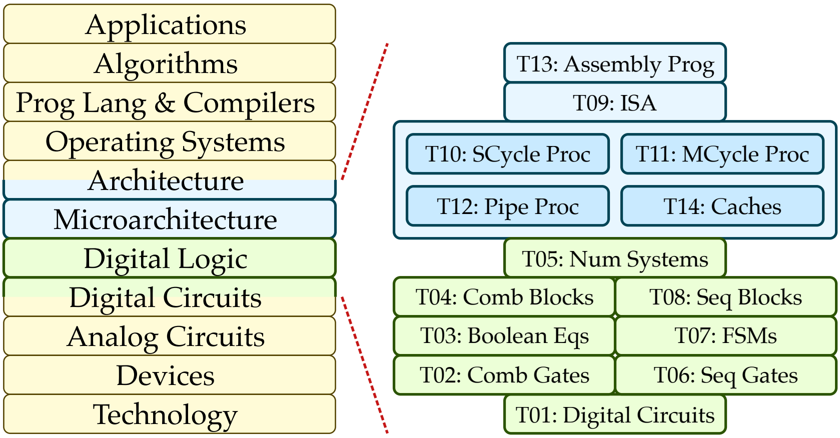
ECE 2300 Digital Logic and Computer Organization
Fall 2024

Topic 11: Multi-Cycle Processors

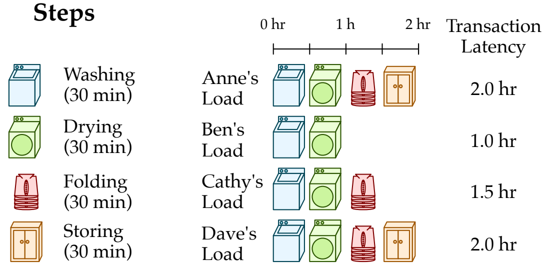
School of Electrical and Computer Engineering
Cornell University

revision: 2024-11-07-11-02


Applications
Algorithms
Prog Lang & Compilers
Operating Systems
Architecture
Microarchitecture
Digital Logic
Digital Circuits
Analog Circuits
Devices
Technology
T13: Assembly Prog
T09: ISA
T10: SCycle Proc
T11: MCycle Proc
T12: Pipe Proc
T14: Caches
T05: Num Systems
T04: Comb Blocks
T08: Seq Blocks
T03: Boolean Eqs
T07: FSMs
T02: Comb Gates
T06: Seq Gates
T01: Digital Circuits

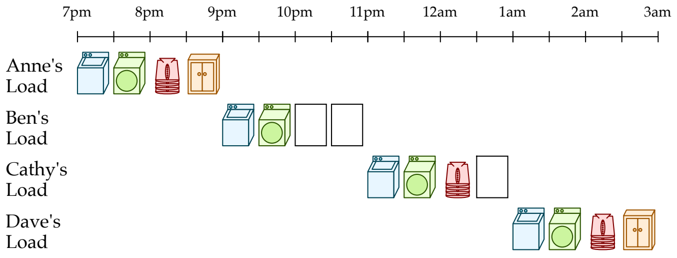
1. High-Level Idea for Single-Cycle Processors

Transaction Steps **Four Types of Transactions**

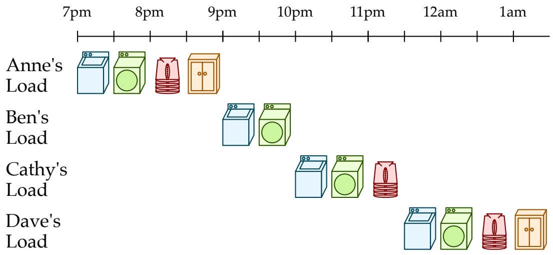


	Transaction Latency	
Anne's Load	2.0 hr	Anne requires all four steps
Ben's Load	1.0 hr	Ben is messy, leaves unfolded clothes in his laundry basket
Cathy's Load	1.5 hr	Cathy does not have a bureau, leaves folded clothes in basket
Dave's Load	2.0 hr	Dave requires all four steps

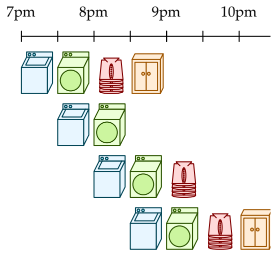
Fixed Time Slot Laundry (Single-Cycle Processors)



Variable Time Slot Laundry (Multi-Cycle Processors)



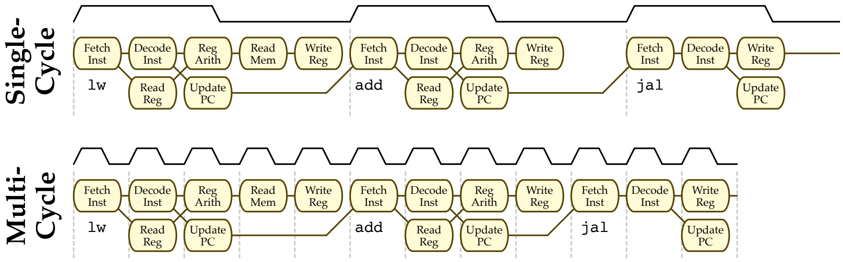
Pipelined Laundry



1.1. Transactions and Steps

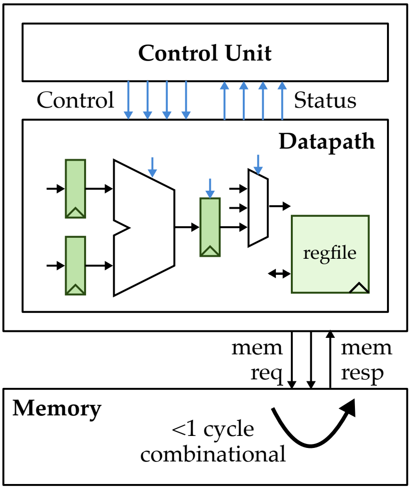
- We can think of each instruction as a transaction
- Executing a transaction involves a sequence of steps

	`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 Constraints

- Assume older 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 memory arrays can only have one read/write port



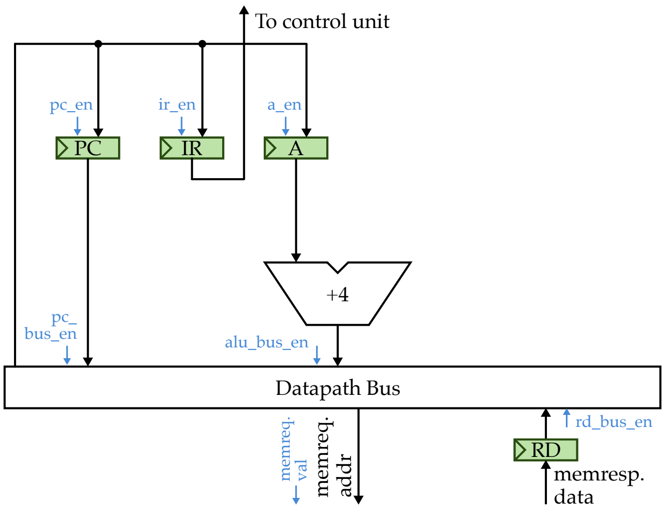
1.3. First-Order Performance Equation

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

- 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. Multi-Cycle Processor

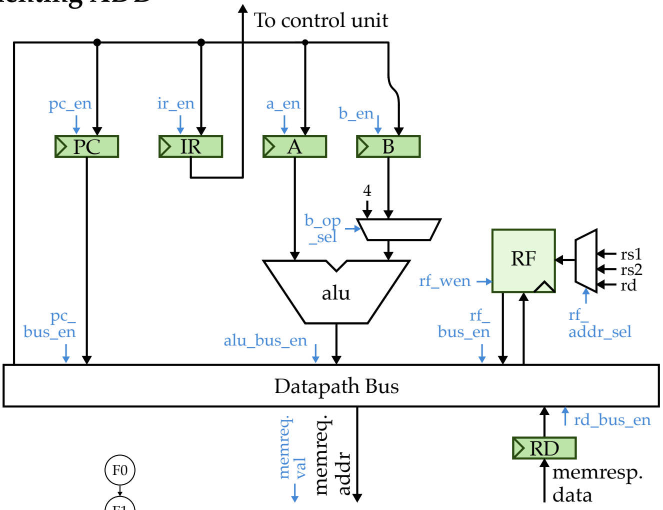


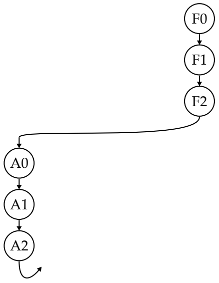
$$\begin{tikzcd} F0 \arrow[d] \\ F1 \arrow[d] \\ F2 \end{tikzcd}$$

(pseudo-control-signal syntax)

	Bus Enables			Reg Enables			MReq
state	**pc**	**alu**	**rd**	**pc**	**ir**	**a**	**val**
F0	1	0	0	0	0	1	1
F1	0	0	1	0	1	0	0
F2	0	1	0	1	0	0	0

Implementing ADD

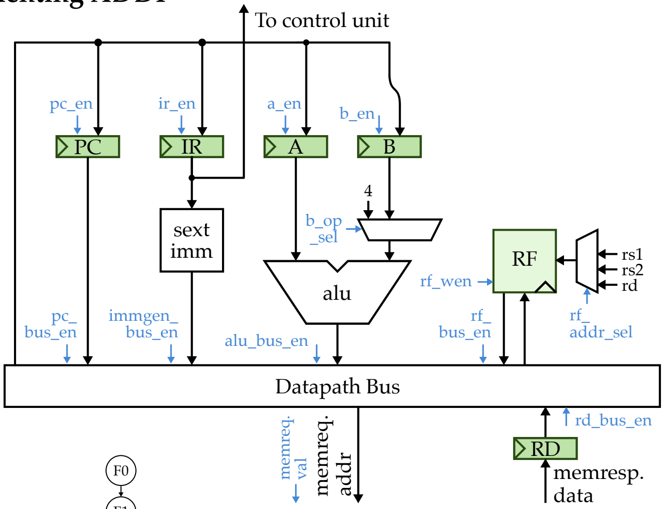


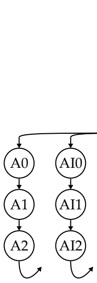


```
add rd, rs1, rs2
```

	Bus Enables				Register Enables				Mux	RF		MReq
state	pc	alu	rf	rd	pc	ir	a	b	bop	sel	wen	val
A0												
A1												
A2												

Implementing ADDI

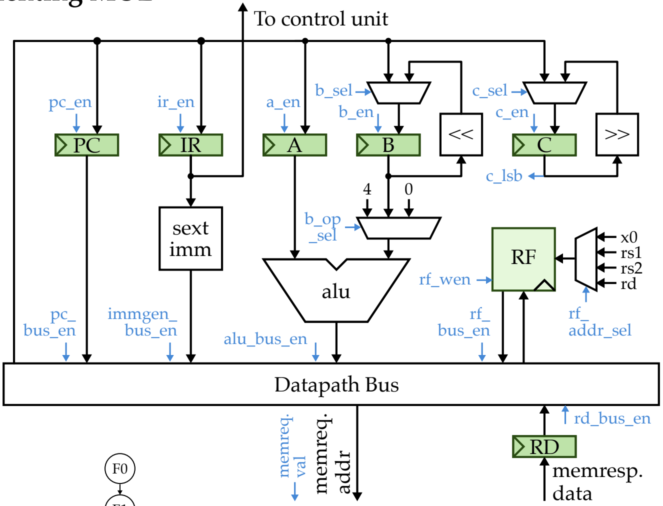


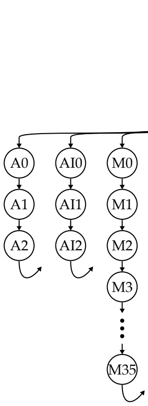


```
add rd, rs1, imm
```

	Bus Enables					Register Enables				Mux	RF		MReq
state	pc	ig	alu	rf	rd	pc	ir	a	b	bop	sel	wen	val
AI0													
AI1													
AI2													

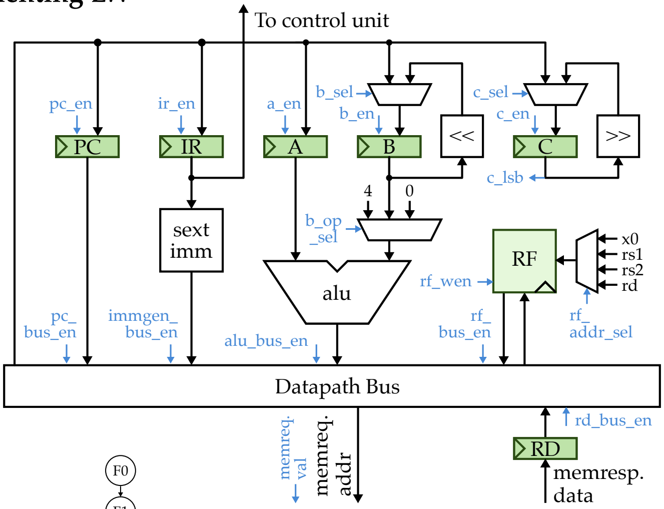
Implementing MUL

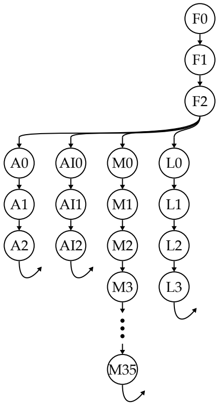




```
mul rd, rs1, rs2
```

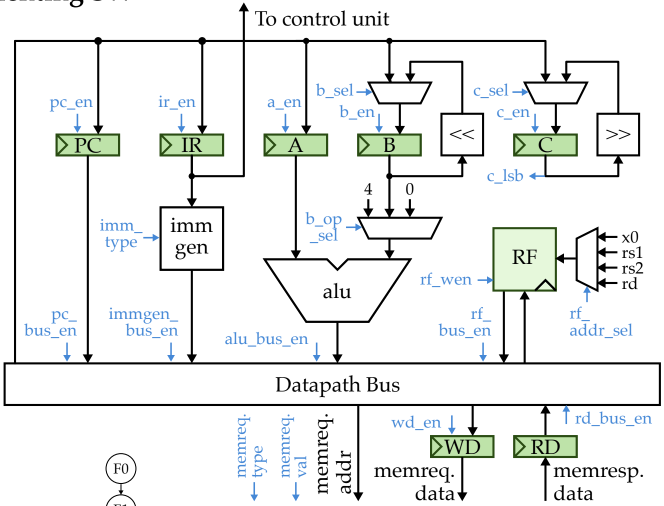
Implementing LW

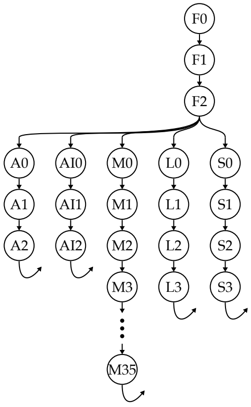




`lw rd, imm(rs1)`

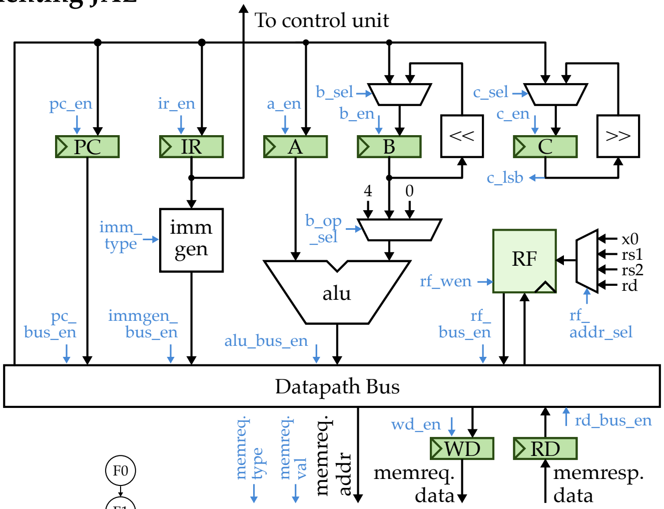
Implementing SW

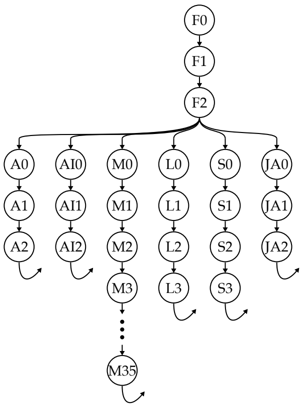




```
sw rs2, imm(rs1)
```

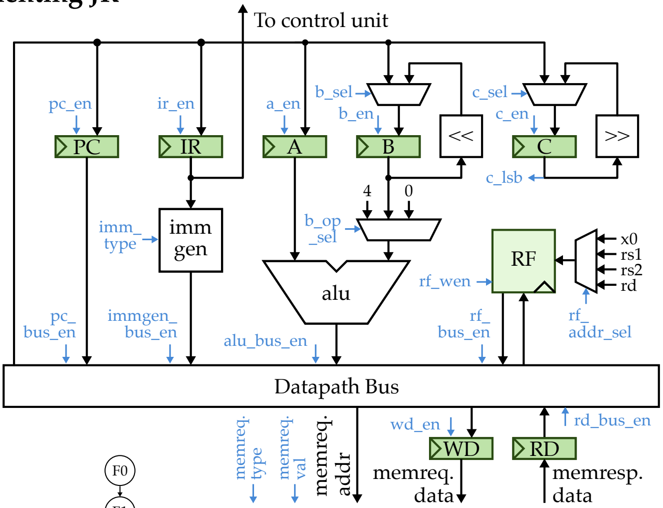
Implementing JAL

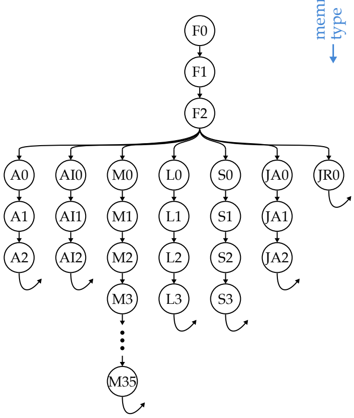




```
jal rd, imm
```

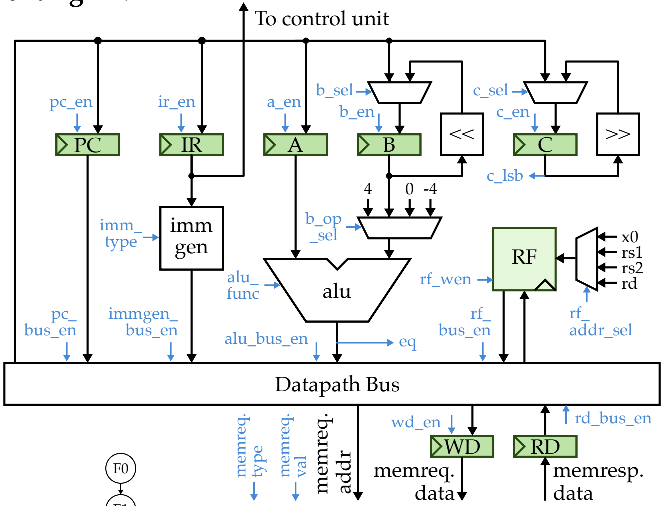
Implementing JR

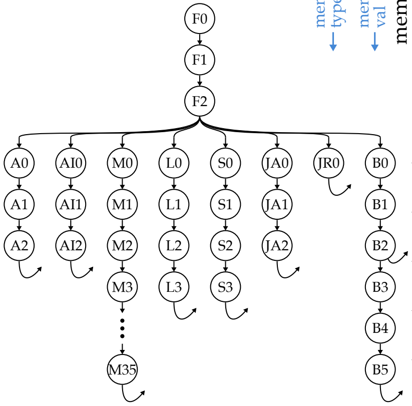




`jr rs1`

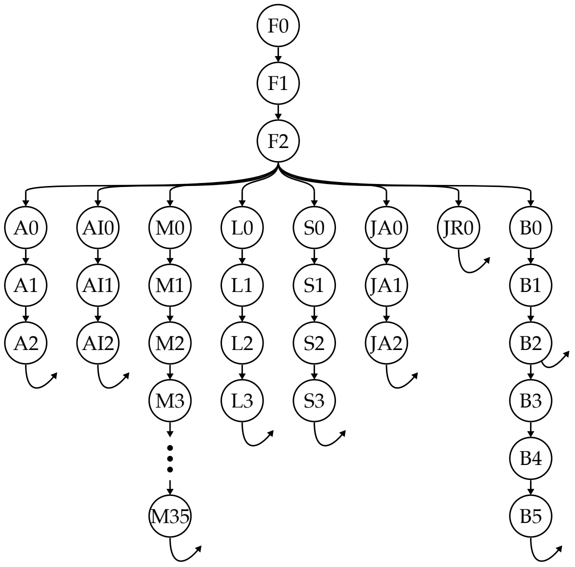
Implementing BNE

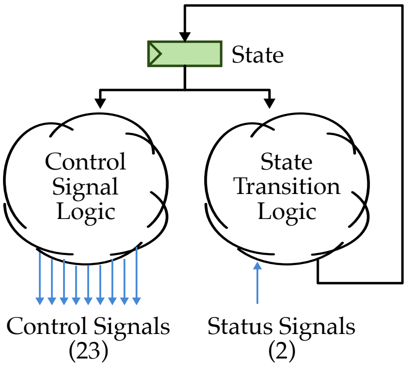


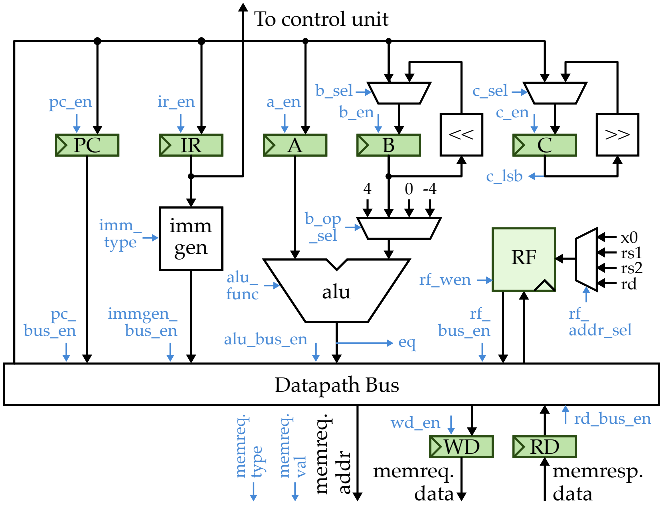


```
bne rs1, rs2, imm
```

Multi-Cycle Processor Final 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

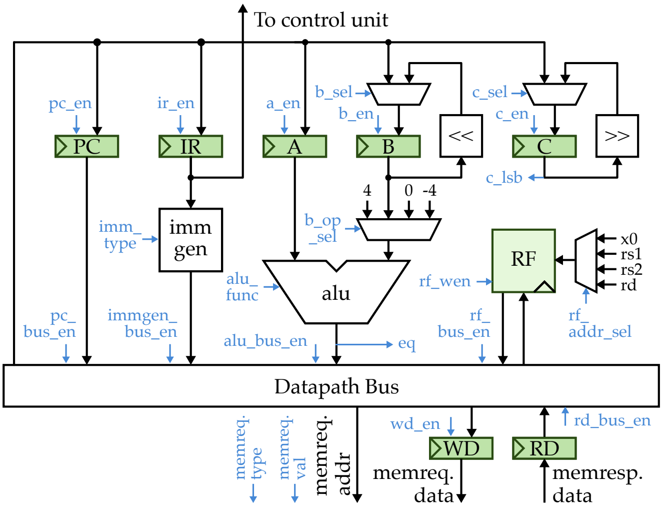
	Bus Enables					Register Enables						Mux			Func		RF		MReq	
state	**pc**	**ig**	**alu**	**rf**	**rd**	**pc**	**ir**	**a**	**b**	**c**	**wd**	**b**	**c**	**bop**	**ig**	**alu**	**sel**	**wen**	**val**	**type**
F0	1	0	0	0	0	0	0	1	0	0	0	x	x	x	x	x	x	0	1	rd
F1	0	0	0	0	1	0	1	0	0	0	0	x	x	x	x	x	x	0	0	x
F2	0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

Adding a Complex Instruction

Adding new complex instructions usually does not require datapath modifications, only additional states.

```
add.mm rd, rs1, rs2
```

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

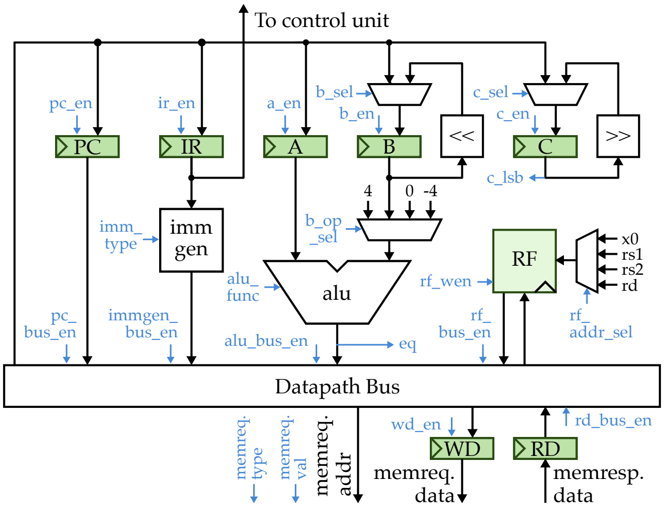


Adding a New Auto-Incrementing Load Instruction

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

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

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

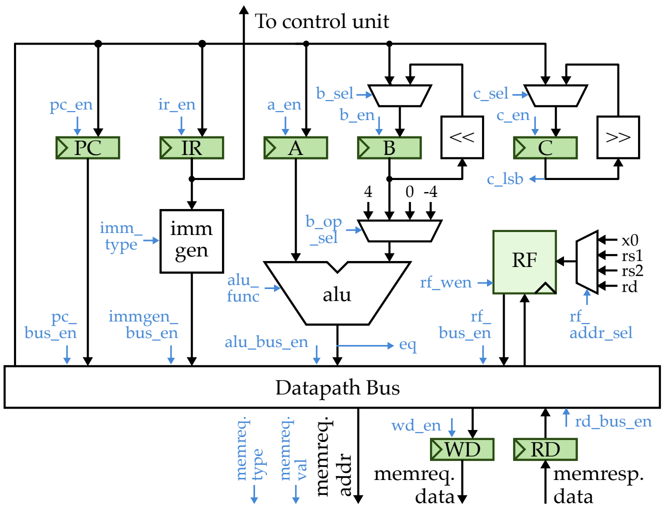


2.1. Analyzing Performance

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

- 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 ALU	64τ
32-bit Shifter	0τ
ImmGen Unit	12τ
Datapath Bus	20τ
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

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

Assembly Code

```
 # addr(dest[i]):x1, addr(src0[i]):x2
 # addr(src1[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`														
`lw   x5, 0(x1)`														
`lw   x6, 0(x2)`														
`add  x7, x5, x6`														
`sw   x7, 0(x3)`														

How long in units of τ will it take to execute the find program assuming `n` is 64 and only the first element matches the given value.

Pseudo-Code

```
found = 0
for i in range(n):
  if ( src0[i] == value )
    found = 1
```

Assembly Code

```
 # addr(src0[i]):x1, n:x2, value:x3
 addi  x5, x0, 0

loop:
 lw    x4, 0(x1)
 bne   x4, x3, neq
 addi  x5, x0, 1

neq:
 addi  x1, x1, 4
 addi  x2, x2, -1
 bne   x2, x0, loop
```

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