ECE 2300 Digital Logic and Computer Organization
Fall 2024

Topic 12: Pipelined Processors

School of Electrical and Computer Engineering
Cornell University

revision: 2024-11-19-10-52

1 High-Level Idea for Single-Cycle Processors **3**

1.1. Transactions and Steps 5

1.2. Technology Constraints 6

1.3. First-Order Performance Equation 6

2 Two-Stage Pipelined Processor **7**

2.1. RAW Data Hazards Through Registers 9

2.2. RAW Data Hazards → Software Scheduling 11

2.3. RAW Data Hazards → Hardware Stalling 12

2.4. RAW Data Hazards → Hardware Bypassing 13

2.5. RAW Data Hazards Through Memory 16

2.6. Control Hazards 17

2.7. Control Hazards → Software Scheduling 20

2.8. Control Hazards → Hardware Speculation 21

2.9. Analyzing Performance 23

3 Five-Stage Pipelined Processor **27**

3.1. RAW Data Hazards Through Registers 28

3.2. RAW Data Hazards → Hardware Stalling 29

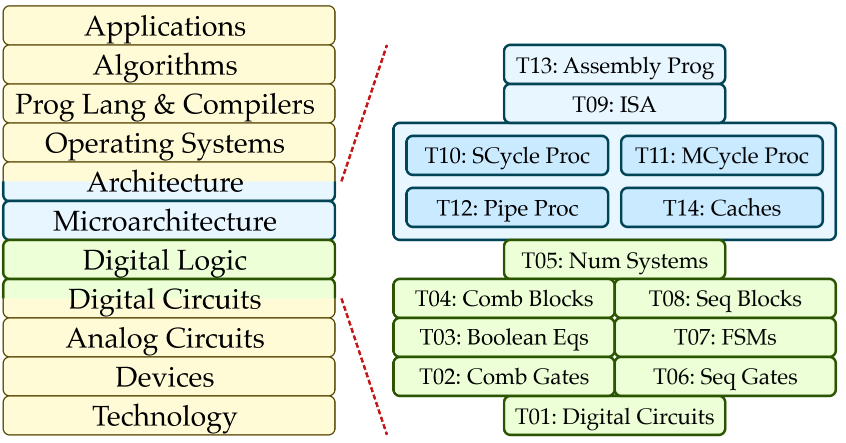
3.3. RAW Data Hazards → Hardware Bypassing 30

3.4. RAW Data Hazards Through Memory 34

3.5. Control Hazards 35

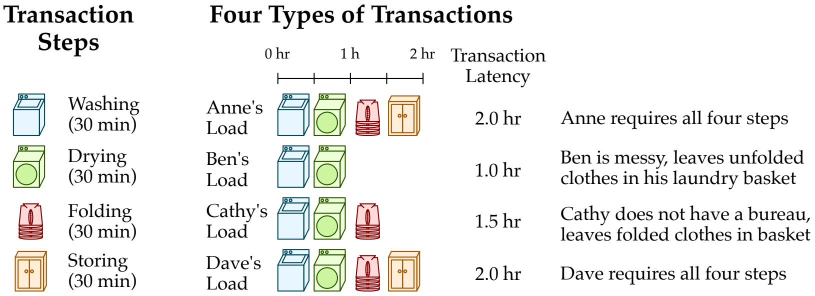
3.6. Control Hazards → Hardware Speculation 36

3.7. Analyzing Performance 38

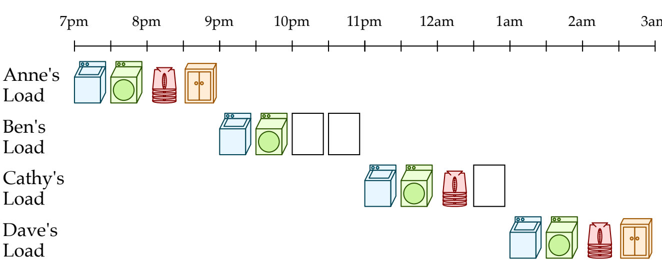


Copyright © 2024 Christopher Batten. All rights reserved. This handout was prepared by Prof. Christopher Batten at Cornell University for ECE 2300 / ENGRD 2300 Digital Logic and Computer Organization. Download and use of this handout is permitted for individual educational non-commercial purposes only. Redistribution either in part or in whole via both commercial or non-commercial means requires written permission.

1. High-Level Idea for Single-Cycle Processors

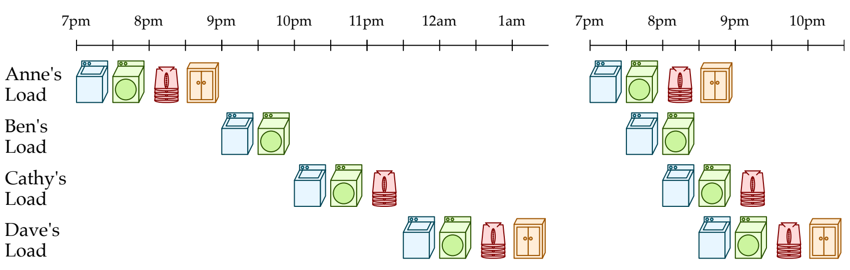


Fixed Time Slot Laundry (Single-Cycle Processors)



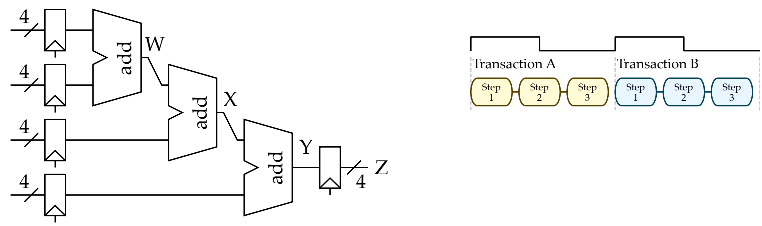
Variable Time Slot Laundry (Multi-Cycle Processors)

Pipelined Laundry

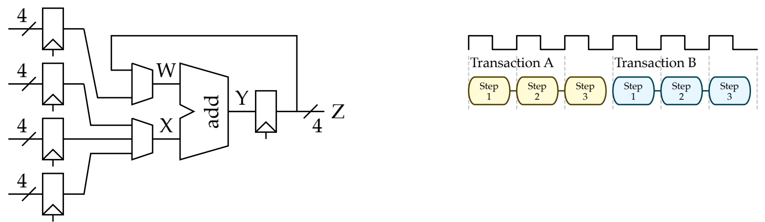


Transaction						Quad Adder	Result
Transaction A	3	1	2	0	⇨		6
Transaction B	5	4	0	6	⇨		15
Transaction C	4	5	3	1	⇨		13
Transaction D	2	3	5	4	⇨		14

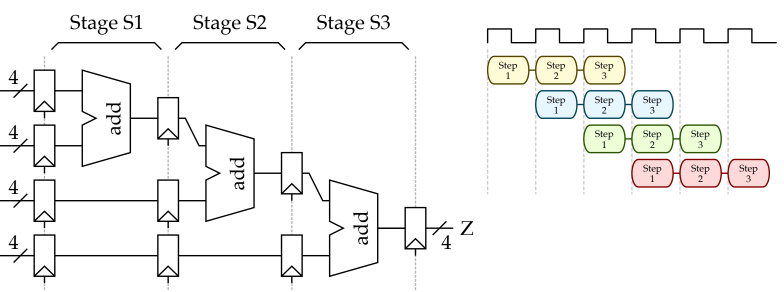
Single-Cycle Quad Adder



Multi-Cycle Quad Adder



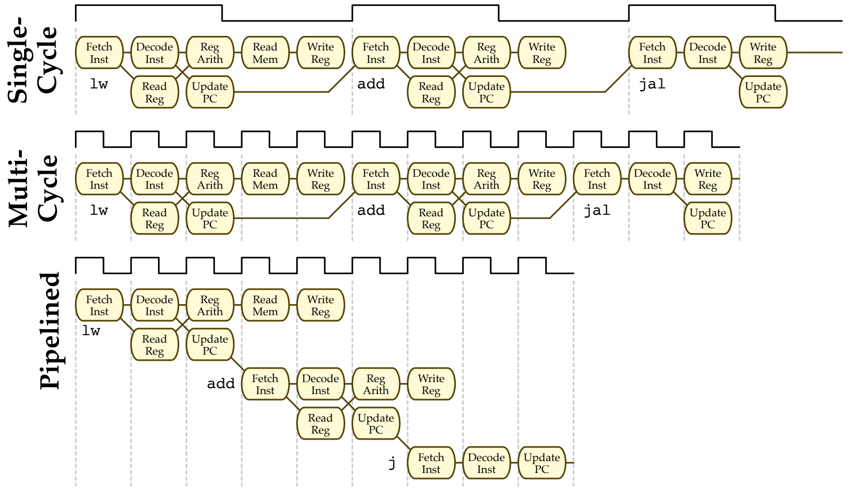
Pipelined Quad Adder



1.1. Transactions and Steps

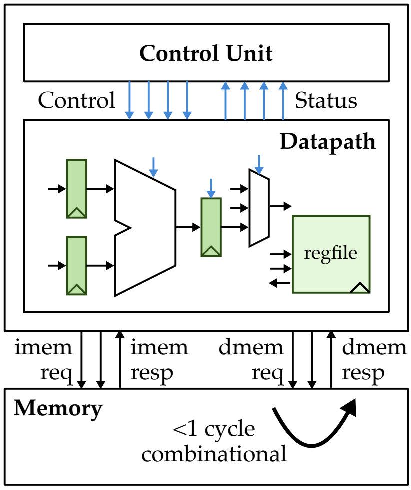
- We can think of each instruction as a transaction
- Executing a transaction involves a sequence of steps

	add	addi	mul	lw	sw	jal	jr	bne
Fetch Instruction	✓	✓	✓	✓	✓	✓	✓	✓
Decode Instruction	✓	✓	✓	✓	✓	✓	✓	✓
Read Registers	✓	✓	✓	✓	✓		✓	✓
Register Arithmetic	✓	✓	✓	✓	✓			✓
Read Memory				✓				
Write Memory					✓			
Write Registers	✓	✓	✓	✓		✓		
Update PC	✓	✓	✓	✓	✓	✓	✓	✓



1.2. Technology Constraints

- Assume modern technology where logic is cheap and fast (e.g., fast integer ALU)
- Assume multi-ported register files with a reasonable number of ports are feasible
- Assume small amount of very fast memory (caches) backed by large, slower memory



1.3. First-Order Performance Equation

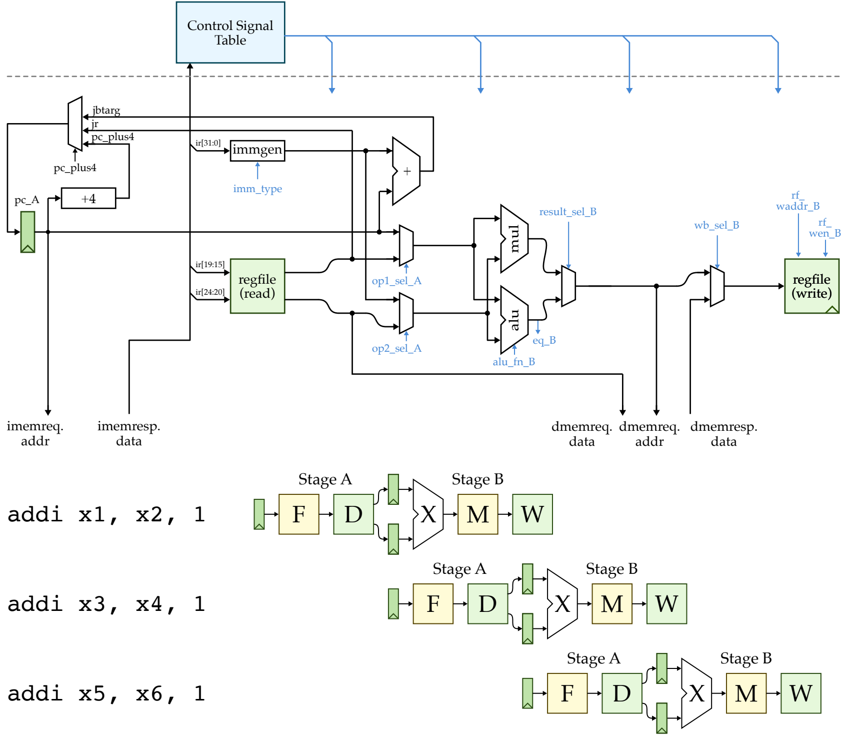
$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Avg Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

- Instructions / program depends on source code, compiler, ISA
- Avg cycles / instruction (CPI) depends on ISA, microarchitecture
- Time / cycle depends upon microarchitecture and implementation

Microarchitecture	CPI	Cycle Time
Single-Cycle Processor	1	long
Multi-Cycle Processor	>1	short
Pipelined Processor	≈ 1	short

2. Two-Stage Pipelined Processor

- Incrementally develop an unpipelined datapath
- Keep data flowing from left to right
- Position control signal table early in the diagram
- Divide datapath/control into stages by inserting pipeline registers
- Keep the pipeline stages roughly balanced
- Forward arrows should avoid “skipping” pipeline registers
- Backward arrows will need careful consideration



Pipeline diagrams

`addi x1, x2, 1`														
`addi x3, x4, 1`														
`addi x5, x6, 1`														

What would be the total execution time if these three instructions were repeated 10 times?

Hazards occur when instructions interact with each other in pipeline

- RAW Data Hazards: An instruction depends on a data value produced by an earlier instruction
- Control Hazards: Whether or not an instruction should be executed depends on a control decision made by an earlier instruction
- Structural Hazards: An instruction in the pipeline needs a resource being used by another instruction in the pipeline
- WAW and WAR Name Hazards: An instruction in the pipeline is writing a register that an earlier instruction in the pipeline is either writing or reading

Stalling and squashing instructions

- Stalling: An instruction *originates* a stall due to a hazard, causing all instructions earlier in the pipeline to also stall. When the hazard is resolved, the instruction no longer needs to stall and the pipeline starts flowing again.
- Squashing: An instruction *originates* a squash due to a hazard, and squashes all previous instructions in the pipeline (but not itself). We restart the pipeline to begin executing a new instruction sequence.

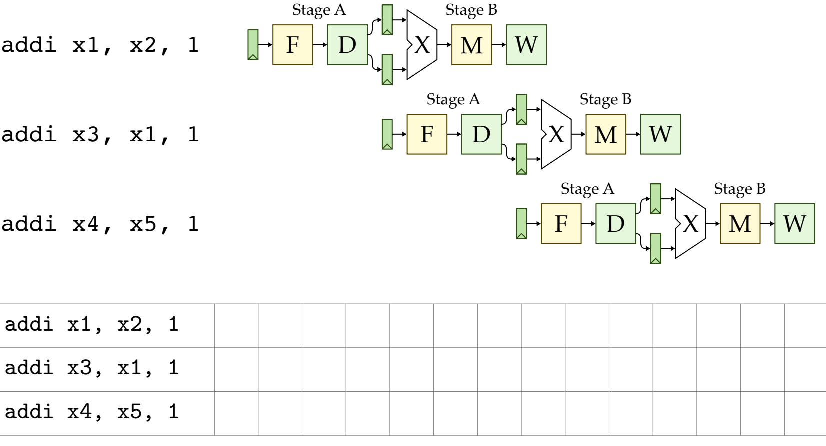
2.1. RAW Data Hazards Through Registers

RAW data hazards occur when one instruction depends on a data value produced by a preceding instruction still in the pipeline. We use architectural dependency arrows to illustrate RAW dependencies in assembly code sequences.

```
addi x1, x2, 1

addi x3, x1, 1

addi x4, x5, 1
```

Using pipeline diagrams to illustrate RAW hazards

We use microarchitectural dependency arrows to illustrate RAW hazards on pipeline diagrams.



addi x1, x2, 1														
addi x3, x1, 1														
addi x4, x5, 1														

Approaches to resolving data hazards

- Software Scheduling: Expose data hazards in ISA forcing assembly level programmer or compiler to explicitly avoid scheduling instructions that would create hazards
- Hardware Stalling: Hardware includes control logic that freezes later instructions until earlier instruction has finished producing data value; software scheduling can still be used to avoid stalling (i.e., software scheduling for performance)
- Hardware Bypassing/Forwarding: Hardware allows values to be sent from an earlier instruction to a later instruction before the earlier instruction has left the pipeline
- Hardware Scheduling: Hardware dynamically schedules instructions to avoid RAW hazards, potentially allowing instructions to execute out of order
- Hardware Speculation: Hardware guesses that there is no hazard and allows later instructions to potentially read invalid data; detects when there is a problem, squashes and then re-executes instructions that operated on invalid data

2.2. RAW Data Hazards → Software Scheduling

- ISA specifies exactly how many instructions must be between a register write and a later read of that register
- Assembly level programmer or compiler must insert independent instructions to delay the read of earlier write
- If cannot find any independent instructions, must insert instructions do nothing (`nops`) to delay read of earlier write. These `nops` count as real instructions increasing instructions per program.
- If hazard is exposed in ISA, software scheduling is required for correctness! A scheduling mistake can cause undefined behavior.

```
addi x1, x2, 1

addi x3, x1, 1

addi x4, x5, 1
```

Resolving RAW hazards using software scheduling

addi x1, x2, 1														
addi x0, x0, 0														
addi x3, x1, 1														
addi x4, x5, 1														

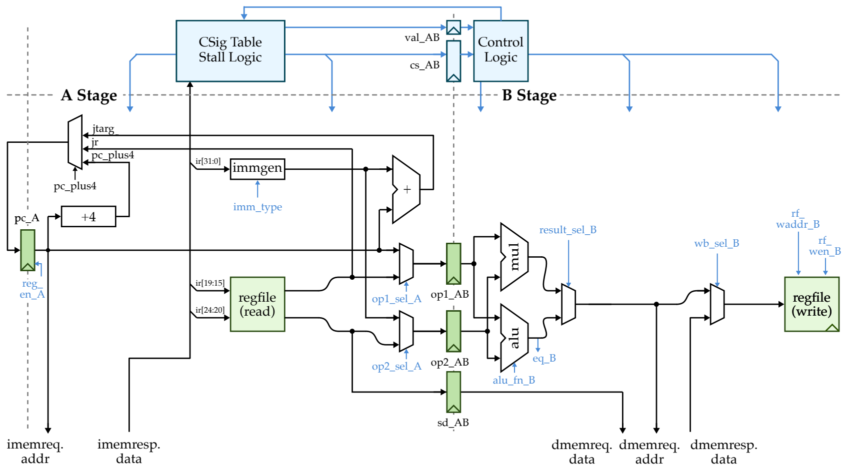
addi x1, x2, 1														
addi x4, x5, 1														
addi x3, x1, 1														

2.3. RAW Data Hazards → Hardware Stalling

- Hardware includes control logic that freezes later instructions (in front of pipeline) until earlier instruction (in back of pipeline) has finished producing data value.
- Software scheduling is not required for correctness, but can improve performance! Programmer or compiler schedules independent instructions to reduce the number of cycles spent stalling.

addi x1, x2, 1														
addi x3, x1, 1														
addi x4, x5, 1														

Modifications to datapath/control to support hardware stalling

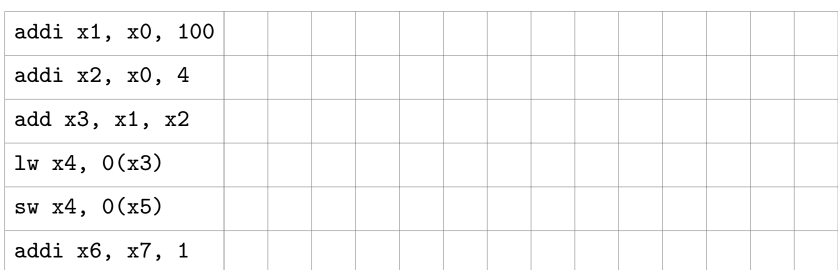


Deriving the stall signal

	add	addi	mul	lw	sw	jal	jr	bne
rs1_en								
rs2_en								
rf_wen								

```
stall_waddr_B_rs1_A = rs1_en_A && val_B && rf_wen_B
  && (inst_rs1_A == rf_waddr_B) && (rf_waddr_B != 0)

stall_waddr_B_rs2_A = rs2_en_A && val_B && rf_wen_B
  && (inst_rs2_A == rf_waddr_B) && (rf_waddr_B != 0)

stall_A = stall_waddr_B_rs1_A || stall_waddr_B_rs2_A;
```

Draw the pipeline diagram assuming RAW hazards are resolved with hardware stalling



addi x1, x0, 100														
addi x2, x0, 4														
add x3, x1, x2														
lw x4, 0(x3)														
sw x4, 0(x5)														
addi x6, x7, 1														

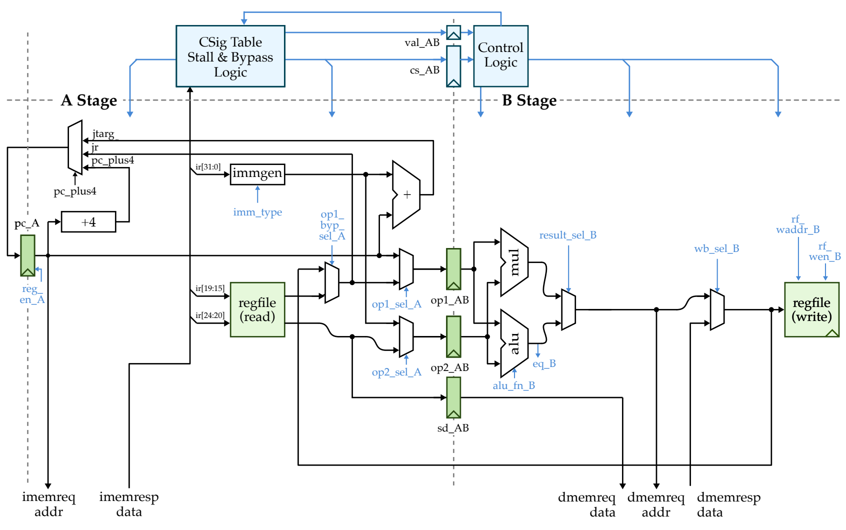
2.4. RAW Data Hazards → Hardware Bypassing

Hardware allows values to be sent from an earlier instruction (in back of pipeline) to a later instruction (in front of pipeline) before the earlier instruction has left the pipeline. Sometimes called “forwarding”.

Pipeline diagram showing hardware bypassing for RAW data hazards

`addi x1, x2, 1`														
`addi x3, x1, 1`														
`addi x4, x5, 1`														

Adding single bypass path to support limited hardware bypassing



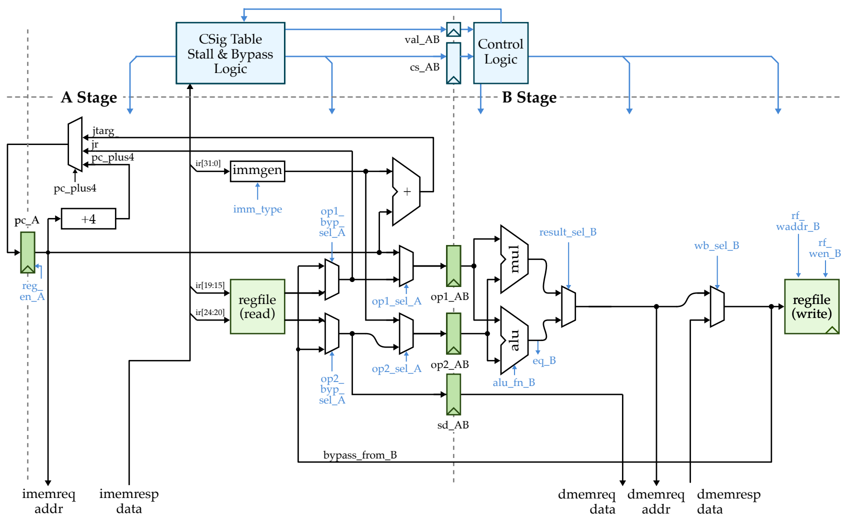
Deriving the bypass and stall signals

```
stall_waddr_B_rs1_A = 0
bypass_waddr_B_rs1_A = rs1_en_A && val_B && rf_wen_B
  && (inst_rs1_A == rf_waddr_B) && (rf_waddr_B != 0)
```

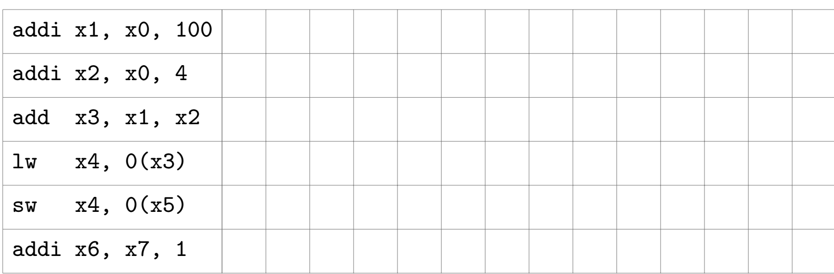
Pipeline diagram showing multiple hardware bypass paths

addi x2, x10, 1														
addi x2, x11, 1														
addi x1, x2, 1														
addi x3, x4, 1														
addi x5, x3, 1														
add x6, x1, x3														
sw x5, 0(x1)														

Adding all bypass path to support full hardware bypassing



Draw the pipeline diagram assuming RAW hazards are resolved with hardware bypassing



`addi x1, x0, 100`														
`addi x2, x0, 4`														
`add  x3, x1, x2`														
`lw   x4, 0(x3)`														
`sw   x4, 0(x5)`														
`addi x6, x7, 1`														

2.5. RAW Data Hazards Through Memory

So far we have only studied RAW data hazards through registers, but we must also carefully consider RAW data hazards through memory.

```
sw x1, 0(x2)
lw x3, 0(x4) # RAW dependency occurs if R[x2] == R[x4]
```

`sw x1, 0(x2)`														
`lw x3, 0(x4)`														

2.6. Control Hazards

Control hazards occur when whether or not an instruction should be executed depends on a control decision made by an earlier instruction. We use architectural dependency arrows to illustrate control dependencies in assembly code sequences.

Static Instr Sequence

```
     addi x1, x0, 1
     jal  x0, foo
     addi x2, x0, 1
foo: bne  x0, x1, bar
     addi x3, x0, 1
bar: addi x4, x0, 1
```

Dynamic Instr Sequence

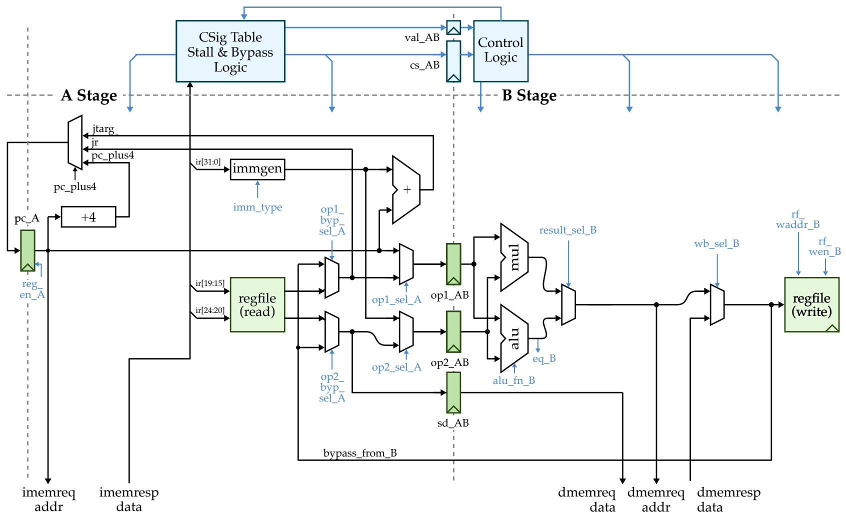
```
addi x1, x0, 1
jal  x0, foo
bne  x0, x1, bar
addi x4, x0, 1
```

Using pipeline diagrams to illustrate control hazards

We use microarchitectural dependency arrows to illustrate control hazards on pipeline diagrams.

addi x1, x0, 1														
jal x0, foo														
bne x0, x1, bar														
addi x4, x0, 1														

What hardware would be required to make the vertical microarchitectural dependency arrow possible?



Approaches to resolving control hazards

- Software Scheduling: Expose control hazards in ISA forcing assembly level programmer or compiler to explicitly avoid scheduling instructions that would create hazards
- Hardware Speculation: Hardware guesses which way the control flow will go and potentially fetches incorrect instructions; detects when there is a problem and re-executes instructions that are along the correct control flow
- Software Predication: Assembly level programmer or compiler converts control flow into data flow by using instructions that conditionally execute based on a data value
- Software Hints: Assembly level programmer or compiler provides hints about whether a conditional branch will be taken or not taken, and hardware can use these hints for more efficient hardware speculation

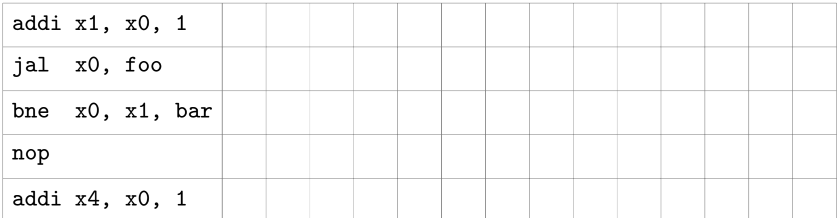
2.7. Control Hazards → Software Scheduling

Expose branch delay slots as part of the instruction set. Branch delay slots are instructions that follow a jump or branch and are *always* executed regardless of whether a jump or branch is taken or not taken. Compiler tries to insert useful instructions, otherwise inserts `nop`s.

```
     addi x1, x0, 1
     jal  x0, foo
     addi x2, x0, 1
foo: bne  x0, x1, bar
     nop
     addi x3, x0, 1
bar: addi x4, x0, 1
```

Assume we modify the TinyRV1 instruction set to specify that BNE instructions have a single-instruction branch delay slot (i.e., one instruction after a BNE is always executed).

Pipeline diagram showing using branch delay slots for control hazards



`addi x1, x0, 1`														
`jal  x0, foo`														
`bne  x0, x1, bar`														
`nop`														
`addi x4, x0, 1`														

2.8. Control Hazards → Hardware Speculation

Hardware guesses which way the control flow will go and potentially fetches incorrect instructions; detects when there is a problem and re-executes instructions the instructions that are along the correct control flow. We will only consider a simple branch prediction scheme where the hardware always predicts not taken.

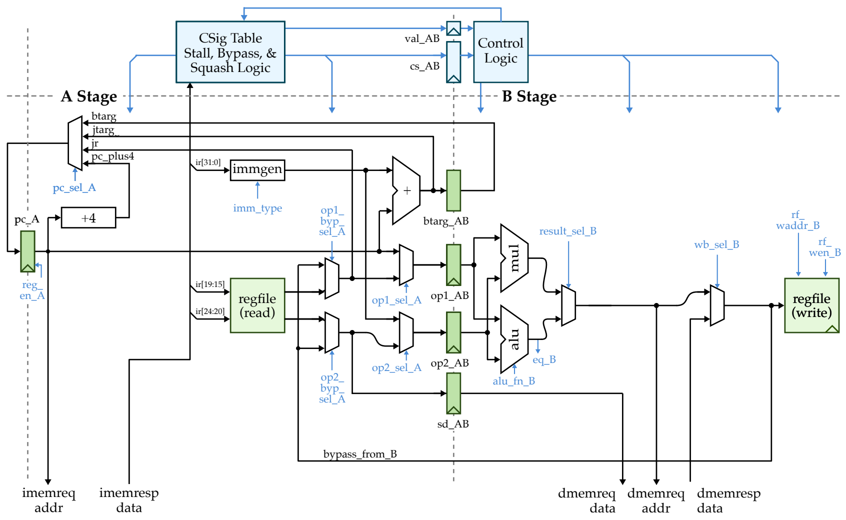
Pipeline diagram when branch is not taken

`addi x1, x0, 1`														
`jal  x0, foo`														
`bne  x0, x1, bar`														
`addi x3, x0, 1`														
`addi x4, x0, 1`														

Pipeline diagram when branch is taken

`addi x1, x0, 1`														
`jal  x0, foo`														
`bne  x0, x1, bar`														
`addi x3, x0, 1`														
`addi x4, x0, 1`														

Modifications to datapath/control to support hardware speculation

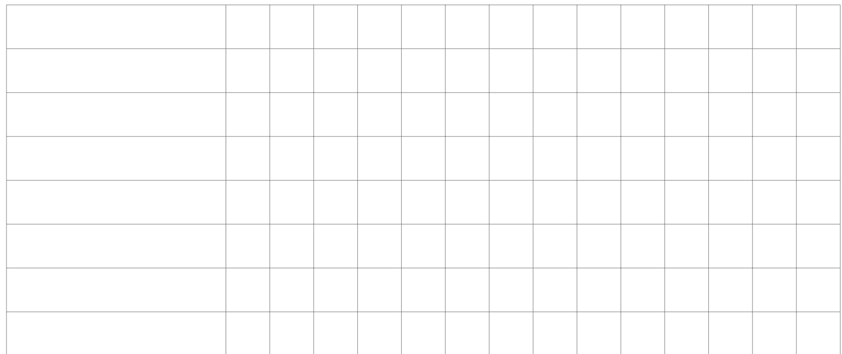


Deriving the squash signals

```
squash_A = val_B && (op_B == bne) && !eq_B
```

Draw the pipeline diagram assuming control hazards are resolved with hardware speculation

```
     addi x1, x0, 0
     bne  x1, x0, foo
     addi x2, x0, 1
     addi x3, x0, 1
foo: bne  x2, x0, bar
     addi x4, x0, 1
     addi x5, x0, 1
bar: addi x6, x0, 1
```

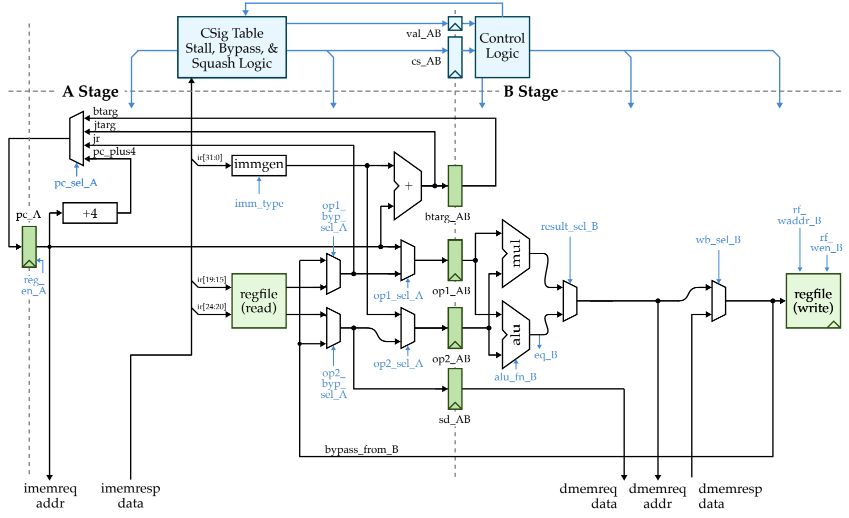


2.9. Analyzing Performance

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycles}}$$

- Instructions / program depends on source code, compiler, ISA
- Cycles / instruction (CPI) depends on ISA, microarchitecture
- Time / cycle depends upon microarchitecture and implementation

Estimating minimum clock period (cycle time)



	t_{pd}
32-bit 2-to-1 Mux	4τ
32-bit 4-to-1 Mux	8τ
32-bit Adder	60τ
32-bit ALU	64τ
32-bit Multiplier	100τ
32-bit +4 Unit	30τ
ImmGen Unit	12τ
32-bit Reg Clk-to-Q	9τ
32-bit Reg Setup	10τ
Register File Read	25τ
Register File Setup	20τ
Memory Read	120τ
Memory Setup	120τ

Estimating execution time

How long in units of τ will it take to execute the vector-vector add program assuming `n` is 64?

Pseudo-Code

```
1 for i in range(n):
2   dest[i] = src0[i] + src1[i]
```

Assembly Code

```
1  # addr(dest[i]):x1, addr(src0[i]):x2
2  # addr(src1[i]):x3, n:x4
3 loop:
4  lw    x5, 0(x1)
5  lw    x6, 0(x2)
6  add   x7, x5, x6
7  sw    x7, 0(x3)
8  addi  x1, x1, 4
9  addi  x2, x2, 4
10 addi  x3, x3, 4
11 addi  x4, x4, -1
12 bne   x4, x0, loop
```

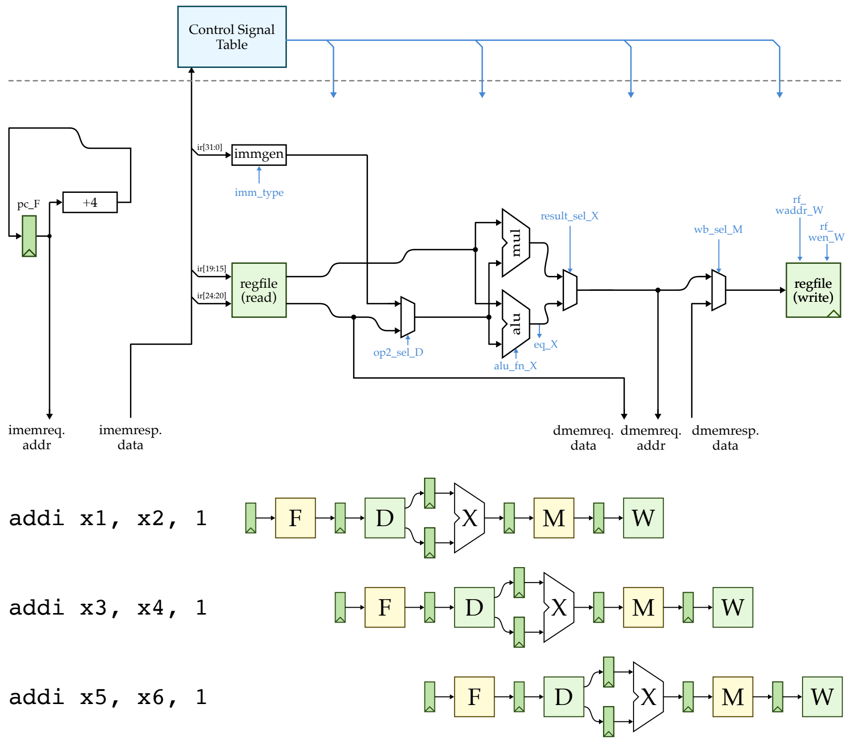
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
`lw x5, 0(x1)`														
`lw x6, 0(x2)`														
`add x7, x5, x6`														
`sw x7, 0(x3)`														
`addi x1, x1, 4`														
`addi x2, x2, 4`														
`addi x3, x3, 4`														
`addi x4, x4, -1`														
`bne x4, x0, loop`														
`opA`														
`lw x5, 0(x1)`														
`lw x6, 0(x2)`														
`add x7, x5, x6`														
`sw x7, 0(x3)`														

Results for vector-vector add example

Microarchitecture	Inst/Prog	Cycle/Inst	Time/Cycle	Exec Time
Single-Cycle	576	1.0	366 τ	210 kτ
Multi-Cycle	576	6.7	231 τ	886 kτ
2-Stage Pipelined	576			

3. Five-Stage Pipelined Processor

- Incrementally develop an unpipelined datapath
- Start with just arithmetic and memory instructions
- Keep data flowing from left to right
- Position control signal table early in the diagram
- Divide datapath/control into stages by inserting pipeline registers
- Keep the pipeline stages roughly balanced
- Forward arrows should avoid "skipping" pipeline registers



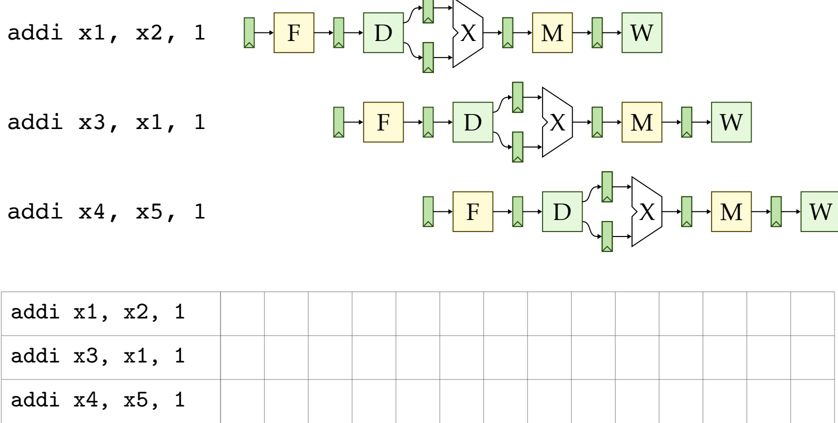
Pipeline diagrams

addi x1, x2, 1														
addi x3, x4, 1														
addi x5, x6, 1														

What would be the total execution time if these three instructions were repeated 10 times?

3.1. RAW Data Hazards Through Registers

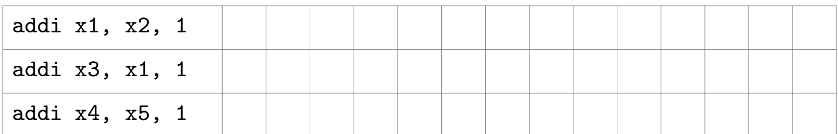
RAW data hazards occur when one instruction depends on a data value produced by a preceding instruction still in the pipeline.



addi x1, x2, 1														
addi x3, x1, 1														
addi x4, x5, 1														

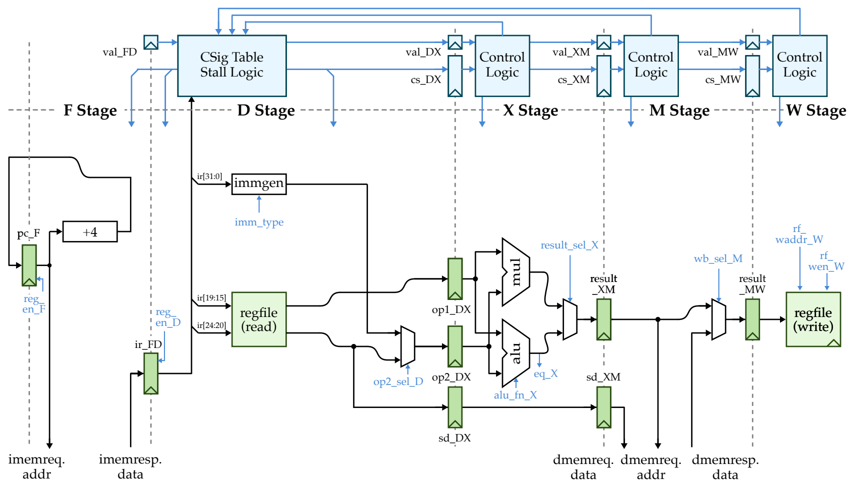
3.2. RAW Data Hazards → Hardware Stalling

- Hardware includes control logic that freezes later instructions (in front of pipeline) until earlier instruction (in back of pipeline) has finished producing data value.



addi x1, x2, 1														
addi x3, x1, 1														
addi x4, x5, 1														

Modifications to datapath/control to support hardware stalling



Deriving the stall signal

```
stall_waddr_X_rs1_D =
  val_D && rs1_en_D && val_X && rf_wen_X
        && (inst_rs1_D == rf_waddr_X) && (rf_waddr_X != 0)

stall_waddr_M_rs1_D =
  val_D && rs1_en_D && val_M && rf_wen_M
        && (inst_rs1_D == rf_waddr_M) && (rf_waddr_M != 0)

stall_waddr_W_rs1_D =
  val_D && rs1_en_D && val_W && rf_wen_W
        && (inst_rs1_D == rf_waddr_W) && (rf_waddr_W != 0)

... similar for stall signals for rs2 source register ...

stall_D = val_D
        && (    stall_waddr_X_rs1_D || stall_waddr_X_rs2_D
             || stall_waddr_M_rs1_D || stall_waddr_M_rs2_D
             || stall_waddr_W_rs1_D || stall_waddr_W_rs2_D )

stall_F = stall_D
```

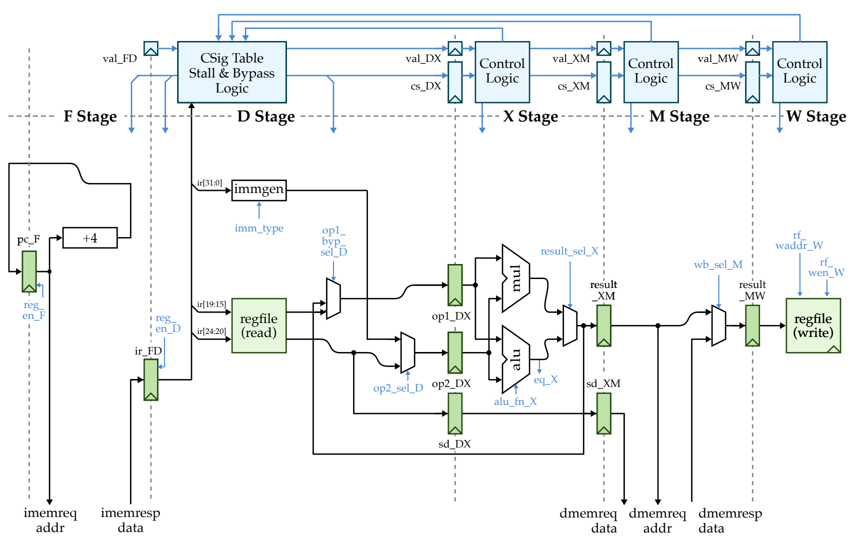
3.3. RAW Data Hazards → Hardware Bypassing

Hardware allows values to be sent from an earlier instruction (in back of pipeline) to a later instruction (in front of pipeline) before the earlier instruction has left the pipeline. Sometimes called “forwarding”.

Pipeline diagram showing hardware bypassing for RAW data hazards

addi x1, x2, 1														
addi x3, x1, 1														
addi x4, x5, 1														

Adding single bypass path to support limited hardware bypassing



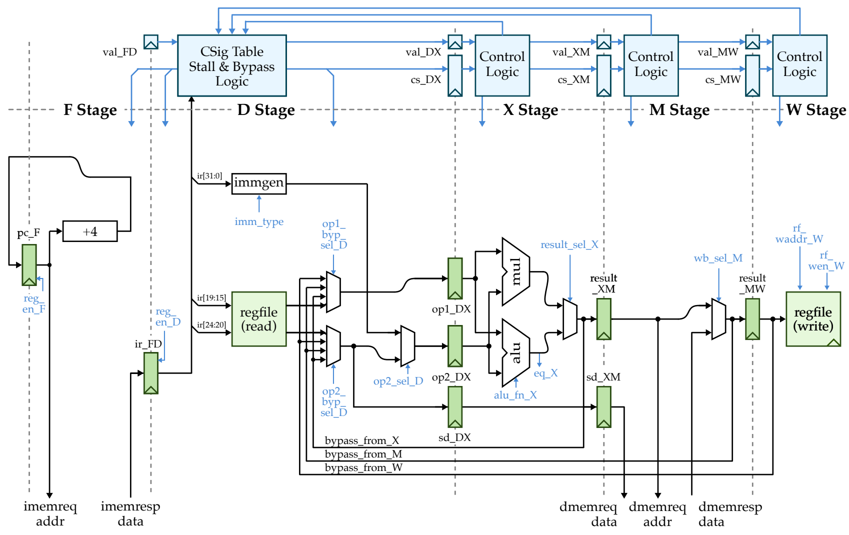
Deriving the bypass and stall signals

```
stall_waddr_X_rs1_D  = 0
bypass_waddr_X_rs1_D =
  val_D && rs1_en_D && val_X && rf_wen_X
        && (inst_rs1_D == rf_waddr_X) && (rf_waddr_X != 0)
```

Pipeline diagram showing multiple hardware bypass paths

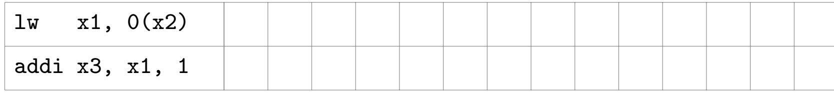
`addi x2, x10, 1`														
`addi x2, x11, 1`														
`addi x1, x2, 1`														
`addi x3, x4, 1`														
`addi x5, x3, 1`														
`add  x6, x1, x3`														
`sw   x5, 0(x1)`														

Adding all bypass path to support full hardware bypassing



Handling load-use RAW dependencies

ALU-use latency is only one cycle, but load-use latency is two cycles.



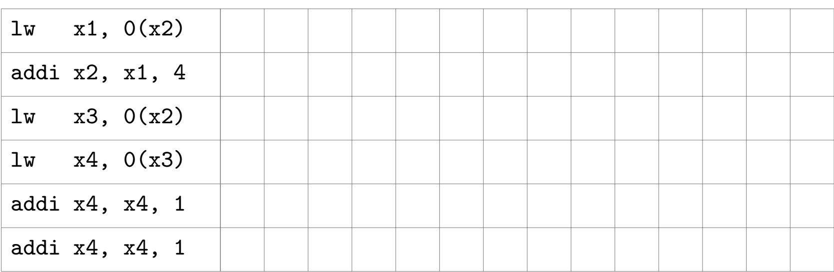
lw x1, 0(x2)														
addi x3, x1, 1														

lw x1, 0(x2)														
addi x3, x1, 1														

```
stall_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)

stall_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)

stall_D =
  val_D && ( stall_load_use_X_rs1_D || stall_load_use_X_rs2_D )

bypass_waddr_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)

bypass_waddr_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)
```

Pipeline diagram for simple assembly sequence

Draw a pipeline diagram illustrating how the following assembly sequence would execute on a fully bypassed pipelined TinyRV1 processor. Include microarchitectural dependency arrows to illustrate how data is transferred along various bypass paths.



lw x1, 0(x2)														
addi x2, x1, 4														
lw x3, 0(x2)														
lw x4, 0(x3)														
addi x4, x4, 1														
addi x4, x4, 1														

3.4. RAW Data Hazards Through Memory

So far we have only studied RAW data hazards through registers, but we must also carefully consider RAW data hazards through memory.

```
sw x1, 0(x2)
lw x3, 0(x4) # RAW dependency occurs if R[x2] == R[x4]
```

sw x1, 0(x2)														
lw x3, 0(x4)														

3.5. Control Hazards

Control hazards occur when whether or not an instruction should be executed depends on a control decision made by an earlier instruction.

Static Instr Sequence

```
     addi x1, x0, 1
     jal  x0, foo
     opA
     opB
foo: addi x2, x3, 1
     bne  x0, x1, bar
     opC
     opD
     opE
bar: addi x4, x5, 1
```

Dynamic Instr Sequence

```
addi x1, x0, 1
jal  x0, foo
addi x2, x3, 1
bne  x0, x1, bar
addi x4, x5, 1
```

addi x1, x0, 1														
jal x0, foo														
addi x2, x3, 1														
bne x0, x1, bar														
addi x4, x5, 1														

3.6. Control Hazards → Hardware Speculation

Hardware guesses which way the control flow will go and potentially fetches incorrect instructions; detects when there is a problem and re-executes instructions the instructions that are along the correct control flow. We will only consider a simple branch prediction scheme where the hardware always predicts not taken.

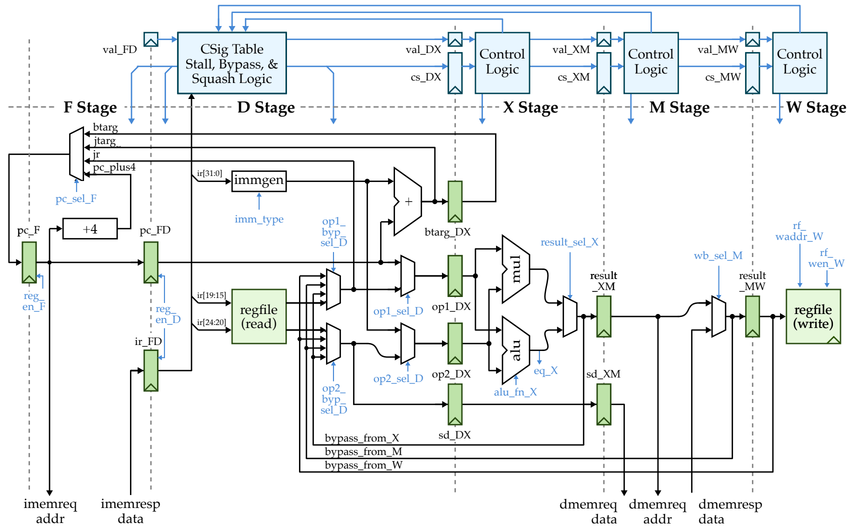
Pipeline diagram when branch is not taken

`addi x1, x0, 1`														
`jal  x0, foo`														
`opA`														
`addi x2, x3, 1`														
`bne  x0, x1, bar`														
`opC`														
`opD`														

Pipeline diagram when branch is taken

`addi x1, x0, 1`														
`jal  x0, foo`														
`opA`														
`addi x2, x3, 1`														
`bne  x0, x1, bar`														
`opC`														
`opD`														
`addi x4, x5, 1`														

Modifications to datapath/control to support hardware speculation



Deriving the squash signals

```
squash_D = val_X && (op_X == bne) && !eq_X
squash_F = squash_D || (val_D && ((op_D == jal) || (op_D == jr)))
```

Important: PC select logic must give priority to older instructions (i.e., prioritize branches over jumps)! *Good exam question?*

Draw the pipeline diagram assuming control hazards are resolved with hardware speculation

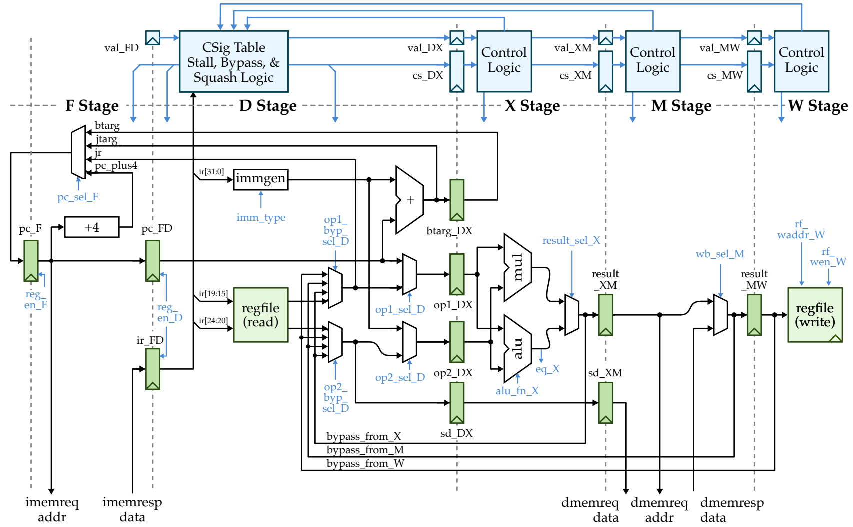
```
     addi x1, x0, 0
     bne  x1, x0, foo
     addi x2, x0, 1
     addi x3, x0, 1
foo: bne  x2, x0, bar
     addi x4, x0, 1
     addi x5, x0, 1
bar: addi x6, x0, 1
```

3.7. Analyzing Performance

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycles}}$$

- Instructions / program depends on source code, compiler, ISA
- Cycles / instruction (CPI) depends on ISA, microarchitecture
- Time / cycle depends upon microarchitecture and implementation

Estimating minimum clock period (cycle time)



	t_{pd}
32-bit 2-to-1 Mux	4τ
32-bit 4-to-1 Mux	8τ
32-bit Adder	60τ
32-bit ALU	64τ
32-bit Multiplier	100τ
32-bit +4 Unit	30τ
ImmGen Unit	12τ
32-bit Reg Clk-to-Q	9τ
32-bit Reg Setup	10τ
Register File Read	25τ
Register File Setup	20τ
Memory Read	120τ
Memory Setup	120τ

Estimating execution time

How long in units of τ will it take to execute the vector-vector add program assuming `n` is 64?

Pseudo-Code

```
1 for i in range(n):
2   dest[i] = src0[i] + src1[i]
```

Assembly Code

```
1  # addr(dest[i]):x1, addr(src0[i]):x2
2  # addr(src1[i]):x3, n:x4
3 loop:
4  lw    x5, 0(x1)
5  lw    x6, 0(x2)
6  add   x7, x5, x6
7  sw    x7, 0(x3)
8  addi  x1, x1, 4
9  addi  x2, x2, 4
10 addi  x3, x3, 4
11 addi  x4, x4, -1
12 bne   x4, x0, loop
```

lw																					
lw																					
add																					
sw																					
addi																					
addi																					
addi																					
addi																					
bne																					
opA																					
opB																					
lw																					
lw																					

Results for vector-vector add example

Microarchitecture	Inst/Prog	Cycle/Inst	Time/Cycle	Exec Time
Single-Cycle	576	1.0	366 τ	210 kτ
Multi-Cycle	576	6.7	231 τ	886 kτ
2-Stage Pipelined	576	1.1	223 τ	143 kτ
5-Stage Pipelined	576			