ECE 4750 Computer Architecture, Fall 2022
Topic 2: Fundamental Processor Microarchitecture

School of Electrical and Computer Engineering
Cornell University

revision: 2022-09-29-22-20

1. Processor Microarchitectural Design Patterns

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

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

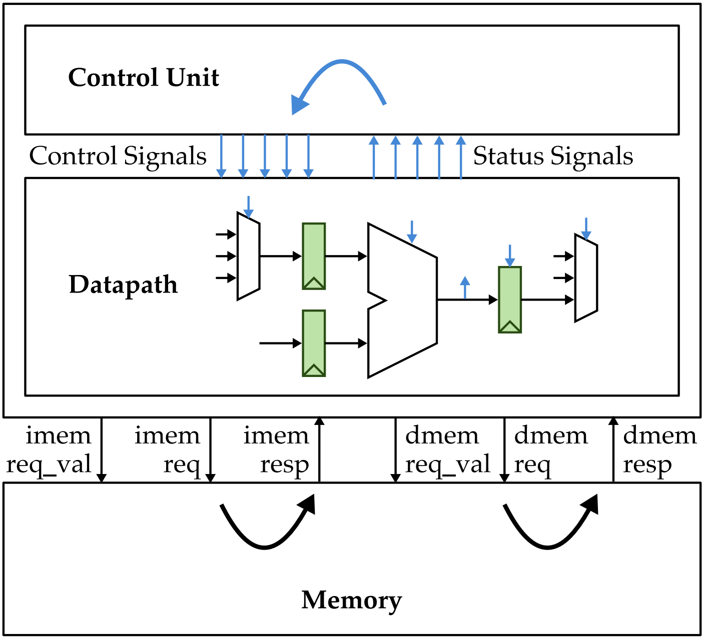
Microarchitecture	**CPI**	**Cycle Time**
Single-Cycle Processor	1	long
FSM Processor	>1	short
Pipelined Processor	≈ 1	short

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. Microarchitecture: Control/Datapath Split



2. TinyRV1 Single-Cycle Processor

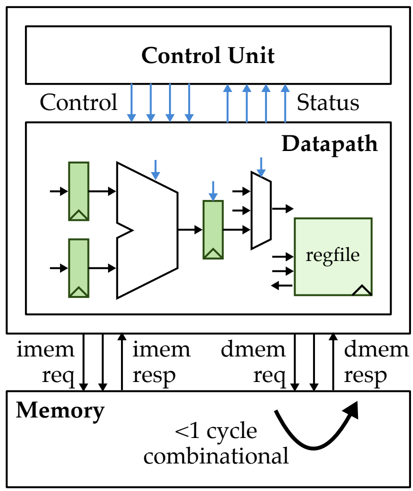
$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Avg Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

- 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
FSM Processor	>1	short
Pipelined Processor	≈ 1	short

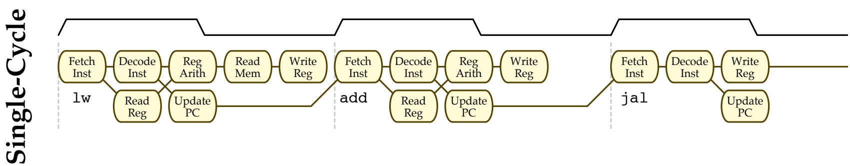
Technology Constraints

- Assume technology where logic is not too expensive, so we do not need to overly minimize the number of registers and combinational logic
- Assume multi-ported register file with a reasonable number of ports is feasible
- Assume a dual-ported combinational memory



2.1. High-Level Idea for Single-Cycle Processors

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



2.2. Single-Cycle Processor Datapath

ADD

`add rd, rs1, rs2`

R[`rd`] ← R[`rs1`] + R[`rs2`]

PC ← PC + 4

31–25	24–20	19–15	14–12	11–7	6–0
0000000	`rs2`	`rs1`	000	`rd`	0110011

ADDI

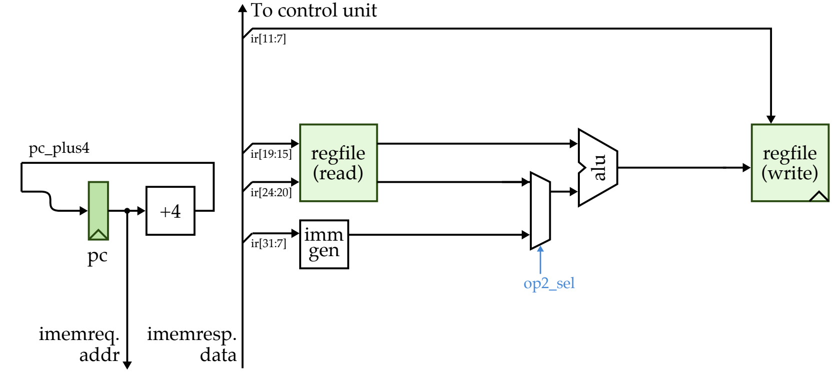
`addi rd, rs1, imm`

R[`rd`] ← R[`rs1`] + sext(`imm`)

PC ← PC + 4

31–20	19–15	14–12	11–7	6–0
`imm`	`rs1`	000	`rd`	0010011

Implementing ADD and ADDI Instructions



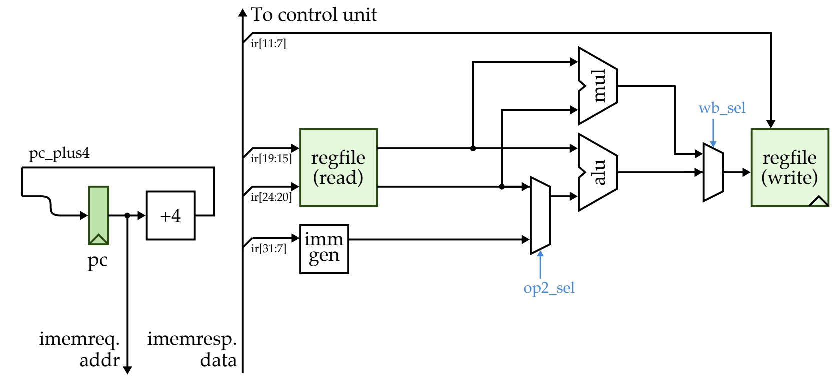
MUL

`mul rd, rs1, rs2`

R[`rd`] ← R[`rs1`] × R[`rs2`]

PC ← PC + 4

31 … 25	24 … 20	19 … 15	14 … 12	11 … 7	6 … 0
0000001	`rs2`	`rs1`	000	`rd`	0110011



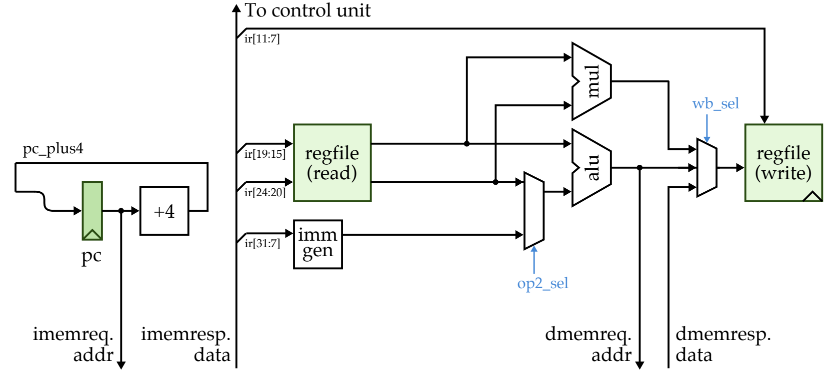
LW

`lw rd, imm(rs1)`

R[`rd`] ← M[R[`rs1`] + sext(`imm`)]

PC ← PC + 4

31 – 20	19 – 15	14 – 12	11 – 7	6 – 0
`imm`	`rs1`	010	`rd`	0000011



SW

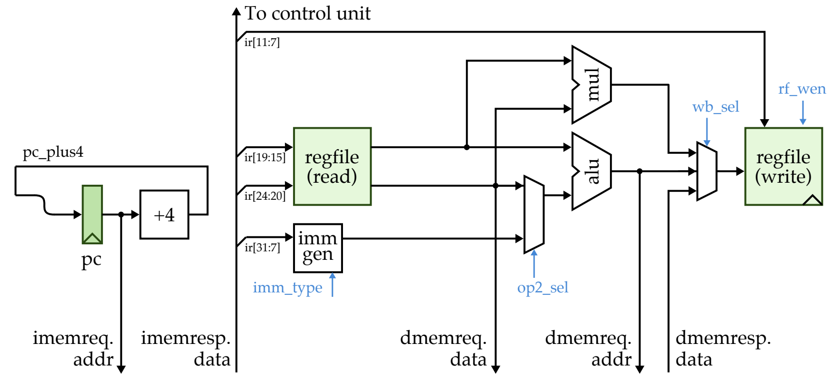
`sw rs2, imm(rs1)`

M[R[`rs1`] + sext(`imm`)] ← R[`rs2`]

PC ← PC + 4

31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0
imm	`rs2`	`rs1`	010	*imm*	0100011

`imm = { inst[31:25], inst[11:7] }`



JAL

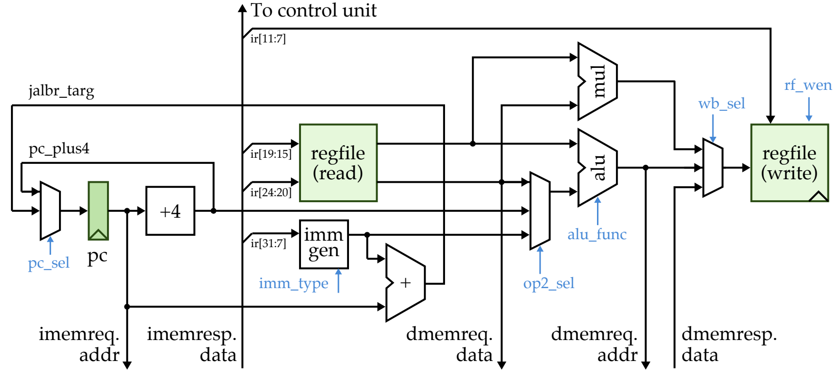
`jal rd, imm`

R[`rd`] ← PC + 4

PC ← PC + sext(`imm`)

31 … 12	11 … 7	6 … 0
imm	rd	1101111

```
imm = { inst[31], inst[19:12],
    inst[20], inst[30:21], 0 }
```

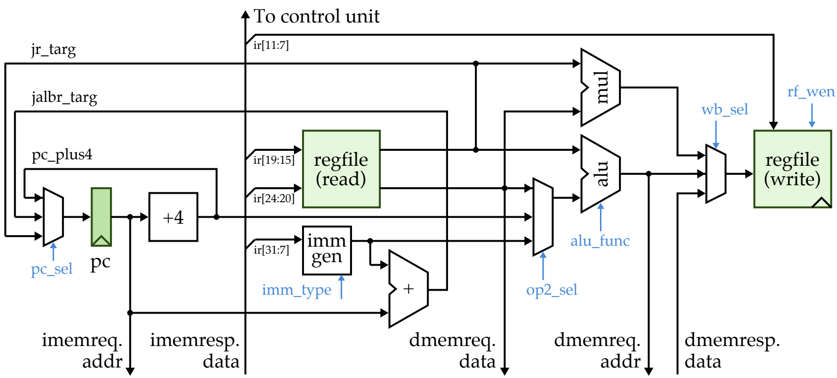


JR

`jr rs1`

PC ← R[`rs1`]

31 … 20	19 … 15	14 … 12	11 … 7	6 … 0
000000000000	rs1	000	00000	1100111



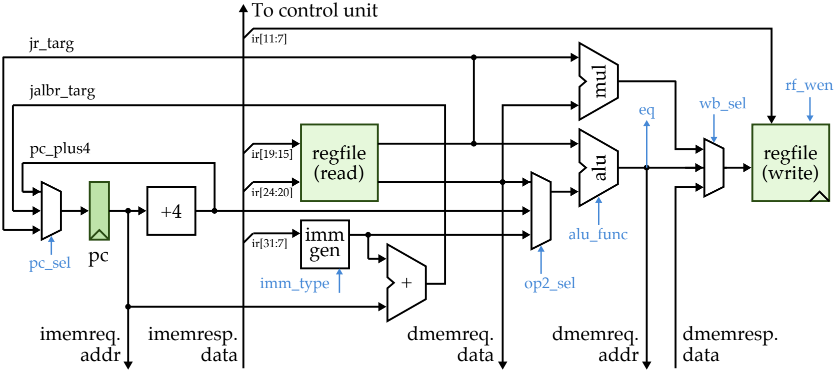
BNE

`bne rs1, rs2, imm`

if (R[`rs1`] != R[`rs2`]) PC ← PC + sext(`imm`)
else PC ← PC + 4

31 25	24 20	19 15	14 12	11 7	6 0
imm	rs2	rs1	001	*imm*	1100011

`imm = { inst[31], inst[7], inst[30:25], inst[11:8], 0 }`



Adding a New Auto-Incrementing Load Instruction

Draw on the datapath diagram what paths we need to use as well as any new paths we will need to add in order to implement the following auto-incrementing load instruction.

LW.AI

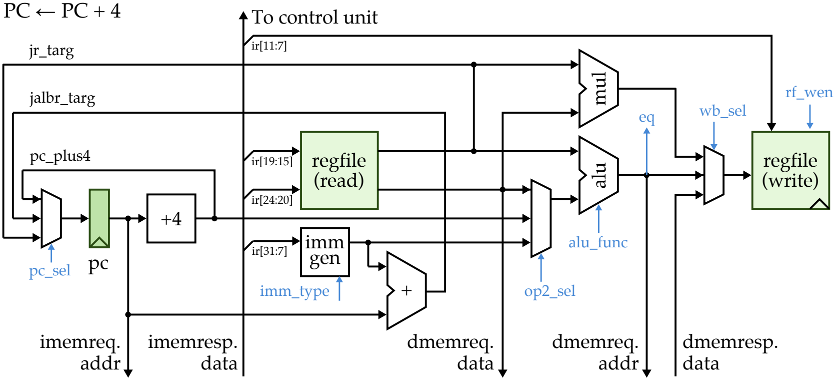
`lw.ai rd, imm(rs1)`

R[`rd`] ← M[R[`rs1`] + sext(`imm`)]

R[`rs1`] ← R[`rs1`] + 4

PC ← PC + 4

31 … 20	19 … 15	14 … 12	11 … 7	6 … 0
`imm`	`rs1`	000	`rd`	0001011



2.3. Single-Cycle Processor Control Unit

inst	pc sel	imm type	op2 sel	alu func	wb sel	rf wen	imem req val	dmem req val
`add`	pc+4	–	rf	+	alu	1	1	0
`addi`								
`mul`	pc+4	–	–	–	mul	1	1	0
`lw`	pc+4	i	imm	+	mem	1	1	1
`sw`								
`jal`								
`jr`	jr	–	–	–	–	0	1	0
`bne`								

Need to factor `eq` status signal into `pc_sel` signal for BNE!

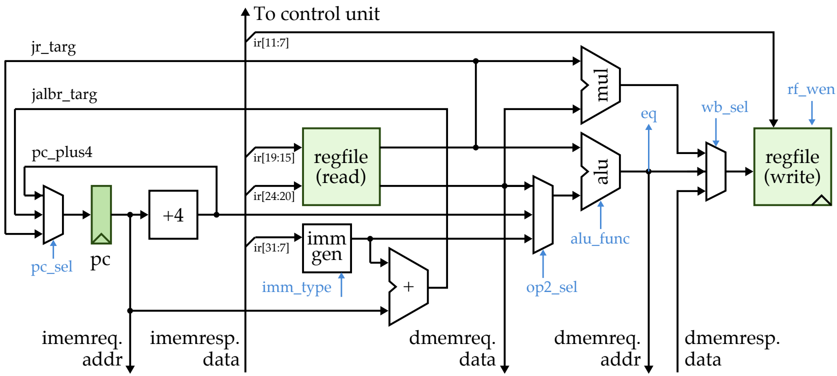
2.4. Analyzing Performance

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

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

Estimating cycle time

There are many paths through the design that start at a state element and end at a state element. The “critical path” is the longest path across all of these paths. We can usually use a simple first-order static timing estimate to estimate the cycle time (i.e., the clock period and thus also the clock frequency).



- register read $= 1\tau$
- register write $= 1\tau$
- regfile read $= 10\tau$
- regfile write $= 10\tau$
- memory read $= 20\tau$
- memory write $= 20\tau$
- +4 unit $= 4\tau$
- immgen $= 2\tau$
- mux $= 3\tau$
- multiplier $= 20\tau$
- alu $= 10\tau$
- adder $= 8\tau$

Estimating execution time

Using our first-order equation for processor performance, how long in units of τ will it take to execute the vector-vector add example assuming `n` is 64?

```
loop:
 lw    x5,  0(x13)
 lw    x6,  0(x14)
 add   x7,  x5,  x6
 sw    x7,  0(x12)
 addi  x13, x12, 4
 addi  x14, x14, 4
 addi  x12, x12, 4
 addi  x15, x15, -1
 bne   x15, x0,  loop
 jr    x1
```

Using our first-order equation for processor performance, how long in units of τ will it take to execute the mystery program assuming `n` is 64 and that we find a match on the last element.

```
 addi  x5,  x0,  0
loop:
 lw    x6,  0(x12)
 bne   x6,  x14, foo
 addi  x10, x5,  0
 jr    x1
foo:
 addi  x12, x12, 4
 addi  x5,  x5,  1
 bne   x5,  x13, loop
 addi  x10, x0,  -1
 jr    x1
```

3. TinyRV1 FSM Processor

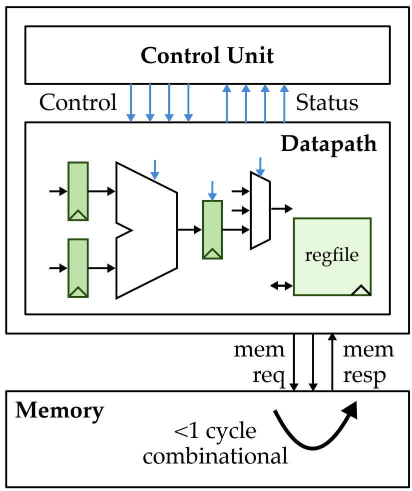
$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Avg Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

- 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
FSM Processor	>1	short
Pipelined Processor	≈ 1	short

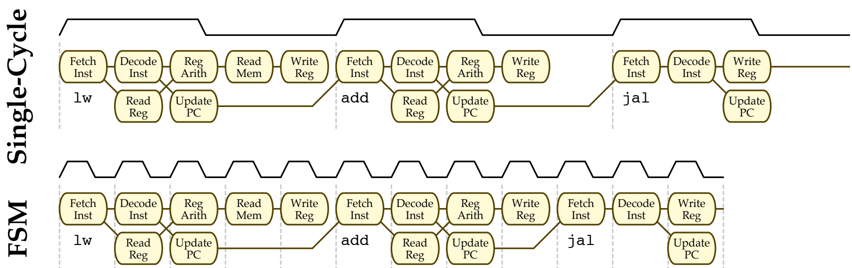
Technology Constraints

- Assume legacy technology where logic is expensive, so we want to minimize the number of registers and combinational logic
- Assume an (unrealistic) combinational memory
- Assume multi-ported register files and memories are too expensive, these structures can only have a single read/write port



3.1. High-Level Idea for FSM Processors

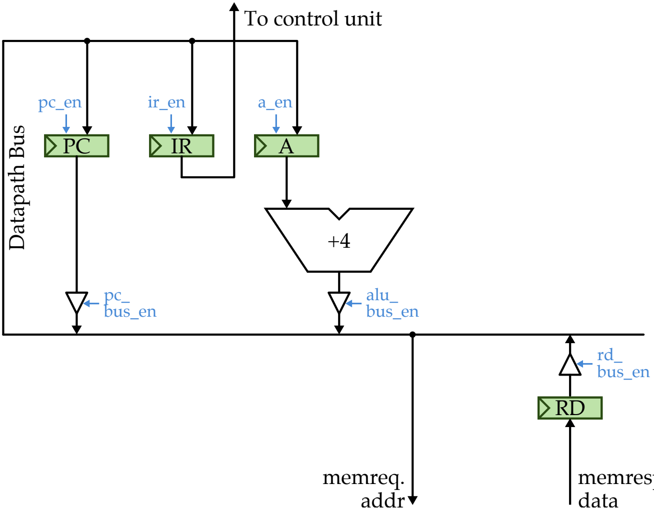
	add	addi	mul	lw	sw	jal	jr	bne
Fetch Instruction	✓	✓	✓	✓	✓	✓	✓	✓
Decode Instruction	✓	✓	✓	✓	✓	✓	✓	✓
Read Registers	✓	✓	✓	✓	✓		✓	✓
Register Arithmetic	✓	✓	✓	✓	✓			✓
Read Memory				✓				
Write Memory					✓			
Write Registers	✓	✓	✓	✓		✓		
Update PC	✓	✓	✓	✓	✓	✓	✓	✓

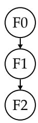


3.2. FSM Processor Datapath

Implementing an FSM datapath requires thinking about the required FSM states, but we will defer discussion of how to implement the control logic to the next section.

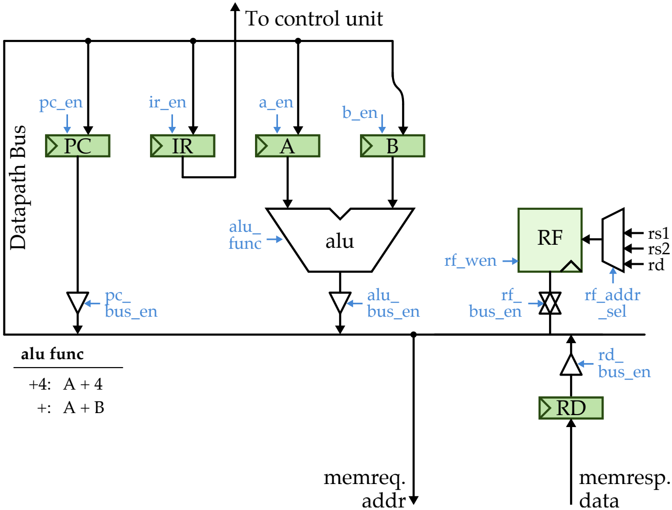
Implementing Fetch Sequence

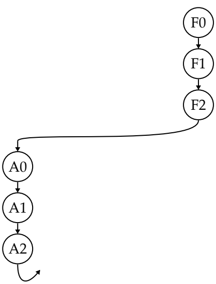




(pseudo-control-signal syntax)

Implementing ADD Instruction

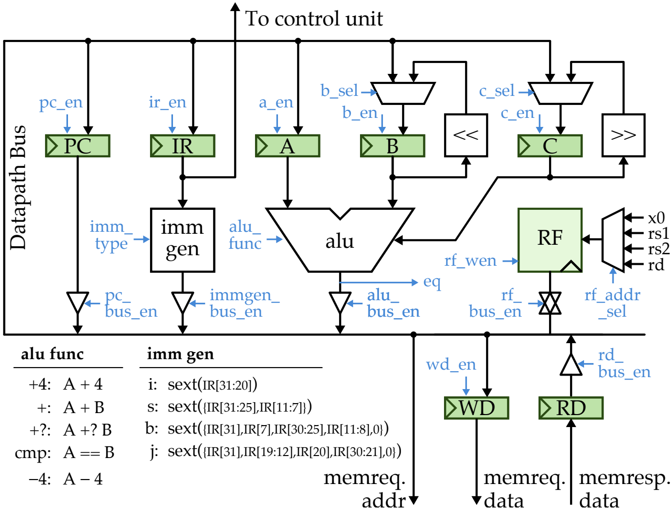


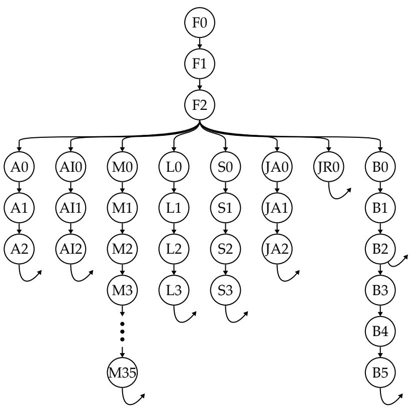


(pseudo-control-signal syntax)

```
add rd, rs1, rs2
```

Full Datapath for TinyRV1 FSM Processor





ADDI Pseudo-Control-Signal Fragment

```
addi rd, rs1, imm
```

MUL Instruction

`mul rd, rs1, rs2`

M0: A ← RF[x0]
M1: B ← RF[`rs1`]
M2: C ← RF[`rs2`]
M3: A ← A +? B;
  B ← B << 1; C ← C >> 1
M4: A ← A +? B;
  B ← B << 1; C ← C >> 1
...
M35: RF[`rd`] ← A +? B; goto F0

LW Instruction

`lw rd, imm(rs1)`

L0: A ← RF[`rs1`]
L1: B ← sext(`imm_i`)
L2: memreq.addr ← A + B
L3: RF[`rd`] ← RD; goto F0

SW Instruction

`sw rs2, imm(rs1)`

S0: WD ← RF[`rs2`]
S1: A ← RF[`rs1`]
S2: B ← sext(`imm_s`)
S3: memreq.addr ← A + B; goto F0

JAL Instruction

`jal rd, imm`

JA0: RF[`rd`] ← PC
JA1: B ← sext(`imm_j`)
JA2: PC ← A + B; goto F0

JR Instruction

`jr rs1`

JR0: PC ← RF[`rs1`]; goto F0

BNE Instruction

`bne rs1, rs2, imm`

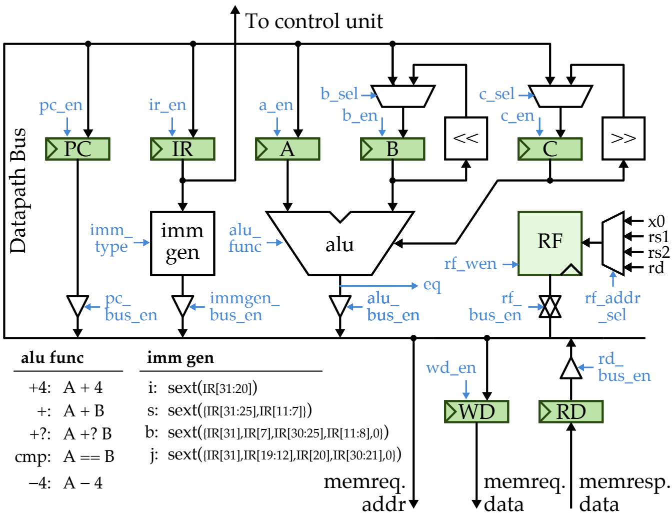
B0: A ← RF[`rs1`]
B1: B ← RF[`rs2`]
B2: B ← sext(`imm_b`);
  if A == B goto F0
B3: A ← PC
B4: A ← A − 4
B5: PC ← A + B; goto F0

Adding a Complex Instruction

FSM processors simplify adding complex instructions. New instructions usually do not require datapath modifications, only additional states.

`add.mm rd, rs1, rs2`

$$\text{M[R[}\texttt{rd}\text{]]} \leftarrow \text{M[R[}\texttt{rs1}\text{]] + M[R[}\texttt{rs2}\text{]]}$$

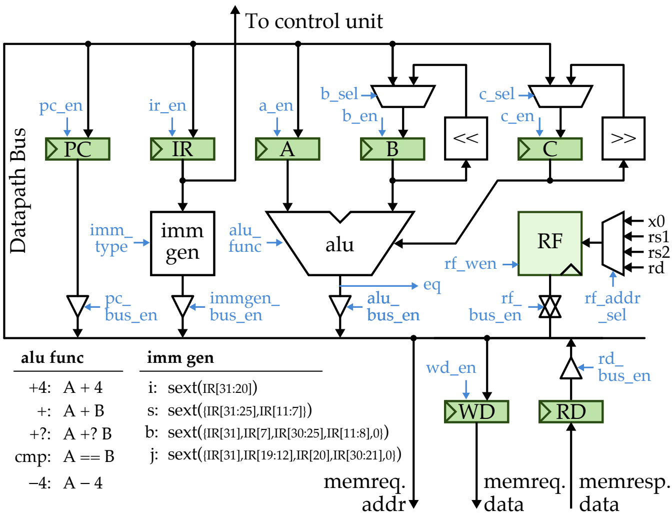


Adding a New Auto-Incrementing Load Instruction

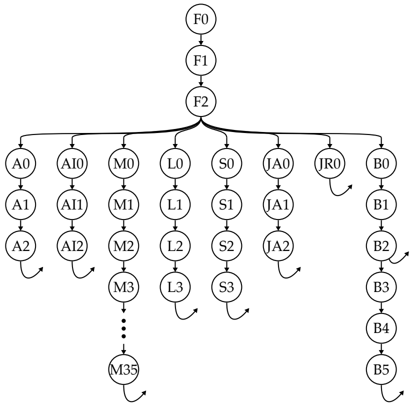
Implement the following auto-incrementing load instruction using pseudo-control-signal syntax. Modify the datapath if necessary.

```
lw.ai rd, imm(rs1)
```

$$R[\texttt{rd}] \leftarrow M[\ R[\texttt{rs1}] + \text{sext}(\texttt{imm\_i})\ ];\ R[\texttt{rs1}] \leftarrow R[\texttt{rs1}] + 4$$



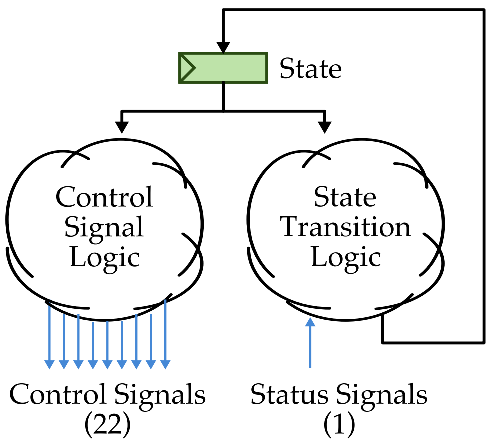
3.3. FSM Processor Control Unit



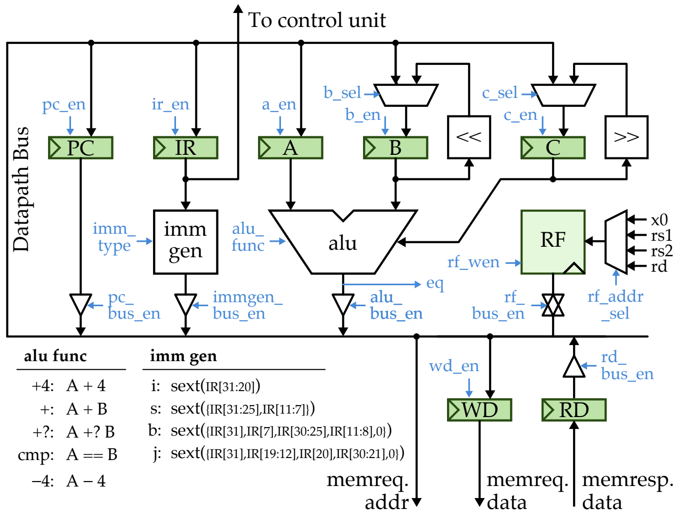
We will study three techniques for implementing FSM control units:

- **Hardwired control units** are high-performance, but inflexible
- **Horizontal µcoding** increases flexibility, requires large control store
- **Vertical µcoding** is an intermediate design point

Hardwired FSM



Control signal output table for hardwired control unit

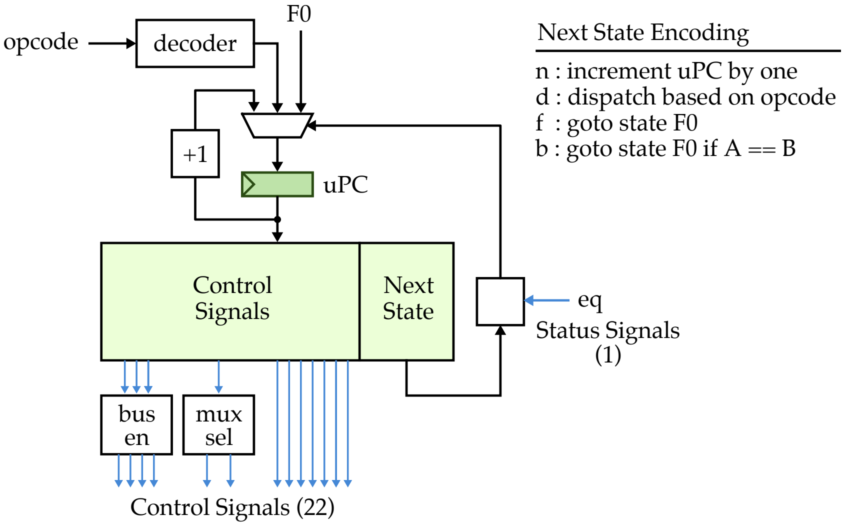


F0: memreq.addr ← PC; A ← PC
F1: IR ← RD
F2: PC ← A + 4; goto inst

A0: A ← RF[`rs1`]
A1: B ← RF[`rs2`]
A2: RF[`rd`] ← A + B; goto F0

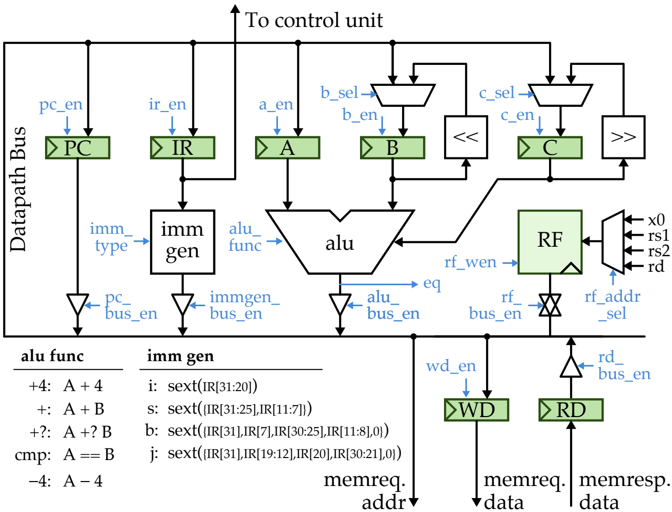
	Bus Enables					**Register Enables**						**Mux**		**Func**		**RF**		**MReq**	
state	**pc**	**ig**	**alu**	**rf**	**rd**	**pc**	**ir**	**a**	**b**	**c**	**wd**	**b**	**c**	**ig**	**alu**	**sel**	**wen**	**val**	**op**
F0	1	0	0	0	0	0	0	1	0	0	0	–	–	–	–	–	0	1	r
F1	0	0	0	0	1	0	1	0	0	0	0	–	–	–	–	–	0	0	–
F2	0	0	1	0	0	1	0	0	0	0	0	–	–	–	+4	–	0	0	–
A0																			
A1																			
A2																			

Vertically Microcoded FSM



- Use memory array (called the control store) instead of random logic to encode both the control signal logic and the state transition logic
- Enables a more systematic approach to implementing complex multi-cycle instructions
- Microcoding can produce good performance if accessing the control store is much faster than accessing main memory
- Read-only control stores might be replaceable enabling in-field updates, while read-write control stores can simplify diagnostics and microcode patches

Control signal store for microcoded control unit



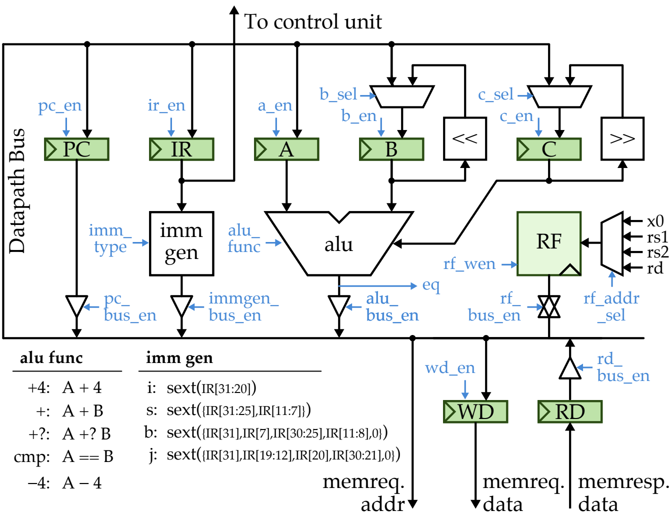
B0: A ← RF[`rs1`]
B1: B ← RF[`rs2`]
B2: B ← sext(`imm_b`); if A == B goto F0
B3: A ← PC
B4: A ← A − 4
B5: PC ← A + B; goto F0

	Bus Enables					Register Enables						Mux		Func		RF		MReq		
state	pc	ig	alu	rf	rd	pc	ir	a	b	c	wd	b	c	ig	alu	sel	wen	val	op	next
B0	0	0	0	1	0	0	0	1	0	0	0	–	–	–	–	rs1	0	0	–	
B1	0	0	0	1	0	0	0	0	1	0	0	b	–	–	–	rs2	0	0	–	
B2	0	1	0	0	0	0	0	0	1	0	0	b	–	b	cmp	–	0	0	–	
B3	1	0	0	0	0	0	0	1	0	0	0	–	–	–	–	–	0	0	–	
B4	0	0	1	0	0	0	0	1	0	0	0	–	–	–	−4	–	0	0	–	
B5	0	0	1	0	0	1	0	0	0	0	0	–	–	–	+	–	0	0	–	

3.4. Analyzing Performance

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

Estimating cycle time



- register read/write $= 1\tau$
- regfile read/write $= 10\tau$
- mem read/write $= 20\tau$
- immgen $= 2\tau$
- mux $= 3\tau$
- alu $= 10\tau$
- 1b shifter $= 1\tau$
- tri-state buf $= 1\tau$

Estimating execution time

Using our first-order equation for processor performance, how long in units of τ will it take to execute the vector-vector add example assuming `n` is 64?

```
loop:
 lw    x5,  0(x13)
 lw    x6,  0(x14)
 add   x7,  x5,  x6
 sw    x7,  0(x12)
 addi  x13, x12, 4
 addi  x14, x14, 4
 addi  x12, x12, 4
 addi  x15, x15, -1
 bne   x15, x0,  loop
 jr    x1
```

Using our first-order equation for processor performance, how long in units of τ will it take to execute the mystery program assuming `n` is 64 and that we find a match on the last element.

```
 addi  x5,  x0,  0
loop:
 lw    x6,  0(x12)
 bne   x6,  x14, foo
 addi  x10, x5,  0
 jr    x1
foo:
 addi  x12, x12, 4
 addi  x5,  x5,  1
 bne   x5,  x13, loop
 addi  x10, x0,  -1
 jr    x1
```

4. TinyRV1 Pipelined Processor

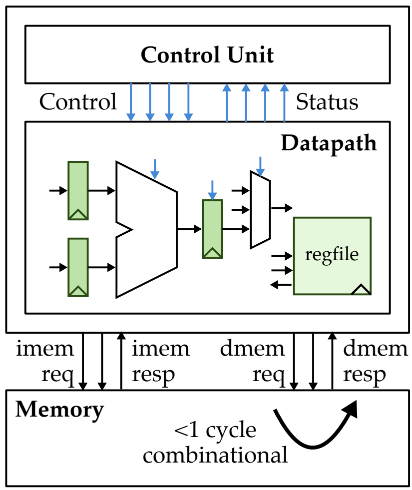
$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Avg Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

- 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
FSM Processor	>1	short
Pipelined Processor	≈ 1	short

Technology Constraints

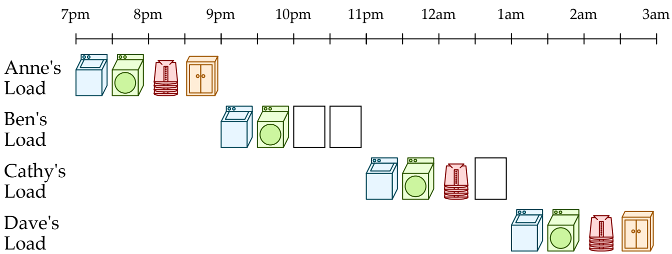
- Assume modern technology where logic is cheap and fast (e.g., fast integer ALU)
- Assume multi-ported register files with a reasonable number of ports are feasible
- Assume small amount of very fast memory (caches) backed by large, slower memory



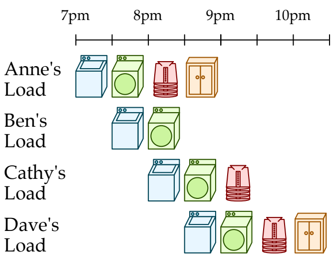
4.1. High-Level Idea for Pipelined Processors

- Anne, Brian, Cathy, and Dave each have one load of clothes
- Washing, drying, folding, and storing each take 30 minutes

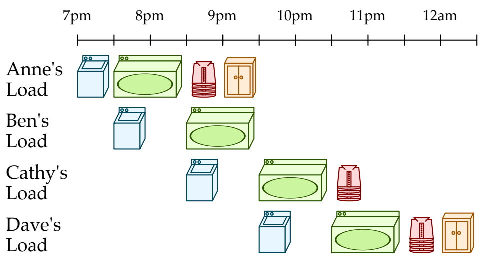
Fixed Time-Slot Laundry



Pipelined Laundry



Pipelined Laundry with Slow Dryers

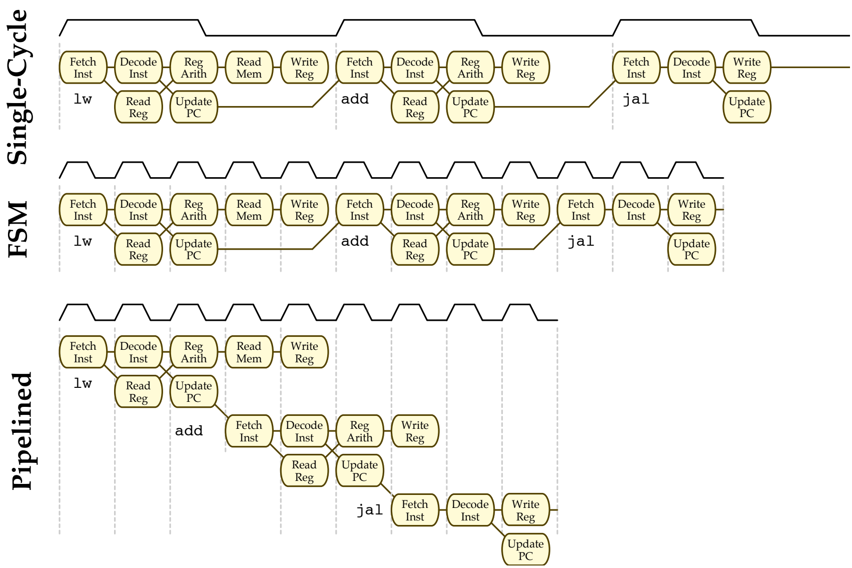


Pipelining lessons

- Multiple transactions operate simultaneously using different resources
- Pipelining does not help the transaction latency
- Pipelining does help the transaction throughput
- Potential speedup is proportional to the number of pipeline stages
- Potential speedup is limited by the slowest pipeline stage
- Potential speedup is reduced by time to fill the pipeline

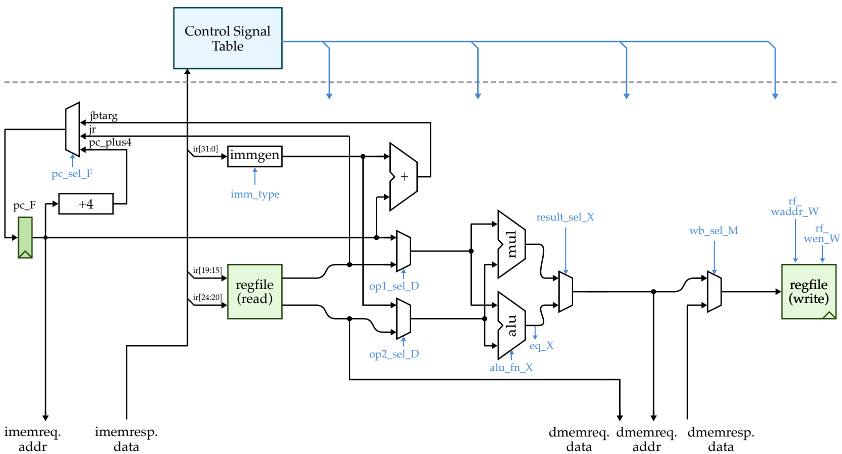
Applying pipelining to processors

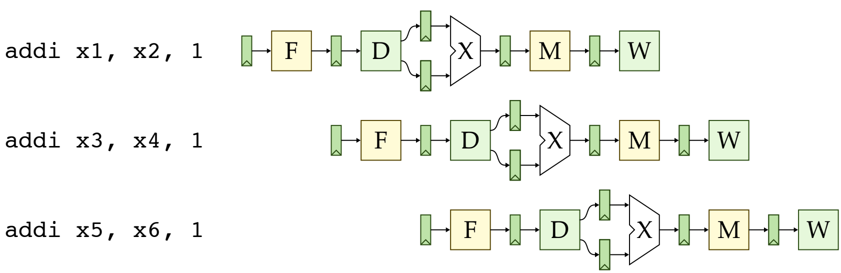
	add	addi	mul	lw	sw	jal	jr	bne
Fetch Instruction	✓	✓	✓	✓	✓	✓	✓	✓
Decode Instruction	✓	✓	✓	✓	✓	✓	✓	✓
Read Registers	✓	✓	✓	✓	✓		✓	✓
Register Arithmetic	✓	✓	✓	✓	✓			✓
Read Memory				✓				
Write Memory					✓			
Write Registers	✓	✓	✓	✓		✓		
Update PC	✓	✓	✓	✓	✓	✓	✓	✓



4.2. Pipelined Processor Datapath and Control Unit

- 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



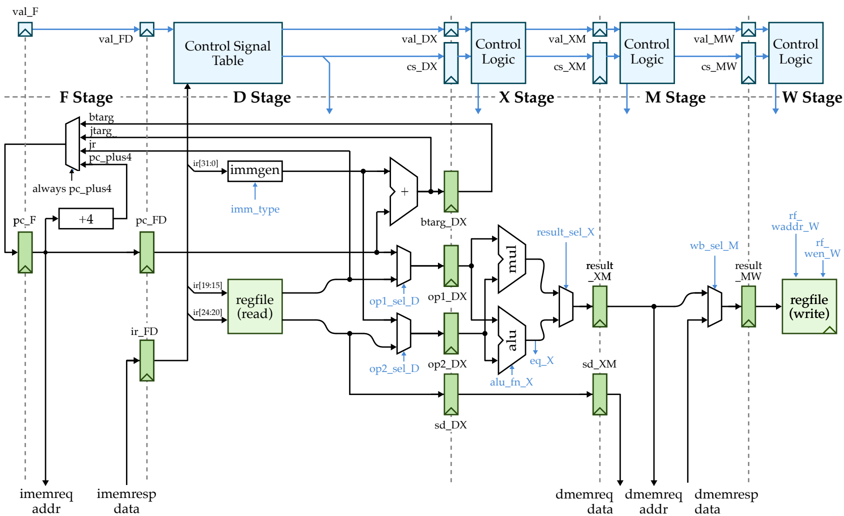


Adding a new auto-incrementing load instruction

Draw on the above datapath diagram what paths we need to use as well as any new paths we will need to add in order to implement the following auto-incrementing load instruction.

```
lw.ai rd, imm(rs1)
```

$$R[\texttt{rd}] \leftarrow M[\ R[\texttt{rs1}] + \text{sext}(\texttt{imm})\ ];\ R[\texttt{rs1}] \leftarrow R[\texttt{rs1}] + 4$$



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?

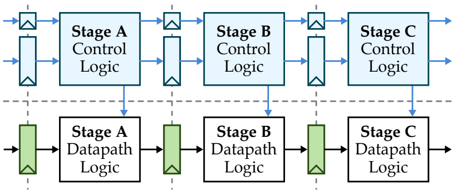
Hazards occur when instructions interact with each other in pipeline

- 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

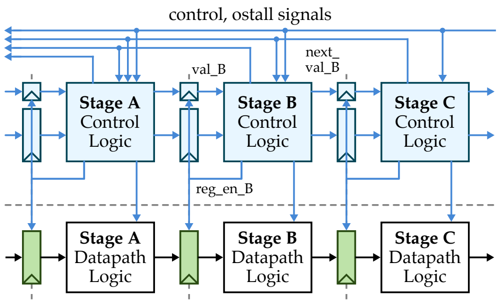
- Stalling: An instruction *originates* a stall due to a hazard, causing all instructions earlier in the pipeline to also stall. When the hazard is resolved, the instruction no longer needs to stall and the pipeline starts flowing again.
- Squashing: An instruction *originates* a squash due to a hazard,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

Control logic with no stalling and no squashing




```
always_ff @( posedge clk )
  if ( reset )
    val_B <= 0
  else
    val_B <= next_val_A

next_val_B = val_B
```

Control logic with stalling and no squashing

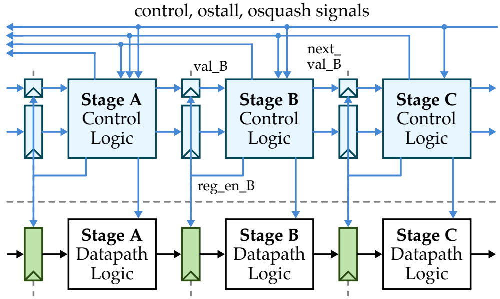



```
reg_en_B = !stall_B

always_ff @( posedge clk )
  if ( reset )
    val_B <= 0
  else if ( reg_en_B )
    val_B <= next_val_A

ostall_B   = val_B && ( ostall_hazard1_B || ostall_hazard2_B )

stall_B    = val_B && ( ostall_B || ostall_C || ... )

next_val_B = val_B && !stall_B
```

ostall_B	Originating stall due to hazards detected in B stage.
stall_B	Should we actually stall B stage? Factors in ostalls due to hazards and ostalls from later pipeline stages.
next_val_B	Only send transaction to next stage if transaction in B stage is valid and we are not stalling B stage.

Control logic with stalling and squashing

```
reg_en_B = !stall_B
        || squash_B

always_ff @( posedge clk )
  if ( reset )
    val_B <= 0
  else if ( reg_en_B )
    val_B <= next_val_A
```




```
ostall_B   = val_B && ( ostall_hazard1_B || ostall_hazard2_B )

stall_B    = val_B && ( ostall_B || ostall_C || ... )

osquash_B  = val_B && !stall_B && ( osquash_hazard1_B || ... )

squash_B   = val_B && ( osquash_C || ... )

next_val_B = val_B && !stall_B && !squash_B
```

`ostall_B`	Originating stall due to hazards detected in B stage.
`stall_B`	Should we actually stall B stage? Factors in ostalls due to hazards and ostalls from later pipeline stages.
`osquash_B`	Originating squash due to hazards detected in B stage. If this stage is stalling, do not originate a squash.
`squash_B`	Should we squash B stage? Factors in the originating squashes from later pipeline stages. An originating squash from B stage means to squash all stages *earlier* than B, so `osquash_B` is *not* factored into `squash_B`.
`next_val_B`	Only send transaction to next stage if transaction in B stage is valid and we are not stalling or squashing B stage.

5. Pipeline Hazards: RAW Data Hazards

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, x3, 1
```

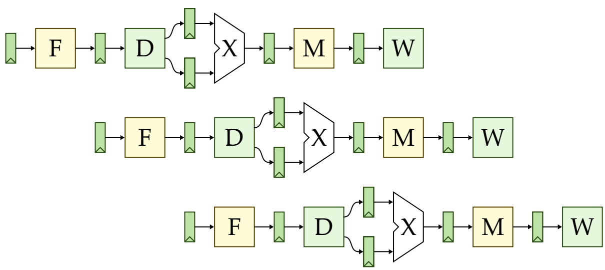
Using pipeline diagrams to illustrate RAW hazards

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

`addi x1, x2, 1`

`addi x3, x1, 1`

`addi x4, x3, 1`



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

Approaches to resolving data hazards

- Expose in Instruction Set Architecture: Expose data hazards in ISA forcing compiler to explicitly avoid scheduling instructions that would create hazards (i.e., software scheduling for correctness)
- Hardware Scheduling: Hardware dynamically schedules instructions to avoid RAW hazards, potentially allowing instructions to execute out of order
- 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 (sometimes called *forwarding*)
- 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

5.1. Expose in Instruction Set Architecture

Insert nops to delay read of earlier write. These nops count as real instructions increasing instructions per program.

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

Insert independent instructions to delay read of earlier write, and only use nops if there is not enough useful work

```
addi x1, x2, 1
addi x6, x7, 1
addi x8, x9, 1
nop
addi x3, x1, 1
nop
nop
nop
addi x4, x3, 1
```

Pipeline diagram showing exposing RAW data hazards in the ISA

addi x1, x2, 1														
addi x6, x7, 1														
addi x8, x9, 1														
nop														
addi x3, x1, 1														
nop														
nop														
nop														
addi x4, x3, 1														

Note: If hazard is exposed in ISA, software scheduling is required for correctness! A scheduling mistake can cause undefined behavior.

5.2. Hardware Stalling

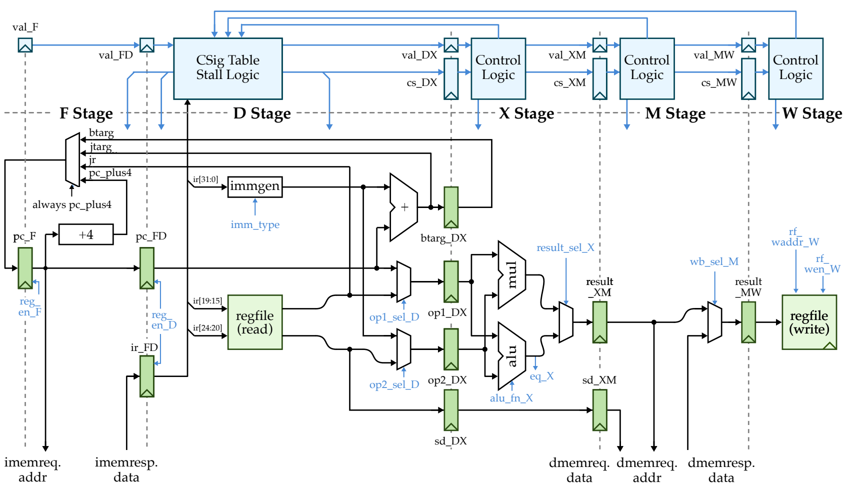
Hardware includes control logic that freezes later instructions (in front of pipeline) until earlier instruction (in back of pipeline) has finished producing data value.

Pipeline diagram showing hardware stalling for RAW data hazards

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

Note: 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.

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								

```
ostall_waddr_X_rs1_D =
  val_D && rs1_en_D && val_X && rf_wen_X
        && (inst_rs1_D == rf_waddr_X) && (rf_waddr_X != 0)

ostall_waddr_M_rs1_D =
  val_D && rs1_en_D && val_M && rf_wen_M
        && (inst_rs1_D == rf_waddr_M) && (rf_waddr_M != 0)

ostall_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 ostall signals for rs2 source register ...

ostall_D = val_D
        && (    ostall_waddr_X_rs1_D || ostall_waddr_X_rs2_D
             || ostall_waddr_M_rs1_D || ostall_waddr_M_rs2_D
             || ostall_waddr_W_rs1_D || ostall_waddr_W_rs2_D )
```

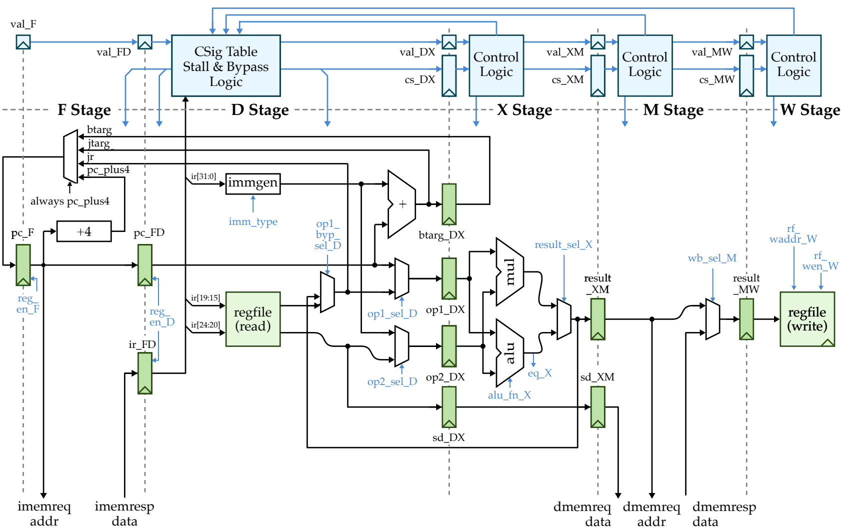
5.3. Hardware Bypassing/Forwarding

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, x3, 1`														

Adding single bypass path to support limited hardware bypassing



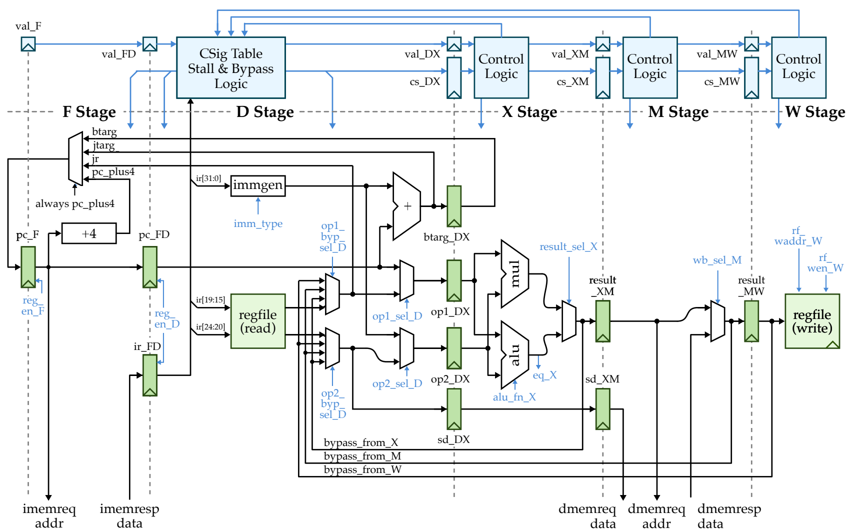
Deriving the bypass and stall signals

```
ostall_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)`														
`jr   x6`														

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`														

```
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 )

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

5.4. 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)`														

6. Pipeline Hazards: 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
      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
```

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

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. Jump resolution latency is two cycles, and branch resolution latency is three cycles.

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

Approaches to resolving control hazards

- Expose in Instruction Set Architecture: Expose control hazards in ISA forcing compiler to explicitly avoid scheduling instructions that would create hazards (i.e., software scheduling for correctness)
- Software Predication: Programmer or compiler converts control flow into data flow by using instructions that conditionally execute based on a data value
- 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 Hints: 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

6.1. Expose in Instruction Set Architecture

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 `nops`.

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

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

Pipeline diagram showing using branch delay slots for control hazards

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

6.2. 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. For now, we will only consider a simple branch prediction scheme where the hardware always predicts not taken.

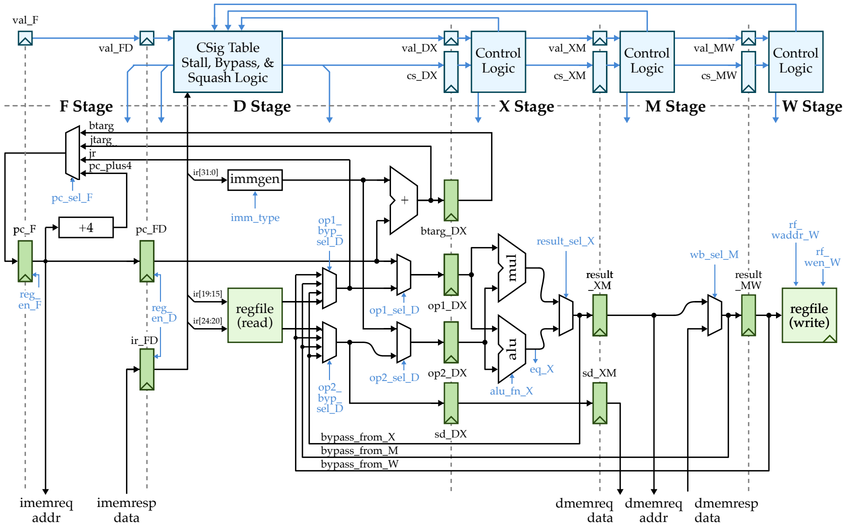
Pipeline diagram when branch is not taken

`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

```
osquash_j_D  = (op_D == jal) || (op_D == jr)
osquash_br_X = (op_X == bne) && !eq_X
```

Our generic stall/squash scheme gives priority to squashes over stalls.

```
osquash_D = val_D && !stall_D && osquash_j_D
squash_D  = val_D && osquash_X

osquash_X = val_D && !stall_X && osquash_br_X
squash_X  = 0
```

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

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 that uses hardware speculation which always predicts not-taken. **Unlike the "standard" TinyRV1 processor, you should also assume that we add a single-instruction branch delay slot to the instruction set.** So this processor will partially expose the control hazard in the instruction, but also use hardware speculation. Include microarchitectural dependency arrows to illustrate both data and control flow.

```
  addi x1, x2, 1
  bne  x0, x3, foo  # assume R[rs] != 0
  addi x4, x5, 1    # instruction is in branch delay slot
  addi x6, x7, 1
  ...
foo:
  add  x8, x1, x4
  addi x9, x1, 1
```


6.3. Interrupts and Exceptions

Interrupts and exceptions alter the normal control flow of the program. They are caused by an external or internal event that needs to be processed by the system, and these events are usually unexpected or rare from the program's point of view.

- **Asynchronous Interrupts**
  - Input/output device needs to be serviced
  - Timer has expired
  - Power distruption or hardware failure
- **Synchronous Exceptions**
  - Undefined opcode, privileged instruction
  - Arithmetic overflow, floating-point exception
  - Misaligned memory access for instruction fetch or data access
  - Memory protection violation
  - Virtual memory page faults
  - System calls (traps) to jump into the operating system kernel

Interrupts and Exception Semantics

- Interrupts are asynchronous with respect to the program, so the microarchitecture can decide when to service the interrupt
- Exceptions are synchronous with respect to the program, so they must be handled immediately

- To handle an interrupt or exception the hardware/software must:
  - Stop program at current instruction (*I*), ensure previous insts finished
  - Save cause of interrupt or exception in privileged arch state
  - Save the PC of the instruction *I* in a special register (EPC)
  - Switch to privileged mode
  - Set the PC to the address of either the interrupt or the exception handler
  - Disable interrupts
  - Save the user architectural state
  - Check the type of interrupt or exception
  - Handle the interrupt or exception
  - Enable interrupts
  - Switch to user mode
  - Set the PC to EPC if *I* should be restarted
  - Potentially set PC to EPC+4 if we should skip *I*

Handling a misaligned data address and syscall exceptions

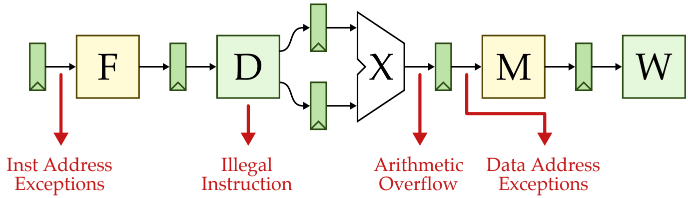
Static Instr Sequence

```
  addi x1, x0, 0x2001
  lw   x2, 0(x1)
  syscall
  opB
  opC
  ...
exception_hander:
  opD  # disable interrupts
  opE  # save user registers
  opF  # check exception type
  opG  # handle exception
  opH  # enable interrupts
  addi EPC, EPC, 4
  eret
```

Dynamic Instr Sequence

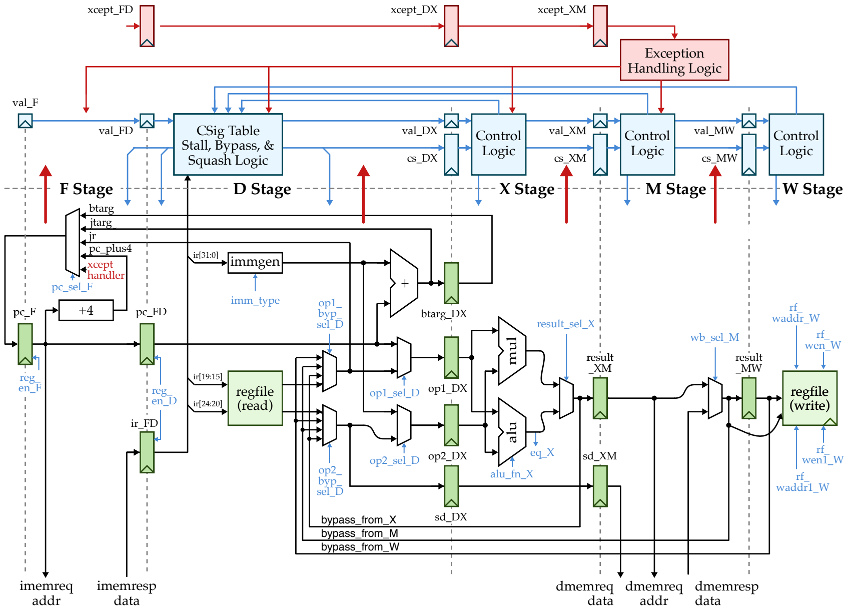
```
addi x1, x0, 0x2001
lw   x2, 0(x1) (excep)
opD
opE
opF
opG
opH
addi EPC, EPC, 4
eret
syscall        (excep)
opD
opE
opF
```

Interrupts and Exceptions in a RISC-V Pipelined Processor



- How should we handle a single instruction which generates multiple exceptions in different stages as it goes down the pipeline?
  - Exceptions in earlier pipeline stages override later exceptions for a given instruction
- How should we handle multiple instructions generating exceptions in different stages at the same or different times?
  - We always want the execution to appear as if we have completely executed one instruction before going onto the next instruction
  - So we want to process the exception corresponding to the earliest instruction in program order first
  - Hold exception flags in pipeline until commit point
  - Commit point is after all exceptions could be generated but before any architectural state has been updated
  - To handle an exception at the commit point: update cause and EPC, squash all stages before the commit point, and set PC to exception handler
- How and where to handle external asynchronous interrupts?
  - Inject asynchronous interrupts at the commit point
  - Asynchronous interrupts will then naturally override exceptions caused by instructions earlier in the pipeline

Modifications to datapath/control to support exceptions



Deriving the squash signals

```
osquash_j_D      = (op_D == jal) || (op_D == jr)
osquash_br_X     = (op_X == bne) && !eq_X
osquash_xcept_M = exception_M
```

Control logic needs to redirect the front end of the pipeline just like for a jump or branch. Again, squashes take priority over stalls, and PC select logic must give priority to older instructions (i.e., priortize exceptions, over branches, over jumps)!

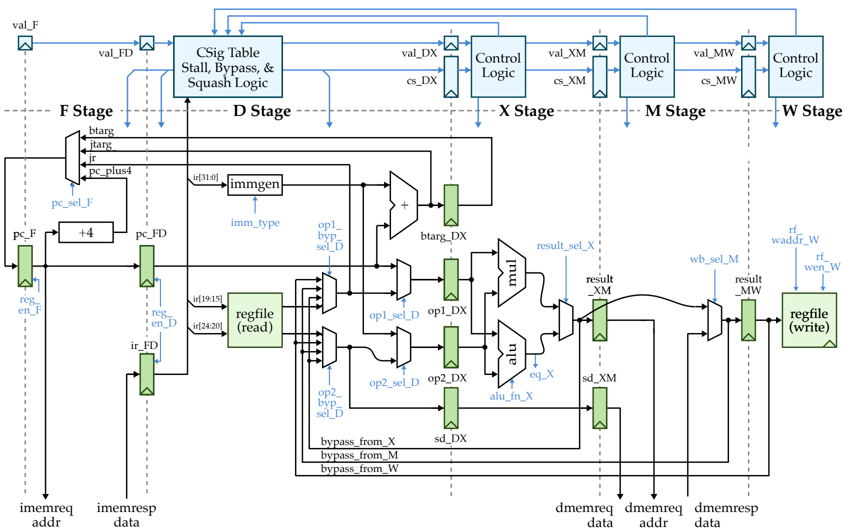
Pipeline diagram of exception handling

```
  addi x1, x0, 0x2001
  lw   x2, 0(x1)    # assume causes misaligned address exception
  syscall           # causes a syscall exception
  opB
  opC
  ...
exception_hander:
  opD  # disable interrupts
  opE  # save user registers
  opF  # check exception type
  opG  # handle exception
  opH  # enable interrupts
  addi EPC, EPC, 4
  eret
```


7. Pipeline Hazards: Structural Hazards

Structural hazards occur when an instruction in the pipeline needs a resource being used by another instruction in the pipeline. The TinyRV1 processor pipeline is specifically designed to avoid structural hazards.

Let's introduce a structural hazard by allowing instructions that do not do any real work in the M stage (i.e., non-memory instructions) to effectively skip that stage. This would require adding an extra path which "skips over" the pipeline reigster between the X and M stages and connects directly to the writeback mux at the end of the M stage. For non-memory instructions we set `wb_sel_M` to choose the value from the end of the X stage, while for memory instructions we set `wb_sel_M` to choose the value coming back from memory.



Using pipeline diagrams to illustrate structural hazards

We use structural dependency arrows to illustrate structural hazards.

`addi x1, x2, 1`														
`addi x3, x4, 1`														
`lw   x5, 0(x6)`														
`addi x7, x8, 1`														

Note that the key shared resources that are causing the structural hazard are the pipeline registers at the end of the M stage. We cannot write these pipeline registers with the transaction that is in the X stage and also the transaction that is the M stage at the same time.

Approaches to resolving structural hazards

- Expose in Instruction Set Architecture: Expose structural hazards in ISA forcing compiler to explicitly avoid scheduling instructions that would create hazards (i.e., software scheduling for correctness)
- Hardware Stalling: Hardware includes control logic that freezes later instructions until earlier instruction has finished using the shared resource; software scheduling can still be used to avoid stalling (i.e., software scheduling for performance)
- Hardware Duplication: Add more hardware so that each instruction can access separate resources at the same time

7.1. Expose in Instruction Set Architecture

Insert independent instructions or `nops` to delay non-memory instructions if they follow a LW or SW instruction.

Pipeline diagram showing exposing structural hazards in the ISA

`addi x1, x2, 1`														
`addi x3, x4, 1`														
`lw   x5, 0(x6)`														
`nop`														
`addi x7, x8, 1`														

7.2. Hardware Stalling

Hardware includes control logic that stalls a non-memory instruction if it follows a LW or SW instruction.

Pipeline diagram showing hardware stalling for structural hazards

`addi x1, x2, 1`														
`addi x3, x4, 1`														
`lw   x5, 0(x6)`														
`addi x7, x8, 1`														

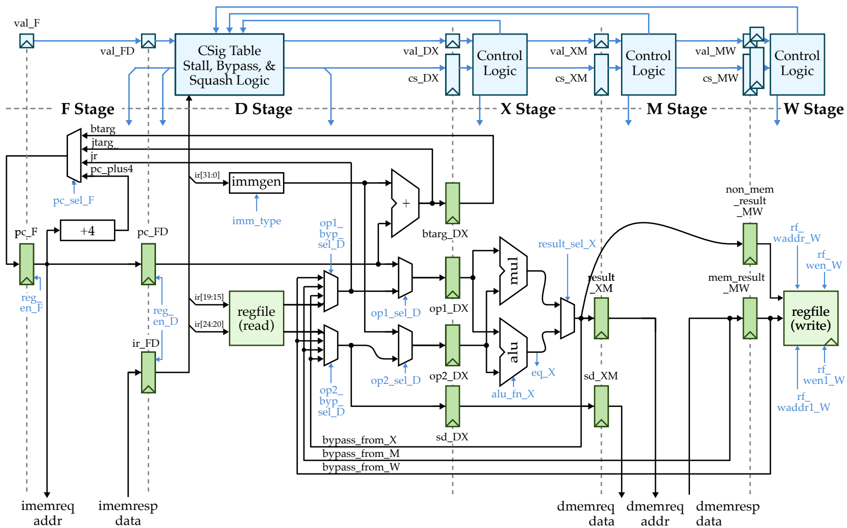
Deriving the stall signal

```
ostall_wport_hazard_D = val_D && !mem_inst_D && val_X && mem_inst_X
```

where `mem_inst` is true for a LW or SW instruction and false otherwise. Stall far before the structural hazard actually occurs, because we know exactly how instructions move down the pipeline. Also possible to use dynamic arbitration in the back-end of the pipeline.

7.3. Hardware Duplication

Add more pipeline registers at the end of M stage and a second write port so that non-memory and memory instructions can writeback to the register file at the same time.



Does allowing early writeback help performance in the first place?

`addi x1, x2, 1`														
`addi x3, x1, 1`														
`addi x4, x3, 1`														
`addi x5, x4, 1`														
`addi x6, x5, 1`														
`addi x7, x6, 1`														

8. Pipeline Hazards: WAW and WAR Name Hazards

WAW dependencies occur when an instruction overwrites a register than an earlier instruction has already written. WAR dependencies occur when an instruction writes a register than an earlier instruction needs to read. We use architectural dependency arrows to illustrate WAW and WAR dependencies in assembly code sequences.

```
mul  x1, x2, x3

addi x4, x1, 1

addi x1, x5, 1
```

WAW name hazards occur when an instruction in the pipeline writes a register before an earlier instruction (in back of the pipeline) has had a chance to write that same register.

WAR name hazards occur when an instruction in the pipeline writes a register before an earlier instuction (in back of pipeline) has had a chance to read that same register.

The TinyRV1 processor pipeline is specifically designed to avoid any WAW or WAR name hazards. Instructions always write the registerfile in-order in the same stage, and instructions always read registers in the front of the pipeline and write registers in the back of the pipeline.

Let's introduce a WAW name hazard by using an iterative variable latency multiplier, and allowing other instructions to continue executing while the multiplier is working.

Using pipeline diagrams to illustrate WAW name hazards

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

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

Approaches to resolving WAW and WAR hazards

- Software Renaming: Programmer or compiler changes the register names to avoid creating name hazards
- Hardware Renaming: Hardware dynamically changes the register names to avoid creating name hazards
- Hardware Stalling: Hardware includes control logic that freezes later instructions until earlier instruction has finished either writing or reading the problematic register name

8.1. Software Renaming

As long as we have enough architectural registers, renaming registers in software is easy. WAW and WAR dependencies occur because we have a finite number of architectural registers.

```
mul  x1, x2, x3
addi x4, x6, 1
addi x7, x5, 1
```

8.2. Hardware Stalling

Simplest approach is to add stall logic in the decode stage similar to what the approach used to resolve other hazards.

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

Deriving the stall signal

```
ostall_struct_hazard_D = val_D && (op_D == MUL) && !imul_rdy_D

ostall_waw_hazard_D =
  val_D && rf_wen_D && val_Z && rf_wen_Z
        && (rf_waddr_D == rf_waddr_Z) && (rf_waddr_Z != 0)
```

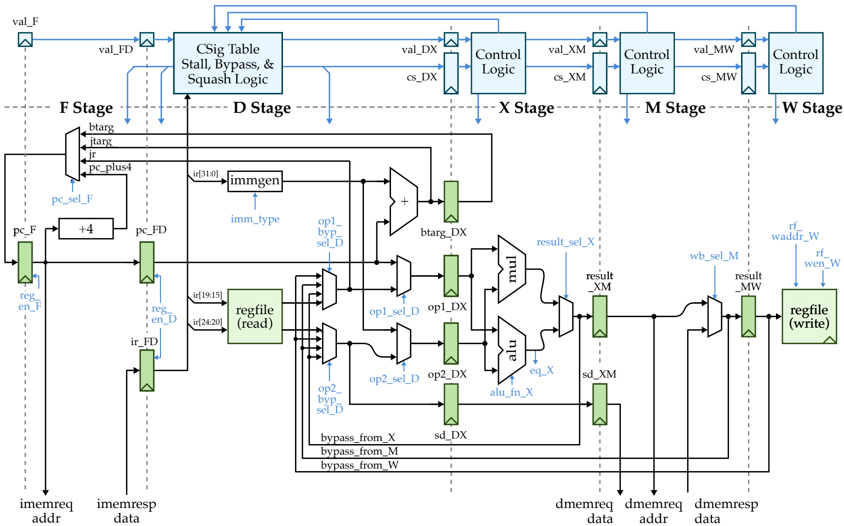
9. Summary of Processor Performance

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

Results for vector-vector add example

Microarchitecture	Inst	CPI	Cycle Time	Exec Time
Single-Cycle Processor	576	1.0	$74\,\tau$	$43\,\text{k}\tau$
FSM Processor	576	6.7	$36\,\tau$	$138\,\text{k}\tau$
Pipelined Processor	576			

Estimating cycle time for pipelined processor



- register read = 1τ
- register write = 1τ
- regfile read = 10τ
- regfile write = 10τ
- memory read = 20τ
- memory write = 20τ
- +4 unit = 4τ
- immgen = 2τ
- mux = 3τ
- multiplier = 20τ
- alu = 10τ
- adder = 8τ

Estimating execution time

Using our first-order equation for processor performance, how long in τ will it take to execute the vvadd example assuming `n` is 64?

```
loop:
 lw    x5,  0(x13)
 lw    x6,  0(x14)
 add   x7,  x5,  x6
 sw    x7,  0(x12)
 addi  x13, x12, 4
 addi  x14, x14, 4
 addi  x12, x12, 4
 addi  x15, x15, -1
 bne   x15, x0,  loop
 jr    x1
```

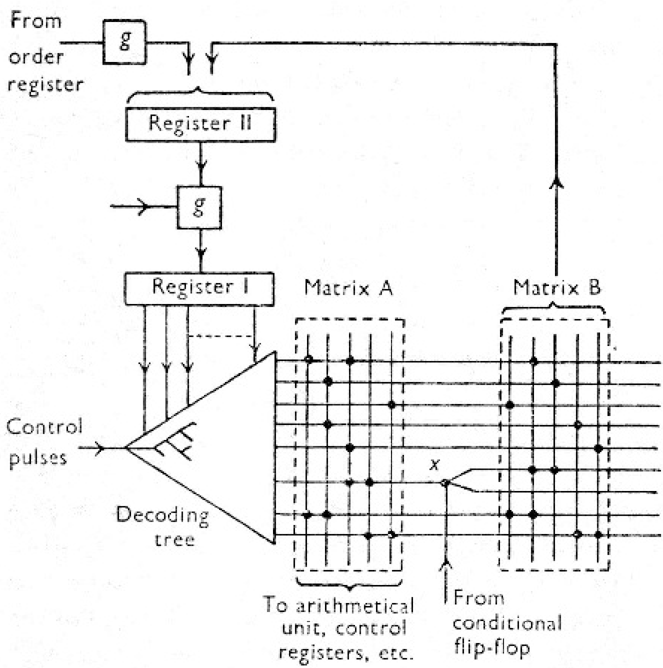
lw																					
lw																					
add																					
sw																					
addi																					
addi																					
addi																					
addi																					
bne																					
opA																					
opB																					
lw																					

Using our first-order equation for processor performance, how long in τ will it take to execute the mystery program assuming `n` is 64 and that we find a match on the last element.

```
 addi  x5,  x0,  0
loop:
 lw    x6,  0(x12)
 bne   x6,  x14, foo
 addi  x10, x5,  0
 jr    x1
foo:
 addi  x12, x12, 4
 addi  x5,  x5,  1
 bne   x5,  x13, loop
 addi  x10, x0,  -1
 jr    x1
```

10. Case Study: Transition from CISC to RISC

- Microcoding thrived in the 1970's
  - ROMs significantly faster than DRAMs
  - For complex instruction sets, microcode was cheaper and simpler
  - New instructions supported without modifying datapath
  - Fixing bugs in controller is easier
  - ISA compatibility across models relatively straight-forward



— Maurice Wilkes, 1954

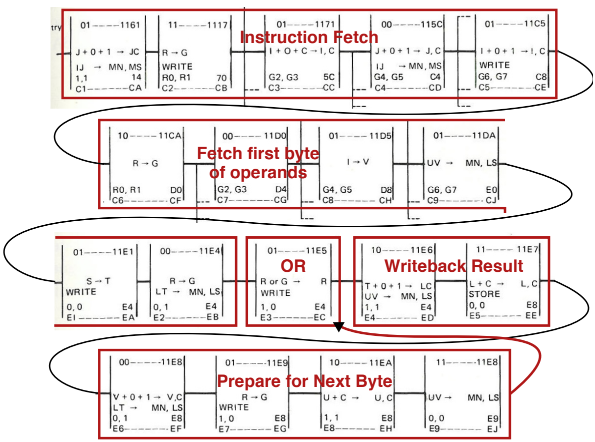
10.1. Example CISC: IBM 360/M30

	M30	M40	M50	M65
Datapath width (bits)	8	16	32	64
µinst width (bits)	50	52	85	87
µcode size (1K µinsts)	4	4	2.75	2.75
µstore technology	CCROS	TROS	BCROS	BCROS
µstore cycle (ns)	750	625	500	200
Memory cycle (ns)	1500	2500	2000	750
Rental fee ($K/month)	4	7	15	35

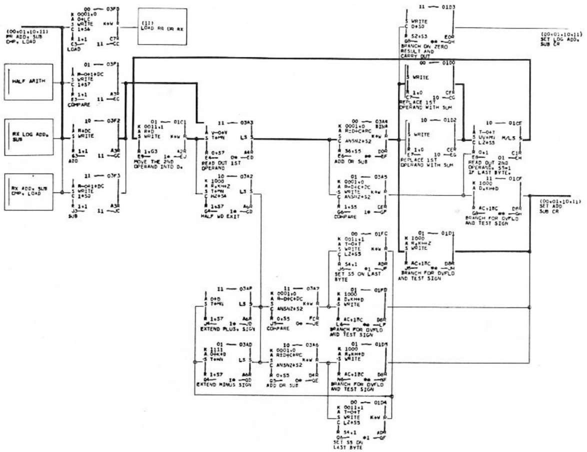
TROS = transformer read-only storage (magnetic storage)
BCROS = balanced capacitor read-only storage (capacitive storage)
CCROS = card capacitor read-only storage (metal punch cards, replace in field)

Only the fastest models (75,95) were hardwired

IBM 360/M30 microprogram for register-register logical OR



IBM 360/M30 microprogram for register-register binary ADD

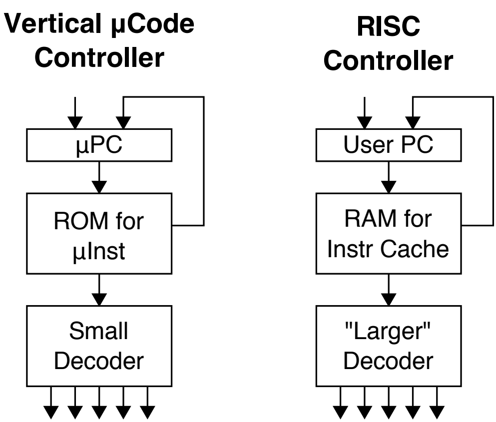


Analyzing Microcoded Machines

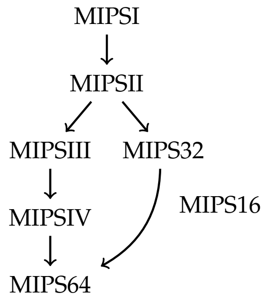
- John Cocke and group at IBM
  - Working on a simple pipelined processor, 801, and advanced compilers
  - Ported experimental PL8 compiler to IBM 370, and only used simple register-register and load/store instructions similar to 801
  - Code ran faster than other existing compilers that used all 370 instructions! (up to 6 MIPS, whereas 2 MIPS considered good before)
- Joel Emer and Douglas Clark at DEC
  - Measured VAX-11/780 using external hardware
  - Found it was actually a 0.5 MIPS machine, not a 1 MIPS machine
  - 20% of VAX instrs = 60% of µcode, but only 0.2% of the dynamic execution
- VAX 8800, high-end VAX in 1984
  - Control store: 16K×147b RAM, Unified Cache: 64K×8b RAM
  - 4.5× more microstore RAM than cache RAM!

From CISC to RISC

- Key changes in technology constraints
  - Logic, RAM, ROM all implemented with MOS transistors
  - RAM ≈ same speed as ROM
- Use fast RAM to build fast instruction cache of user-visible instructions, not fixed hardware microfragments
  - Change contents of fast instruction memory to fit what app needs
- Use simple ISA to enable hardwired pipelined implementation
  - Most compiled code only used a few of CISC instructions
  - Simpler encoding allowed pipelined implementations
  - Load/Store Reg-Reg ISA as opposed to Mem-Mem ISA
- Further benefit with integration
  - Early 1980's → fit 32-bit datapath, small caches on single chip
  - No chip crossing in common case allows faster operation



10.2. Example RISC: MIPS R2K



- MIPS R2K is one of the first popular pipelined RISC processors
- MIPS R2K implements the MIPS I instruction set
- MIPS = Microprocessor without Interlocked Pipeline Stages
- MIPS I used software scheduling to avoid some RAW hazards by including a single-instruction load-use delay slot
- MIPS I used software scheduling to avoid some control hazards by including a single-instruction branch delay slot

One-Instr Branch Delay Slot

```
  addiu r1, r2, 1
  j     foo
  addiu r3, r4, 1  # BDS
  ...
foo:
  addiu r5, r6, 1
  bne   r7, r8, bar
  addiu r9, r10, 1 # BDS
  ...
bar:
```

Present in all MIPS instruction sets; not possible to depricate and still enable legacy code to execute on new microarchitectures

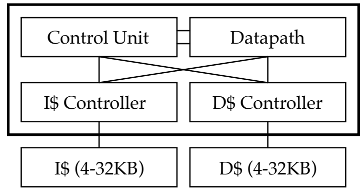
One-Instr Load-Use Delay Slot

```
  lw    r1, 0(r2)
  lw    r3, 0(r4)
  addiu r2, r2, 4  # LDS
  addu  r5, r1, r3
```

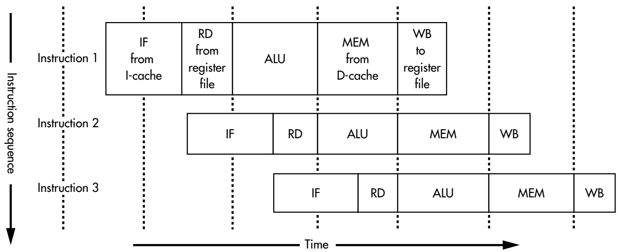
Deprecated in MIPS II instruction set; legacy code can still execute on new microarchitectures, but code using the MIPS II instruction set can rely in hardware stalling

MIPS R2K Microarchitecture

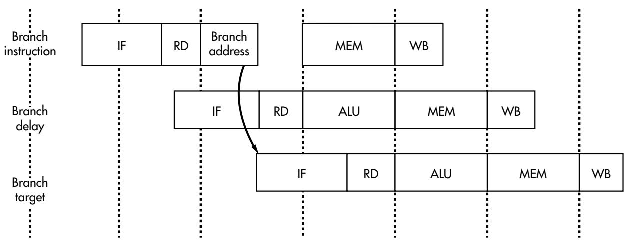
The pipelined datapath and control were located on a single die. Cache control and memory management unit were also integrated on-die, but the actual tag and data storage for the cache was located off-chip.



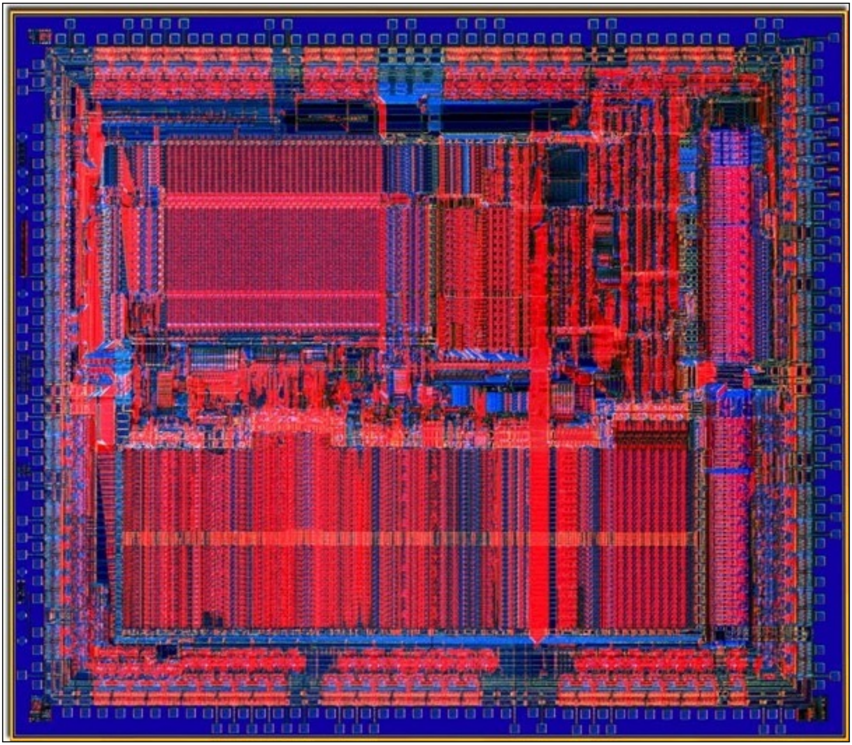
Used two-phase clocking to enable five pipeline stages to fit into four clock cycles. This avoided the need for explicit bypassing from the W stage to the end of the D stage.



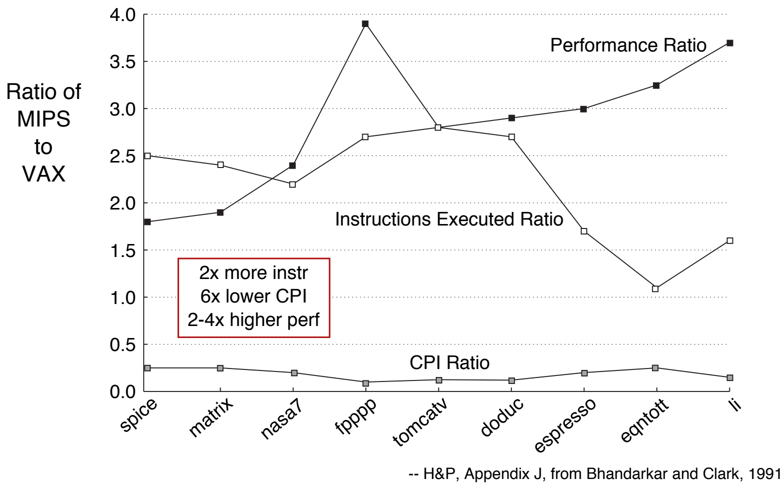
Two-phase clocking enabled a single-cycle branch resolution latency since register read, branch address generation, and branch comparison can fit in a single cycle.



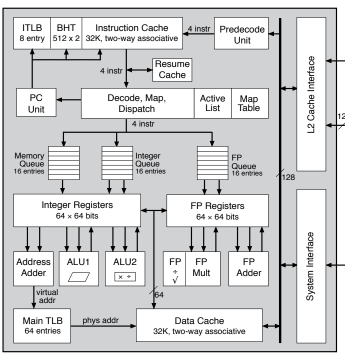
MIPS R2K VLSI Design



Process: 2 µm, two metal layers
Clock Frequency: 8–15 MHz
Size: 110K transistors, 80 mm^2

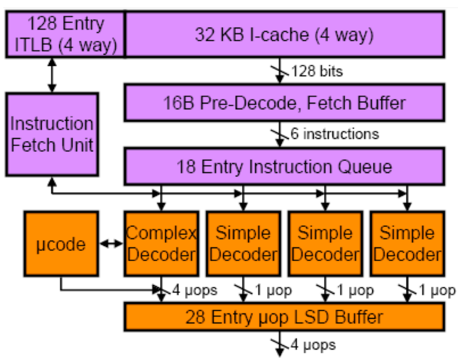


CISC/RISC Convergence



MIPS R10K uses sophisticated out-of-order engine; branch delay slot not useful

– Gwennap, MPR, 1994



Intel Nehalem frontend breaks x86 CISC into smaller RISC-like µops; µcode engine handles rarely used complex instr

– Kanter, Real World Technologies, 2009

Microprogamming Today

- Microprogramming is far from extinct
- Played a crucial role in microprocessors of the 1980s
  (DEC VAX, Motorola 68K series, Intel 386/486)
- Microprogramming plays assisting role in many modern processors
  (AMD Phenom, Intel Nehalem, Intel Atom, IBM Z196)
  - 761 Z196 instructions executed with hardwired control
  - 219 Z196 "complex" instructions always executed with microcode
  - 24 Z196 instructions conditionally executed with microcode
- Patchable microcode common for post-fabrication bug fixes (Intel processors load µcode patches at bootup)