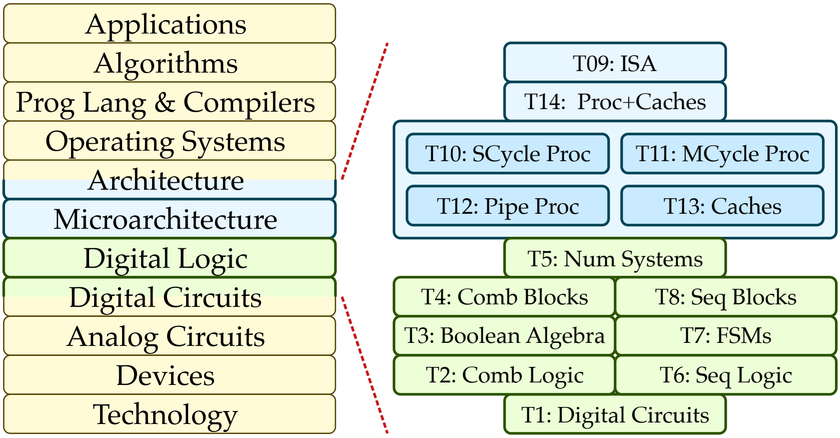
ECE 2300 Digital Logic and Computer Organization Fall 2025

Topic 12: Pipelined Processors

School of Electrical and Computer Engineering
Cornell University

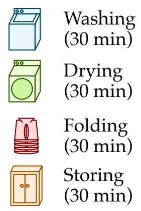
revision: 2025-11-20-11-00



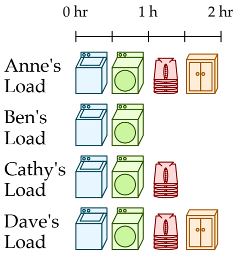
Copyright © 2025 Christopher Batten. All rights reserved. This handout was prepared by Prof. Christopher Batten at Cornell University for ECE 2300 / ENGRD 2300 Digital Logic and Computer Organization. Download and use of this handout is permitted for individual educational non-commercial purposes only. 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

Transaction Steps

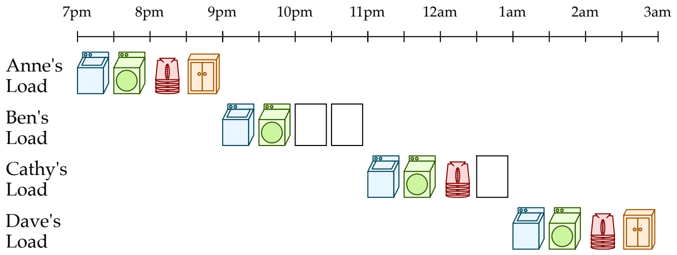


Four Types of Transactions

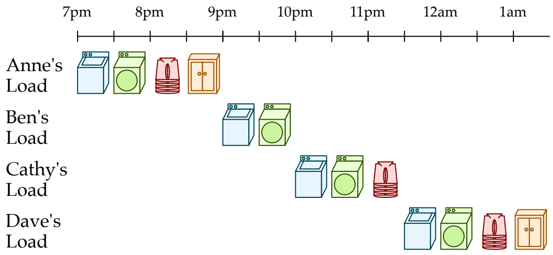


	Transaction Latency	
Anne's Load	2.0 hr	Anne requires all four steps
Ben's Load	1.0 hr	Ben is messy, leaves unfolded clothes in his laundry basket
Cathy's Load	1.5 hr	Cathy does not have a bureau, leaves folded clothes in basket
Dave's Load	2.0 hr	Dave requires all four steps

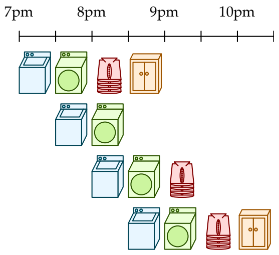
Fixed Time Slot Laundry (Single-Cycle Processors)



Variable Time Slot Laundry (Multi-Cycle Processors)



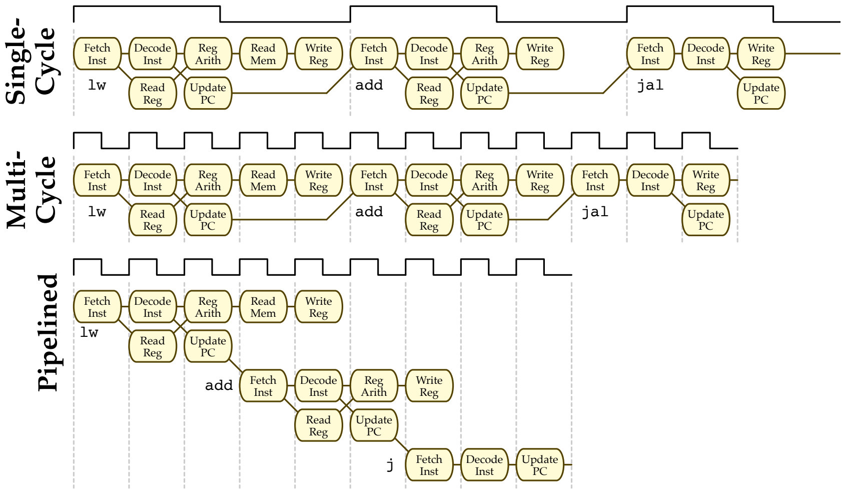
Pipelined Laundry



1.1. Transactions and Steps

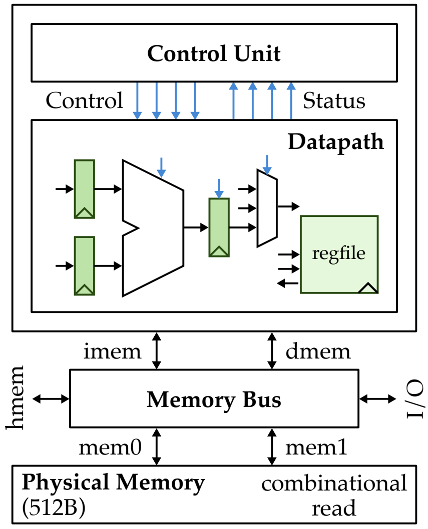
- We can think of each instruction as a transaction
- Executing a transaction involves a sequence of steps

	`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 and System Constraints

- Assume modern technology where logic is cheap and fast (e.g., fast integer ALU)
- Assume multi-ported register file with a reasonable number of ports is feasible
- Assume a dual-ported physical memory with combinational read, 512B of capacity, and wait signal
- Assume a memory bus interconnects instruction, data, host memory interfaces with physical memory and external memory-mapped input/output (I/O) devices

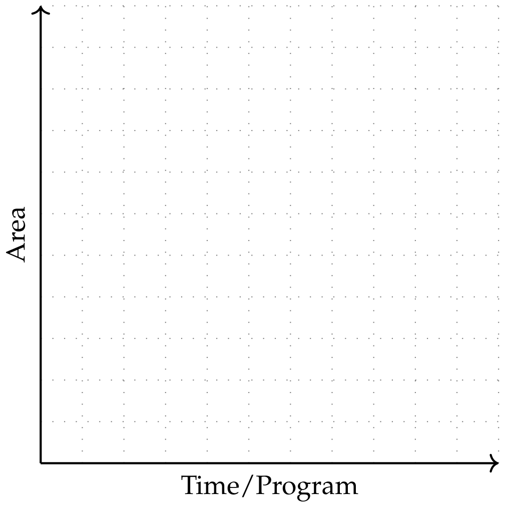


1.3. First-Order Performance Equation

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

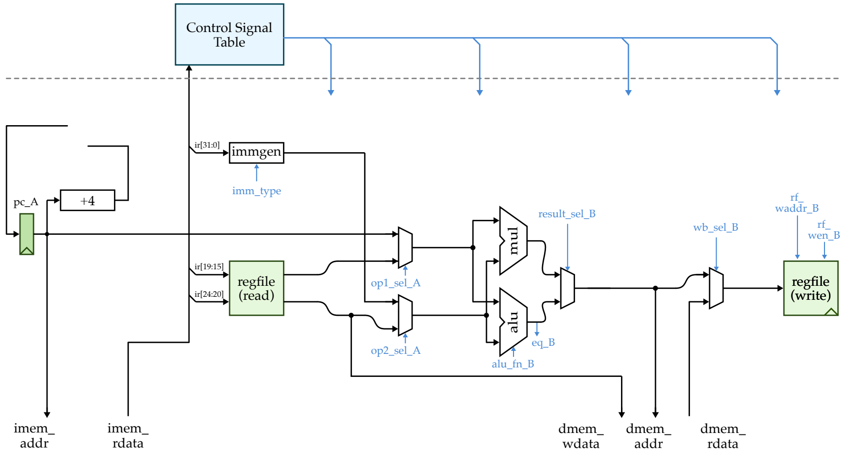
- Instructions / program depends on source code, compiler, ISA
- Avg cycles / instruction (CPI) depends on ISA, microarchitecture
- Time / cycle depends upon microarchitecture and implementation

Microarchitecture	**Cycles / Instruction**	**Time / Cycle**	**Time / Program**	**Area**
Single-Cycle Processor	1	long	medium	medium
Multi-Cycle Processor	>1	short	high	low
Pipelined Processor	≈ 1	short	low	high

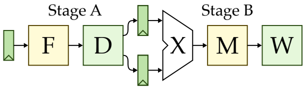


2. Two-Stage Pipelined Processor

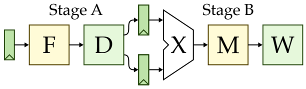
- Incrementally develop an unpipelined datapath
- Keep data flowing from left to right
- Position control signal table early in the diagram
- Divide datapath/control into stages by inserting pipeline registers
- Keep the pipeline stages roughly balanced
- Forward arrows should avoid "skipping" pipeline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



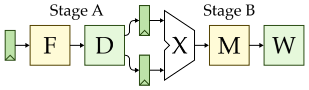
```
addi x1, x2, 1
```



```
addi x3, x4, 1
```



```
addi x5, x6, 1
```



Using transaction diagrams to illustrate pipelined processors

`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?

Instructions can interact with each other in the pipeline creating four kinds of dependencies

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

2.1. RAW Data Dependencies Through Registers

RAW dependencies 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.

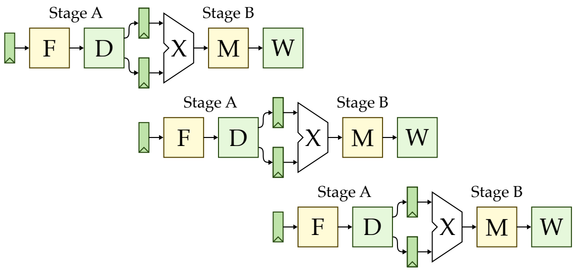
```
addi x1, x2, 1

addi x3, x1, 1

addi x4, x5, 1
```

Using pipeline diagrams to illustrate RAW hazards

We use microarchitectural dependency arrows to illustrate RAW dependencies on pipeline diagrams. RAW dependencies where we would read the incorrect value are called RAW hazards.



```
addi x1, x2, 1

addi x3, x1, 1

addi x4, x5, 1
```

`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`														

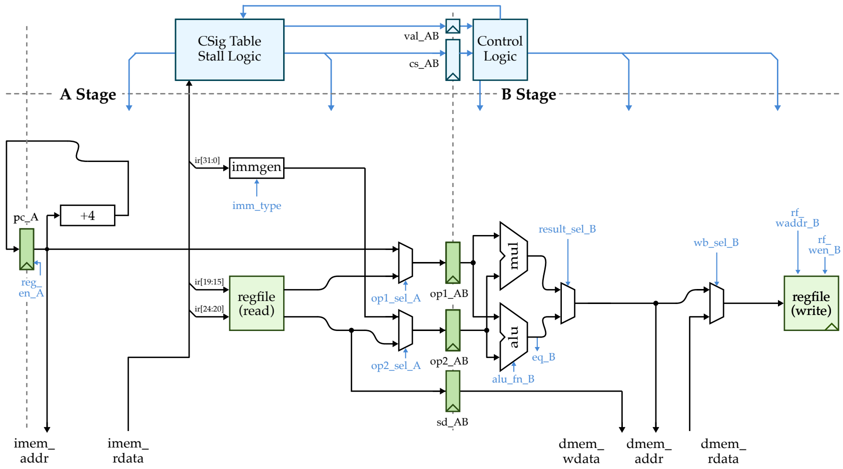
`addi x1, x2, 1`														
`addi x4, x5, 1`														
`addi x3, x1, 1`														

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														

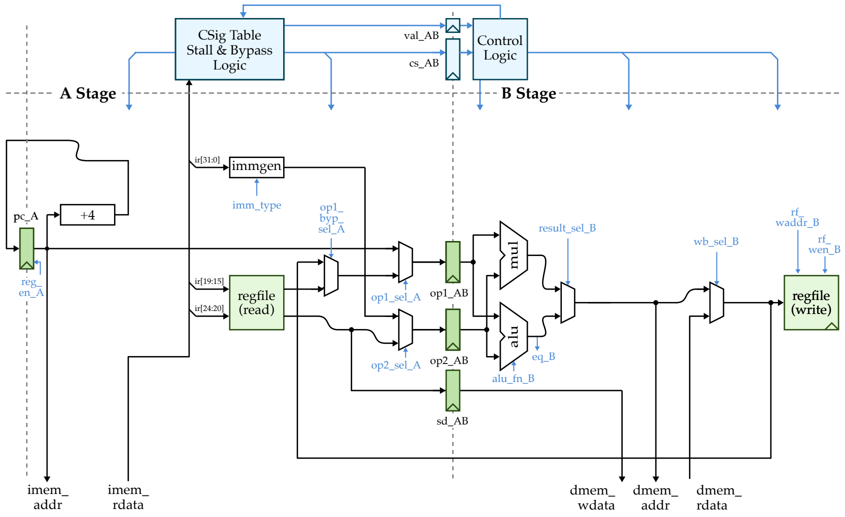
2.4. RAW Data Hazards → Hardware Bypassing

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



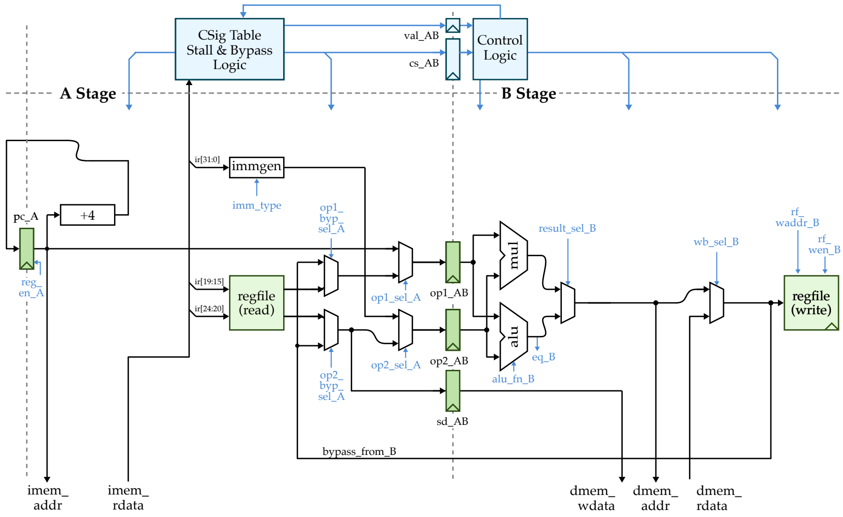
Deriving the bypass and stall signals

```
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 Dependencies Through Memory

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

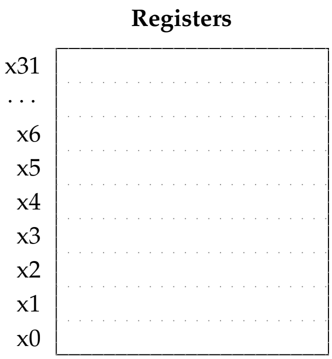
```
  sw x1, 0(x2)
  lw x3, 0(x4) # RAW dependency occurs if R[x2] == R[x4]
```

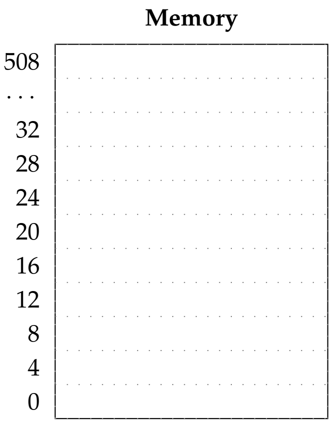
`sw x1, 0(x2)`														
`lw x3, 0(x4)`														

2.6. Control Dependencies

Control dependencies 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.

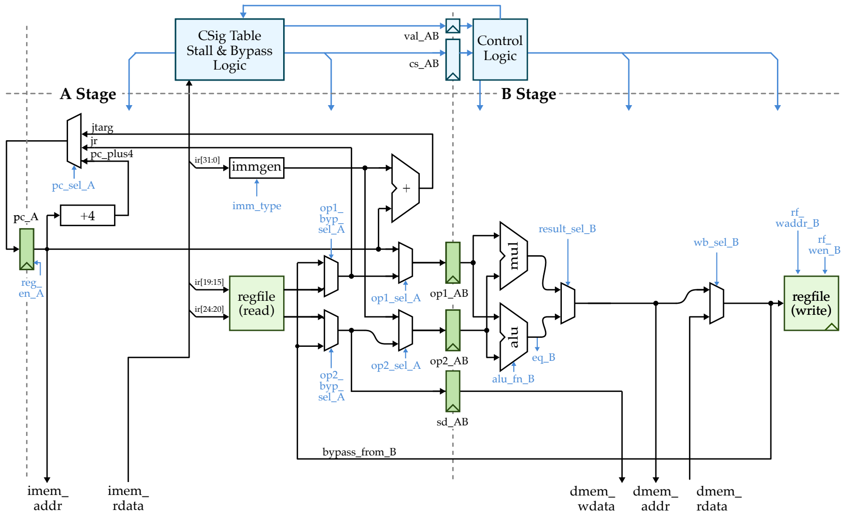
```
    addi x1, x0, 1
    addi x2, x0, 2
    jal  x0, L1
    addi x3, x0, 3
    addi x4, x0, 4
L1: addi x5, x0, 5
    addi x6, x0, 6
```





- **Static instruction sequence** are instructions stored in memory
- **Dynamic instruction sequence** are instructions actually executed

Modifications to datapath/control to support jumps



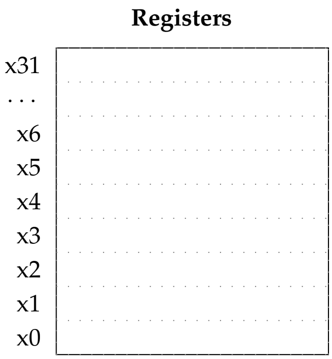
Using pipeline diagrams to illustrate control dependencies

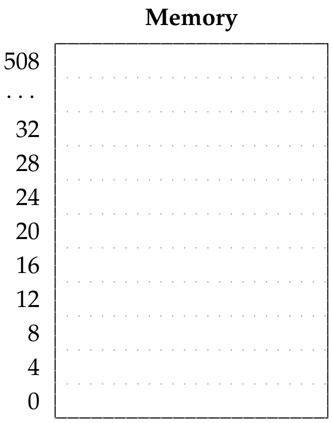
We use microarchitectural dependency arrows to illustrate control dependencies on pipeline diagrams.

addi x1, x0, 1													
addi x2, x0, 2													
jal x0, L1													
addi x5, x0, 5													
addi x6, x0, 6													

How should we handle control dependencies from conditional branches?

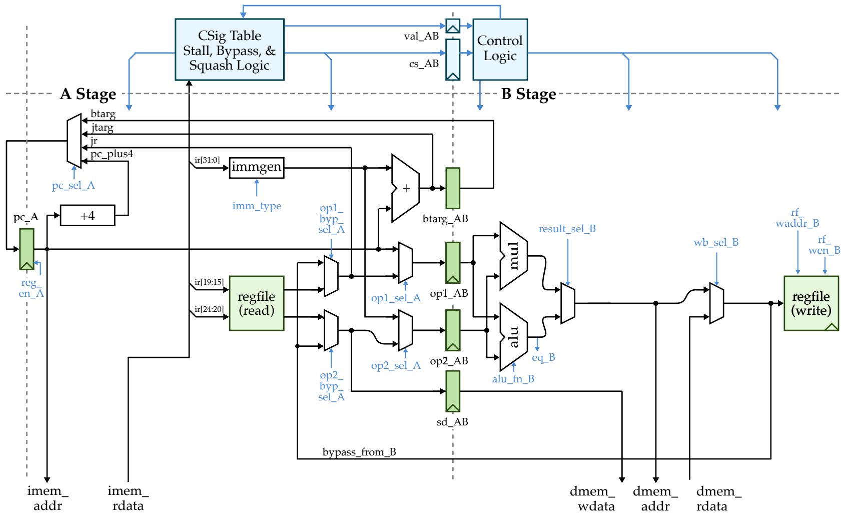
```
     addi x1, x0, 1
     addi x2, x0, 2
     bne  x1, x2, L1
     addi x3, x0, 3
     addi x4, x0, 4
L1:  addi x5, x0, 5
     addi x6, x0, 6
```





- **Static instruction sequence** are instructions stored in memory
- **Dynamic instruction sequence** are instructions actually executed

Modifications to datapath/control to support branches



Using pipeline diagrams to illustrate control dependencies

`addi x1, x0, 1`														
`addi x2, x0, 2`														
`bne  x1, x2, L1`														
`addi x5, x0, 5`														
`addi x6, x0, 6`														

We will not allow a vertical control dependency arrow, since it would create a very long critical path. A vertical control dependency arrow will be considered a control hazard.

Approaches to resolving control hazards

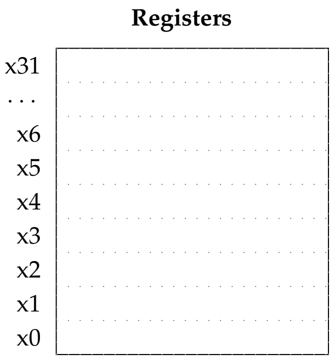
- Software Scheduling: Expose control hazards in ISA forcing assembly level programmer or compiler to explicitly avoid scheduling instructions that would create hazards
- Hardware Speculation: Hardware guesses which way the control flow will go and potentially fetches incorrect instructions; detects 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

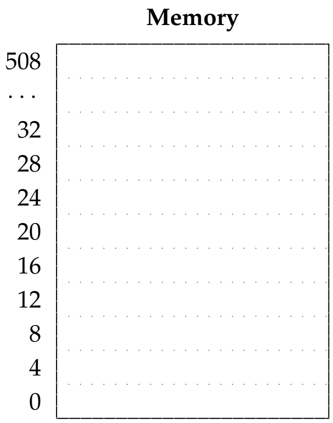
2.7. Control Hazards → Software Scheduling

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.

```
☐☐☐     addi x1, x0, 1
☐☐☐     addi x2, x0, 2
☐☐☐     bne  x1, x2, L1
☐☐☐     nop
☐☐☐     addi x3, x0, 3
☐☐☐     addi x4, x0, 4
☐☐☐ L1: addi x5, x0, 5
☐☐☐     addi x6, x0, 6
```

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).





`addi x1, x0, 1`														
`addi x2, x0, 2`														
`bne  x1, x2, L1`														
`nop`														
`addi x5, x0, 5`														
`addi x6, x0, 6`														

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.

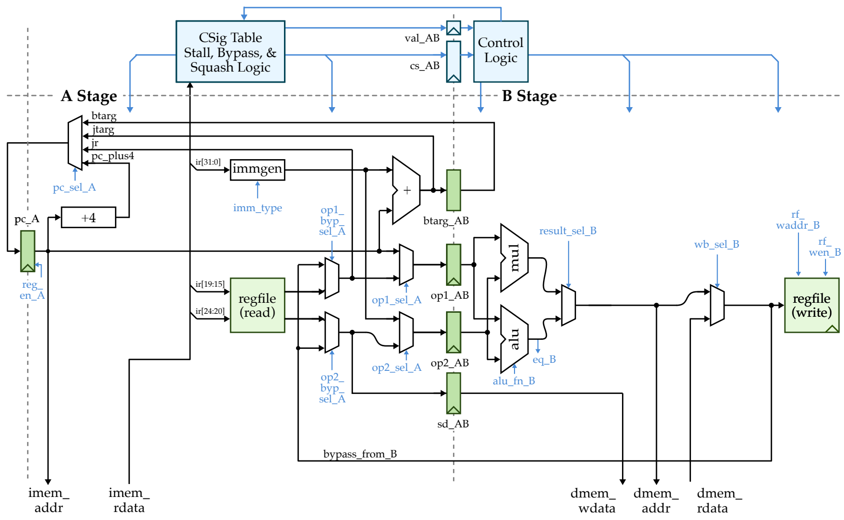
Pipeline diagram when branch is taken

`addi x1, x0, 1`														
`addi x2, x0, 2`														
`bne  x1, x2, L1`														
`addi x3, x0, 3`														
`addi x5, x0, 5`														
`addi x6, x0, 6`														

Pipeline diagram when branch is not taken

`addi x1, x0, 1`														
`addi x2, x0, 2`														
`bne  x1, x2, L1`														
`addi x3, x0, 3`														
`addi x5, x0, 5`														
`addi x6, x0, 6`														

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

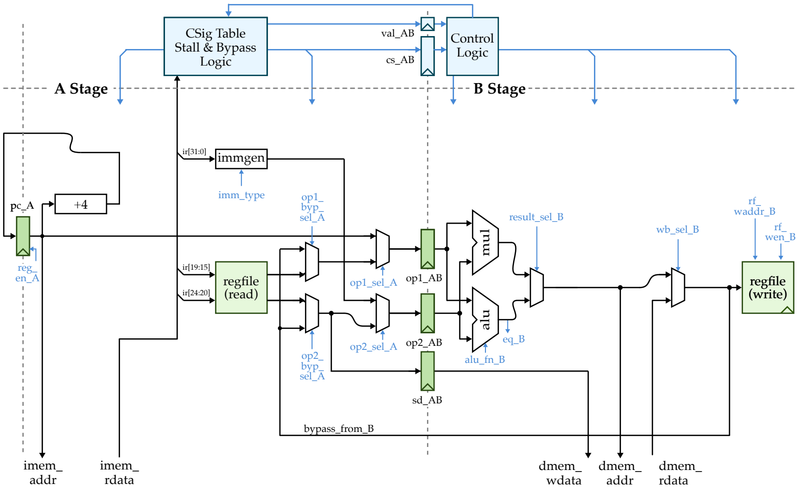
```
    addi x1, x0, 0
    bne  x1, x0, L1
    addi x2, x0, 1
    addi x3, x0, 1
L1: bne  x2, x0, L2
    addi x4, x0, 1
    addi x5, x0, 1
L2: addi x6, x0, 1
```


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 (t_{cq})	9τ
Register File Read	25τ
Memory Read	120τ

32-bit Reg (t_{setup})	10τ
Register File (t_{setup})	20τ
Memory (t_{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

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

Assembly Code

```
 # addr(src0[i]):x1, addr(src1[i]):x2
 # addr(dest[i]):x3, n:x4
loop:
 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
```

	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)`														

Results for vector-vector-add micro-benchmark

Microarchitecture	Inst/Prog	Cycle/Inst	Time/Cycle	Exec Time
Single-Cycle	576	1.0	366 τ	211 kτ
15-State Multi-Cycle	576	4.2	151 τ	367 kτ
63-State Multi-Cycle	576	6.7	231 τ	891 kτ
2-Stage Pipelined				