# ECE 4750 Computer Architecture

# Topic 2: Processor Microarchitecture

http://www.csl.cornell.edu/courses/ece4750
School of Electrical and Computer Engineering
Cornell University

revision: 2024-09-17-10-07

## List of Problems

**1 Short Answer** **3**

1.A Architectural RAW, WAR, and WAW Dependencies . . . 3

1.B X-Stage Bypass Muxing . . . 4

1.C Implementing Multiply-Add Instruction in a Pipelined Processor . . . 5

1.D Exception Handling . . . 6

**2 Microcoded TinyRV1 Processor** – ***Correction: topic* is covered Fall 2024** **7**

2.A Implementing Conditional Move Instructions . . . 9

2.B Implementing Indirect Load Instruction . . . 9

2.C Implementing Memory-Memory Swap Instruction . . . 9

2.D Implementing Memory-Memory Increment Instruction . . . 10

2.E Implementing String Length Instruction . . . 10

**3 Multiplier Microarchitecture** **10**

3.A Iterative Microarchitecture . . . 12

3.B Two-Cycle Microarchitecture . . . 13

3.C Four-Cycle Microarchitecture . . . 13

3.D Variable-Latency Microarchitecture . . . 13

3.E Comparison of Microarchitectures . . . 14

**4 Two-Cycle Pipelined Integer ALU and Multiplier** **14**

4.A Control and Data Hazard Latencies . . . 16

4.B Resolving Data Hazards with Software Scheduling . . . 16

4.C Resolving Data Hazards with Stalling . . . 17

4.D New Stall Signal . . . . . 17
4.E Resolving Control Hazards with Speculation . . . . . 17

**5 Moving Branch Resolution from X Stage to D Stage 19**
5.A Performance when Resolving Branch in X Stage . . . . . 21
5.B Performance when Resolving Branch in D Stage . . . . . 22

**6 Reduced Register-File Ports 23**
6.A Performance of 2r1w Processor Microarchitecture . . . . . 24
6.B Performance of 1r1w Processor Microarchitecture . . . . . 25
6.C Software Scheduling for 1r1w Processor Microarchitecture . . . . . 27

## Problem 1. Short Answer

### Part 1.A Architectural RAW, WAR, and WAW Dependencies

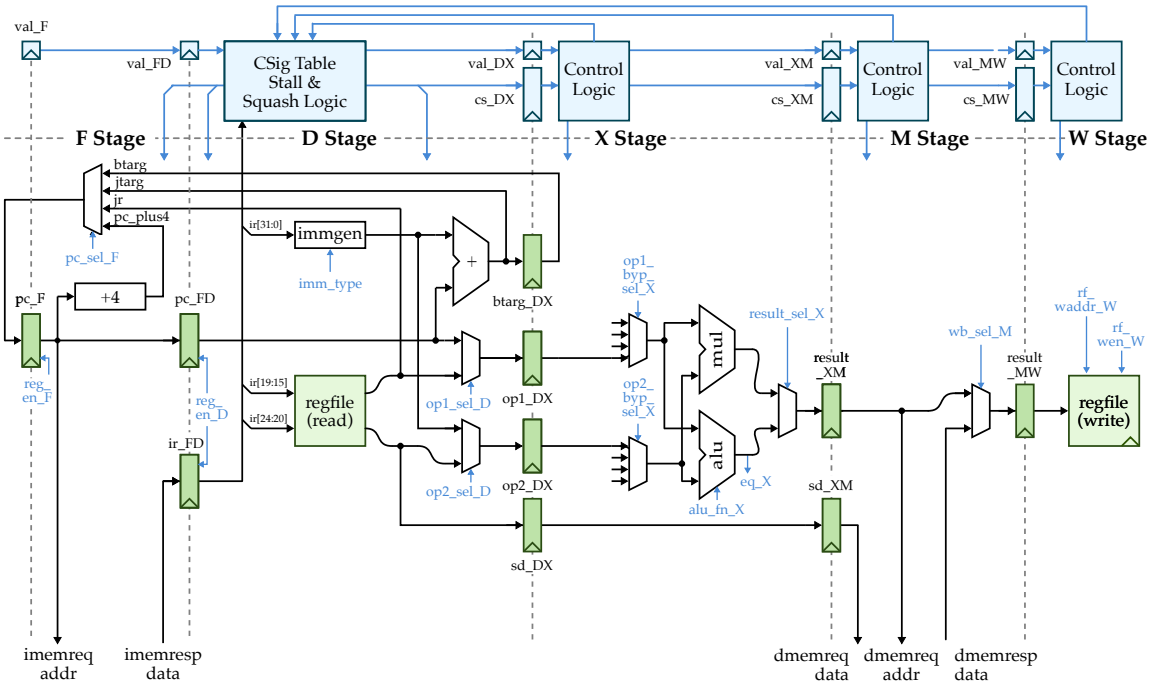
Consider the following short assembly sequence. **Draw and label arrows to indicate the architectural RAW, WAR, and WAW dependencies between instructions.** You must label your arrows so we can clearly distinguish between the three different types of dependencies.

```
sub     x1,     x2,     x3

add     x4,     x5,     x6

add     x4,     x4,     x3

sub     x8,     x9,     x1

sub     x1,     x2,     x9
```

### Part 1.B X-Stage Bypass Muxing

Recall that the canonical TinyRV1 pipelined processor discussed in lecture implemented bypassing by adding bypass muxes to the D stage. Architects sometimes choose to move these bypass muxes into the X stage. Consider the following instruction sequence and datapath diagram. The datapath diagram includes the X-stage bypass muxes, but does not include the actual wires that implement the bypass paths. **Modify the datapath diagram to include just those bypass paths which are required to avoid any stalls due to RAW hazards for the given instruction sequence. Draw a pipeline diagram for this instruction sequence that illustrates the execution assuming we include the specific bypass paths you added to the datapath diagram to avoid any stalls. You must include all relevant RAW microarchitectural dependency arrows, and these arrows must accurately reflect how data moves through the datapath.** Note that you cannot add any more muxes to the datapath diagram. You should just add wires that connect to the currently unconnected bypass mux inputs. Each arrow should correspond to a bypass path in the datapath diagram. You are required to add the *minimum* number of bypass paths to avoid stalls for this specific instruction sequence.

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `addi x1, x0, 1` | | | | | | | | | | | | | | | | | |
| `addi x1, x1, 2` | | | | | | | | | | | | | | | | | |
| `addi x1, x1, 3` | | | | | | | | | | | | | | | | | |
| `addi x1, x1, 4` | | | | | | | | | | | | | | | | | |

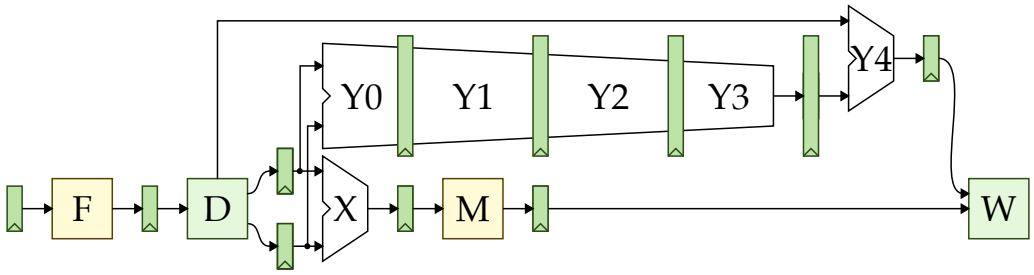


**Part 1.C Implementing Multiply-Add Instruction in a Pipelined Processor**

For this problem, we will use the fully bypassed five-stage TinyRV1 processor discussed in lecture. Consider adding a new multiply-accumulate `madd` instruction with the following semantics.

`madd rd, ry, rs1, rs2` $\quad$ R[`rd`] $\leftarrow$ ( R[`rs2`] $\times$ R[`rs1`] ) + R[`ry`]

Note that this instruction has three source operands. The following abstract datapath diagram illustrates one potential way to implement this instruction. We add a four-stage pipelined multiplier (stages Y0, Y1, Y2, Y3) followed by a dedicated single-cycle adder (stage Y4). Note that we read `rs1` and `rs2` in the D stage as with all standard TinyRV1 instructions, but this implementation delays reading `ry` until the Y4 stage. We add an extra register file read port and an extra register file write port to avoid any structural hazards. Assume that we add additional bypass paths from the end of Y4, and add bypass muxing for the `ry` source operand. Assume that instructions other than `madd` do not delay their writeback; they writeback to the register file just as they would in the standard fully bypassed five-stage TinyRV1 processor. Assume that we do not add any extra control logic to detect or prevent WAW or WAR hazards.




**Carefully choose a two-instruction assembly sequence that illustrates both a WAW and WAR hazard. You must show all register specifiers. Draw a pipeline diagram for this two instruction assembly sequence, and draw arrows on the pipeline diagram to clearly indicate the WAW and WAR hazards.** Your WAW and WAR arrows should start and end where the appropriate instruction reads or writes the register file (i.e., do not worry about bypassing when determining where to draw the arrows). **Include a few sentences about why the WAW/WAR hazards occur in this specific design.**

| | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | |

### Part 1.D Exception Handling

Consider the following instruction sequence. Assume that `op` instructions are just generic instructions with no data hazards. Assume that `xxx` is an instruction that is currently not implemented in the processor, so it will cause an exception in the D stage. Assume that the commit point is in the X stage, not in the M stage. The exception handler is at address `0x200`.

```
  0x100 opA
  0x104 opB
  0x108 xxx  # causes illegal instruction exception in D stage
  0x10c opC
  0x110 opD
  0x114 opE
  0x118 opF
  ...
exception_handler:
  0x200 opG
  0x204 opH
```

**Draw a pipeline diagram in the space provided below illustrating how this code will execute on a fully bypassed TinyRV2 processor with support for exceptions. Draw a control dependency arrow to indicate the control flow.** Stop your pipeline diagram when you get to `opH`. You will not necessarily need to use all of the rows and/or columns provided. Use a dash (–) to indicate squashed instructions going down the pipeline.

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | |

## Problem 2. Microcoded TinyRV1 Processor – *Correction: topic* is covered Fall 2024

Consider the TinyRV1 FSM processor with a microcoded control unit described in lecture. The datapath is shown in Figure 1. Figure 2 shows the encoding you should use for all of the control signal fields that are not just 0 or 1. Note that we have added three new operations to our ALU compared to lecture: increment the A input by one, copy the A input to the output, and copy the B input to the output.

In this problem, we explore adding several new instructions by using the same datapath and just adding new microcode sequences to the control store. Your solutions should be elegant and efficient; minimize the number of new states needed. You cannot add new datapath components or modify the datapath components beyond this one change (although see the final problem for an exception!). If you use any new pseudo-control-signal syntax please clearly explain what this syntax means. When filling in microcode, use don't cares (marked with an x or –) for fields where it is safe to use don't cares. Study the processor described in lecture well, and make sure all of your microinstructions are legal. Please comment your code clearly. Your code should exhibit "clean" behavior and not modify any architectural registers in the course of the execution. Finally, make sure that each new macroinstruction fetches the next macroinstruction with a microjump to F0.

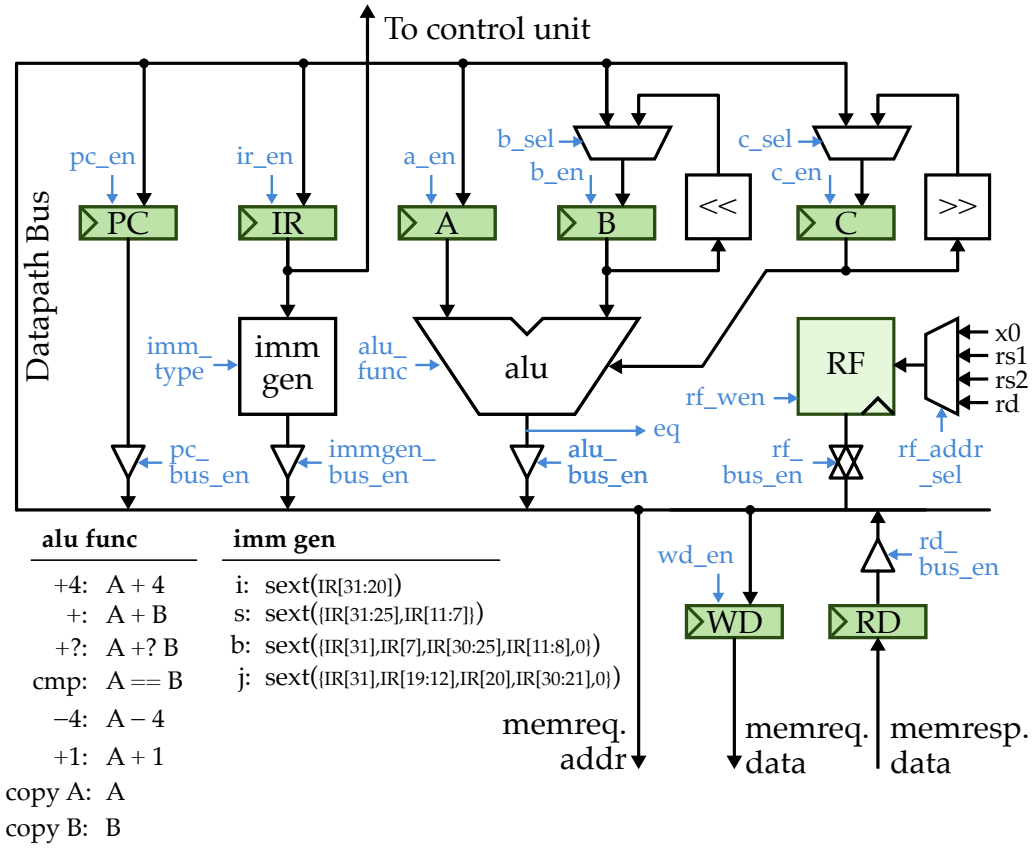



**Figure 1: TinyRV1 FSM Processor Datapath**

| | | |
|---|---|---|
| **B/C Mux Select Encoding** | b | select mux input from bus |
| | s | select mux input from shifter |
| **Immediate Generation Unit (IG) Types** | i | sext( IR[31:20] ) |
| | s | sext( IR[31:25], IR[11:7] ) |
| | b | sext( IR[31], IR[7], IR[30:25], IR[11:8], 0 ) |
| | j | sext( IR[31], IR[19:12], IR[20], IR[30:21], 0 ) |
| **Arithmetic/Logic Unit (ALU) Functions** | +4 | A + 4 |
| | + | A + B |
| | +? | C[0] ? A + B : copy A |
| | cmp | A == B |
| | -4 | A - 4 |
| | +1 | A + 1 |
| | copy A | A |
| | copy B | B |
| **Register File Select Encoding** | x0 | select `x0` |
| | rs1 | select register based on `rs1` field in IR |
| | rs2 | select register based on `rs2` field in IR |
| | rd | select register based on `rd` field in IR |
| **Memory Request Op Encoding** | r | read memory request |
| | w | write memory request |
| **Next State Encoding** | n | goto next state by incrementing µPC by one |
| | d | dispatch to instruction sequence based on opcode |
| | f | goto state F0 |
| | b | goto state F0 if A == B |

**Figure 2: Control Signal Encodings**

| | | Bus Enables | | | | | Register Enables | | | | | | Mux | | Func | | RF | | mreq | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **State** | **Pseudo Control Sigs** | **pc** | **ig** | **alu** | **rf** | **rd** | **pc** | **ir** | **a** | **b** | **c** | **wd** | **b** | **c** | **ig** | **alu** | **sel** | **wen** | **val** | **op** | **next** |
| F0: | mreq.addr ← PC; A ← PC | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | – | – | – | – | – | 0 | 1 | r | n |
| F1: | IR ← RD | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | – | – | – | – | – | 0 | 0 | – | n |
| F2: | PC ← A+4 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | – | – | – | +4 | – | 0 | 0 | – | d |
| | ... | | | | | | | | | | | | | | | | | | | | |

**Figure 3: Microcode Table**

### Part 2.A Implementing Conditional Move Instructions

For this part, you are to add two new conditional move instructions. The `movn` instruction only copies the source register to the destination register if a second source register is not zero. The assembly format and semantics for the new RISC-V instruction are as follows:

`movn rd, rs1, rs2` if ( R[rs2] ! = 0 ) R[rd] ← R[rs1]

The `movz` instruction only copies the source register to the destination register if a second source register is zero. The assembly format and semantics for the new RISC-V instruction are as follows:

`movz rd, rs1, rs2` if ( R[rs2] == 0 ) R[rd] ← R[rs1]

**Create a table like the one shown in Figure 3 to represent the contents of the control store, and fill in the state, pseudo-control-signal syntax, actual control signals, and next state fields for a microinstruction fragment that implements both the `movn` and `movz` instructions.** The fetch fragment has already been provided for you.

### Part 2.B Implementing Indirect Load Instruction

For this part, you are to add a new indirect load (`lwx`) instruction. The assembly format and semantics for the new RISC-V instruction are as follows:

`lwx rd, imm(rs1)` R[rd] ← M[ M[ R[rs1] + sext(imm) ] ]

**Create a table like the one shown in Figure 3 to represent the contents of the control store, and fill in the state and pseudo-control-signal syntax for a microinstruction fragment that implements the `lwx` instruction. You do not need to fill in the actual control signals or the next state fields!** The fetch fragment has already been provided for you. Minimize the number of microinstructions in your sequence. Note that a naive implementation will always do the swap even when the values being swapped are identical.

### Part 2.C Implementing Memory-Memory Swap Instruction

For this part, you are to add a new memory-memory swap (`swap.mm`) instruction. The assembly format and semantics for the new RISC-V instruction are as follows:

`swap.mm rd, rs1` *temp* ← M[ R[rs1] ]; M[ R[rs1] ] ← M[ R[rd] ]; M[ R[rd] ] ← *temp*

Note that this instruction should not modify any architectural state other than what is indicated by the semantics (i.e., you must carefully select what you use for temporary state to avoid corrupting architectural state). **Create a table like the one shown in Figure 3 to represent the contents of the control store, and fill in the state and pseudo-control-signal syntax for a microinstruction fragment that implements the `swap.mm` instruction. You do not need to fill in the actual control signals or the next state fields!** The fetch fragment has already been provided for you. Minimize the number of microinstructions in your sequence. Note that a naive implementation will always do the swap even when the values being swapped are identical. **Optimize your microinstruction sequence to reduce the execution time when the values being swapped identical, but avoid increasing the execution time in the common case when the values being swapped are not identical.**

### Part 2.D Implementing Memory-Memory Increment Instruction

For this part, you are to implement a new instruction that is similar in spirit to the x86 memory-memory `inc` instruction discussed in a practice problem for the previous topic. This instruction will use a relatively complicated addressing mode to read a value from memory, increment that value by one, and then write the value back to the same location in memory. The assembly format and semantics for the new RISC-V instruction are as follows:

`inc rd, rs1, imm` $\quad$ *addr* $\leftarrow$ R[`rd`] + (R[`rs1`] $\times$ `imm`[3:0]); M[*addr*] $\leftarrow$ M[*addr*] + 1

Note that *addr* is simply a temporary to simplify the instruction semantics. It is not architectural state. Note that this instruction only uses the least significant four bits of the immediate when calculating the effective address. **Create a table like the one in Figure 3 to represent the contents of the control store, and fill in the state, pseudo-control-signal syntax, actual control signals, and next state fields for a microinstruction fragment that implements the `inc` instruction.** The fetch fragment has already been provided for you.

### Part 2.E Implementing String Length Instruction

For this part, you are to implement a new string length (`strlen`) instruction. The assembly format and semantics for the new RISC-V instruction are as follows:

`strlen rd, rs1` $\quad$ R[`rd`] $\leftarrow$ 0; while ( M[ R[`rs1`] ] != 0 ) { R[`rs1`] $\leftarrow$ R[`rs1`] + 1; R[`rd`] $\leftarrow$ R[`rd`] + 1 }

In other words, the `strlen` instruction should count the number of characters in a string pointed to by `rs1` and return the final count in the `rd` register. After the instruction has finished, the `rs1` register will contain a pointer to the null character at the end of the string. *Note that we will need to assume we can do byte reads from memory and you will need to assume you can encode a micro-branch to any other micro-instruction (which is new functionality we did not support in lecture!).* **Create a table like the one shown in Figure 3 to represent the contents of the control store, and fill in the state and pseudo-control-signal syntax for a microinstruction fragment that implements the `strlen` instruction. You do not need to fill in the actual control signals or the next state fields!** The fetch fragment has already been provided for you.

## Problem 3. Multiplier Microarchitecture

In this problem, we consider several different implementations of an unsigned two-input integer multiplier capable of multiplying a 32-bit operand by a 4-bit operand to produce a truncated 32-bit result. Figure 4 abstractly illustrates the datapaths for five microarchitectures: a single-cycle microarchitecture; a four-cycle iterative microarchitecture; a two-cycle microarchitecture that can be either unpipelined or pipelined; a four-cycle microarchitecture that again can be either unpipelined or pipelined; and a variable-latency pipelined microarchitecture. The "iron-law" of processor performance is applicable to far more than just processors. We will be using the following generalized form to examine the performance of each of these microarchitectures:

$$\frac{\text{Time}}{\text{Transaction Sequence}} = \frac{\text{Transactions}}{\text{Transaction Sequence}} \times \frac{\text{Cycles}}{\text{Transaction}} \times \frac{\text{Time}}{\text{Cycle}}$$

In our multiplier, a transaction is simply a multiplication request. **Create a table similar to the one shown in Figure 5.** Feel free to use a spreadsheet and copy the final table into your submission.

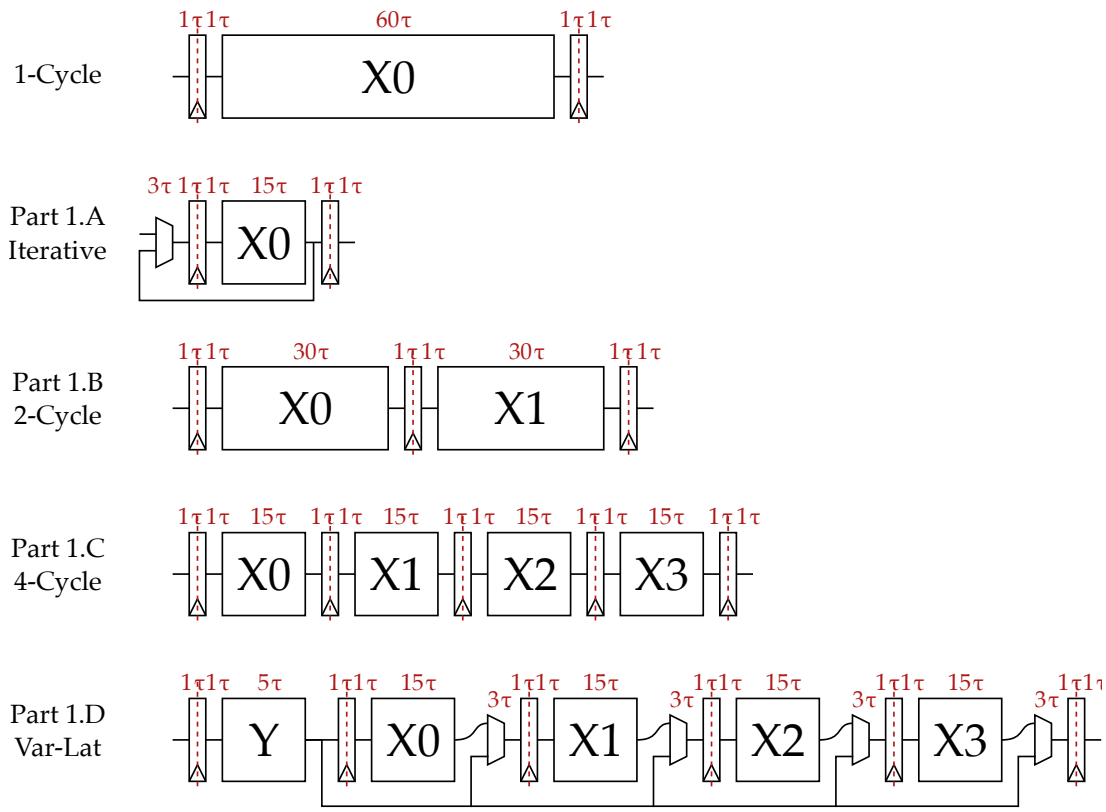



**Figure 4: Various Multiplier Microarchitectures**

In each part, we will study one of these microarchitectures, and our goal is to gradually fill in this table. The table already includes the results for the single-cycle multiplier microarchitecture.

The cycle time (i.e., clock period) is measured in normalized gate delays, where $1\,\tau$ is the delay of a single inverter driving four identical inverters. Rough approximations of the delay of each component are shown on the datapath diagrams in Figure 4. Note that a register has a clock-to-data delay of $1\,\tau$ (i.e., the combinational delay from the rising clock edge to when the output data is stable), and a setup time of $1\,\tau$ (i.e., how much time before the clock edge the input data must be stable).

The transaction latency is the number of cycles we need to execute a single transaction going through the multiplier in isolation. For the variable latency multiplier, the transaction latency depends on the data so you should include a range in the table that captures the best and worst case transaction latency.

The transaction throughput can be measured as either the average number of transactions we process per cycle or the average number of cycles a transaction occupies the microarchitecture. One is just the inverse of the other. Latency and throughput are two very different (although related) concepts; please make sure you clearly understand these two concepts. When calculating the transaction throughput for this problem, we will assume that our multiplier is processing a sequence of 60 transactions. The 60 transactions are formed by repeating the following three transactions 20 times.

| | | Num Trans | Cycle Time | Transaction Latency | | Transaction Throughput | | Total Execution Time | |
|---|---|---|---|---|---|---|---|---|---|
| | **Microarchitecture** | **(#)** | **($\tau$)** | **(cyc)** | **($\tau$)** | **(trans/cyc)** | **(cyc/trans)** | **(cycles)** | **($\tau$)** |
| | 1-Cycle | 60 | 62 | 1 | 62 | 1 | 1 | 60 | 3720 |
| Part A | Iterative | 60 | | | | | | | |
| Part B | 2-Cycle Unpiplined | 60 | | | | | | | |
| Part B | 2-Cycle Pipelined | 60 | | | | | | | |
| Part C | 4-Cycle Unpiplined | 60 | | | | | | | |
| Part C | 4-Cycle Pipelined | 60 | | | | | | | |
| Part D | Var-Lat Pipelined | 60 | | | | | | | |

**Figure 5: Evaluation of Various Multiplier Microarchitectures**

```
mul 0xdeadbeef, 0xf
mul 0xf5fe4fbc, 0x7
mul 0x0a01b044, 0x3
```

So the 60 transaction sequence will look like this:

```
mul 0xdeadbeef, 0xf
mul 0xf5fe4fbc, 0x7
mul 0x0a01b044, 0x3
mul 0xdeadbeef, 0xf
mul 0xf5fe4fbc, 0x7
mul 0x0a01b044, 0x3
mul 0xdeadbeef, 0xf
mul 0xf5fe4fbc, 0x7
...
```

The total execution time is the total time (in units of $\tau$) to execute the sequence of 60 transactions.

### Part 3.A Iterative Microarchitecture

Consider the iterative multiplier microarchitecture shown in Figure 4. This microarchitecture is very similar to the one you implemented in the first lab assignment, except that we only need to iterate for four cycles. This is because we are multiplying a 32-bit operand by a just a 4-bit operand. Assume that we have optimized the implementation so that we do not need any additional cycles to handle the val/rdy interface. The multiplier only handles unsigned numbers so we don't need to worry about sign/unsign logic. **In other words, we can complete each transaction in exactly four cycles, and we are ready to start the next transaction after exactly four cycles.**

**Draw a transaction vs. time diagram illustrating the execution of the first four multiplication transactions on this microarchitecture**. A transaction vs. time diagram is like a pipeline diagram, but of course this microarchitecture is not pipelined. There should be one column per cycle and one row per transaction. Use the symbol X0 to indicate on which cycle each transaction is using the iterative multiplier. Use your transaction vs. time diagram to fill in the appropriate row of the table in Figure 5.

### Part 3.B Two-Cycle Microarchitecture

Consider the two-cycle multiplier microarchitecture shown in Figure 4. In this microarchitecture, we use the same basic approach as the iterative multiplier, but we do the computation in space instead of time by unrolling the shift and add operations. We do two of the shift and addition operations in a single cycle. We will consider an unpipelined variant where only a single transaction can be using any part of the multiplier at once, and a pipelined variant where there can be two different transactions using the multiplier at the same time (i.e., one in the X0 stage and one in the X1 stage).

**Draw two transaction vs. time diagrams illustrating the execution of the first four multiplication transactions on both the unpipelined and pipelined variants.** Use the symbols X0 and X1 to indicate on which cycle each transaction is using that part of the multiplier. Use your transaction vs. time diagram to fill in the appropriate rows of the table in Figure 5.

### Part 3.C Four-Cycle Microarchitecture

Consider the four-cycle multiplier microarchitecture shown in Figure 4. In this microarchitecture, we use the same basic approach as the two-cycle multiplier, but with a single shift and addition operation per cycle. We will consider an unpipelined variant where only a single transaction can be using any part of the multiplier at once, and a pipelined variant where there can be four different transactions using the multiplier at the same time (i.e., different transactions in X0, X1, X2, and X3).

**Draw two transaction vs. time diagrams illustrating the execution of the first four multiplication transactions on both the unpipelined and pipelined variants.** Use the symbols X0, X1, X2, and X3 to indicate on which cycle each transaction is using that part of the multiplier. Use your transaction vs. time diagram to fill in the appropriate rows of the table in Figure 5.

### Part 3.D Variable-Latency Microarchitecture

Consider the variable-latency multiplier microarchitecture shown in Figure 4. This microarchitecture exploits the fact that when some of the bits in the four-bit operand are zero, we don't actually have to do any work. We add a new stage at the beginning of the pipeline (denoted with the Y symbol) that is responsible for determining the bit position of the most significant one in the four-bit operand. This control information is used to set the mux select in a later pipeline stage so as to skip over some of the early stages in the pipeline. For example, if the four-bit operand is two (0b0010), then the transaction would go through stage Y, skip over stages X0 and X1, and go through stages X2 and X3. Note that this requires an extra mux at the end of each X stage, and we will need to carefully handle the structural hazard caused by multiple stages writing the same register. Again, if the four-bit operand is two, then we need to avoid two transactions writing the pipeline register at the end of the X1 stage at the same time. **Assume that the multiplier stalls in the Y stage whenever it detects that letting the current transaction go down the pipeline would cause a structural hazard.**

**Draw a transaction vs. time diagram illustrating the execution of the first four multiplication transactions on the pipelined variable-latency microarchitecture.** Use the symbols Y, X0, X1, X2, and X3 to indicate on which cycle each transaction is using that part of the multiplier. Look at the four-bit operand in each of the four multiplication transactions to determine how many stages of computation are actually required. Ensure that two transactions are never in the same stage at

the same time. Use your transaction vs. time diagram to fill in the appropriate row of the table in Figure 5.

### Part 3.E Comparison of Microarchitectures

Which microarchitecture has the highest performance? **In a few paragraphs, explain some of the trade-offs in terms of area and performance between these microarchitectures.** Would we ever want to consider a multiplier with many more stages (e.g., a 20-cycle pipelined microarchitecture)? How does the fixed-latency pipelined multiplier compare to the variable-latency pipelined multiplier? Discuss when we would want to choose fixed-latency over variable-latency, and when we would want to choose variable-latency over fixed-latency. How would this trade-off change if one multiply transaction needs to wait for the result of an earlier transaction before starting? Make sure you generalize your conclusions so that they are valid over many different transaction sequences, not just the specific sequence studied in this problem.

## Problem 4. Two-Cycle Pipelined Integer ALU and Multiplier

Assume in a given emerging technology, the logic delay is significantly slower than the memory delay as compared to the standard CMOS technology used in modern processors. In this situation, the execute stage of the standard five-stage pipeline might be significantly longer than the other stages, and as a consequence, we might want to split this stage into two pipeline stages. In this problem we will be investigating the implications of using a two-cycle pipelined integer ALU and multiplier. Our new pipelined TinyRV1 processor will have the following six stages:

- F – instruction fetch
- D – decode and read registers
- X0 – first half of the ALU and multiply operation
- X1 – second half of the ALU and multiply operation
- M – data memory read/write
- W – write registers

Figure 7 illustrates the new six-stage datapath. Spend some time studying this datapath to understand how the two-cycle pipelined integer ALU and multiplier afffect the structural, data, and control hazards in the pipeline. Assume that only those bypass paths shown in the diagram are present. More specifically, notice that this datapath does not allow bypassing between back-to-

**Static Instr Sequence**

```
1   bne  x1, x0, done
2   lw   x5, 0(x2)
3   lw   x6, 0(x3)
4   add  x7, x5, x6
5   addi x8, x4, 4
6   sw   x7, 0(x8)
7   ...
8 done:
9   addi x10, x9, 1
```

| Dynamic Transaction | Cycle 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 `bne x1, x0, done` | F | D | X0 | X1 | M | W | | | | | |
| 2 `lw x5, 0(x2)` | | F | D | X0 | X1 | M | W | | | | |
| 3 `lw x6, 0(x3)` | | | F | D | X0 | X1 | M | W | | | |
| 4 `add x7, x5, x6` | | | | F | D | X0 | X1 | M | W | | |
| 5 `addi x8, x4, 4` | | | | | F | D | X0 | X1 | M | W | |
| 6 `sw x7, 0(x8)` | | | | | | F | D | X0 | X1 | M | W |

**Figure 6: Execution of Six-Stage Pipelined MIPS32 Processor**

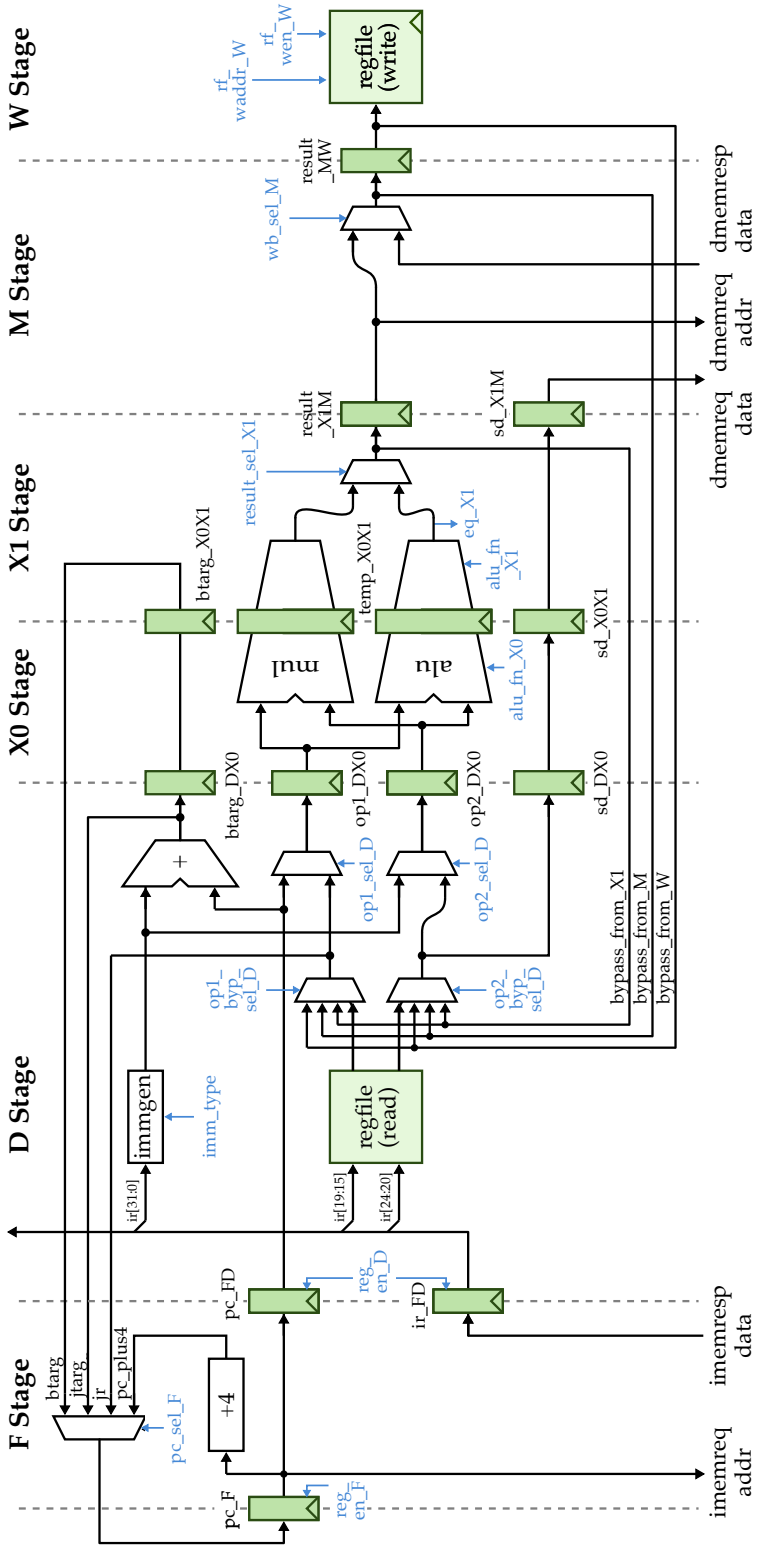



**Figure 7: Datapath for Six-Stage Pipelined TinyRV1 Processor**

back dependent integer ALU operations without requiring some kind of stall. Also notice the store data can only bypassed into the end of the D stage, and that conditional branches are resolved by the end of the X1 stage.

Figure 6 shows a simple code sequence and illustrates the read-after-write (RAW) data dependencies present given the current pipeline assuming the branch is not taken. An arrow indicates a microarchitectural RAW dependency, and since some of these arrows point backwards they create data hazards. Please note the backwards arrow representing the RAW hazard between the X1 stage of the `addi` instruction and the D stage of the `sw` instruction. The result of the `addi` is not ready until the end of the X1 stage, but the `sw` instruction needs the store data at the end of the D stage so that it can bypass it into the operand pipeline register between the D and X0 stages – spend some time studying what bypass paths are available in Figure 7. Technically, we could add a special bypass path just for the store data from the end of the X1 stage to the end of the X0 stage (and also from the end of the M stage into the end X0 stage and the end of the M stage into the end of the X1 stage). Since this bypass path can only be used in very specific cases, it is not included in our design (and thus not shown in Figure 7). In this microarchitecture, the store base addresss and store data must both be ready at the the end of the D stage so they can be muxed into the operand pipeline reigster between the D and X0 stages. Spend some more time understanding the example in Figure 6.

### Part 4.A Control and Data Hazard Latencies

The jump resolution latency and branch resolution latency are the number of cycles we need to delay the fetch of the next instruction in order to avoid any kind of control hazard (assuming we do not use speculation). Note that with a single-issue processor, we always delay the fetch of the next instruction by one cycle anyways. The ALU-use delay latency is the number of cycles we need to delay the execution of an instruction that uses the result of an integer ALU instruction to avoid a data hazard. The load-use delay latency is the number of cycles we need to delay an instruction that uses the result of a load to avoid a data hazard. For the standard five-stage pipeline, the jump resolution latency is two cycles, the branch resolution latency is three cycles if the branch condition is checked in the execute stage, the branch resolution latency is two cycles if the branch condition is checked in the decode stage, the ALU-use delay latency is one cycle, and the load-use delay latency is two cycles. Since a single-issue processor always delays the fetch of the next instruction by one cycle, we do not need to stall even though the ALU-use delay latency is one cycle.

**What is the jump resolution latency, branch resolution latency, the ALU-use delay latency, and the load-use delay latency for the new six-stage pipeline shown in Figure 7?**

### Part 4.B Resolving Data Hazards with Software Scheduling

Assume we have a fully bypassed datapath, but we expose those stalls that are unavoidable (even with bypassing) in the instruction set architecture. Reschedule the code shown in Figure 6 by moving instructions and/or adding `nop` instructions to avoid any data hazards that are not handled by bypassing. Try to minimize the execution time of the instruction sequence. Your rescheduled code should be functionally equivalent to the original code, but should have no stalls! **Show the new static code sequence and describe your optimizations. Draw a pipeline diagram similar to the one shown in Figure 6.** Either draw the microarchitectural RAW dependencies as in Figure 6 or list them in the form *Instruction X's D stage depends on Instruction Y's M stage*. Verify that none of the RAW dependency arrows point backward in time. Assume the branch is not taken, and that the microarchitecture always speculatively executes the not taken path.

### Part 4.C Resolving Data Hazards with Stalling

Assume we wish to use hardware stalling to correctly prevent RAW hazards that we cannot avoid with bypassing. Note that we are using a combination of stalling and bypassing in this problem. **Draw a pipeline diagram similar to the one shown in Figure 6 that shows which instructions have to stall in which stages.** Show stalls by simply repeating the pipeline stage character (e.g., D) for multiple consecutive cycles. Either draw the microarchitectural RAW dependencies as in Figure 6 or list them in the form *Instruction X's D stage depends on Instruction Y's M stage*. Verify that none of the RAW dependency arrows point backward in time. Assume the branch is not taken, and that the microarchitecture always speculatively executes the not taken path.

### Part 4.D New Stall Signal

The stall signal for a fully bypassed five-stage pipeline was discussed in class and is included below for you reference:

```
ostall_load_use_X_rs1_D =
  val_D && rs1_en_D && val_X && rf_wen_X
        && (inst_rs1_D == rf_waddr_X) && (rf_waddr_X != 0)
        && (op_X == LW)

ostall_load_use_X_rs2_D =
  val_D && rs2_en_D && val_X && rf_wen_X
        && (inst_rs2_D == rf_waddr_X) && (rf_waddr_X != 0)
        && (op_X == LW)

ostall_D =
  val_D && ( ostall_load_use_X_rs1_D || ostall_load_use_X_rs2_D )
```

Each signal has a suffix indicating which pipeline stage the signal originates from. `rs1_en` and `rs2_en` are true for instructions that read from either the `rs1` or `rs2` registers respectively; `rs1` and `rs2` are the actual read register specifiers for both read ports; `rf_waddr` is the write destination register; and `op` is the opcode. Understand this stall signal thoroughly before attempting to complete this part.

**Derive the new stall signal for the six-stage pipeline with the datapath and associated bypassing shown in Figure 7.** This stall signal essentially implements the stalls that you identified in the previous part. You should use a similar syntax as the original stall signal above. Define new `ostall` hazard signals as necessary.

### Part 4.E Resolving Control Hazards with Speculation

In this problem, you will explore resolving control hazards using speculation by drawing two pipeline diagrams. The first pipeline diagram should assume the branch is taken. **Draw a pipeline diagram similar to the one shown in Figure 6 that shows which instructions have to be squashed in which stages.** Use a dash symbol (–) to indicate pipeline bubbles caused by squashing instructions. For the second pipeline diagram assume we replace the `bne` instruction with a `jal` instruction. **Draw a pipeline diagram similar to the one shown in Figure 6 except with a `jal` instruction**

**in place of the `bne` instruction. Your diagram should clearly show which instructions have to be squashed in which stages.**

## Problem 5. Moving Branch Resolution from X Stage to D Stage

In this problem, you will explore the two different processor microarchitectures shown in Figures 10 and 11. The microarchitecture in Figure 10 is the standard fully bypassed five-stage TinyRV1 processor discussed in lecture. The microarchitecture in Figure 11 moves branch resolution into the D stage by adding a dedicated branch comparator after the bypass muxes. Figure 8 contains the delay of the various components in the datapath in units of $\tau$.

We will examine how these two microarchitectures execute the following function written in assembly, which is a variant of one of the sequences we studied in lecture. The function compares each element in an array of integers to each of three search values. If a value is found, then the function returns the index of the value. If none of the three values are found, then the function returns -1. Assume the following initial register values: `x4` initially holds the pointer to the array of integers; `x5` holds the size of the array; `x6`, `x7`, and `x8` hold the three search values. `x2` holds the return value. Assume that `x5` is initially 64 and that none of the search values are actually present in the array (i.e., the loop executes 64 times).

```
0x1000    addi x12, x0, 0

        loop:
0x1004    lw   x13, 0(x4)
0x1008    bne  x13, x6, L1 # check value 1
0x100c    jal  x0, done
        L1:
0x1010    bne  x13, x7, L2 # check value 2
0x1014    jal  x0, done
        L2:
0x1018    bne  x13, x8, L3 # check value 3
0x101c    jal  x0, done
        L3:
0x1020    addi x4,  x4,  4
0x1024    addi x12, x12, 1
0x1028    bne  x12, x5, loop
0x102c    addi x2,  x0, -1
0x1030    jalr x0,  x31

        done:
0x1034    addi x2,  x12, 0
0x1038    jalr x31
```

| Component | Delay ($\tau$) |
|---|---|
| register read | 1 |
| register write | 1 |
| regfile read | 10 |
| regfile write | 10 |
| memory read | 20 |
| memory write | 20 |
| +4 unit | 4 |
| immgen | 2 |
| mux | 3 |
| multiplier | 20 |
| alu | 10 |
| adder | 8 |
| br cmp | 4 |
| squash logic | 3 |

**Figure 8: Datapath Component Delays**

| Part | Branch Resolution | Instructions / Program | Avg Cycles / Instruction | Time ($\tau$) / Cycle | Time ($\tau$) / Program |
|---|---|---|---|---|---|
| **2.A** | **X** | | | | |
| **2.B** | **D** | | | | |

**Figure 9: Processor Performance for Assembly Sequence with Two Different Branch Resolutions**

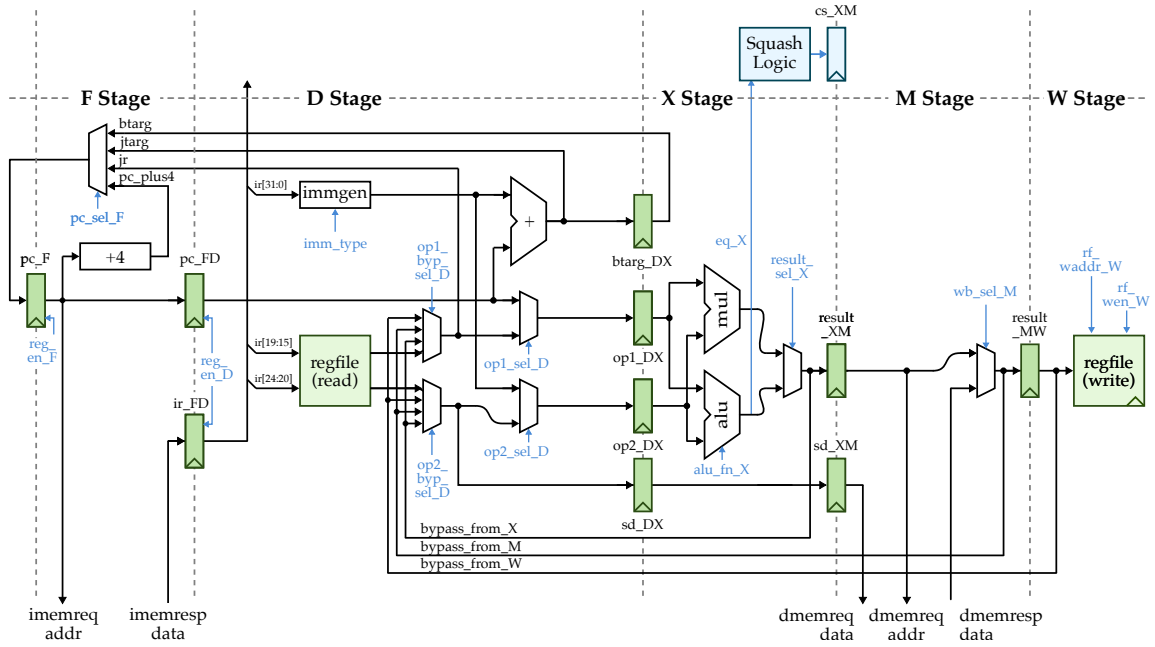



**Figure 10: Processor Datapath for Branch Resolution in X**

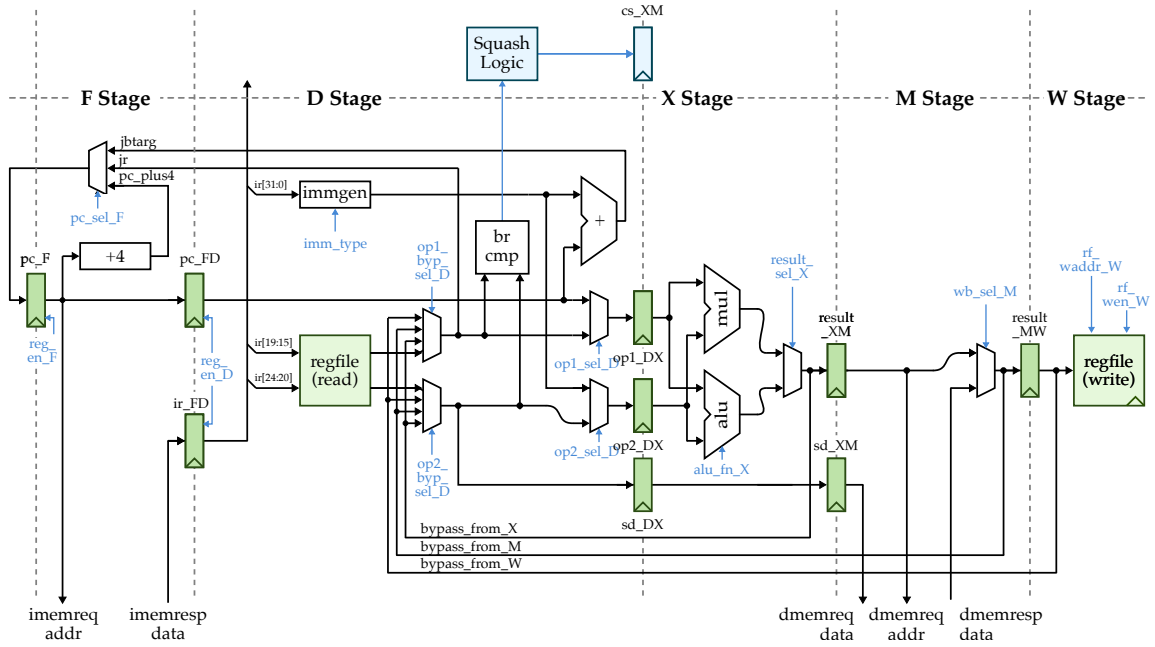



**Figure 11: Processor Datapath for Branch Resolution in D**

**Part 5.A Performance when Resolving Branch in X Stage**

In this part, we will calculate the execution time for the assembly sequence given above executing on the microarchitecture shown in Figure 10. Again, this microarchitecture is identical to the fully bypassed five-stage TinyRV1 pipeline discussed in lecture. There are no special features; the processor is exactly as described in lecture. Assume the processor is directly connected to a magic combinational memory providing single-cycle instruction and data access latency. Remember that branches are resolved in the X stage and there is no branch delay slot.

**Calculate the critical path for this microarchitecture and clearly highlight this path on Figure 10 given the datapath component delays in Figure 8.** *Hint: the critical path likely includes one of the bypass paths.* **Draw a pipeline diagram illustrating how the first iteration of the loop executes on this microarchitecture.** You can use microarchitectural dependency arrows to illustrate how data is transferred between instructions, although this is not required. If you do draw arrows, then they must be correct. You may want to include the first instruction of the second iteration to illustrate the impact of the branch resolution latency. **Based on this pipeline diagram estimate the execution time in units of $\tau$ for 64 iterations of this loop. Fill in the appropriate row of the table in Figure 14. You must show your work.**

**Part 5.B Performance when Resolving Branch in D Stage**

In this part, we will calculate the execution time for the assembly sequence given above executing on the microarchitecture shown in Figure 11. This microarchitecture is identical to the fully bypassed five-stage TinyRV1 pipeline in the previous part except for one key difference: the branch is resolved in D using a dedicated branch comparator (labeled "br cmp" in the datapath diagram).

**Calculate the critical path for this microarchitecture and clearly highlight this path on Figure 11 given the datapath component delays in Figure 8. Draw a pipeline diagram illustrating how the first iteration of the loop executes on this microarchitecture.** You can use microarchitectural dependency arrows to illustrate how data is transferred between instructions, although this is not required. If you do draw arrows, then they must be correct. You may want to include the first instruction of the second iteration to illustrate the impact of the branch resolution latency. **Based on this pipeline diagram estimate the execution time in units of $\tau$ for 64 iterations of this loop. Fill in the appropriate row of the table in Figure 14. You must show your work.**

## Problem 6. Reduced Register-File Ports

In this problem, you will explore the two different processor microarchitectures shown abstractly in Figures 12 and 13. The 2r1w microarchitecture shown in Figure 12 is the standard fully bypassed five-stage TinyRV1 processor discussed in lecture. In lecture, we also discussed how the number of register file ports can impact the cycle time if the register file is on the critical path. The 1r1w microarchitecture shown in Figure 13 uses a simpler register file with only one read port and one write port to reduce the cycle time.

We will examine how these two microarchitectures execute the following assembly loop. Assume the following initial register values: `x1` initially points to an array in memory that holds pointers to pairs of values; `x7` is a constant; `x9` is initially 64 (i.e., the loop executes 64 times). There are 10 instructions per iteration so the total number of instructions per program is $10 \times 64 = 640$.

```
loop:
  lw   x2, 0(x1)     # load pointer to pair of values
  lw   x3, 0(x2)     # load first element of pair
  lw   x4, 4(x2)     # load second element of pair
  addi x1, x1, 4     # pointer increment for array
  addi x9, x9, -1    # subtract one from loop counter
  add  x5, x3, x7    # add constant to first element
  add  x6, x4, x7    # add constant to second element
  sw   x6, 4(x2)     # store new second element of pair
  sw   x5, 0(x2)     # store new first element of pair
  bne  x9, x0, loop  # backwards loop branch
```

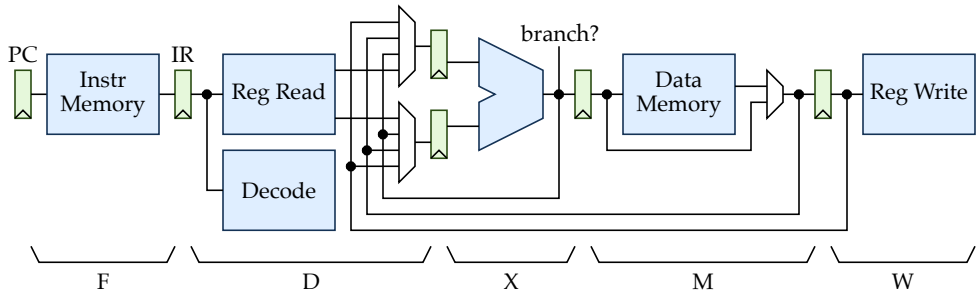



**Figure 12: 2r1w Processor Microarchitecture – Standard five-stage PARCv2 pipeline**

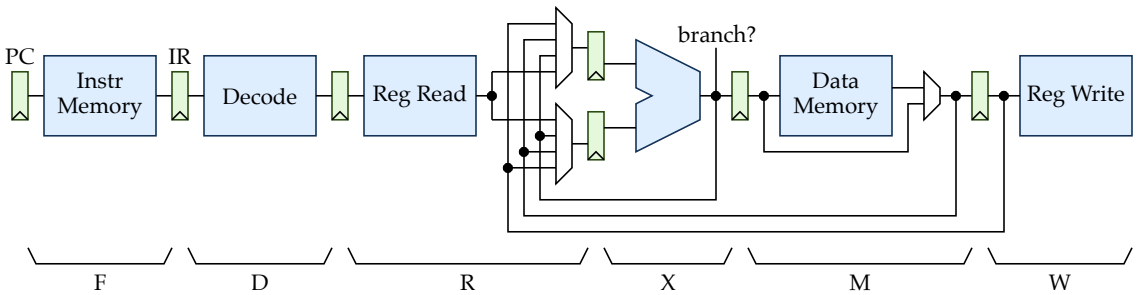



**Figure 13: 1r1w Processor Microarchitecture – Six-stage pipeline with one register-file read port**

**Part 6.A Performance of 2r1w Processor Microarchitecture**

In this part, we will calculate the execution time for the assembly loop given above running on the 2r1w microarchitecture shown in Figure 12. Assume that this microarchitecture is identical to the fully bypassed five-stage PARCv2 pipeline discussed in lecture. Assume the processor is directly connected to a magic combinational memory providing single-cycle instruction and data access latency. Remember that branches are resolved in the X stage and there is no branch delay slot. The cycle time for this microarchitecture is 1 ns and is limited by the critical path through the register file.

**Draw a pipeline diagram illustrating how the first iteration of the loop executes on the 2r1w processor microarchitecture. Use microarchitectural dependency arrows to illustrate how data is transferred between instructions.** You may want to include the first instruction of the second iteration to illustrate the impact of the branch resolution latency. **Based on this pipeline diagram estimate the execution time for 64 iterations of this loop. Fill in the appropriate row of the table in Figure 14. You must show your work.**

| | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `lw x2, 0(x1)` | F | D | X | M | W | | | | | | | | | | | | | | | | | | | | | | |
| `lw x3, 0(x2)` | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| `lw x4, 4(x2)` | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| `addi x1, x1, 4` | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| `addi x9, x9, -1` | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| `add x5, x3, x7` | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| `add x6, x4, x7` | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| `sw x6, 4(x2)` | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| `sw x5, 0(x2)` | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| `bne x9, loop` | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | | | | | | |

| Part | Microarchitecture | Instructions / Program | Avg Cycles / Instruction | Time (ns) / Cycle | Time (ns) / Program |
|---|---|---|---|---|---|
| **2.A** | **2r1w** | 640 | | 1.0 | |
| **2.B** | **1r1w** | 640 | | 0.7 | |
| **2.C** | **1r1w** | 640 | | 0.7 | |

**Figure 14: Processor Performance for Assembly Loop on Two Different Microarchitectures**

**Part 6.B Performance of 1r1w Processor Microarchitecture**

In this part, we will calculate the execution time for the assembly loop given above running on the 1r1w microarchitecture shown in Figure 13. This microarchitecture uses a simpler register file with only one read port and one write port. It is not possible to read two values from this register file in the same cycle. It is now necessary to first decode an instruction, determine the register read specifier, and only then can we read the register file. This means we must add an extra R stage to our pipeline. The new pipeline includes the following six stages: fetch (F), decode (D), register-file read (R), execute (X), memory access (M), and register-file writeback (W). Branches are still resolved in the X stage and there is still no branch delay slot.

Some instructions only read one register value (e.g., `ori`) and other instructions don't actually read any register values (e.g., `j`). For these instructions the reduced number of read ports will not be an issue. For instructions that actually do need to read two values from the register file, the control logic will need to stall in the R stage for an extra cycle. We read the first register value in the first cycle and read the second register value in the second cycle. Notice that we bypass into the R stage and not the D stage with the 1r1w microarchitecture, so assume that we stall in R if we cannot bypass and need to resolve a RAW hazards. Due to the reduced number of register file ports, the cycle time for this microarchitecture is 0.7 ns and is limited by the critical path through the register file.

There is one final issue that is subtle but very important. As shown in Figure 13, using the bypass path does not require using a read port of the register file! This means that we do not need to stall an instruction that reads two register values if one or both of the values come from the bypass paths as opposed to actually coming from the register file.

**Draw a pipeline diagram illustrating how the first iteration of the loop executes on the 1r1w processor microarchitecture. Use microarchitectural dependency arrows to illustrate how data is transferred between instructions.** You may want to include the first instruction of the second iteration to illustrate the impact of the branch resolution latency. **Based on this pipeline diagram estimate the execution time for 64 iterations of this loop. Fill in the appropriate row of the table in Figure 14. You must show your work below.**

| | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `lw x2, 0(x1)` | F | D | R | X | M | W | | | | | | | | | | | | | | | | | | | | |
| `lw x3, 0(x2)` | | | | | | | | | | | | | | | | | | | | | | | | | | |
| `lw x4, 4(x2)` | | | | | | | | | | | | | | | | | | | | | | | | | | |
| `addi x1, x1, 4` | | | | | | | | | | | | | | | | | | | | | | | | | | |
| `addi x9, x9, -1` | | | | | | | | | | | | | | | | | | | | | | | | | | |
| `add x5, x3, x7` | | | | | | | | | | | | | | | | | | | | | | | | | | |
| `add x6, x4, x7` | | | | | | | | | | | | | | | | | | | | | | | | | | |
| `sw x6, 4(x2)` | | | | | | | | | | | | | | | | | | | | | | | | | | |
| `sw x5, 0(x2)` | | | | | | | | | | | | | | | | | | | | | | | | | | |
| `bne x9, loop` | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | | | | | |

**Part 6.C Software Scheduling for 1r1w Processor Microarchitecture**

Software can improve the performance of the assembly loop by rescheduling the instructions to avoid stalls due to structural and data hazards. Note that this does not imply changing the instruction set architecture; instead, the hardware architect would provide optimization guidelines to the compiler writer to enable performance optimizations for the specific 1r1w microarchitecture described in the previous section. In the space below, **carefully craft succinct optimization guidelines for compiler writers that if followed will improve the performance of assembly code running on the 1r1w microarchitecture**. *Your optimization guidelines should mitigate the impact of stalls due to structural and data hazards.* **Explain your optimization guidelines.**

Use your optimization guidelines to reschedule the assembly loop. **Show the optimized static instruction sequence below. Explain your optimizations.** Note that we have provided space for a pipeline diagram on the next page but you are not required to use it. Do not remove or add new assembly instructions; simply reschedule the assembly instructions given in the loop. **Estimate the execution time for 64 iterations of this loop. Fill in the appropriate row of the table in Figure 14. You must show your work.**

Note that we have provided space for a pipeline diagram below but you are not required to use it.