16 bits of the operand will be shifted to the lower 16 bit position of the result.

4.1.2.c Critical Path

By pushing the design through the ASIC flow, we get a minimum cycle time of 1.25 ns and the critical path is the bypassing path from the X1 stage to the D stage (illustrated in *figure 15* and *figure 16*): it starts from the operand register in X1 stage, goes through the higher-16-bit

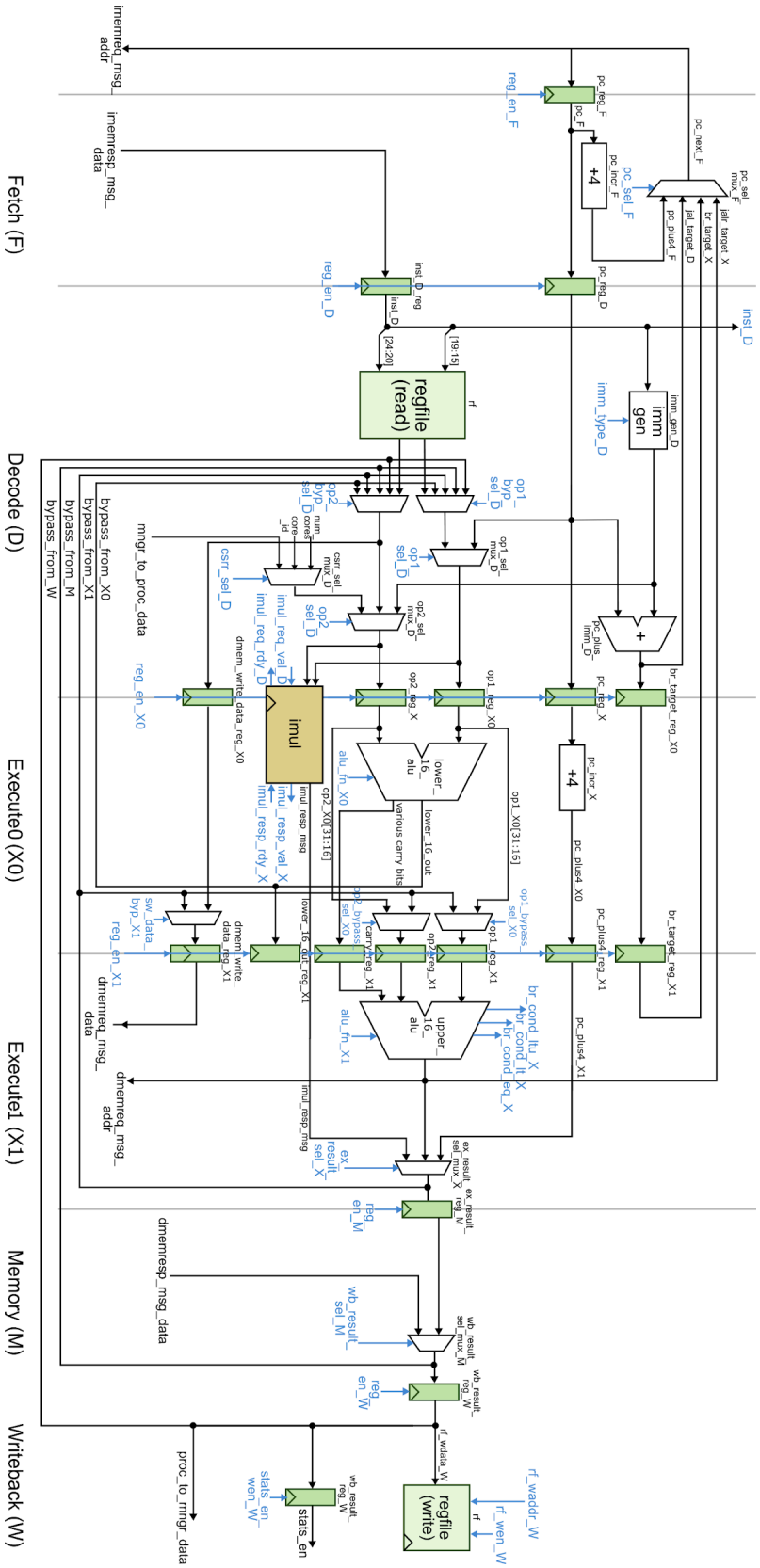


Figure 14. Datapath for X0/X1 Splitted Processor Without Full Bypassing

ALU, enters the execution result select mux, follows the bypassing path from X1 to D to the muxes in D stage, and ends at the operand register between D stage and X0 stage. Combining this critical path with the critical path we find for the non-bypassing 6-stage processor, we decide the next step is splitting the D stage into D0/D1.

	clk[0] ^		0.024		-0.138	-0.145
CTS_ccl_a_BUF_ideal_clock_G0_L1_1		CLKBUF_X3	0.024	0.001	-0.137	-0.145
CTS_ccl_a_BUF_ideal_clock_G0_L1_1	A ^ -> Z ^	CLKBUF_X3	0.030	0.061	-0.077	-0.084
proc/v/dpath/CTS_ccl_a_BUF_ideal_clock_G0_L2_2		CLKBUF_X1	0.030	0.003	-0.073	-0.081
proc/v/dpath/CTS_ccl_a_BUF_ideal_clock_G0_L2_2	A ^ -> Z ^	CLKBUF_X1	0.026	0.059	-0.015	-0.022
proc/v/dpath/op1_reg_X2/clk_gate_q_reg/latch		CLKGATETST_X2	0.026	0.000	-0.014	-0.022
proc/v/dpath/op1_reg_X2/clk_gate_q_reg/latch	CK ^ -> GCK ^	CLKGATETST_X2	0.007	0.029	0.015	0.008
proc/v/dpath/op1_reg_X2/clk_gate_q_reg/FE_USKC1279 _net5499		CLKBUF_X2	0.007	0.000	0.015	0.008
proc/v/dpath/op1_reg_X2/clk_gate_q_reg/FE_USKC1279 _net5499	A ^ -> Z ^	CLKBUF_X2	0.029	0.050	0.065	0.058
proc/v/dpath/op1_reg_X2/q_reg_10_		DFF_X1	0.029	0.001	0.065	0.058
proc/v/dpath/op1_reg_X2/q_reg_10_	CK ^ -> Q v	DFF_X1	0.027	0.116	0.181	0.174
proc/v/dpath/alu_higher16/U50		INV_X1	0.027	0.004	0.185	0.178
proc/v/dpath/alu_higher16/U50	A v -> ZN ^	INV_X1	0.034	0.051	0.236	0.229
proc/v/dpath/alu_higher16/U136		AOI22_X1	0.034	0.001	0.237	0.230
proc/v/dpath/alu_higher16/U136	B1 ^ -> ZN v	AOI22_X1	0.031	0.038	0.274	0.267
proc/v/dpath/alu_higher16/U137		AOI22_X1	0.031	0.000	0.274	0.267
proc/v/dpath/alu_higher16/U137	B2 v -> ZN ^	AOI22_X1	0.025	0.056	0.331	0.324
proc/v/dpath/alu_higher16/U138		OAI21_X1	0.025	0.000	0.331	0.324
proc/v/dpath/alu_higher16/U138	A ^ -> ZN v	OAI21_X1	0.020	0.035	0.365	0.358
proc/v/dpath/alu_higher16/U300		AND2_X1	0.020	0.000	0.365	0.358
proc/v/dpath/alu_higher16/U300	A1 v -> ZN v	AND2_X1	0.008	0.038	0.403	0.396
proc/v/dpath/alu_higher16/U311		AOI21_X1	0.008	0.000	0.403	0.396
proc/v/dpath/alu_higher16/U311	A v -> ZN ^	AOI21_X1	0.064	0.093	0.496	0.489
proc/v/dpath/alu_higher16/U312		AOI21_X1	0.064	0.000	0.497	0.489
proc/v/dpath/alu_higher16/U312	B2 ^ -> ZN v	AOI21_X1	0.020	0.031	0.528	0.520
proc/v/dpath/alu_higher16/U685		AOI22_X1	0.020	0.000	0.528	0.521
proc/v/dpath/alu_higher16/U685	A1 v -> ZN ^	AOI22_X1	0.035	0.047	0.574	0.567
proc/v/dpath/alu_higher16/U686		NAND2_X2	0.035	0.000	0.574	0.567
proc/v/dpath/alu_higher16/U686	A2 ^ -> ZN v	NAND2_X2	0.029	0.044	0.618	0.611
proc/v/dpath/ex_result_sel_mux_X/U89		AOI222_X1	0.029	0.002	0.621	0.613
proc/v/dpath/ex_result_sel_mux_X/U89	C2 v -> ZN ^	AOI222_X1	0.054	0.112	0.732	0.725
proc/v/dpath/ex_result_sel_mux_X/U90		INV_X1	0.054	0.000	0.732	0.725
proc/v/dpath/ex_result_sel_mux_X/U90	A ^ -> ZN v	INV_X1	0.023	0.035	0.767	0.760
proc/v/dpath/op2_byp_mux_D/U145		AOI22_X1	0.023	0.001	0.769	0.762
proc/v/dpath/op2_byp_mux_D/U145	B2 v -> ZN ^	AOI22_X1	0.029	0.052	0.821	0.814
proc/v/dpath/op2_byp_mux_D/U147		OAI211_X1	0.029	0.000	0.821	0.814
proc/v/dpath/op2_byp_mux_D/U147	A ^ -> ZN v	OAI211_X1	0.035	0.060	0.881	0.874
proc/v/dpath/op2_sel_mux_D/U93		AOI222_X1	0.035	0.000	0.881	0.874
proc/v/dpath/op2_sel_mux_D/U93	C2 v -> ZN ^	AOI222_X1	0.049	0.105	0.986	0.979
proc/v/dpath/op2_sel_mux_D/U94		INV_X1	0.049	0.000	0.986	0.979
proc/v/dpath/op2_sel_mux_D/U94	A ^ -> ZN v	INV_X1	0.018	0.028	1.014	1.007
proc/v/dpath/op2_reg_X1/U41		AND2_X1	0.018	0.000	1.014	1.007
proc/v/dpath/op2_reg_X1/U41	A1 v -> ZN v	AND2_X1	0.010	0.036	1.050	1.043
proc/v/dpath/op2_reg_X1/q_reg_2_		DFF_X1	0.010	0.000	1.050	1.043

Figure 15. Gate Level Critical Path for Bypassing X0/X1 Splitted Processor (X1 in figure corresponds to X0 in report)

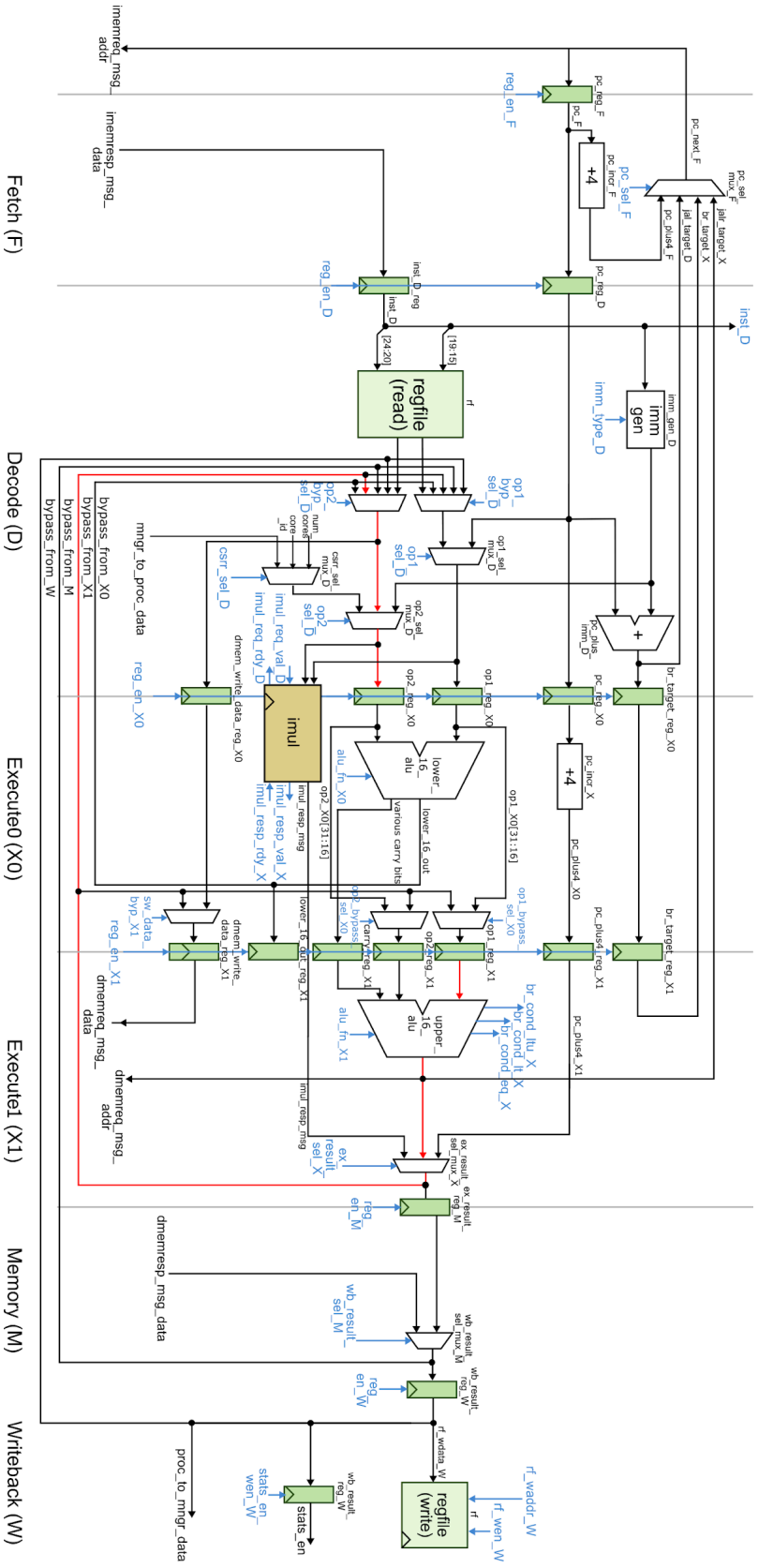


Figure 16. Critical Path for Bypassing X0/X1 Splitted Processor

4.2 Splitting D Stage into D0/D1

The critical path in section 4.1.1 goes through the register file in stage D as the register file module that we use has sequential write but combinational read, which means everything from reading from the register file to choosing the source of different operands using large muxes has to fit in one cycle. As *figure 15* and *figure 16* shows, both demonstrate a substantial contribution to the critical path. Therefore, we decide to split the D stage right after the register file to reduce the length of the critical path. However, as the register file is physically larger than the area of the bypassing and data selection muxes, the unbalanced D0 and D1 stage might bring a new critical path through the register file. As we are not sure whether the ASIC flow optimizes the register-based design into latch-based design to enable timing-borrowing between stages, the plan is to implement pipeline registers between the register file and the muxes and treat this as an incremental approach to a latch-based balanced design if necessary.

When we pushed the design in 4.1.2 through the ASIC flow, we discovered that the bypassing path from X1 to D becomes a critical path. We thought about rerouting the path to D0, but it achieves nothing as the bypassing path from M to D1 serves the same purpose. M stage also has less combinational logic than X1 so the path is less likely to be a problem. Not being able to come up with a better solution, we decided to erase this path and insert a bubble instead.

Another problem with the new D0/D1 design is that if there is a dependency between instructions in W stage and D0 stage, the data read from the register file will not match the data being written into the register file. Thus we add a bypassing path from W stage to D0 stage specifically for this RAW hazard.

4.2.1 Register-based Design

4.2.1.a Datapath

As seen in *figure 17*, we decided to insert registers right after `op1_bypass_mux_D0` and `op2_bypass_mux_D1`. The rationale behind this is that we are trying to balance the two stages. While time borrowing using latches might help relax the timing constraint, we still need to properly balance the logic for it to work. The bypassing path mentioned earlier from X1 to D1 is deleted from the data path, and a new bypassing path from W to D0 appears with two muxes to select between register file outputs and bypassing results.

One special case we discover makes it necessary to add a bypassing path from M stage to D0 stage. If dependencies between instructions force the instruction `I2` to stall at D0 when `I1`, the instruction it depends on, is at M stage and stall at D1 until `I1` writes back, it will read the wrong data from the register file and has no chance to read the data correctly bypassed from W to D1. Thus the bypassing mux in D0 has three inputs: one from the register file, one from M stage, and one from W stage.

4.2.1.b Control Logic

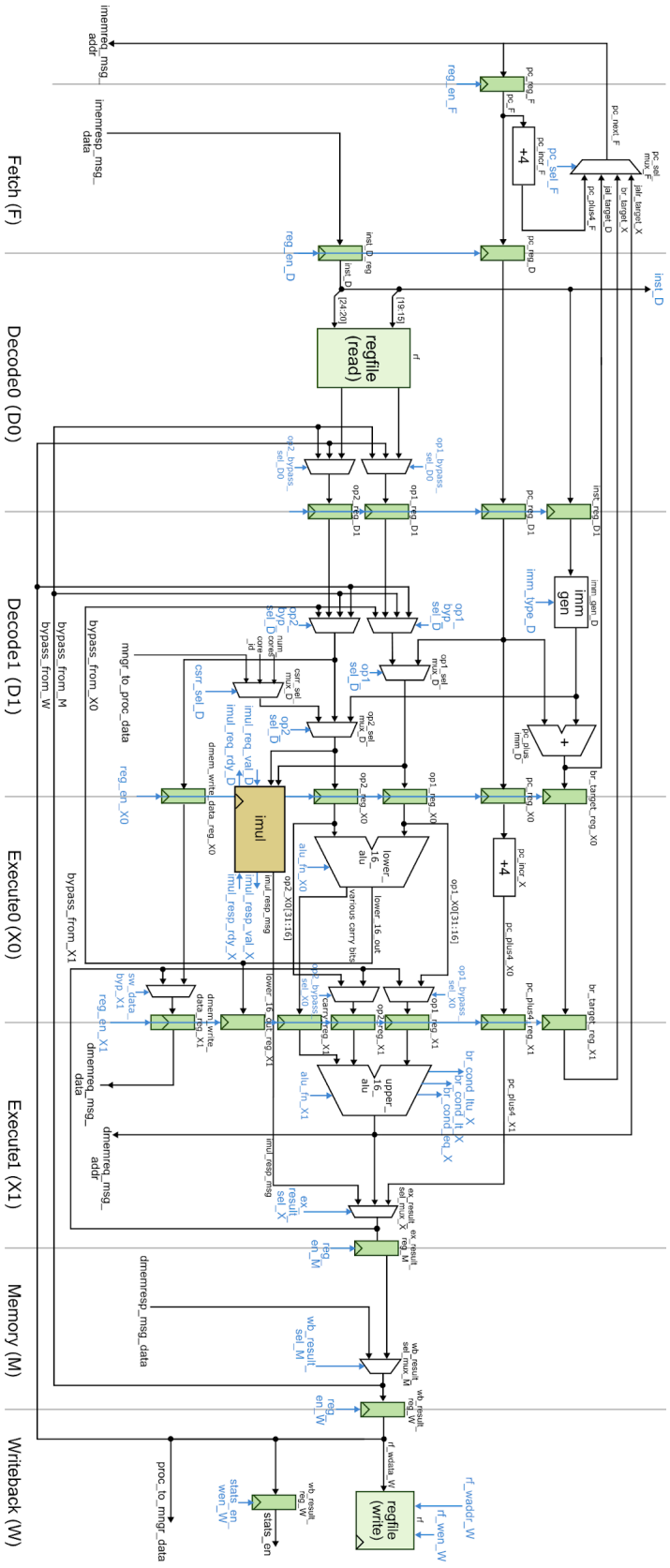


Figure 17. Datapath for D0/D1 Splitted Processor

As we add yet another stage to the processor, we need to again carefully consider the stall, bypass, and squash logic for the processor. Since we eliminated the bypassing path from X1 to D1, when the processor compares the destination register of stage X1 and the operand registers in stage D1 and find a match, it stalls to avoid hazard caused by read-after-write dependency; just like before, if the instruction in X1 stage is a `lw`, we stall until we the data is back from the memory. Stage D0 does not originate a stall as the instruction will go to D1 no matter what because there are no hazards between D0 and D1. The squash logic in D1 is inherited completely from the old stage D.

The bypassing logic from M to D0 and from W to D0 are almost exactly the same as other bypassing logics. When the processor compares the destination register and the input registers, it enables the bypassing result to go through the mux. One modification we made, however, is that we determine this without considering whether the result will be used or not. There are two reasons for this. First, whether we are using results from the register file is determined by a control table that we placed in D1, and D0 has no access to it. Second, it does not matter whether the result would be used. As long as the correct register results are provided, D1 will correctly choose which inputs to use.

4.2.1.c Critical Path

After we tested the functionality of our implementation, we pushed the design through ASIC flow to see if we improved timing and which path we had to fix next. We get a minimum cycle time of 1.13ns and unexpectedly the critical path is from X1 to F, as shown in *figure 18* and *figure 19*. The critical path originates from op2_reg_X1, through the upper-16 ALU and pc_sel_mux_F, and ends at a queue from imemreq. This is likely due to the extra logic inserted between the two stages affecting the place and route. The path is physically longer and creates a larger delay. Although not as much as we had hoped for, this is the first improvement in timing that we see.

	clk[0] ^		0.027		-0.140	-0.135
v/CTS_ccl_a_BUF_ideal_clock_G0_L1_1		CLKBUF_X2	0.027	0.002	-0.138	-0.134
v/CTS_ccl_a_BUF_ideal_clock_G0_L1_1	A ^ -> Z ^	CLKBUF_X2	0.033	0.062	-0.075	-0.071
v/dpath/CTS_ccl_a_BUF_ideal_clock_G0_L2_3		CLKBUF_X1	0.033	0.002	-0.074	-0.070
v/dpath/CTS_ccl_a_BUF_ideal_clock_G0_L2_3	A ^ -> Z ^	CLKBUF_X1	0.031	0.065	-0.008	-0.004
v/dpath/op2_reg_X2/clk_gate_q_reg/latch		CLKGATETST_X2	0.031	0.000	-0.008	-0.004
v/dpath/op2_reg_X2/clk_gate_q_reg/latch	CK ^ -> GCK ^	CLKGATETST_X2	0.027	0.056	0.048	0.052
v/dpath/op2_reg_X2/q_reg_1_		DFF_X1	0.027	0.001	0.049	0.053
v/dpath/op2_reg_X2/q_reg_1_	CK ^ -> Q ^	DFF_X1	0.020	0.108	0.156	0.160
v/dpath/alu_higher16/FE_OCPC994_op2_X2_1		CLKBUF_X1	0.020	0.000	0.156	0.160
v/dpath/alu_higher16/FE_OCPC994_op2_X2_1	A ^ -> Z ^	CLKBUF_X1	0.025	0.054	0.210	0.214
v/dpath/alu_higher16/DP_OP_61J6_122_3532/U288		NAND2_X1	0.025	0.000	0.211	0.215
v/dpath/alu_higher16/DP_OP_61J6_122_3532/U288	A2 ^ -> ZN v	NAND2_X1	0.015	0.026	0.237	0.241
v/dpath/alu_higher16/DP_OP_61J6_122_3532/U331		OAI21_X1	0.015	0.000	0.237	0.241
v/dpath/alu_higher16/DP_OP_61J6_122_3532/U331	B2 v -> ZN ^	OAI21_X1	0.019	0.035	0.271	0.276
v/dpath/alu_higher16/DP_OP_61J6_122_3532/U332		AOI21_X1	0.019	0.000	0.271	0.276
v/dpath/alu_higher16/DP_OP_61J6_122_3532/U332	A ^ -> ZN v	AOI21_X1	0.014	0.020	0.292	0.296
v/dpath/alu_higher16/DP_OP_61J6_122_3532/U345		OAI21_X1	0.014	0.000	0.292	0.296
v/dpath/alu_higher16/DP_OP_61J6_122_3532/U345	B1 v -> ZN ^	OAI21_X1	0.042	0.060	0.352	0.356
v/dpath/alu_higher16/DP_OP_61J6_122_3532/U350		AOI21_X2	0.042	0.000	0.352	0.356
v/dpath/alu_higher16/DP_OP_61J6_122_3532/U350	B1 ^ -> ZN v	AOI21_X2	0.016	0.024	0.376	0.380
v/dpath/alu_higher16/DP_OP_61J6_122_3532/U353		OAI21_X1	0.016	0.000	0.377	0.381
v/dpath/alu_higher16/DP_OP_61J6_122_3532/U353	B1 v -> ZN ^	OAI21_X1	0.033	0.046	0.423	0.427
v/dpath/alu_higher16/DP_OP_61J6_122_3532/U364		AOI21_X1	0.033	0.000	0.423	0.427
v/dpath/alu_higher16/DP_OP_61J6_122_3532/U364	B1 ^ -> ZN v	AOI21_X1	0.017	0.029	0.452	0.456
v/dpath/alu_higher16/DP_OP_61J6_122_3532/U367		OAI21_X1	0.017	0.000	0.452	0.456
v/dpath/alu_higher16/DP_OP_61J6_122_3532/U367	B1 v -> ZN ^	OAI21_X1	0.032	0.045	0.497	0.501
v/dpath/alu_higher16/DP_OP_61J6_122_3532/U394		AOI21_X1	0.032	0.000	0.497	0.501
v/dpath/alu_higher16/DP_OP_61J6_122_3532/U394	B1 ^ -> ZN v	AOI21_X1	0.015	0.026	0.523	0.527
v/dpath/alu_higher16/DP_OP_61J6_122_3532/U397		OAI21_X1	0.015	0.000	0.523	0.527
v/dpath/alu_higher16/DP_OP_61J6_122_3532/U397	B1 v -> ZN ^	OAI21_X1	0.032	0.044	0.567	0.571
v/dpath/alu_higher16/DP_OP_61J6_122_3532/U416		AOI21_X1	0.032	0.000	0.567	0.571
v/dpath/alu_higher16/DP_OP_61J6_122_3532/U416	B1 ^ -> ZN v	AOI21_X1	0.016	0.027	0.595	0.599
v/dpath/alu_higher16/DP_OP_61J6_122_3532/U419		OAI21_X1	0.016	0.000	0.595	0.599
v/dpath/alu_higher16/DP_OP_61J6_122_3532/U419	B1 v -> ZN ^	OAI21_X1	0.025	0.037	0.631	0.636
v/dpath/alu_higher16/DP_OP_61J6_122_3532/U455		FA_X1	0.025	0.000	0.632	0.636
v/dpath/alu_higher16/DP_OP_61J6_122_3532/U455	CI ^ -> S v	FA_X1	0.015	0.095	0.726	0.730
v/dpath/alu_higher16/U497		AOI21_X1	0.015	0.000	0.726	0.730
v/dpath/alu_higher16/U497	B1 v -> ZN ^	AOI21_X1	0.032	0.039	0.765	0.769
v/dpath/alu_higher16/FE_RC_1032_0		OAI21_X2	0.032	0.000	0.765	0.769
v/dpath/alu_higher16/FE_RC_1032_0	A ^ -> ZN v	OAI21_X2	0.027	0.040	0.805	0.809
v/dpath/pc_sel_mux_F/U84		AOI22_X1	0.027	0.001	0.806	0.810
v/dpath/pc_sel_mux_F/U84	A2 v -> ZN ^	AOI22_X1	0.026	0.046	0.851	0.856
v/dpath/pc_sel_mux_F/U86		NAND3_X1	0.026	0.000	0.852	0.856
v/dpath/pc_sel_mux_F/U86	A2 ^ -> ZN v	NAND3_X1	0.041	0.073	0.925	0.929
v/imemreq_q/genblk1_dpath/genblk1_bypass_mux/U24		MUX2_X1	0.041	0.002	0.926	0.930
v/imemreq_q/genblk1_dpath/genblk1_bypass_mux/U24	B v -> Z v	MUX2_X1	0.020	0.086	1.012	1.016
		proc_ProcRTL_1core	0.020	0.001	1.013	1.017

Figure 18. Gate Level Critical Path for D0/D1 Splitted Processor (X2 in figure corresponds to X1 in report)

4.2.2 Modification Related to the Instruction Memory Drop Unit

By testing our 4.2.1 design, we found a special scenario that induces a bug not considered before. The error occurs in stage F. When a dependency causes a stall in the pipeline (in our case, the stall originates from D1) and a branch instruction that will be taken arrives at stage X1 the next cycle, the instruction memory request that needs to be squashed is not actually sent but the squash signal sets the `imemresp_drop` signal to `1`. The `imem` drop unit incorrectly thinks that

the next instruction memory response is an instruction that needs to be dropped, and thus drops a valid instruction. We fix this by adding a comparison signal in the processor, which stops the `imemresp_drop` from being set to 1.

By pushing the new design through the ASIC flow, we find that the additional logic changes the routing of the original design and hurts the timing significantly. The new critical path, shown in *figure 20* and *figure 21*, originates from op2_reg_X1, through the branching flag ports in the upper-16 ALU and the control logic, and ends at the signal `next_resp_addr` that we used for instruction address comparison. The minimum cycle time is 1.26ns, worse than the previous design. Since in both 4.2.1 and 4.2.2 designs the critical path is related to the PC redirection, we decided to move branch resolution to stage M, eliminating the overhead brought by the ALU unit in the path.

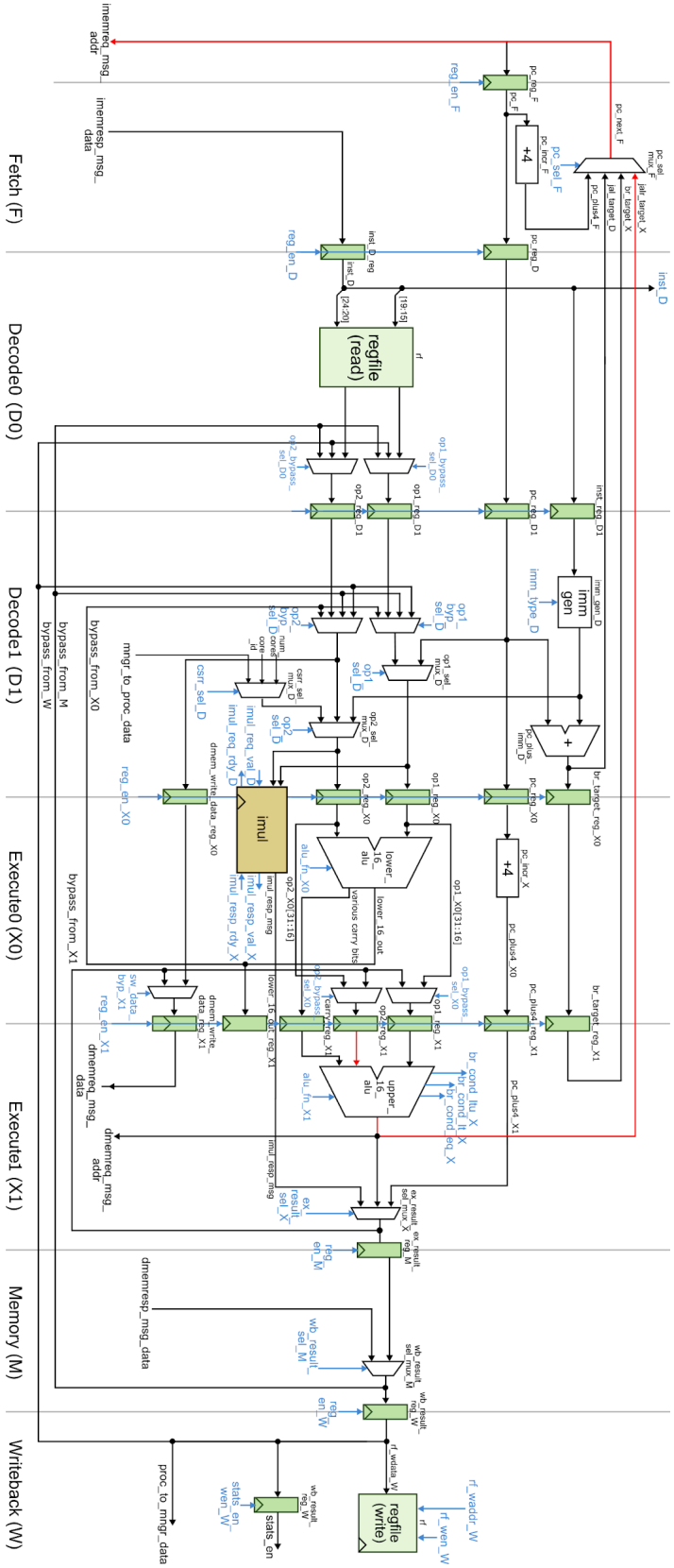


Figure 19. Critical Path for D0/D1 Splitted Processor

proc/v/dpath_op1_reg_X2/clk_gate_q_reg/latch	CK ^ -> GCK ^	CLKGATETST_X4	0.016	0.042	0.053	0.054
proc/v/dpath_op1_reg_X2/q_reg_0_		DFF_X1	0.016	0.001	0.054	0.055
proc/v/dpath_op1_reg_X2/q_reg_0_	CK ^ -> Q ^	DFF_X1	0.016	0.100	0.154	0.155
proc/v/dpath_alu_higher16/FE_OFC403_dpath_op1_X2_0_dup		INV_X2	0.016	0.000	0.154	0.155
proc/v/dpath_alu_higher16/FE_OFC403_dpath_op1_X2_0_dup	A ^ -> ZN v	INV_X2	0.009	0.016	0.170	0.171
proc/v/dpath_alu_higher16/FE_OFC404_dpath_op1_X2_0		INV_X2	0.009	0.001	0.170	0.172
proc/v/dpath_alu_higher16/FE_OFC404_dpath_op1_X2_0	A v -> ZN ^	INV_X2	0.034	0.042	0.213	0.214
proc/v/dpath_alu_higher16/cond_eq_comp/U19		XNOR2_X1	0.034	0.001	0.213	0.215
proc/v/dpath_alu_higher16/cond_eq_comp/U19	B ^ -> ZN ^	XNOR2_X1	0.025	0.051	0.264	0.266
proc/v/dpath_alu_higher16/cond_eq_comp/U20		NAND4_X1	0.025	0.000	0.264	0.266
proc/v/dpath_alu_higher16/cond_eq_comp/U20	A4 ^ -> ZN v	NAND4_X1	0.029	0.049	0.314	0.315
proc/v/dpath_alu_higher16/cond_eq_comp/U21		NOR4_X2	0.029	0.000	0.314	0.316
proc/v/dpath_alu_higher16/cond_eq_comp/U21	A4 v -> ZN ^	NOR4_X2	0.053	0.118	0.432	0.434
proc/v/dpath_alu_higher16/U539		AND2_X2	0.053	0.001	0.433	0.434
proc/v/dpath_alu_higher16/U539	A2 ^ -> ZN ^	AND2_X2	0.013	0.050	0.482	0.484
proc/v/ctrl/U5		NAND3_X1	0.013	0.000	0.483	0.484
proc/v/ctrl/U5	A3 ^ -> ZN v	NAND3_X1	0.011	0.022	0.505	0.506
proc/v/ctrl/U21		OAI21_X1	0.011	0.000	0.505	0.507
proc/v/ctrl/U21	A v -> ZN ^	OAI21_X1	0.021	0.022	0.527	0.529
proc/v/ctrl/FE_RC_587_0		OAI21_X1	0.021	0.000	0.528	0.529
proc/v/ctrl/FE_RC_587_0	B2 ^ -> ZN v	OAI21_X1	0.013	0.023	0.551	0.552
proc/v/ctrl/FE_RC_33_0		NAND3_X2	0.013	0.000	0.551	0.552
proc/v/ctrl/FE_RC_33_0	A1 v -> ZN ^	NAND3_X2	0.019	0.025	0.576	0.578
proc/v/ctrl/U8		NAND2_X2	0.019	0.000	0.577	0.578
proc/v/ctrl/U8	A1 ^ -> ZN v	NAND2_X2	0.013	0.016	0.592	0.594
proc/v/ctrl/U15		NAND2_X2	0.013	0.000	0.593	0.594
proc/v/ctrl/U15	A1 v -> ZN ^	NAND2_X2	0.010	0.017	0.609	0.611
proc/v/ctrl/FE_RC_153_0		OAI21_X2	0.010	0.000	0.609	0.611
proc/v/ctrl/FE_RC_153_0	A ^ -> ZN v	OAI21_X2	0.011	0.020	0.630	0.631
proc/v/dpath_pc_sel_mux_F/U9		OR2_X1	0.011	0.000	0.630	0.631
proc/v/dpath_pc_sel_mux_F/U9	A1 v -> ZN v	OR2_X1	0.011	0.049	0.679	0.681
proc/v/dpath_pc_sel_mux_F/FE_OFC270_n4		INV_X2	0.011	0.000	0.679	0.681
proc/v/dpath_pc_sel_mux_F/FE_OFC270_n4	A v -> ZN ^	INV_X2	0.021	0.030	0.709	0.711
proc/v/dpath_pc_sel_mux_F/FE_OFC379_n84_dup		BUF_X4	0.021	0.001	0.710	0.711
proc/v/dpath_pc_sel_mux_F/FE_OFC379_n84_dup	A ^ -> Z ^	BUF_X4	0.020	0.037	0.747	0.749
proc/v/dpath_pc_sel_mux_F/U33		AOI22_X1	0.020	0.002	0.749	0.750
proc/v/dpath_pc_sel_mux_F/U33	B1 ^ -> ZN v	AOI22_X1	0.016	0.027	0.776	0.777
proc/v/dpath_pc_sel_mux_F/U34		NAND2_X2	0.016	0.000	0.776	0.777
proc/v/dpath_pc_sel_mux_F/U34	A2 v -> ZN ^	NAND2_X2	0.028	0.039	0.815	0.816
proc/v/U144		XOR2_X1	0.028	0.001	0.816	0.817
proc/v/U144	B ^ -> Z ^	XOR2_X1	0.023	0.051	0.867	0.868
proc/v/FE_RC_592_0		NOR2_X1	0.023	0.000	0.867	0.868
proc/v/FE_RC_592_0	A1 ^ -> ZN v	NOR2_X1	0.008	0.010	0.876	0.878
proc/v/FE_RC_591_0		AND2_X1	0.008	0.000	0.876	0.878
proc/v/FE_RC_591_0	A1 v -> ZN v	AND2_X1	0.006	0.028	0.904	0.906
proc/v/U154		NAND4_X1	0.006	0.000	0.905	0.906
proc/v/U154	A3 v -> ZN ^	NAND4_X1	0.015	0.021	0.926	0.928
proc/v/U168		NOR2_X1	0.015	0.000	0.926	0.928
proc/v/U168	A1 ^ -> ZN v	NOR2_X1	0.006	0.010	0.936	0.937
proc/v/U181		NOR2_X1	0.006	0.000	0.936	0.937
proc/v/U181	A1 v -> ZN ^	NOR2_X1	0.019	0.027	0.963	0.964
proc/v/clk_gate_next_resp_addr_reg/latch		CLKGATETST_X4	0.019	0.000	0.963	0.964

Figure 20. Gate Level Critical Path for Processor with Modified `imem` Drop Unit (X2 in figure corresponds to X1 in report)

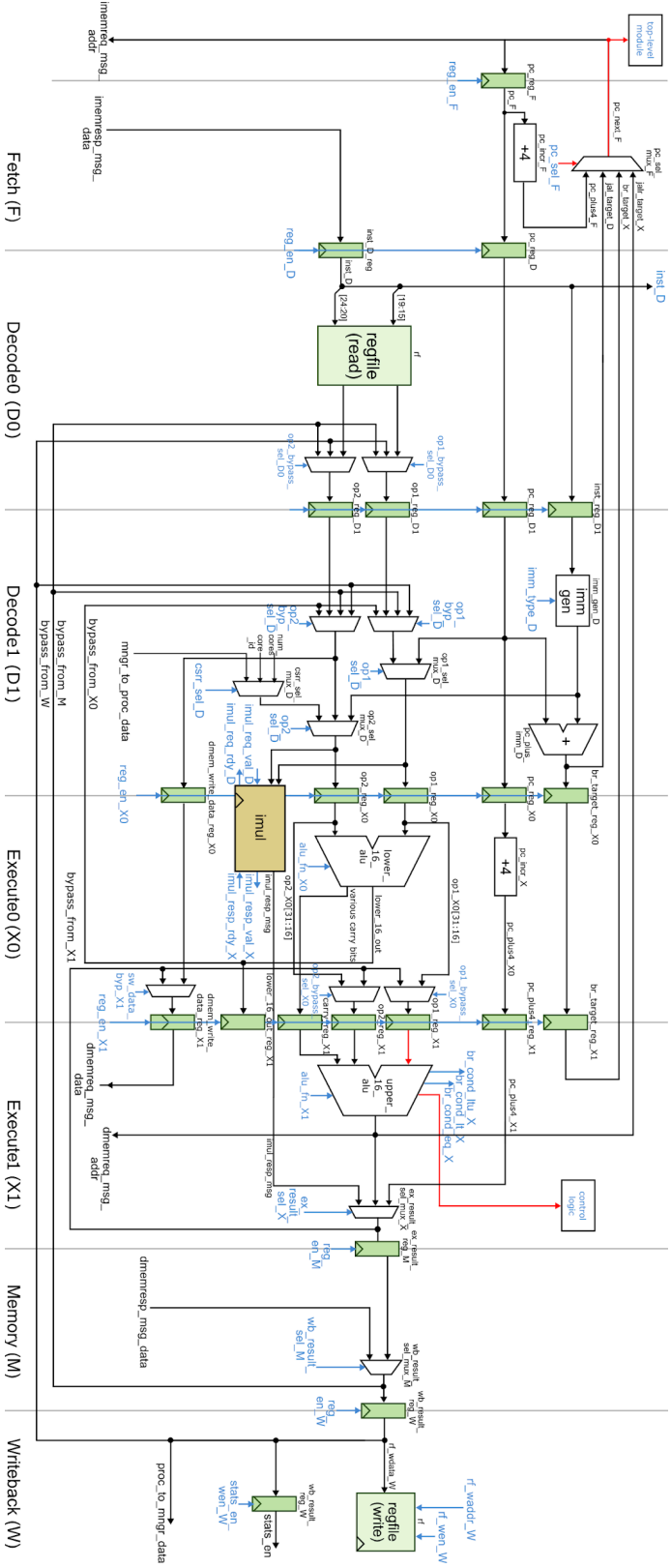


Figure 21. Critical Path for Processor with Modified imem Drop Unit

4.2.3 Potential Latch-based Design

Since the critical path through the register file is no longer an issue, we decided to temporarily suspend the latch-based approach and focus on the new critical path. If this path resurfaces in the future, we will revisit the latch-based approach.

4.3 Dealing with PC Redirection

As indicated in *figure 19*, the critical path is caused by the branch target from X1 stage to F stage for pc redirection. In *figure 21*, the critical path is also caused by the flags generated by the ALU. To reduce the critical path length, we plan to move the pc redirection path from X1 stage to M stage, saving the time for the signal to pass through the ALU unit. This will bring one extra cycle to every branch or jump instruction that is taken. Most of the changes happen in the control logic, and some additional registers are needed to pass the PC target to the M stage. Besides, as data memory requests are sent in the X1 stage, we need to drop the data sent back from the memory if we are squashing the previous `lw` instruction. However, we need to stall `sw` instructions following a branching instruction, just so the incorrect memory request is never sent.

As the modification brings a higher CPI to branch and jump instructions, we start to look for methods to reduce the CPI count. After examining the datapath diagram we realize that we can move the JAL resolution from D1 stage to D0 stage, saving one cycle for function calls in the microbenchmarks.

Furthermore, the more stages we have before branch resolution, the higher branch misprediction penalty is. After splitting X stage and D stage and moving branch prediction to M stages, there will be five mis-fetched instructions for each mispredicted branch instruction. Thus improving branch prediction accuracy will significantly reduce CPI of the program and a simple four-entry branch target buffer is a reasonable choice.

4.3.1 Branch Resolution in Stage M

4.3.1.a Datapath

To move branch resolution to M stage, we need to keep the data for branch target and jump-and-link-register target until M stage. Thus two pipeline registers are added between X1 and M stage: one for the branch target calculated in D1 stage and the other for the ALU computation result used for jump-and-link-register target. The two target data are fed back into the PC select mux which is used to generate the instruction memory request address. As needed by the control logic, the flags output by ALU is also passed into M stage, adding two more pipeline registers to M stage.

One other important change is related to data memory requests. If there is a `lw` instruction right after a PC redirection instruction (branch or jump) and the wrong result is predicted, by the time we resolve the misprediction in M stage, a memory read request has already been set in X1 stage and the processor is supposed to be waiting for a memory response data at data memory port. Thus we added a data memory drop unit, similar to the instruction memory drop unit used for dropping the next instruction in flight when squashing, to drop the data response.

4.3.1.b Control Logic

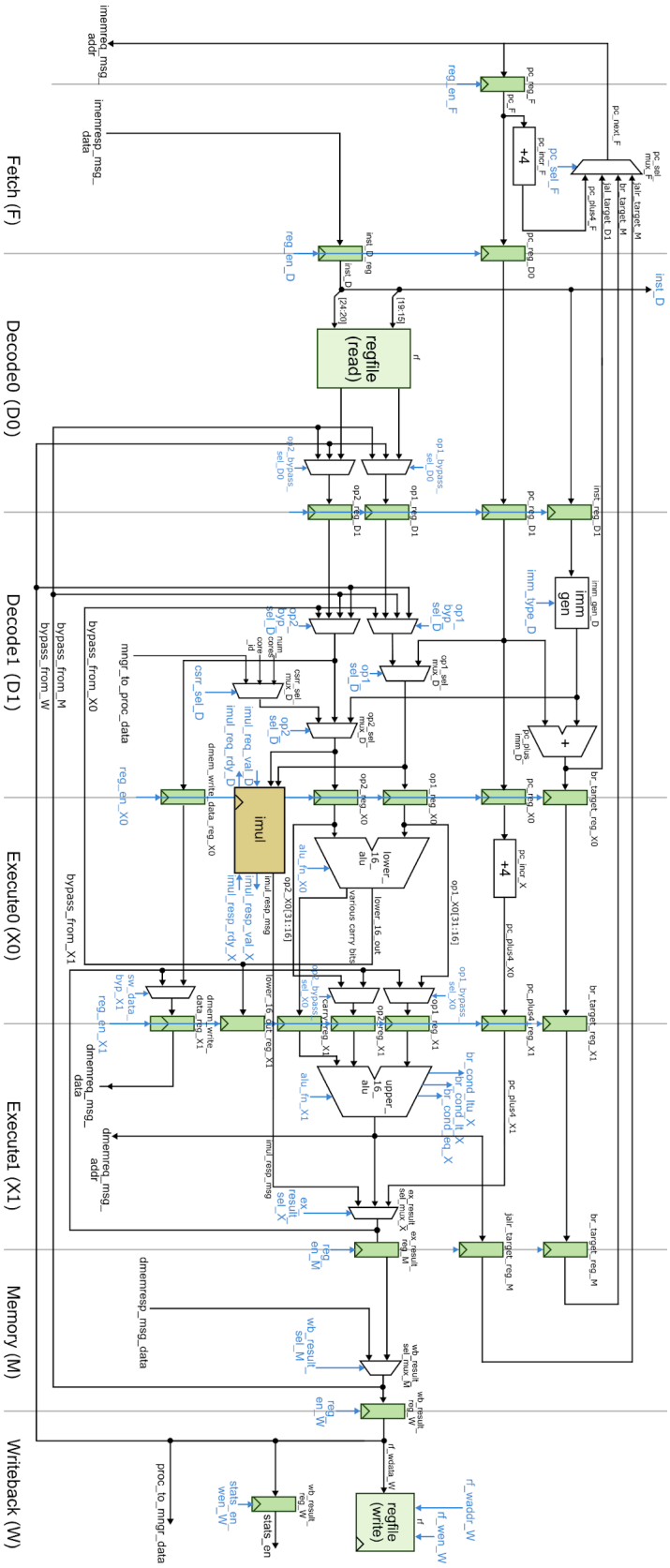


Figure 22. Datapath for M-stage PC Redirect Processor

As we move the entire pc redirection to M stage, the control signal for branching and `JALR` is also moved to M stage as `PC_redirect_M`. The instruction branching type is determined by the decoding table in D0 stage, and is passed all the way through the pipeline into M stage to decide which ALU flag should be used as the branch condition. A `dmemresp_drop` signal is implemented as an input to the `dmem` drop unit. If the instruction in X1 is a valid memory instruction and is being squashed, the response of the instruction will be dropped.

4.3.1.c Critical Path

The critical path of the design, indicated in *figure 23* and *figure 24*, has a length of 1.27ns. The critical path goes through the accelerator, which is simply a placeholder, the writeback mux in M, back to the op2 bypass mux and op2 select mux in D1, and ends at the pipeline register between D1 and X0. As we can see in *figure x*, the accelerator itself creates a delay of about 0.4 ns. Since this is merely a placeholder, everything is combinational and we get an unintended critical path. This is likely due to the routing algorithm moving gates around due to the different logic and ended up increasing the length between these gates, resulting in a worse cycle time.

Instance	Arc	Cell	Slew	Delay	Arrival Time	
	clk[0] ^		0.023		-0.140	
CTS_ccl_a_BUF_ideal_clock_G0_L1_1		CLKBUF_X3	0.023	0.001	-0.139	
CTS_ccl_a_BUF_ideal_clock_G0_L1_1	A ^ -> Z ^	CLKBUF_X3	0.029	0.059	-0.079	
CTS_ccl_a_BUF_ideal_clock_G0_L2_4		CLKBUF_X2	0.029	0.003	-0.076	
CTS_ccl_a_BUF_ideal_clock_G0_L2_4	A ^ -> Z ^	CLKBUF_X2	0.007	0.032	-0.044	
FE_USKC824_CTS_3		CLKBUF_X2	0.007	0.000	-0.044	
FE_USKC824_CTS_3	A ^ -> Z ^	CLKBUF_X2	0.036	0.058	0.014	
CTS_ccl_a_BUF_ideal_clock_G0_L3_1		CLKBUF_X3	0.036	0.001	0.015	
CTS_ccl_a_BUF_ideal_clock_G0_L3_1	A ^ -> Z ^	CLKBUF_X3	0.028	0.064	0.078	
xcel/xcelreq_q/ctrl/head_reg_0_		SDFF_X1	0.028	0.001	0.079	
xcel/xcelreq_q/ctrl/head_reg_0_	CK ^ -> QN ^	SDFF_X1	0.104	0.192	0.271	
xcel/xcelreq_q/dpath/queue/U37		MUX2_X1	0.104	0.001	0.272	
xcel/xcelreq_q/dpath/queue/U37	S ^ -> Z ^	MUX2_X1	0.017	0.058	0.331	
xcel/U3		INV_X1	0.017	0.000	0.331	
xcel/U3	A ^ -> ZN v	INV_X1	0.021	0.033	0.364	
xcel/FE_OFC231_n6		CLKBUF_X1	0.021	0.001	0.365	
xcel/FE_OFC231_n6	A v -> Z v	CLKBUF_X1	0.054	0.093	0.457	
xcel/U30		AND2_X1	0.054	0.001	0.459	
xcel/U30	A2 v -> ZN v	AND2_X1	0.008	0.054	0.512	
proc/v/xcelresp_q/genblkl_dpath/genblkl_bypass_mux /U25		MUX2_X1	0.008	0.000	0.512	
proc/v/xcelresp_q/genblkl_dpath/genblkl_bypass_mux /U25	B v -> Z v	MUX2_X1	0.012	0.058	0.571	
proc/v/dpath_wb_result_sel_mux_M/U25		AOI222_X1	0.012	0.000	0.571	
proc/v/dpath_wb_result_sel_mux_M/U25	A2 v -> ZN ^	AOI222_X1	0.097	0.147	0.718	
proc/v/dpath_wb_result_sel_mux_M/FE_OFC255_n11		CLKBUF_X1	0.097	0.000	0.718	
proc/v/dpath_wb_result_sel_mux_M/FE_OFC255_n11	A ^ -> Z ^	CLKBUF_X1	0.014	0.057	0.775	
proc/v/dpath_wb_result_sel_mux_M/U26		INV_X1	0.014	0.000	0.775	
proc/v/dpath_wb_result_sel_mux_M/U26	A ^ -> ZN v	INV_X1	0.016	0.025	0.800	
proc/v/dpath_op2_byp_mux_D2/U31		AOI22_X1	0.016	0.001	0.801	
proc/v/dpath_op2_byp_mux_D2/U31	A2 v -> ZN ^	AOI22_X1	0.025	0.039	0.840	
proc/v/dpath_op2_byp_mux_D2/U32		NAND2_X1	0.025	0.000	0.840	
proc/v/dpath_op2_byp_mux_D2/U32	A2 ^ -> ZN v	NAND2_X1	0.014	0.024	0.864	
proc/v/dpath_op2_sel_mux_D2/U25		AOI222_X1	0.014	0.000	0.864	
proc/v/dpath_op2_sel_mux_D2/U25	C2 v -> ZN ^	AOI222_X1	0.078	0.167	1.031	
proc/v/dpath_op2_sel_mux_D2/U26		INV_X1	0.078	0.000	1.031	
proc/v/dpath_op2_sel_mux_D2/U26	A ^ -> ZN v	INV_X1	0.019	0.018	1.049	
proc/v/dpath_op2_reg_X1/U27		AND2_X1	0.019	0.000	1.049	
proc/v/dpath_op2_reg_X1/U27	A1 v -> ZN v	AND2_X1	0.008	0.034	1.083	
proc/v/dpath_op2_reg_X1/q_reg_22_		DFF_X1	0.008	0.000	1.083	

Figure 23. Gate Level Critical Path for M-stage PC Redirect Processor

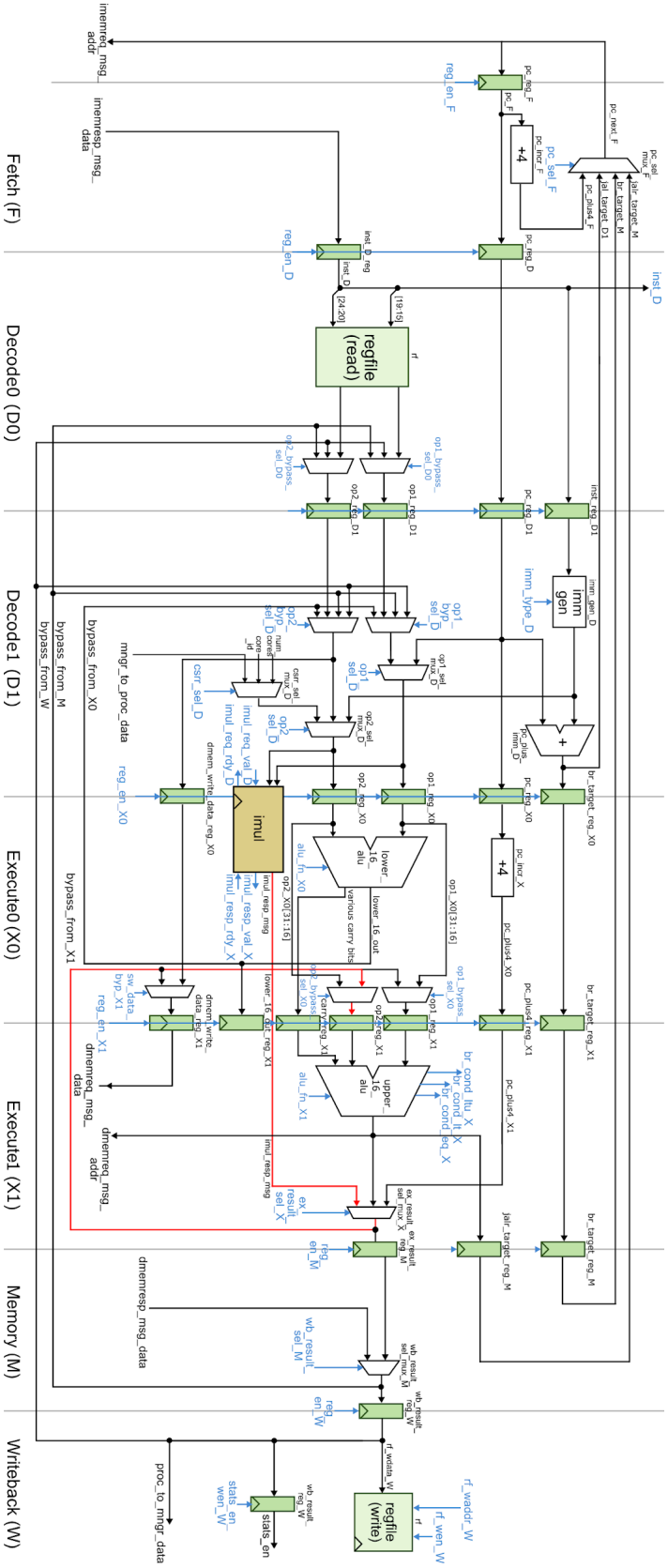


Figure 24. Critical Path for M-stage PC Redirect Processor

4.3.2 JAL in D0 Stage

4.3.2.a Datapath

To resolve `JAL` in the D0 stage, the immediate generator and the `PC_plus_imm` adder are both moved to the D0 stage. The immediate generated is passed through a pipeline register to be used as a potential input operand for ALU in D1 stage.

4.3.2.b Control Logic

D0 stage originates a squash if the instruction decoded is `JAL` and `PC_redirect_D0` is set to 1. The PC select mux then chooses the correct PC to fetch and the processor starts another assembly sequence.

4.3.2.c Critical Path

By pushing the design through the ASIC flow, we get a minimum cycle time of 1.26ns, and the critical path is indicated below in *figure 26* and *figure 27*. The critical path penetrates through the multiplier and ends at the pipeline register between X1 stage and M stage. As illustrated in section 4.1, we purposely send the multiplication request at X0 stage and receive multiplication response at X1 stage, giving the multiplier two cycles to finish its task. However, we believe that the tool only gives the multiplier one cycle and thus makes it to be the longest path in the design. The plan is to modify the ASIC flow and enable register retiming feature for the multiplier so that the tool can auto-balance the stage by us simply adding a register before the output port of the multiplier unit. As this requires us to modify mflowgen, a lightweight modular flow specification and build-system generator for ASIC and FPGA design-space exploration built by Christopher Torng, we are going to look more in depth into the flow.

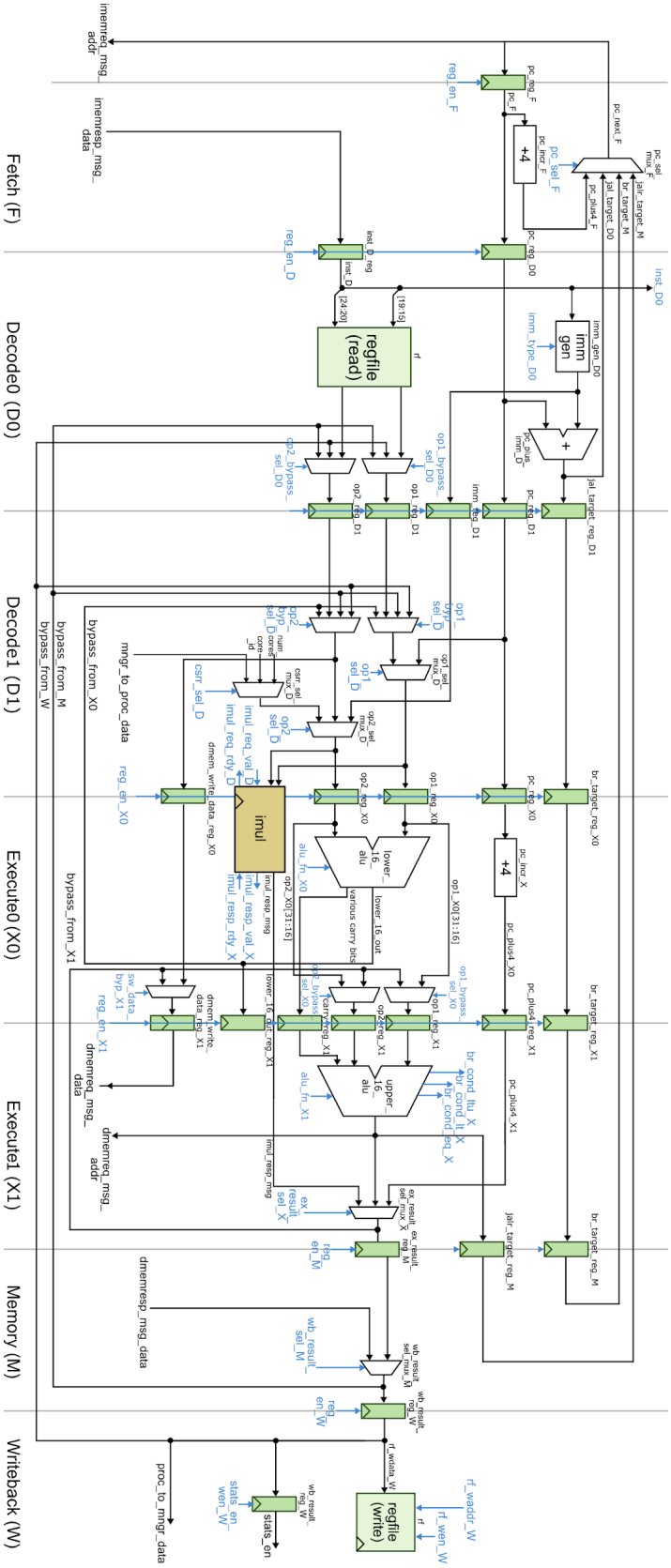
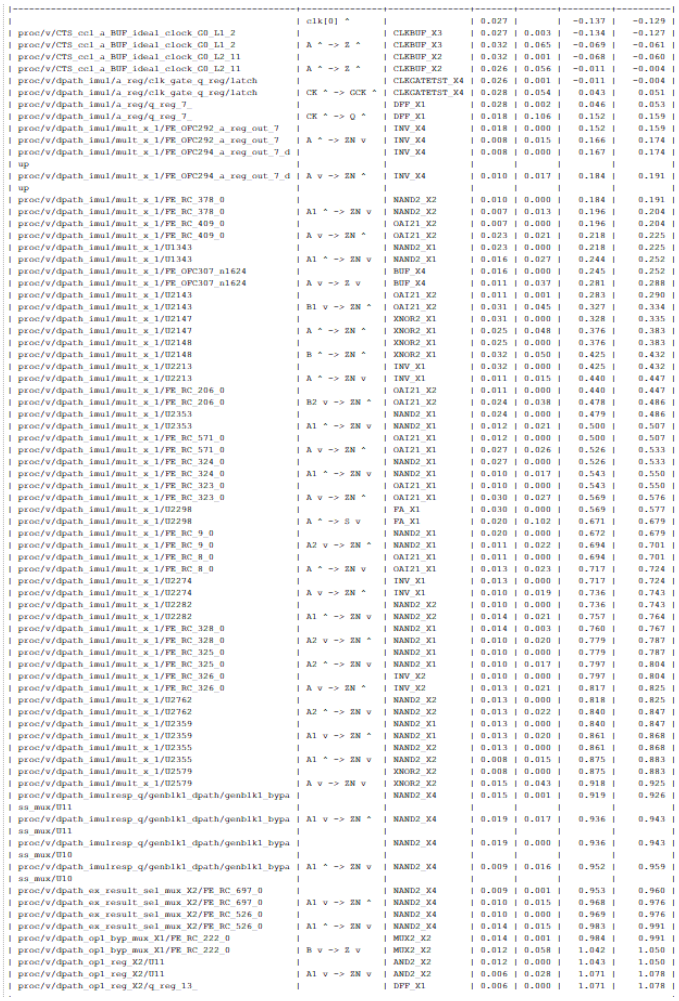


Figure 25. Datapath for 7-stage Processor with `JAL` in D0



	clk[0] ^		0.027		-0.137	-0.129
proc/v/CTS_cc1_a_BUF_ideal_clock_G0_L1_2		CLKBUF_X3	0.027	0.003	-0.134	-0.127
proc/v/CTS_cc1_a_BUF_ideal_clock_G0_L1_2	A ^ -> Z ^	CLKBUF_X3	0.032	0.065	-0.069	-0.061
proc/v/CTS_cc1_a_BUF_ideal_clock_G0_L2_11		CLKBUF_X2	0.032	0.001	-0.068	-0.060
proc/v/CTS_cc1_a_BUF_ideal_clock_G0_L2_11	A ^ -> Z ^	CLKBUF_X2	0.026	0.056	-0.011	-0.004
proc/v/dpath_imul/a_reg/clk_gate_q_reg/latch		CLKGATETST_X4	0.026	0.001	-0.011	-0.004
proc/v/dpath_imul/a_reg/clk_gate_q_reg/latch	CK ^ -> GCK ^	CLKGATETST_X4	0.028	0.054	0.043	0.051
proc/v/dpath_imul/a_reg/q_reg_7_		DFF_X1	0.028	0.002	0.046	0.053
proc/v/dpath_imul/a_reg/q_reg_7_	CK ^ -> Q ^	DFF_X1	0.018	0.106	0.152	0.159
proc/v/dpath_imul/mult_x_1/FE_OFC292_a_reg_out_7		INV_X4	0.018	0.000	0.152	0.159
proc/v/dpath_imul/mult_x_1/FE_OFC292_a_reg_out_7	A ^ -> ZN v	INV_X4	0.008	0.015	0.166	0.174
proc/v/dpath_imul/mult_x_1/FE_OFC294_a_reg_out_7_d		INV_X4	0.008	0.000	0.167	0.174
up						
proc/v/dpath_imul/mult_x_1/FE_OFC294_a_reg_out_7_d	A v -> ZN ^	INV_X4	0.010	0.017	0.184	0.191
up						
proc/v/dpath_imul/mult_x_1/FE_RC_378_0		NAND2_X2	0.010	0.000	0.184	0.191
proc/v/dpath_imul/mult_x_1/FE_RC_378_0	A1 ^ -> ZN v	NAND2_X2	0.007	0.013	0.196	0.204
proc/v/dpath_imul/mult_x_1/FE_RC_409_0		OAI21_X2	0.007	0.000	0.196	0.204
proc/v/dpath_imul/mult_x_1/FE_RC_409_0	A v -> ZN ^	OAI21_X2	0.023	0.021	0.218	0.225
proc/v/dpath_imul/mult_x_1/U1343		NAND2_X1	0.023	0.000	0.218	0.225
proc/v/dpath_imul/mult_x_1/U1343	A1 ^ -> ZN v	NAND2_X1	0.016	0.027	0.244	0.252
proc/v/dpath_imul/mult_x_1/FE_OFC307_n1624		BUF_X4	0.016	0.000	0.245	0.252
proc/v/dpath_imul/mult_x_1/FE_OFC307_n1624	A v -> Z v	BUF_X4	0.011	0.037	0.281	0.288
proc/v/dpath_imul/mult_x_1/U2143		OAI21_X2	0.011	0.001	0.283	0.290
proc/v/dpath_imul/mult_x_1/U2143	B1 v -> ZN ^	OAI21_X2	0.031	0.045	0.327	0.334
proc/v/dpath_imul/mult_x_1/U2147		XNOR2_X1	0.031	0.000	0.328	0.335
proc/v/dpath_imul/mult_x_1/U2147	A ^ -> ZN ^	XNOR2_X1	0.025	0.048	0.376	0.383
proc/v/dpath_imul/mult_x_1/U2148		XNOR2_X1	0.025	0.000	0.376	0.383
proc/v/dpath_imul/mult_x_1/U2148	B ^ -> ZN ^	XNOR2_X1	0.032	0.050	0.425	0.432
proc/v/dpath_imul/mult_x_1/U2213		INV_X1	0.032	0.000	0.425	0.432
proc/v/dpath_imul/mult_x_1/U2213	A ^ -> ZN v	INV_X1	0.011	0.015	0.440	0.447
proc/v/dpath_imul/mult_x_1/FE_RC_206_0		OAI21_X2	0.011	0.000	0.440	0.447
proc/v/dpath_imul/mult_x_1/FE_RC_206_0	B2 v -> ZN ^	OAI21_X2	0.024	0.038	0.478	0.486
proc/v/dpath_imul/mult_x_1/U2353		NAND2_X1	0.024	0.000	0.479	0.486
proc/v/dpath_imul/mult_x_1/U2353	A1 ^ -> ZN v	NAND2_X1	0.012	0.021	0.500	0.507
proc/v/dpath_imul/mult_x_1/FE_RC_571_0		OAI21_X1	0.012	0.000	0.500	0.507
proc/v/dpath_imul/mult_x_1/FE_RC_571_0	A v -> ZN ^	OAI21_X1	0.027	0.026	0.526	0.533
proc/v/dpath_imul/mult_x_1/FE_RC_324_0		NAND2_X1	0.027	0.000	0.526	0.533
proc/v/dpath_imul/mult_x_1/FE_RC_324_0	A1 ^ -> ZN v	NAND2_X1	0.010	0.017	0.543	0.550
proc/v/dpath_imul/mult_x_1/FE_RC_323_0		OAI21_X1	0.010	0.000	0.543	0.550
proc/v/dpath_imul/mult_x_1/FE_RC_323_0	A v -> ZN ^	OAI21_X1	0.030	0.027	0.569	0.576
proc/v/dpath_imul/mult_x_1/U2298		FA_X1	0.030	0.000	0.569	0.577
proc/v/dpath_imul/mult_x_1/U2298	A ^ -> S v	FA_X1	0.020	0.102	0.671	0.679
proc/v/dpath_imul/mult_x_1/FE_RC_9_0		NAND2_X1	0.020	0.000	0.672	0.679
proc/v/dpath_imul/mult_x_1/FE_RC_9_0	A2 v -> ZN ^	NAND2_X1	0.011	0.022	0.694	0.701
proc/v/dpath_imul/mult_x_1/FE_RC_8_0		OAI21_X1	0.011	0.000	0.694	0.701
proc/v/dpath_imul/mult_x_1/FE_RC_8_0	A ^ -> ZN v	OAI21_X1	0.013	0.023	0.717	0.724
proc/v/dpath_imul/mult_x_1/U2274		INV_X1	0.013	0.000	0.717	0.724
proc/v/dpath_imul/mult_x_1/U2274	A v -> ZN ^	INV_X1	0.010	0.019	0.736	0.743
proc/v/dpath_imul/mult_x_1/U2282		NAND2_X2	0.010	0.000	0.736	0.743
proc/v/dpath_imul/mult_x_1/U2282	A1 ^ -> ZN v	NAND2_X2	0.014	0.021	0.757	0.764
proc/v/dpath_imul/mult_x_1/FE_RC_328_0		NAND2_X1	0.014	0.003	0.760	0.767
proc/v/dpath_imul/mult_x_1/FE_RC_328_0	A2 v -> ZN ^	NAND2_X1	0.010	0.020	0.779	0.787
proc/v/dpath_imul/mult_x_1/FE_RC_325_0		NAND2_X1	0.010	0.000	0.779	0.787
proc/v/dpath_imul/mult_x_1/FE_RC_325_0	A2 ^ -> ZN v	NAND2_X1	0.010	0.017	0.797	0.804
proc/v/dpath_imul/mult_x_1/FE_RC_326_0		INV_X2	0.010	0.000	0.797	0.804
proc/v/dpath_imul/mult_x_1/FE_RC_326_0	A v -> ZN ^	INV_X2	0.013	0.021	0.817	0.825
proc/v/dpath_imul/mult_x_1/U2762		NAND2_X2	0.013	0.000	0.818	0.825
proc/v/dpath_imul/mult_x_1/U2762	A2 ^ -> ZN v	NAND2_X2	0.013	0.022	0.840	0.847
proc/v/dpath_imul/mult_x_1/U2359		NAND2_X1	0.013	0.000	0.840	0.847
proc/v/dpath_imul/mult_x_1/U2359	A1 v -> ZN ^	NAND2_X1	0.013	0.020	0.861	0.868
proc/v/dpath_imul/mult_x_1/U2355		NAND2_X2	0.013	0.000	0.861	0.868
proc/v/dpath_imul/mult_x_1/U2355	A1 ^ -> ZN v	NAND2_X2	0.008	0.015	0.875	0.883
proc/v/dpath_imul/mult_x_1/U2579		XNOR2_X2	0.008	0.000	0.875	0.883
proc/v/dpath_imul/mult_x_1/U2579	A v -> ZN v	XNOR2_X2	0.015	0.043	0.918	0.925
proc/v/dpath_imulresp_q/genblk1_dpath/genblk1_bypa		NAND2_X4	0.015	0.001	0.919	0.926
ss_mux/U11						
proc/v/dpath_imulresp_q/genblk1_dpath/genblk1_bypa	A1 v -> ZN ^	NAND2_X4	0.019	0.017	0.936	0.943
ss_mux/U11						
proc/v/dpath_imulresp_q/genblk1_dpath/genblk1_bypa		NAND2_X4	0.019	0.000	0.936	0.943
ss_mux/U10						
proc/v/dpath_imulresp_q/genblk1_dpath/genblk1_bypa	A1 ^ -> ZN v	NAND2_X4	0.009	0.016	0.952	0.959
ss_mux/U10						
proc/v/dpath_ex_result_sel_mux_X2/FE_RC_697_0		NAND2_X4	0.009	0.001	0.953	0.960
proc/v/dpath_ex_result_sel_mux_X2/FE_RC_697_0	A1 v -> ZN ^	NAND2_X4	0.010	0.015	0.968	0.976
proc/v/dpath_ex_result_sel_mux_X2/FE_RC_526_0		NAND2_X4	0.010	0.000	0.969	0.976
proc/v/dpath_ex_result_sel_mux_X2/FE_RC_526_0	A1 ^ -> ZN v	NAND2_X4	0.014	0.015	0.983	0.991
proc/v/dpath_op1_byp_mux_X1/FE_RC_222_0		MUX2_X2	0.014	0.001	0.984	0.991
proc/v/dpath_op1_byp_mux_X1/FE_RC_222_0	B v -> Z v	MUX2_X2	0.012	0.058	1.042	1.050
proc/v/dpath_op1_reg_X2/U11		AND2_X2	0.012	0.000	1.043	1.050
proc/v/dpath_op1_reg_X2/U11	A1 v -> ZN v	AND2_X2	0.006	0.028	1.071	1.078
proc/v/dpath_op1_reg_X2/q_reg_13_		DFF_X1	0.006	0.000	1.071	1.078

Figure 26. Gate Level Critical Path for 7-stage Processor with `JAL` in D0

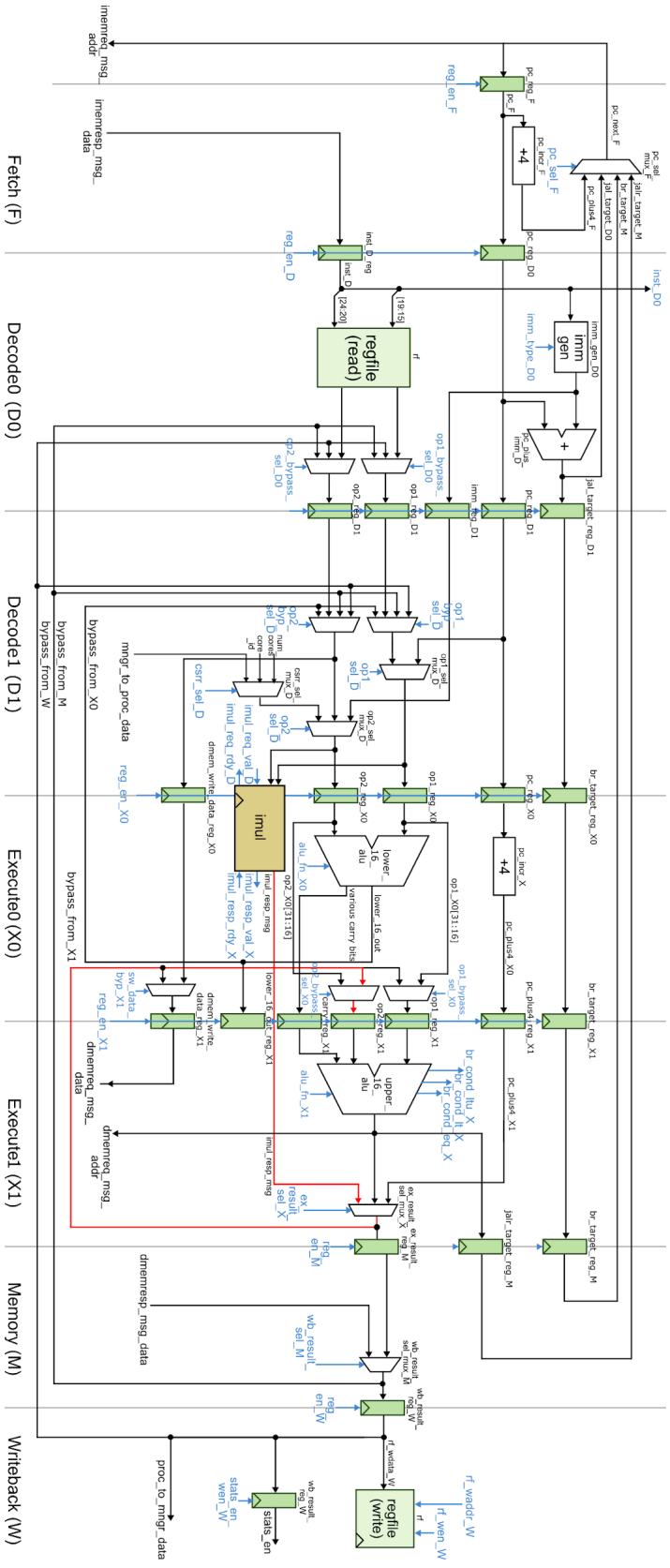


Figure 27. Critical Path for 7-stage Processor with `JAL` in D0

4.3.3 Branch Target Buffer

To recover some of the CPIs we lost by moving branch resolution to stage M, we intended to implement a branch target buffer (BTB) with four entries. Each entry consists of a 32-bit address, 1-bit valid flag and a 2-bit age field. In stage D0, our processor decodes the branch target and checks it against the four entries in the BTB. If there is a match, it squashes the instructions already fetched and redirects to the new PC. That prediction is propagated to stage M, where the actual resolution happens. If the prediction does not match the actual resolution, it has to squash again and redirect to the actual PC. Then, depending on the specific scenario, it would update the entries, maybe replacing the least recently used entry with age 3.

A typical microbenchmark program such as vector-vector add has the assembly sequence shown in *figure 28*. The original design makes no branch prediction and simply fetches the next PC. For vector-vector add, the loop repeats for 100 iterations, and that means 5 instructions squashed for every iteration. With our BTB, the processor predicts the correct PC 98 out of those 100 times, and saves close to 400 cycles, more than a third of the total cycles. This BTB correctly predicts all `for` loops except for the first and the last iteration.

```
00000248 <vvadd_scalar(int*, int*, int*, int)>:
 248:   bge     x0,x13,274 <vvadd_scalar(int*, int*, int*, int)+0x2c>
 24c:   slli    x13,x13,0x2
 250:   add     x13,x11,x13
 254:   lw      x15,0(x11)
 258:   lw      x14,0(x12)
 25c:   addi    x11,x11,4
 260:   addi    x12,x12,4
 264:   add     x15,x15,x14
 268:   sw      x15,0(x10)
 26c:   addi    x10,x10,4
 270:   bne     x11,x13,254 <vvadd_scalar(int*, int*, int*, int)+0xc>
 274:   jalr    x0,x1,0
```

Figure 28. Assembly Snippet Compiled from vvadd.c

Unfortunately, we did not have enough time to actually implement this short but effective improvement.

5. Testing Strategy

5.1 Baseline Testing

The testing for the baseline design involves test-driven development methodology and directed testing on individual data sets associated with controlled input data and expected outputs. To test functionality, we have unit tests for the ALU as well as each instruction under one of the six categories: register-register, register-immediate, branch, jump, memory, and csr. There are a total of 685 tests written that all passed at the time of submission. With unit tests available, we could take an incremental approach to develop the processor. That is, after connecting the necessary datapath and control/status signals for one instruction, we could run the tests for that specific instruction. Upon failure, we would check our outputted line traces and waveforms to confirm proper pipelined execution and expected register/wire values. We faced many problems and corner cases when implementing the alternative design. The layout of the unit tests allows us to single in on error cases and pinpoint the exact location of the bug.

There are numerous test cases, albeit repetitive, since FL, BaseRTL, and AltRTL run essentially the same tests. As mentioned above, the instructions fall under one of the six major categories. Register-register instructions are those that take two register values as operands and store the result in a register, such as `add`, `mul`, `xor`, or `sra`. Register-immediate instructions are those that take a register value and an immediate value as operands. Branch instructions are ones that either jump to a label or not based on the condition specific to the instruction. For instructions under this category, the helper functions from *inst_utils.py* such as *gen_rr_dest_dep_test()* are used. The helper functions are basically wrapper functions that generate an assembly program in string using the parameters provided. By using the wrapper functions, we could generate many tests of the same category easily and neatly.

For each instruction, there are about 40 directed test cases, a random test, and a random delay test. A lot of the directed test cases are corner cases that could potentially fail. For example, how well does `slt`, the instruction that compares two signed numbers, handle 2's complement. Can it compare 0x7fffffff and 0x80000000 correctly? By writing a lot of tests like this, we found a few bugs that could go unnoticed. Random value tests were conducted to ensure robustness of testing unachievable by manual number generation. Tests with random delay were also important as, for this design, the memory is not combinational and the imul takes more than one cycle to complete calculation. Therefore, there are a few different reasons, including data hazards, that could create stalls, which means our processor needs to be properly tested. In the end, we created just under 700 tests, and they all pass on our processor.

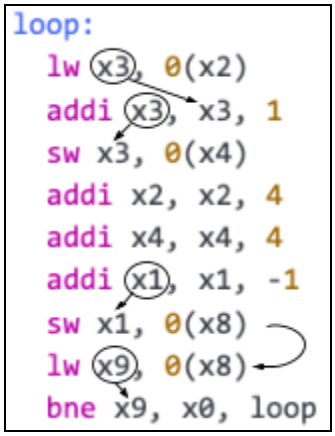


Figure 29. Load-use Dependency Code Snippet

5.2 Alternative Design Testing

5.2.1 Testing Methodology

We reused all of the tests from the baseline design for the alternative design. Since the ISA remains unchanged, the expected behavior of every test is the same. We also added additional mixed-instruction tests that try to expose the structural hazards specific to our superpipelined processor. For example, to shorten the critical path through the ALU, we would break it into two stages, each handling half of the arithmetic. In stage X0, the ALU calculates the lower 16 bits of the operands, while the rest is calculated in stage X1. We must properly handle the stall and bypass logic between the two stages when there are data dependencies to avoid hazards shown in *table 1*. Therefore, back-to-back dependent additions are created to ensure the correctness, as shown in *figure 30*. Similarly, we would test additional stalling and bypassing logic in stages such as M0 and M1, as shown in *figure 31*.

add x1, x1, x1	V	W	**X0**	**X1**	Y	Z		
add x1, x1, x1		V	W	**X0**	**X0**	**X1**	Y	Z
add x1, x1, x1			V	W	**X0**	**X0**	**X0**	**X1**

Table 1. Pipeline Diagram for X0-X1 Hazard

Figure 30. Add-after-Add Dependency Code Snippet

Figure 31. Load-after-load Dependency Code Snippet

5.2.2 Testing Separated ALUs

As stated in section 4.1, we splitted the ALU to two components: lower-16-bit ALU and upper-16-bit ALU each handles the computation of 16 bits. To make sure our pipelined ALU works, we designed separate test cases. For both of them, we created unit tests for each arithmetic instruction of ALU: `add`, `sub`, `sll`, `or`, `and`, `nor`, `xor`, `srl`, `sra`. Besides the operational instructions, ALU also generates flags indicating the relationship between two inputs and even behave like a mux so comparison instructions: `slt`, `sltu`, `eq` and multiplexing instructions `cp0`, `cp1` tests are designed accordingly.

For lower-16-bit ALU, apart from the expected outputs from provided inputs, we also need to test the correctness of the carry out bit for `add`, `sub`, `slt` and `sltu`. When adding,

there might be an overflow. When subtracting, if operand 0 is smaller than operand 1, there will be a borrowed 1 from higher 16 bits. `slt` and `sltu` have carry-out outputs to higher 16 bits because it is possible that the higher 16 bits of two inputs are the same and we need the results from lower 16 bits. In `sll`, we need to pass extra shifted-out bits to higher-16-bit ALU so we have tests for those outputs as well.

For higher 16-bit ALU, apart from similar tests associated with inputs and expected outputs, we need to take the results, carry out and shift out bits from lower 16 bits as inputs. For each instruction, we designed tests with various results from the lower 16-bit ALU. In `add`, `sub`, `slt` and `sltu`, since we have carry out as an input, we have tests that set it as 0 or 1. In `sll`, we generate different shifted out bits from lower 16-bit ALU to test if higher 16-bit ALU calculates the output correctly. For all of the shift operations (`sll`, `srl`, `sra`), we also have test cases for both positive and negative numbers so that we make sure we handle arithmetic and logical shifts correctly. By looking into the failing negative `sra` test cases, we realized that we cannot *or* data with `signed` representation and `unsigned` representation together. So instead we `unsigned` the `signed` shifted upper 16 bits before *or* it together with the output from the lower-16-bit ALU.

Since ALU also generates flags to control logic, we have corresponding test cases with `ops_lt`, `ops_ltu`, `ops_eq` designed to test these three flags. Higher 16-bit ALU will take carried flags from lower 16-bit ALU and generate final flags based on carried flags and higher 16 bits; therefore, we include tests with different carried flags for `slt`, `sltu` and `eq`.

After the testing individual components, we also created a wrapper module to directly connect them together and tested the fully functioning ALU with the original ALU operation tests. This integrated test guarantees that our ALU modules perform the operations exactly as we expected from the design specification.

5.2.3 Testing the X0/X1 Splitted Processor Without Additional Bypassing

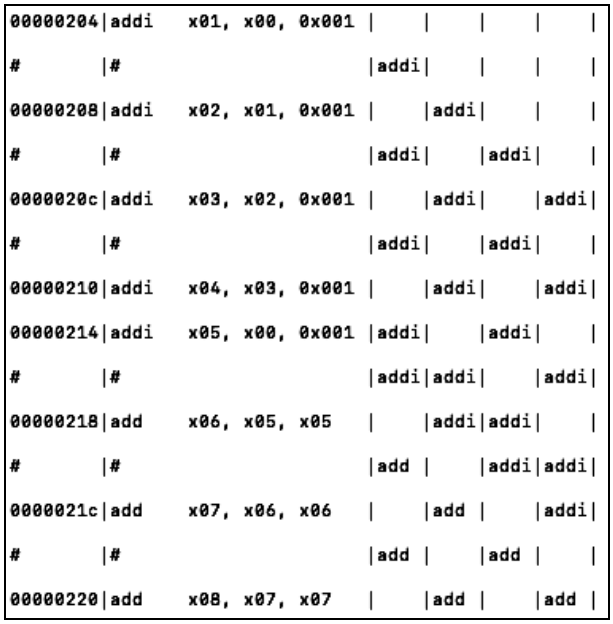
After fully testing the splitted ALU, we put it into the processor with a splitted X stage. To ensure the functionality of the processor, we used the same testing as we have for the baseline processor. As the testing fully represents the design specification we have for the processor, the fact that our 6-stage processor with a splitted ALU passes those tests means our design meets the requirement. Besides the direct, random, and delay test we have for each of the 14 instructions, we also created dependency tests specifically for the ALU. As the extra execution stage creates an additional read-after-write hazard between X0 and X1, we used back-to-back addition sequences and load word sequences to examine if it performs as we expected. As indicated in *figure 32* and *figure 33*, there are bubbles after each RAW add instruction, which matches with our stall signal implemented in the control logic. With the accelerator tests which use `CSRR` and `CSRW` to write data between register file and control register, we fixed a bug in our stall logic. As long as there is a `CSRRX` instruction in-flight in the X0/X1 stage, we need to stall in the D stage.

Otherwise the instruction dependent on the `CSRRX` instruction will receive wrong data since the value will not be returned by the accelerator until the M stage.

By examining the line tracing for independent additions (*figure 32*), back-to-back dependent additions (*figure 33*), and load-use hazard assembly microbenchmarks (*figure 34*), it is clear how our 6-stage processor functions: it streams the addition if there is no dependency; for instructions with back-to-back dependency, there is one cycle of bubble between each instruction; for load word instruction sequence, each instruction comes with two bubbles for waiting memory response from M stage. As we not only want to improve the cycle time but also aim to improve the throughput of the entire processor, it is necessary for us to implement a 6-stage processor with bypassing from X0 to D and from X1 to X0 so that there is only load-use hazard being stalled by memory access.

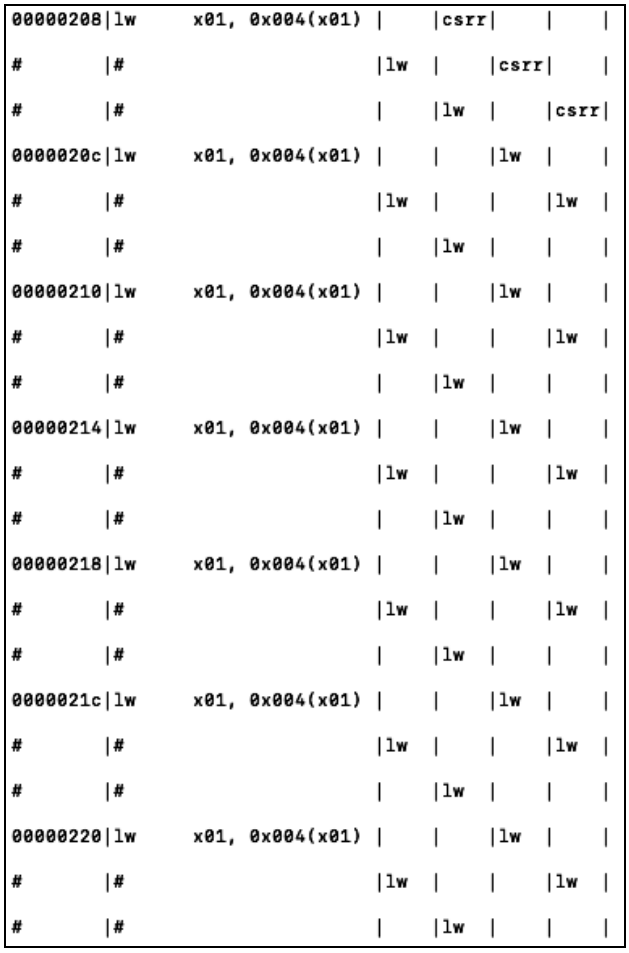
```
00000200|                       |    |    |    |    |
00000204|addi   x01, x00, 0x001 |    |    |    |    |
#       |#                      |addi|    |    |    |
00000208|addi   x02, x01, 0x001 |    |addi|    |    |
0000020c|addi   x03, x01, 0x001 |addi|    |addi|    |
00000210|addi   x04, x01, 0x001 |addi|addi|    |addi|
00000214|addi   x05, x00, 0x001 |addi|addi|addi|    |
#       |#                      |addi|addi|addi|addi|
00000218|add    x06, x05, x05   |    |addi|addi|addi|
0000021c|add    x07, x05, x05   |add |    |addi|addi|
00000220|add    x08, x05, x05   |add |add |    |addi|
00000224|csrw       0x7c0, x04  |add |add |add |    |
00000228|csrw       0x7c0, x08  |csrw|add |add |add |
```

Figure 32. Independent ALU Instruction Line Tracing



```
00000204|addi   x01, x00, 0x001 |    |    |    |    |
#       |#                     |addi|    |    |    |
00000208|addi   x02, x01, 0x001 |    |addi|    |    |
#       |#                     |addi|    |addi|    |
0000020c|addi   x03, x02, 0x001 |    |addi|    |addi|
#       |#                     |addi|    |addi|    |
00000210|addi   x04, x03, 0x001 |    |addi|    |addi|
00000214|addi   x05, x00, 0x001 |addi|    |addi|    |
#       |#                     |addi|addi|    |addi|
00000218|add    x06, x05, x05   |    |addi|addi|    |
#       |#                     |add |    |addi|addi|
0000021c|add    x07, x06, x06   |    |add |    |addi|
#       |#                     |add |    |add |    |
00000220|add    x08, x07, x07   |    |add |    |add |
```

Figure 33. Back-to-back RAW ALU Dependency Line Tracing (Stalling)



```
00000208|lw     x01, 0x004(x01) |    |csrr|    |    |
#       |#                     |lw  |    |csrr|    |
#       |#                     |    |lw  |    |csrr|
0000020c|lw     x01, 0x004(x01) |    |    |lw  |    |
#       |#                     |lw  |    |    |lw  |
#       |#                     |    |lw  |    |    |
00000210|lw     x01, 0x004(x01) |    |    |lw  |    |
#       |#                     |lw  |    |    |lw  |
#       |#                     |    |lw  |    |    |
00000214|lw     x01, 0x004(x01) |    |    |lw  |    |
#       |#                     |lw  |    |    |lw  |
#       |#                     |    |lw  |    |    |
00000218|lw     x01, 0x004(x01) |    |    |lw  |    |
#       |#                     |lw  |    |    |lw  |
#       |#                     |    |lw  |    |    |
0000021c|lw     x01, 0x004(x01) |    |    |lw  |    |
#       |#                     |lw  |    |    |lw  |
#       |#                     |    |lw  |    |    |
00000220|lw     x01, 0x004(x01) |    |    |lw  |    |
#       |#                     |lw  |    |    |lw  |
#       |#                     |    |lw  |    |    |
```

Figure 34. Load-use Dependency Line Tracing

5.2.4 Testing the Bypassing X0/X1 Splitted Processor

As we are taking an incremental approach to implement the superpipelined processor, we can incorporate our old unit and integrated tests with the new dependency tests to prove the functionality and the additional feature of the 6-stage bypassing processor.

One important testing method we used in the line tracing of the pymtl testbench. Line tracing allows us to visualize the pipeline diagram of the test, making it easier to debug the design. As the major difference between a 6-stage bypassing processor and a 6-stage stalling processor is whether there are bubbles between two instruction with dependency, we add dependency tests on different instructions and use the line tracing to show that data is successfully bypassed from execution stages and there are less bubbles in the line tracing.

```
00000200|                          |    |    |    |    |
00000204|addi   x01, x00, 0x001    |    |    |    |    |
00000208|addi   x02, x01, 0x001    |addi|    |    |    |
0000020c|addi   x03, x02, 0x001    |addi|addi|    |    |
00000210|addi   x04, x03, 0x001    |addi|addi|addi|    |
00000214|addi   x05, x00, 0x001    |addi|addi|addi|addi|
00000218|add    x06, x05, x05      |addi|addi|addi|addi|
0000021c|add    x07, x06, x06      |add |addi|addi|addi|
00000220|add    x08, x07, x07      |add |add |addi|addi|
00000224|csrw       0x7c0, x04     |add |add |add |addi|
00000228|csrw       0x7c0, x08     |csrw|add |add |add |
```

Figure 35. Back-to-back RAW Add Dependency Line Tracing (Bypassing)

Comparing the line tracing we get from the X0/X1 splitted processor with stalling execution stage in *figure 33*, line tracing in *figure 35* shows that we get rid of the bubble between RAW `add` instructions by bypassing data from X0 to D and from X1 to X0.

```
00000260|sra    x03, x01, 0x02   |addi|sub |addi|csrw|
#       |#                       |sra |addi|sub |addi|
00000264|sra    x04, x03, 0x02   |    |sra |addi|sub |
#       |#                       |sra |    |sra |addi|
00000268|sra    x05, x04, 0x02   |    |sra |    |sra |
0000026c|srl    x06, x01, 0x02   |sra |    |sra |    |
#       |#                       |srl |sra |    |sra |
00000270|srl    x07, x06, 0x02   |    |srl |sra |    |
#       |#                       |srl |    |srl |sra |
00000274|srl    x08, x07, 0x02   |    |srl |    |srl |
```

Figure 36. Back-to-back Right Shift Bubble Line Tracing (Bypassing)

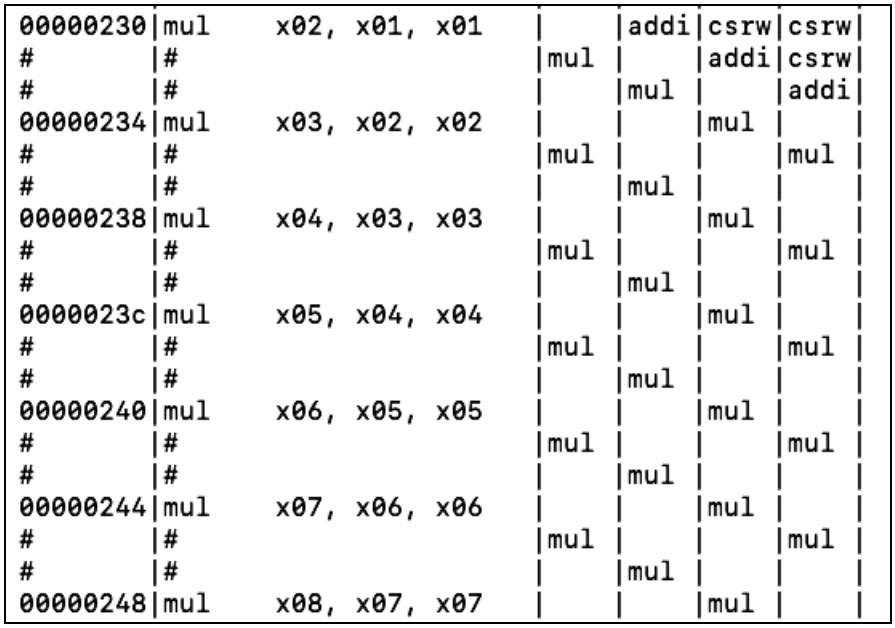


Figure 37. Back-to-back RAW Mul Bubble Line Tracing (Bypassing)

As stated in 4.1.2.b, the dependency between X0/X1 computation results in bubbles between back-to-back multiplication, right shift, and comparison. The line tracing in *figure 36* shows a back-to-back right shifting assembly sequence. For right shift instructions without RAW hazard, there are no bubbles between the instruction; otherwise, the shift instruction stalls in D stage for one cycle to wait to its operand coming back at the end of X1 stage. In *figure 37* there are two bubbles between each pair of multiplication instructions brought by the two cycle latency multiplier and RAW dependency.

After the pipelined processor design passes all of the directed, random, and delay tests, we run to further test our alternative design. These benchmarks provide non-trivial and realistic sequences of instructions, so passing this verification is a good sanity check that our processor is working as expected. In total there are 6 microbenchmarks being used for realistic testing and their detailed information are listed in *table 2*. In `ummark-cmult` we find a bug related to RAW dependency between ALU instruction and `sw` instruction (shown in *figure 38*). Instruction `sub x15,x14,x15` stores data in register `x15` and `sw x15,0xff8(x10)` stores data in `x15` to the memory. Thus for `sw` instruction we need data to be bypassed from register file to memory request data port. As we only implemented bypassing to instruction operand at that time, we passed computation result of the ALU into the operand register between X0 and X1, incorrectly modified the upper 16 bits of the memory address we are going to write to and caused a byte array access out of index error. To solve the problem, we added a bypass path from X1 to `dmem_req_data` port in X0 and fixed the control logic accordingly so that data can be bypassed correctly encountering a RAW dependency between ALU instructions and `sw` instruction.

ubmark-sort	Quicksort algorithm
ubmark-accum	Integer accumulation
ubmark-bsearch	Binary search in a linear array of key/value pairs
ubmark-vvadd	Element-wise vector-vector add
ubmark-mflit	Masked convolution on a small image
ubmark-cmult	Element-wise complex multiplication

Table 2. Six Microbenchmarks for Testing

```
264:   mul     x14,x14,x17
268:   addi    x10,x10,8
26c:   addi    x11,x11,8
270:   addi    x12,x12,8
274:   mul     x15,x16,x15
278:   sub     x15,x14,x15
27c:   sw      x15,-8(x10)
```

Figure 38. RAW Dependency for sw in ubmark-cmult

By running `ubmark-mflit`, we fixed the bug that existed in our multiplication response enable logic. The `bne x15,x12,39c` always predict taken, meaning the next instruction fetched after it will be `mul x12,x12,x12` at PC 388. If the program takes a branch after a multiplication instruction, all instructions in-flight in the pipeline are squashed. As we did not enable the multiplication response enable signal for such a scenario, the `mul` instruction result will stay in the multiplier, making it unable to receive the next request. If a multiplication comes after the squash, it will stall in D stage and the program will never continue running. Thus we enable the processor to receive multiplication response when there is a `mul` instruction in X1 stage and X1 is going to be squashed. The multiplier is then cleared and ready for the next `mul` instruction.

```
3b8:   addi    x15,x15,1
3bc:   addi    x10,x10,4
3c0:   addi    x11,x11,4
3c4:   bne     x15,x12,39c <verify_results(unsigned int*, unsigned int*, int)+0x14>
```

```
00000388 <verify_results(unsigned int*, unsigned int*, int)>:
 388:   mul     x12,x12,x12
 38c:   beq     x12,x0,3c8 <verify_results(unsigned int*, unsigned int*, int)+0x40>
 390:   lui     x16,0x20
```

Figure 39. Multiplication after Squash in Ubmark-mfilt

5.2.5 Testing the D0/D1 Splitted 7-Stage Processor

As we splitted the D stage into D0/D1 and each stage handles different bypassing scenarios, we add more dependency tests to make sure our processor design can handle all kinds of hazards. In the `gen_stall_test`, we have multiple sequences of `add` and `addi` instructions each targeting at dependency between different stages. If there is no `nop` between instructions, our bypassing between X0 and D1 and between X1 and X2 successfully handles the hazard. If there is one `nop`, as shown in *figure 40*, the `addi` instruction in D1 stage will stall for one cycle until the `addi` in X1 reaches M stage. This is due to the fact that we remove the bypassing path from X1 to D stage and instead solve the hazard by bypassing from M to D1. If there are two `nops` between the additions, there is a RAW dependency between X1 stage and D0 stage, shown in *figure 41*. Instruction in D0 will stall for a cycle until data can be bypassed from M to D0. These are the two regular RAW hazards brought by splitting D stage and removing bypassing from X1 to D that we need to stall. When there are 4 `nops` between two dependent instructions, we realize that we cannot wait for the second instruction to reach D1 because the first instruction writes back one cycle before and the second instruction has already read the wrong data in the register file. Thus we add a bypassing path from W to D0 stage to account for the RAW instructions with four instructions in between.

```
F:0000022c|D0:addi   x01, x00, 0x001 |D1:csrw|X0:    |X1:csrw|M:add |W:add |
>
F:00000230|D0:nop                     |D1:addi|X0:csrw|X1:    |M:csrw|W:add |
>
F:00000234|D0:addi   x02, x01, 0x001 |D1:nop |X0:addi|X1:csrw|M:    |W:csrw|
> (00000004)
F:#       |D0:#                      |D1:#   |X0:nop |X1:addi|M:csrw|W:    |
>
F:00000238|D0:nop                     |D1:addi|X0:    |X1:nop |M:addi|W:csrw|
> (00000008)
F:0000023c|D0:addi   x03, x02, 0x001 |D1:nop |X0:addi|X1:    |M:nop |W:addi|
>
F:#       |D0:#                      |D1:#   |X0:nop |X1:addi|M:    |W:nop |
>
F:00000240|D0:nop                     |D1:addi|X0:    |X1:nop |M:addi|W:    |
>
F:00000244|D0:addi   x04, x03, 0x001 |D1:nop |X0:addi|X1:    |M:nop |W:addi|
>
F:#       |D0:#                      |D1:#   |X0:nop |X1:addi|M:    |W:nop |
>
F:00000248|D0:nop                     |D1:addi|X0:    |X1:nop |M:addi|W:    |
```

Figure 40. X1/D1 RAW Dependency

```
F:00000270|D0:addi   x01, x00, 0x001 |D1:csrw|X0:    |X1:csrw|M:add |W:    |
>
F:00000274|D0:nop                     |D1:addi|X0:csrw|X1:    |M:csrw|W:add |
>
F:00000278|D0:nop                     |D1:nop |X0:addi|X1:csrw|M:    |W:csrw|
> (00000004)
F:#       |D0:#                      |D1:nop |X0:nop |X1:addi|M:csrw|W:    |
>
F:0000027c|D0:addi   x02, x01, 0x001 |D1:    |X0:nop |X1:nop |M:addi|W:csrw|
> (00000008)
F:00000280|D0:nop                     |D1:addi|X0:    |X1:nop |M:nop |W:addi|
>
F:00000284|D0:nop                     |D1:nop |X0:addi|X1:    |M:nop |W:nop |
>
F:#       |D0:#                      |D1:nop |X0:nop |X1:addi|M:    |W:nop |
>
F:00000288|D0:addi   x03, x02, 0x001 |D1:    |X0:nop |X1:nop |M:addi|W:    |
>
F:0000028c|D0:nop                     |D1:addi|X0:    |X1:nop |M:nop |W:addi|
>
F:00000290|D0:nop                     |D1:nop |X0:addi|X1:    |M:nop |W:nop |
>
F:#       |D0:#                      |D1:nop |X0:nop |X1:addi|M:    |W:nop |
```

Figure 41. X1/D0 RAW Dependency

By running the microbenchmark tests on our design, we find an irregular dependency caused by instruction stalling due to other hazards. In `ubmark_cmult`, shown in *figure 42* and

figure 43, as the multiplication in `I5` is not ready to respond, `I6` is forced to stall at cycle 10. Since we already have a bypassing path from W to D0, the data dependency hazard between `I3` and `I6` is resolved at cycle 9. `I4` returns `x4` value at cycle 8 and writes back at cycle 9. However, `I6` is only valid for data bypassing at cycle 11. At cycle 11 `I4` has already written back to the register file, which causes I6 to read wrong data for `x4`. Thus we implement a bypassing path from M to D0 so that at cycle 9 both `I3` and `I4` can both bypass the valid data to D0.

```
280:    lw      x14,-8(x11)
284:    lw      x17,-4(x12)
288:    lw      x16,-4(x11)
28c:    lw      x15,-8(x12)
290:    mul     x14,x14,x17
294:    mul     x15,x16,x15
```

Figure 42. Dependency in `ubmark_cmult`

		1	2	3	4	5	6	7	8	9	10	11	
I1	lw x1	F	D0	D1	X0	X1	M	W					
I2	lw x2		F	D0	D1	X0	X1	M	W				
I3	lw x3			F	D0	D1	X0	X1	M	***W***			
I4	lw x4				F	D0	D1	X0	X1	***M***	***W***		
I5	mul x1, x1, x2					F	D0	D0	D1	X0	X1	M	W
I6	mul x3, x3, x4						F	F	D0	**D0**	D1	**D1**	X0

Figure 43. Pipeline Diagram for the ubmark_cmult Dependency (Instruction Simplified)

5.2.6 Testing the D0/D1 Splitted 7-Stage Processor with modified Unit Signal

When testing our design in section 4.2.1, we noticed that some of the branching tests were failing by exceeding the maximum cycle number. After examining the waveform, we found the specific scenario that triggers the bug: stage F being stalled first and then squashed immediately after. This results in stage F waiting for an instruction dropped by the drop unit indefinitely. We resolved this bug by modifying the `imem_resp_drop` signal as described in section 4.2.2.

5.2.7 Testing 7-Stage Pipeline Processor With M Stage Branch Resolution

Even though moving the PC redirection unit from X1 stage to M stage is not a complicated change, we still test it thoroughly to make sure there is no unexpected dependency. We carefully look into the line tracing of `ProcRTL_branch_test` to make sure that every falsely predicted branching is resolved at M stage. By examining the microbenchmark test cycle count, we find a matching between the number of for loops inside the benchmark program and the increase of cycle count between 4.3.1 design and 4.2.2 design (result shown in *table 3*). This

proves that our design has one extra cycle of latency for every branch that is not correctly predicted.

	4.2.2	4.3.1	for loop count
ubmark-sort	20863	22170	*
ubmark-accum	1014	1114	100
ubmark-bsearch	4325	4610	*
ubmark-vvadd	1313	1413	100
ubmark-mflit	8236	8560	200
ubmark-cmult	3213	3313	100

Table 3. Benchmark Cycle Count for 4.3.1 and 4.2.2

When we test our design with `gen_load_after_branch_test`, we realize a bug caused by squashing a `lw` right after a branch instruction. As the branch is resolved in M stage, lw has already sent out a data memory request at X1. The data memory response writes back after the PC points to the valid instruction, corrupting the register file. Thus we add a dmem drop unit which has exactly the same functionality as the imem drop unit: dropping an invalid data memory response if the `lw` instruction is squashed. By carefully examining all dependency conditions, we realize that it is not possible to stall a `lw` instruction at X1 stage. Thus we don't need to add more comparison signals for the dmem drop unit as there will not exist a `lw` instruction at X1 stage but did not send any data.

5.2.8 Testing 7-Stage Pipeline Processor With JAL at D0

Similar to the testing method we have in 5.2.7, the 4.3.2 design only involves moving `JAL` related components from D1 to D0. JALR, however, is still resolved at M stage since it needs the computation result from the ALU. By examining the line tracing for the `ProcRTL_jump_test` (shown in *figure 44*), it is clear that our design meets the specification and requirement as stage F is squashed when the instruction `jal` is at D0.

```
F:00000228|D0:addi   x03, x03, 0x040 |D1:addi|X0:    |X1:    |M:jal |W:addi|
     >
F:~       |D0:jal    x05, 0x1fffe8   |D1:addi|X0:addi|X1:    |M:    |W:jal |
```

```
F:00000238|D0:jalr   x31, x01, 0x000 |D1:nop |X0:nop |X1:nop |M:nop |W:nop |
     >
F:0000023c|D0:addi   x03, x03, 0x001 |D1:jalr|X0:nop |X1:nop |M:nop |W:nop |
     >
F:00000240|D0:nop                    |D1:addi|X0:jalr|X1:nop |M:nop |W:nop |
     >
F:00000244|D0:nop                    |D1:nop |X0:addi|X1:jalr|M:nop |W:nop |
     >
F:~       |D0:~                      |D1:~   |X0:~   |X1:~   |M:jalr|W:nop |
     >
F:00000264|D0:                       |D1:    |X0:    |X1:    |M:    |W:jalr|
     >
F:00000268|D0:addi   x03, x03, 0x002 |D1:    |X0:    |X1:    |M:    |W:    |
     >
```

Figure 44. `JAL` and `JALR` Line Tracing

2.

6. Evaluation

6.1 Cycle Time

	Baseline	X0/X1 with Stalling (4.1.1)	X0/X1 with Bypassing (4.1.2)	D0/D1 (4.2.1)	D0/D1 (4.2.2)	Branch Resolution in W (4.3.1)	JAL in D0 (4.3.2)
Cycle time/ns	1.2	1.23	1.22	1.13	1.23	1.27	1.26

Table 4. Minimum Cycle Time

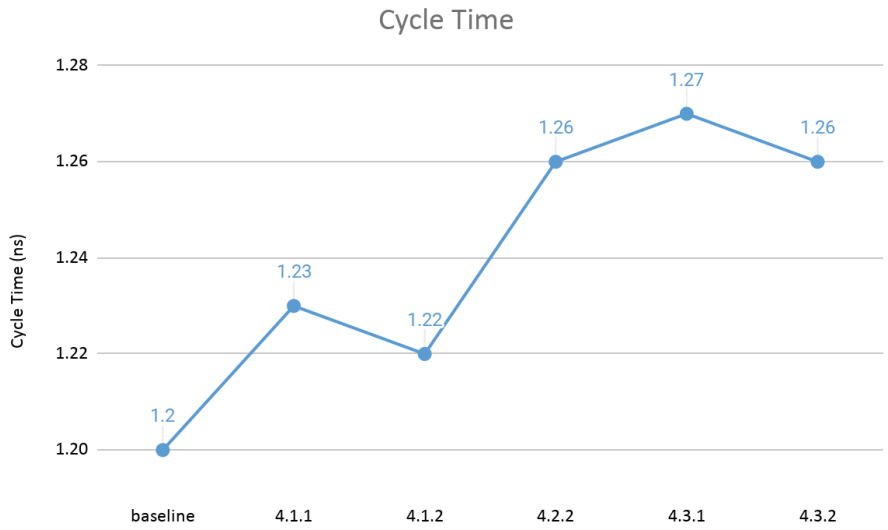


Figure 45. Cycle Time

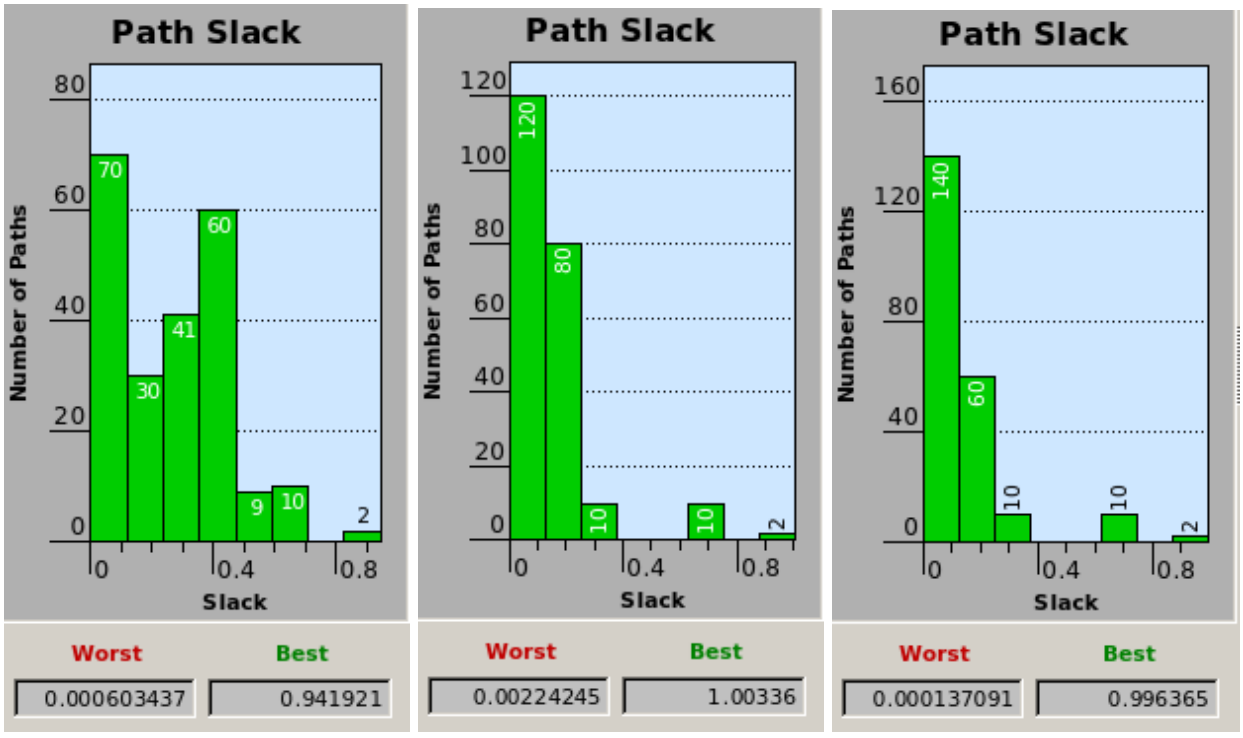
Looking at the cycle times (*table 4* and *figure 45*), we noticed an initial increase in 4.1.1 after we divide the X stage into X0 and X1 without bypassing. We took a look at our critical path and found out that it is not the critical path of our original baseline design and after comparing the components and routing, we realized that the placement of these components are influenced due to our additional logic for dividing X stage.

Compared with 4.1.1, our 4.1.2 design has a relatively smaller cycle time. As we add additional control logic and bypassing muxes, the ASIC flow tool tries to optimize the placement of standard cells and somehow influences routing of the processor. This helps to reduce the critical path of our bypassing design.

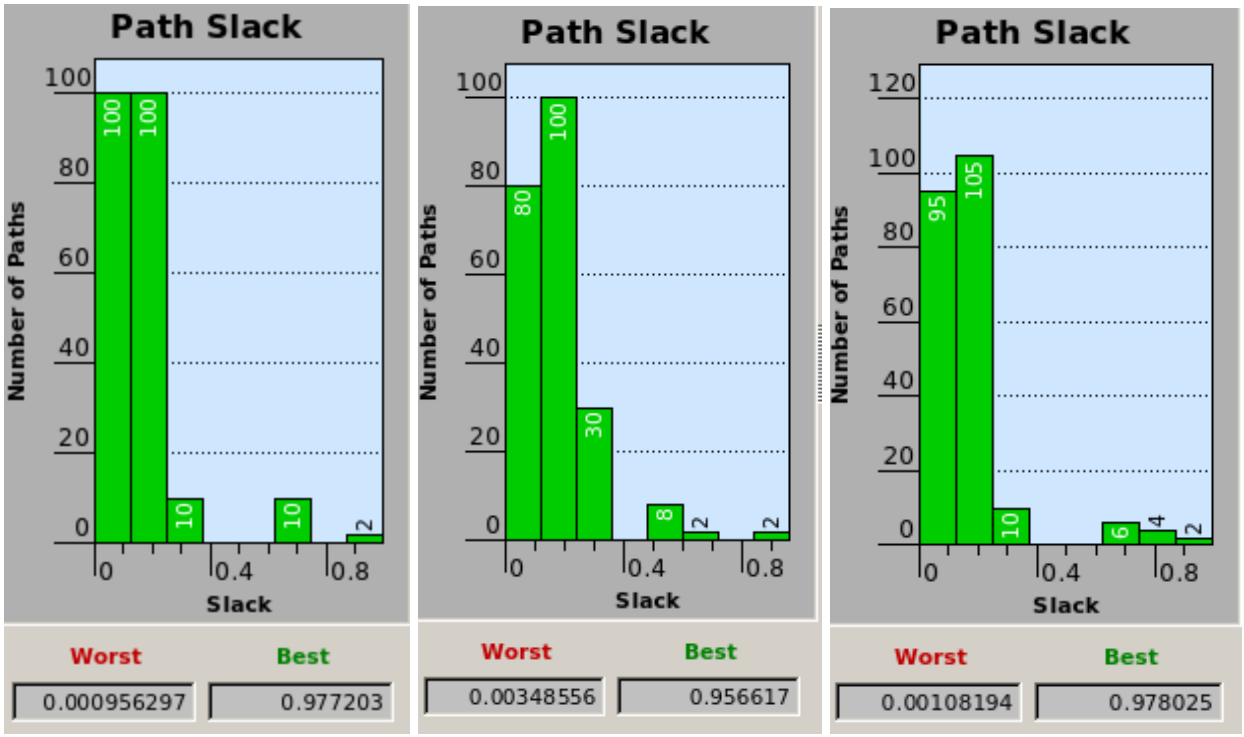
From 4.1.2 to 4.2.1, we first time achieve a cycle time smaller than the original baseline design. This is due to the fact that we divide the critical path in 4.2.1 (in D stage through register file) into two stages, significantly reducing the path length. This matches with our expectation for the incremental implementation approach: gradually dividing critical paths to eliminate the

longest path in the design. However, as we correct our design by adding control signals to the imem drop unit, the cycle time significantly increases from 1.13ns to 1.23ns for design 4.2.2. Taking into account that the critical path shifts from datapath to control logic, we believe the cause is the following: the additional control logic we add in design 4.2.2 completely changes the how Cadence Innovus, the place and route tool, place the components and route the control logic. The increased wiring distance between important components causes the change in critical path.

In 4.3.1 design, we focus on moving the PC redirection unit from X1 stage to M stage, hoping we can save the 0.3ns overhead caused by the ALU unit. However, the cycle time significantly increases again from 1.23ns to 1.27ns, even through the critical path changes. By moving the `JAL` resolution from D1 stage to D0 stage in 4.3.2 design we also decrease the critical path length and change the critical path position. This change is supposed to be a functional improvement of the CPI, but such a minor change also causes change in cycle time. Here we can conclude that place-and-route is very sensitive to the design and any minor change can cause reroute of the entire design; besides, a change from 1.27ns to 1.26ns can cause the critical path position to change, making us suspect that the tool might put in effort to evenly distribute path length of the design. This helps the tool to minimize the longest path in the design and meet the timing constraint; but at the same time this makes it harder for us to see an improvement in the cycle time of our design, since there might be multiple paths of similar length.



Baseline *4.1.1* *4.1.2*



4.2.2 *4.3.1* *4.3.2*

Figure 46. Path Slack Histograms
(top to bottom, left to right: baseline, 4.1.1, 4.1.2, 4.2.2, 4.3.1, 4.3.2)

By examining the path slack histogram (shown in *figure 46*) generated by Cadence Innovus for each of the design we have, we further prove our assumption about how place-and-route works: the tool tries to average datapath length for every single path we have in out design so that it can achieve the minimum cycle time. As we keep improving our design and implementation, more and more paths reach the timing constraint we set when we push the design through the flow. This is one of the major reason why we keep getting critical path of similar or even longer length for different design: there are multiple paths that we need to eliminate to see a significant improvement in the processor cycle time and sometimes slight change in logic can have an impact on routing, making the critical path even longer than before.

One other dilemma we realize is the dual effect of adding more stages into the processor. While it can significantly help improve the cycle (from 4.1.2 to 4.2.1), it also brings more dependency between stages and results in more stalling. To improve CPI, which is also an important factor considering processor performance, we need to add more aggressive bypassing to the design. Looking at the critical paths of our designs, it is obvious that most of them are results of bypassing and data forwarding.

As we mentioned in section 4.3.2, register retiming is an effective tool Cadence Innovus has to automatically balance stages by changing pipeline register position. The reason why we still incrementally divide critical path instead of place registers at the output and let the tool do

the work is that register retiming is only helpful when there is no feedback loop in the design. Forwarding and bypassing loops are usually harder to resolve under the regime of ASIC design. If we take a look at *figure 7* and think about how the industry designs processors with a cycle time of 10-20 FO4 delays, it is reasonable to believe that ASIC design is not the entire picture of processor design. Full custom design allows engineers to manually place components and route wires in the way they want, which seems like a valid method to reduce the length of the data forwarding path. Both ASIC and full custom design are different approaches to solve the same problem and their combination is the most appropriate way to achieve the best design.

6.2 CPI

	Base-line	**X0/X1 with Stalling (4.1.1)**	**X0/X1 with Bypassing (4.1.2)**	**D0/D1 (4.2.2)**	**Branch Resolution in W (4.3.1)**	**JAL in D0 (4.3.2)**
sort/cycles	14874	19017	17603	20863	22170	22137
accum/cycles	612	813	812	1014	1114	1113
bsearch/cycles	2856	4311	3478	4325	4610	4608
vvadd/cycles	1012	1215	1112	1313	1413	1412
mfilt/cycles	6160	7322	6890	8236	8560	8541
cmult/cycles	2212	2915	2712	3213	3313	3312

Table 5. Benchmark Cycle Count

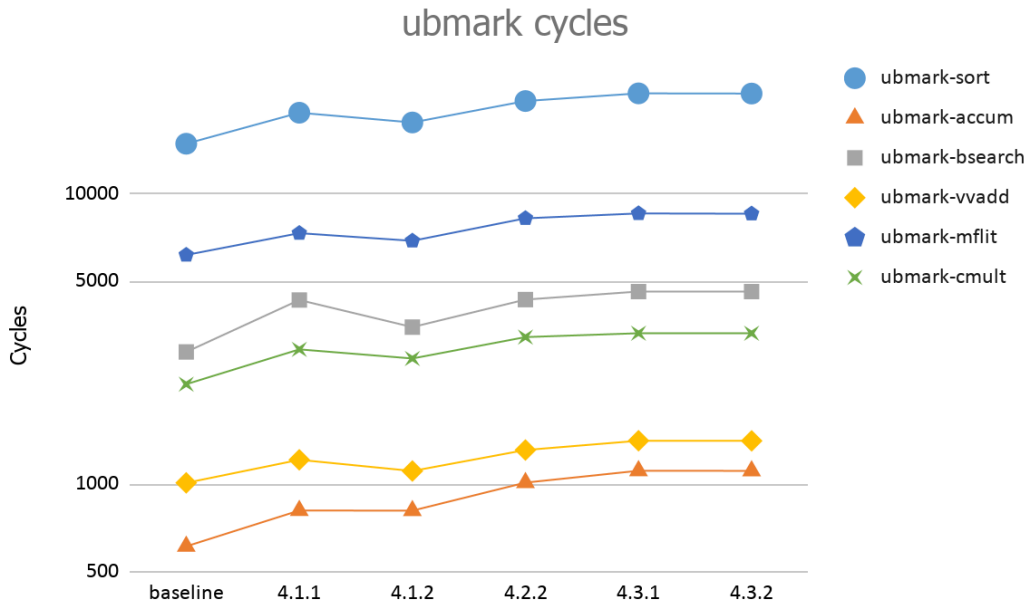


Figure 47. Ubmark Cycle Count

Applying the same microbenchmark to each design is a good way to measure the power, energy, area and CPI of them. Since we do not have an instruction count yet to calculate CPI (cycle count / instruction count), we will evaluate CPI based on the amount of cycles for each design given that instruction count is the same for each benchmark. As we divide our designs into more stages and reduce the length of the critical path, it is important to keep track of the trends of these statistics. As stated in section 1, The execution time of a program is determined by the following equation: $T = i \times cpi \times t$. Even though we primarily focus on improving t, the cycle time of the processor, the ultimate purpose is to reduce the total execution time of the software. Keeping track of cpi helps us understand how many bubbles our processor design creates and helps us decide whether implementing an aggressive bypassing is beneficial, just like the transition we have from the stalling splitted X0/X1 processor (4.1.1) to bypassing splitted X0/X1 processor (4.1.2). A lot of this is contributed by branch penalty. By adding an extra stage before branch resolution, the process wastes an extra cycle for every mispredicted branch instruction. Since we had not implemented a branch predictor, the extra cycle hurts our CPI tremendously.

For design 4.1.1, there is a noticeable increase in the number of cycles, about 30% ,compared to that of our baseline design which is reasonable because we use simple stalling to deal with read-after-write dependency between X0 and X1, and thus did not take full advantage of pipelining.

For design 4.1.2, we see an obvious decrease in the number of cycles, about 10%, for each benchmark because of the additional bypass path. The path successfully reduces the amount

of bubbles required between instructions with RAW hazards. Even though we still need to stall for multiplication, shifting, and comparison, eliminating bubbles for most common instructions such as `add` significantly contributes to our *cpi*.

For design 4.2.2, we continue to see an increase in the number of cycles for about 20%. This is again due to the extra stall logic and branch misprediction penalty. As the number of instructions squashed goes from 3 to 4, we lose many cycles as the benchmarks contain for loops that are usually taken.

For design 4.3.1, the trend continues as expected. In an effort to decrease cycle time, we sacrifice CPI by moving branch resolution to stage M, further increasing the misprediction penalty from 4 to 5, resulting in the number of cycles increasing for about 100.

For design 4.3.2, we see a slight decrease in the number of cycles. However, for benchmarks such as sort, there are many function calls and therefore the number of cycles goes down by 30.

Generally, as we come up with a new design, we see an increase in the number of cycles. This is to be expected, as our goal is to sacrifice CPI for cycle time. By separating stages, we were hoping to see a dramatic decrease in cycle time and therefore the overall execution time can be improved. As we see in *table 6*, this is not the case, unfortunately.

	Base-line	**X0/X1 with Stalling (4.1.1)**	**X0/X1 with Bypassing (4.1.2)**	**D0/D1 (4.2)**	**Branch Resolution in W (4.3.1)**	**JAL in D0 (4.3.2)**
sort/ns	17848.8	23390.91	21475.66	25661.49	28155.9	27892.62
accum/ns	734.4	999.99	990.64	1247.22	1414.78	1402.38
bsearch/ns	3427.2	5302.53	4243.16	5319.75	5854.7	5806.08
vvadd/ns	1214.4	1494.45	1356.64	1614.99	1794.51	1779.12
mfilt/ns	7392	9006.06	8405.8	10130.28	10871.2	10761.66
cmult/ns	2654.4	3585.45	3308.64	3951.99	4207.51	4173.12

Table 6. Benchmark Execution Time

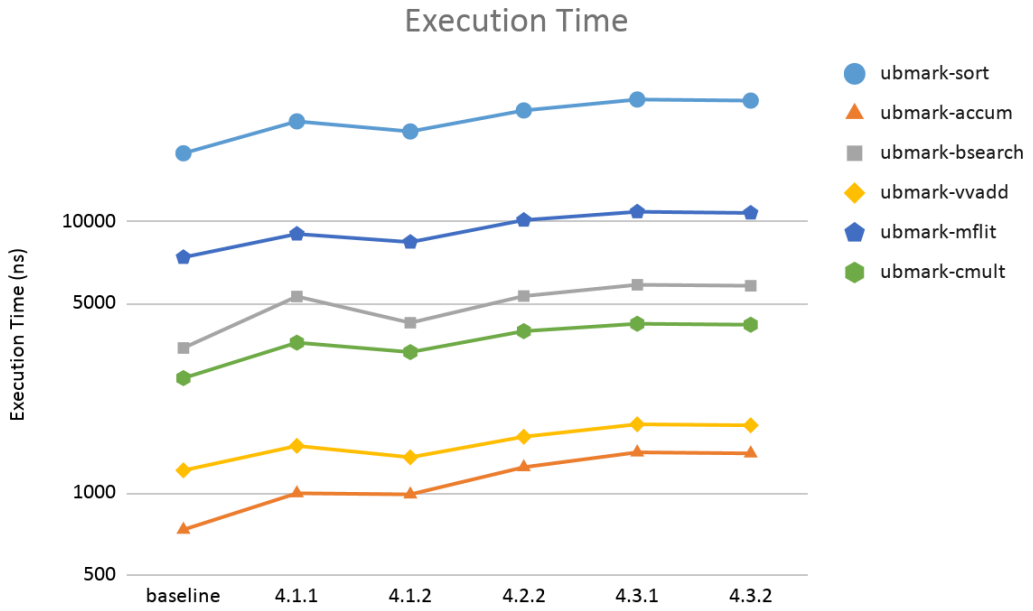


Figure 48. Execution Time for Each Design with Minimum Time Constraint

Since the number of cycles with our latest design has increased for all benchmarks compared to the baseline design, we need a decrease in cycle time in greater percentage in order to see a decrease in overall execution time, which is the product of the cycle time and the cycle count. As discussed in 6.1, the cycle time has not evolved as expected. Instead of going down for each design, the cycle time generally increases with some fluctuations. Since both cycle time and CPI increased, it is not surprising that the execution time for each benchmark increases. It is fair to say that we did not achieve our original goal and we see no speedup at all despite our efforts.

6.3 Area

From an area perspective, we push the design through the flow twice, one with its minimum cycle time and the other with uniform timing constraint, hoping we can get different insights from the result.

	Baseline	**X0/X1 with Stalling (4.1.1)**	**X0/X1 with Bypassing (4.1.2)**	**D0/D1 (4.2)**	**Branch Resolution in W (4.3.1)**	**JAL in D0 (4.3.2)**
Design area/um^2	19051.71	20283.298	20467.37	21631.918	21632.982	22170.036

Table 7. Design Area with Minimum Time Constraint

	Baseline	X0/X1 with Stalling (4.1.1)	X0/X1 with Bypassing (4.1.2)	D0/D1 (4.2)	Branch Resolution in W (4.3.1)	JAL in D0(4.3.2)
Design area/um^2	18799.434	19300.162	20146.236	21547.209	21632.982	22014.593

Table 8. Design Area with Uniform Time Constraint (1.27ns)

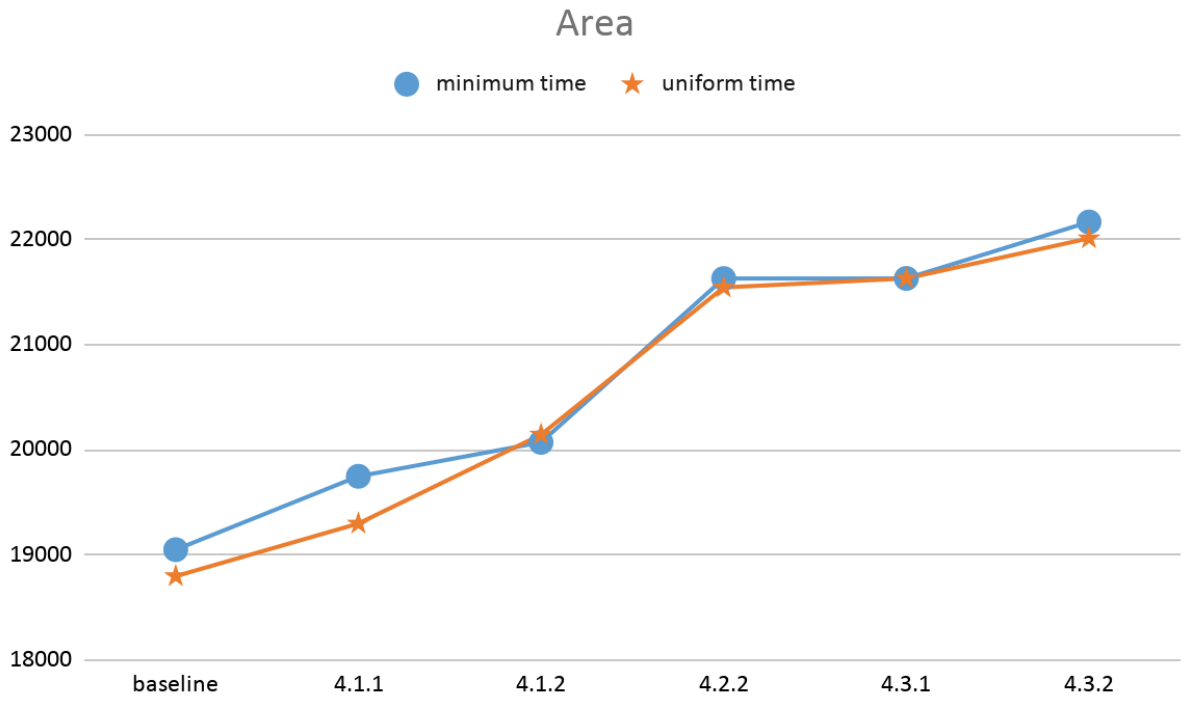


Figure 49. Area for Each Design with Minimum Time Constraint

In *table 7* and *figure 49*, we listed the area of each design with their optimal timing. It is reasonable for the design area to increase each time we make modifications. There are great leaps in the design area from both baseline to 4.1.1 and from 4.1.2 to 4.2. As we add more stages to the design, we require more pipeline registers in both the datapath and the control unit. Besides, as we have higher demand for aggressive bypassing, we need more and larger muxes. Both factors contribute to the design area. From 4.1.1 to 4.1.2, the muxes needed for bypassing increases design area by 0.91%. From 4.2 to 4.3.1, since we only move the position of PC redirection without adding extra logic, the area only change by $1um^2$. As indicated in *table 8* and *figure 49*, as we release the timing constraint for some of the designs, we can see an

improvement in design area. It is confusing why we have a 1.21% increase in area when changing from 4.3.1 to 4.3.2 design since the only change we make is to move JAL components from D1 stage to D0 stage. For 4.1.2 the design area is even slightly higher under looser time constraints. We suspect that changes in design influence the placement of standard cells and the routing of the wire, which reflects the uncertainty of the ASIC flow.

6.4 Power and Energy

	Baseline	X0/X1 with Stalling (4.1.1)	X0/X1 with Bypassing (4.1.2)	D0/D1 (4.2)	Branch Resolution in W (4.3.1)	JAL in D0(4.3.2)
energy/nJ	384.5	483.4	487	488.4	529.7	547.8

Table 9. Design Energy Consumption for ubmark-sort with Minimum Time Constraint

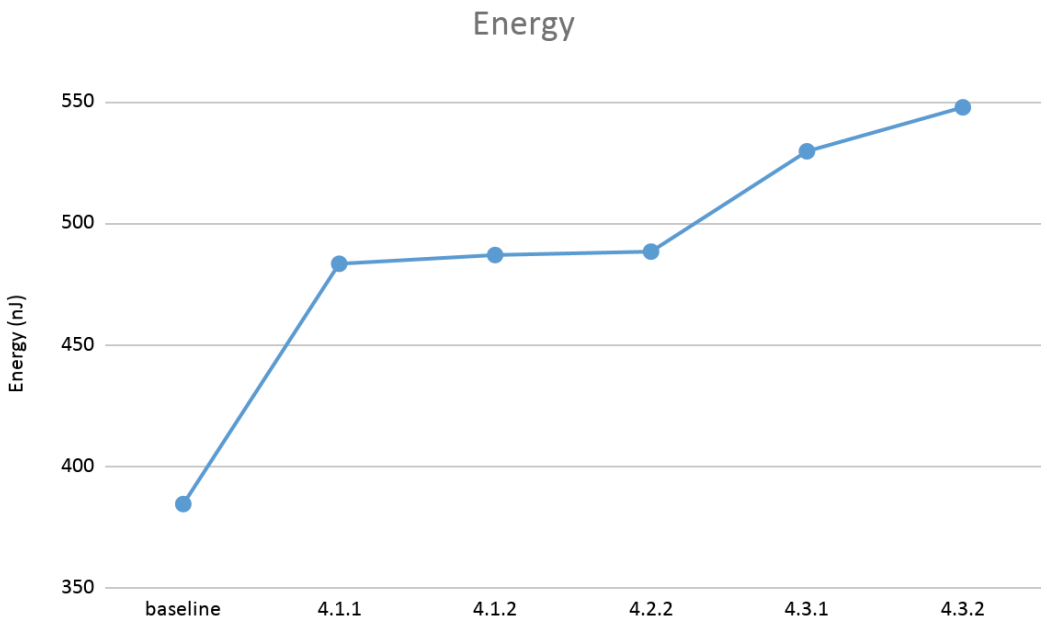


Figure 50. Energy for Each Design

To analyze the power and energy for each design, we use the waveform generated by the same benchmark, `ubmark-sort`, run by each design. As seen in *table 9* and *figure 50*, there exists an increase in energy across all designs when compared to their previous iterations. This is expected because we added extra logic for every iteration and they cost extra energy. The most significant increases we see are from baseline to 4.1.1 and from 4.2 to 4.3.1. From baseline to 4.1.1, the ALU structure is completely different and hence we see the massive increase as more computation is required. From 4.2 to 4.3.1, a drop unit is added and we believe that might be the

cause as it is doing extra work every time a data memory request goes through it. At other iterations, the change in energy is pretty insignificant as we only made minor changes such as adding a few registers to the datapath.

	Base line	X0/X1 with Stalling (4.1.1)	X0/X1 with Bypassing (4.1.2)	D0/D1 (4.2)	Branch Resolution in W (4.3.1)	JAL in D0(4.3.2)
power/mW	18.6	17.6	19.5	15.9	16	16.7

Table 10. Design Power Consumption for ubmark-sort with Minimum Time Constraint

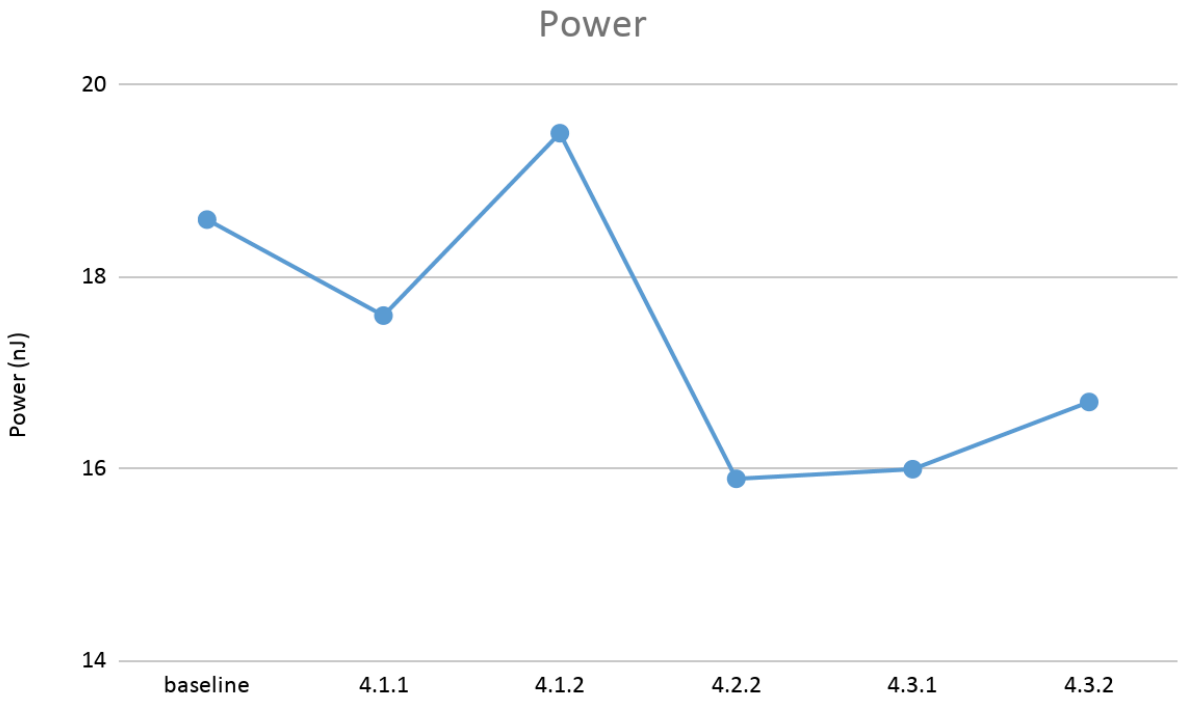


Figure 51. Power for Each Design

As seen in *table 10* and *figure 51*, the power experiences an inexplicable trend. Without 4.1.2, it looks somewhat like a quadratic trend, with the trough at 4.2.2. However, power does not matter as much as energy and it fluctuates a lot depending on the dataset as well as the implementation itself as it is affected by many factors like overall energy as well as cycle time.

In order to increase performance of the original 5-stage pipelined processor, we tried different ways to decrease the cycle time. We thought that some stages might have more logic than others and we could separate those stages in halves so that the critical path would be

shortened by half effectively. Therefore, we separated X into X0/X1 and D into D0/D1, hoping that the critical path would be shortened. We also erased some bypassing paths that are critical paths hoping that we could decrease cycle time in exchange of the increasing CPI. However, as we gradually discovered, the ASIC flow does an amazing job in optimizing the timing of different paths. As seen in *figure 46*, for each design, there are not one but dozens of paths that meet the timing constraint by just a little. When eliminating the one path, we inadvertently interfere with place and route and thus increases the delay in all these other paths. The sad truth as described in section 6.1 and 6.2 is that the numerous efforts and approaches we took did not decrease the execution time of any of the benchmarks. Every one of them increased by at least 50% compared to the baseline design as both the CPI and the cycle time increased. The area and energy also increased, albeit not significantly. Our failed attempt is a valuable lesson that timing is very tricky when it comes to ASIC design and modern ASIC tools are very effective when it comes to balancing paths and meeting timing.

8. Literature Review

[1] A. Bashteen, I. Lui, and J. Mullan, "A superpipeline approach to the MIPS architecture," *COMPCON Spring 91 Digest of Papers*.

A superpipeline approach to the MIPS architecture, published in 1991 by three employers of MIPS Computer Systems, Inc., talks about the rationale behind implementing superpipelining instead of superscalar and VLIW to achieve a higher level of performance on the new generation of MIPS processors. The old generation of MIPS processors has five stages, much like the five-stage processors that we learned about in ECE 4750. The authors noted that many stages only take half a clock cycle and the entire pipeline can be broken down into smaller stages with a deeper pipeline. Mainly, accesses to the instruction cache as well as to the data cache can be split into two states, decreasing the clock cycle requirement.

Superscalar and VLIW processors try to increase performance by running multiple instructions in parallel, and they have their disadvantages, mainly due to having to replicate hardware function units and extra logics required to handle dependencies between instructions. They also lack extensibility and backward compatibility. Superpipelined processors do not face these challenges and result in a faster cycle time while maintaining backward compatibility, but that is not to say that superpipelining is perfect. In the superpipelined version of the MIPS processor, taking branches costs "as much as 20-30%" when discussing the performance.

[2] N. P. Jouppi, and D. W. Wall, "Available instruction-level parallelism for superscalar and superpipelined machines," *ACM SIGARCH Computer Architecture News, April 1989*

Available instruction-level parallelism for superscalar and superpipelined machines, published in 1989 by Norman Jouppi and David Wall from Digital Equipment Corporation,

Western Research Lab, discusses the approach they took to evaluate parametrized superscalar and superpipelined machines with an emphasis on how their performances are improved and limited by instruction-level parallelism.

Ideally, a superpipelined machine should have almost the same performance as the superscalar machine. A superscalar machine of degree *n* can issue *n* instructions in the same cycle and thus have a throughput of *n* and a superpipelined machine of degree *m*, by making the assumption that the cycle time of a machine is many times larger than the add or load latency, has *m* times more stages but *1/m* cycle time of the base machine. What is flawed about the assumption is that a normal program has instruction-level parallelism of 2, and only by carefully manipulating the Assembly code can we bring it up above 4. Besides, we assume that the latency of each stage is equal -- for operations like memory access that may take multiple cycles when misses happen, the deeper the pipeline level the longer we need to wait. Furthermore, they also briefly mentioned how cache performance and cache misses can affect machine performance. All of these concerns provide us with ideas on what factors should be considered for evaluation and how we can improve our superpipelined machine based on the performance bottlenecks.

[3] Ching-Long Su and A.M. Despain, "Minimizing branch misprediction penalties for superpipelined processors," in *Proceedings of MICRO-27. The 27th Annual IEEE/ACM International Symposium on Microarchitecture*, San Jose, CA, USA, 1994

This paper discusses and evaluates the methods to reduce branch misprediction rates and branch penalties in order to reduce branch misprediction penalties.

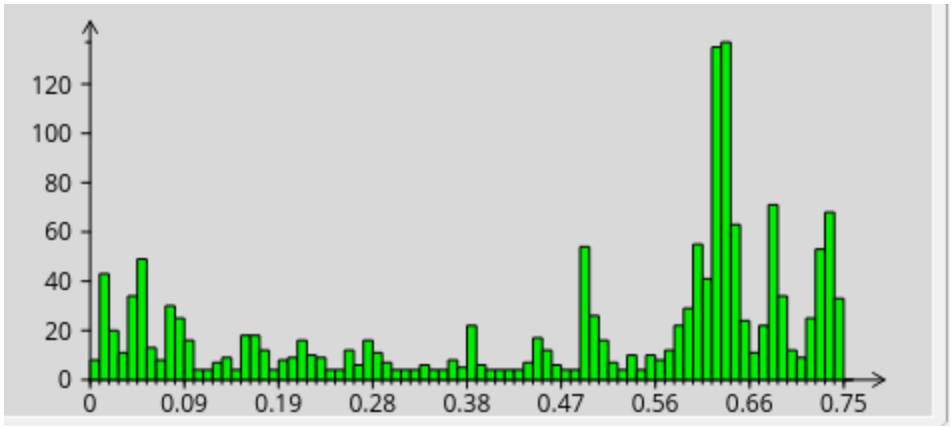
Firstly, there are two approaches to deal with branch misprediction rate. One is dynamic branch schemes implemented in hardware to predict branch behavior at run-time. The other is static branch schemes achieved by the compiler to schedule safe instructions into branch delay slots at compile-time. The authors implemented two static branch schemes: compiler predicting the branch outcomes based on program behavior at compile-time and run-time profile information. Compared with only the program behavior, a combined static branch prediction method improves the accuracy from 71.85% to 86.38% on average. Regarding the dynamic branch schemes, the authors implement a two-level adaptive branch target buffer and a correlation-based scheme that takes advantage of the "relationship between nearby branches to improve accuracy." Secondly, Branch With Masked Squashing(BWMS) is used to reduce branch penalties by filling branch delay slots as much as possible with safe instructions first and then unsafe ones from target blocks.

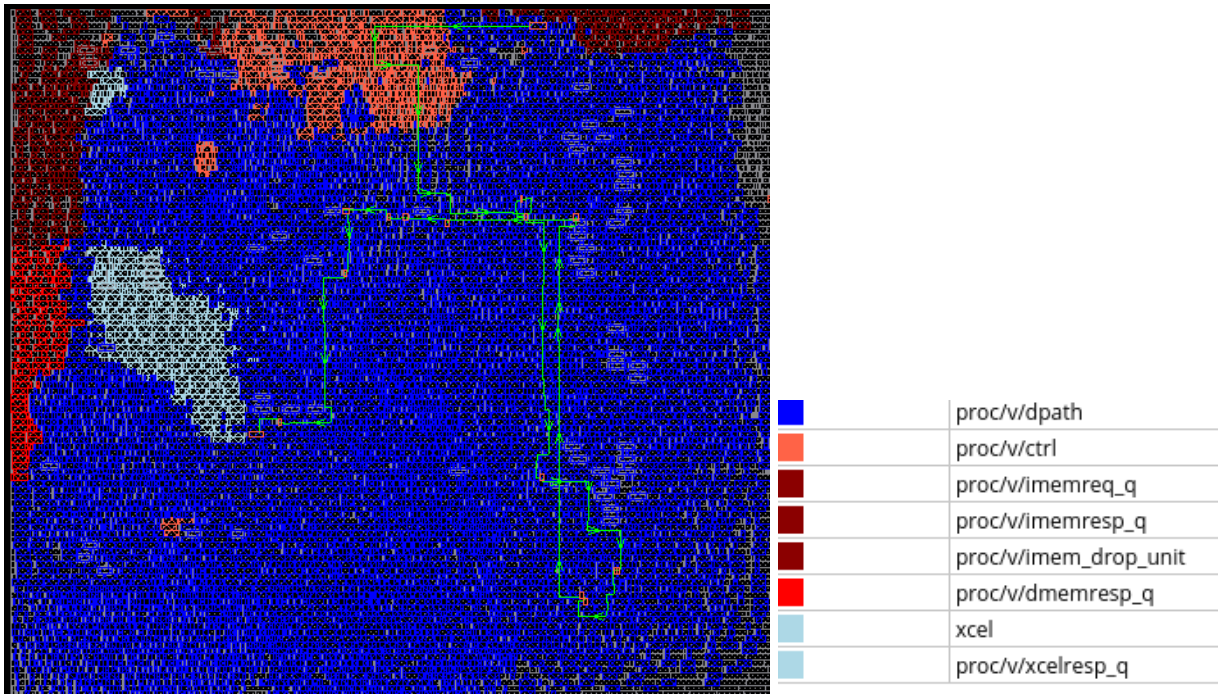
In general, the authors conclude that the prediction accuracy of the dynamic prediction schemes with a small Branch Target Buffer (BTB) can be higher than of the static prediction scheme. However, if we want dynamic prediction schemes to perform better than advanced static branch schemes (BWMS), a large BTB ("more than 2048 entries") is required.

[4] M. S. Hrishikesh, N. P. Jouppi, K. I. Farkas, D. Burger, S. Keckler, and P. Shivakumar, “The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays,” *Proceedings 29th Annual International Symposium on Computer Architecture*.

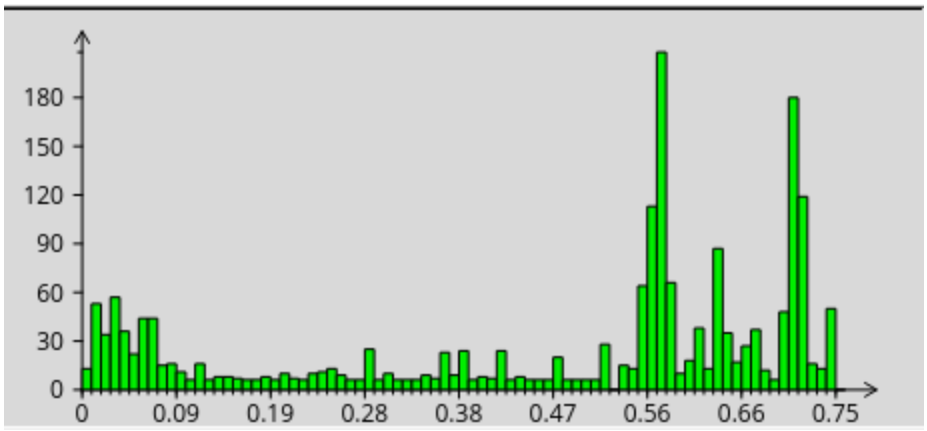
8. Appendix

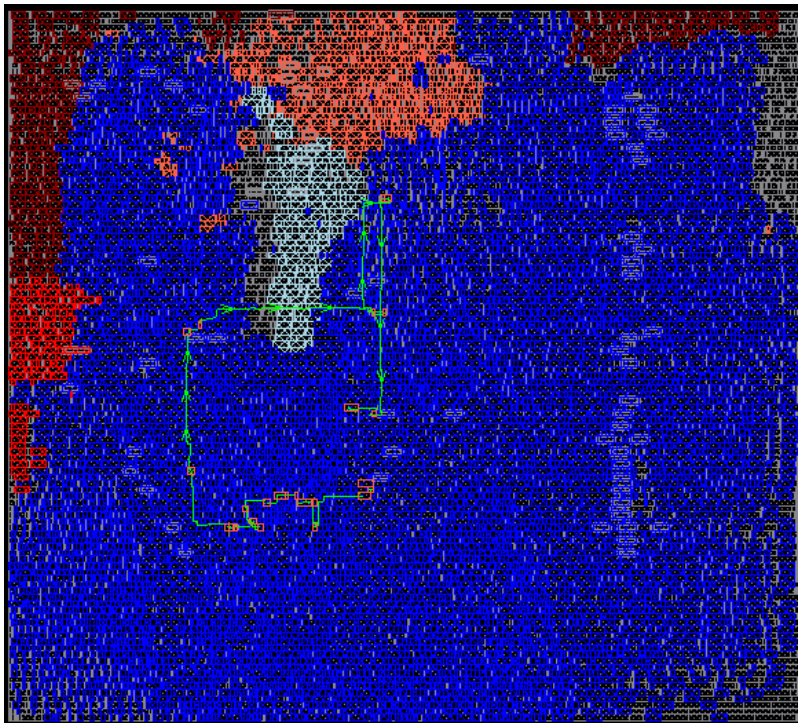
4.1.1: Path Slack Histogram; Amoeba Plot; Color Reference





4.1.2: Path Slack Histogram; Amoeba Plot; Color Reference





4.2: Path Slack Histogram; Amoeba Plot; Color Reference