Lecture 11
Processor Microarchitecture (Part 2)

Xuan ‘Silvia’ Zhang
Washington University in St. Louis

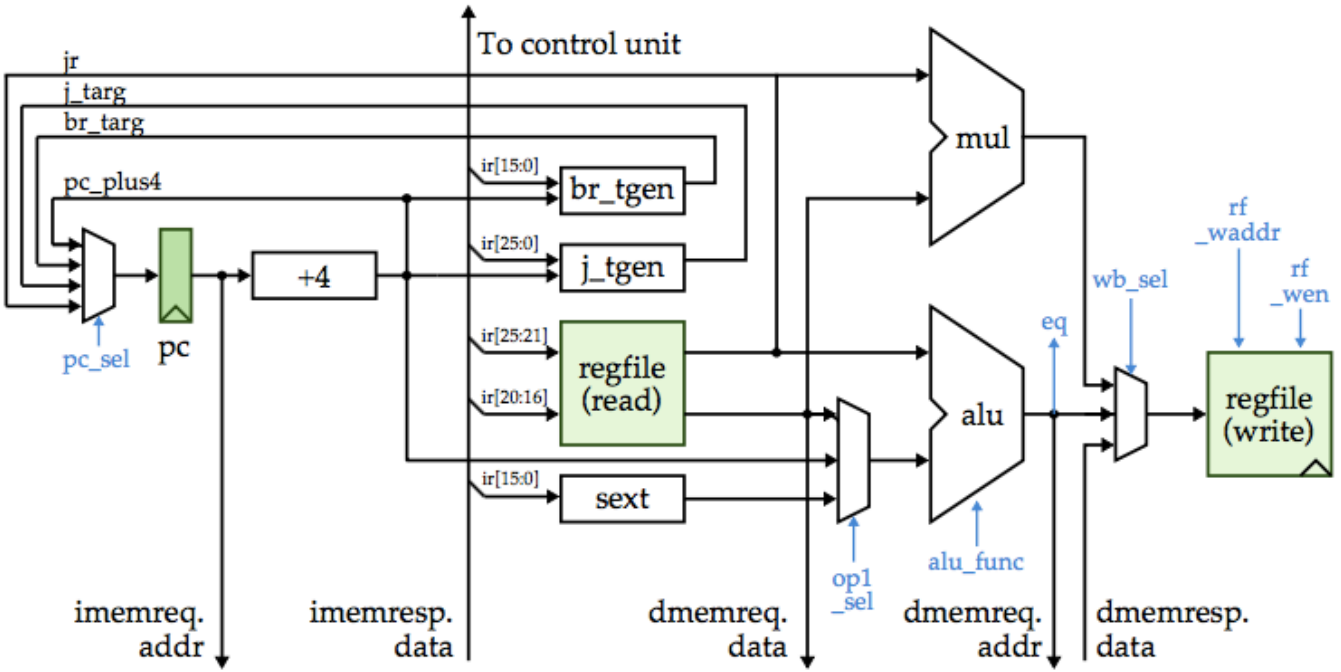
http://classes.engineering.wustl.edu/ese566/

Quiz: 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 rt, offset(rs)
```

$$R[rt] \leftarrow M[\ R[rs] + sext(offset)\ ];\ R[rs] \leftarrow R[rs] + 4$$

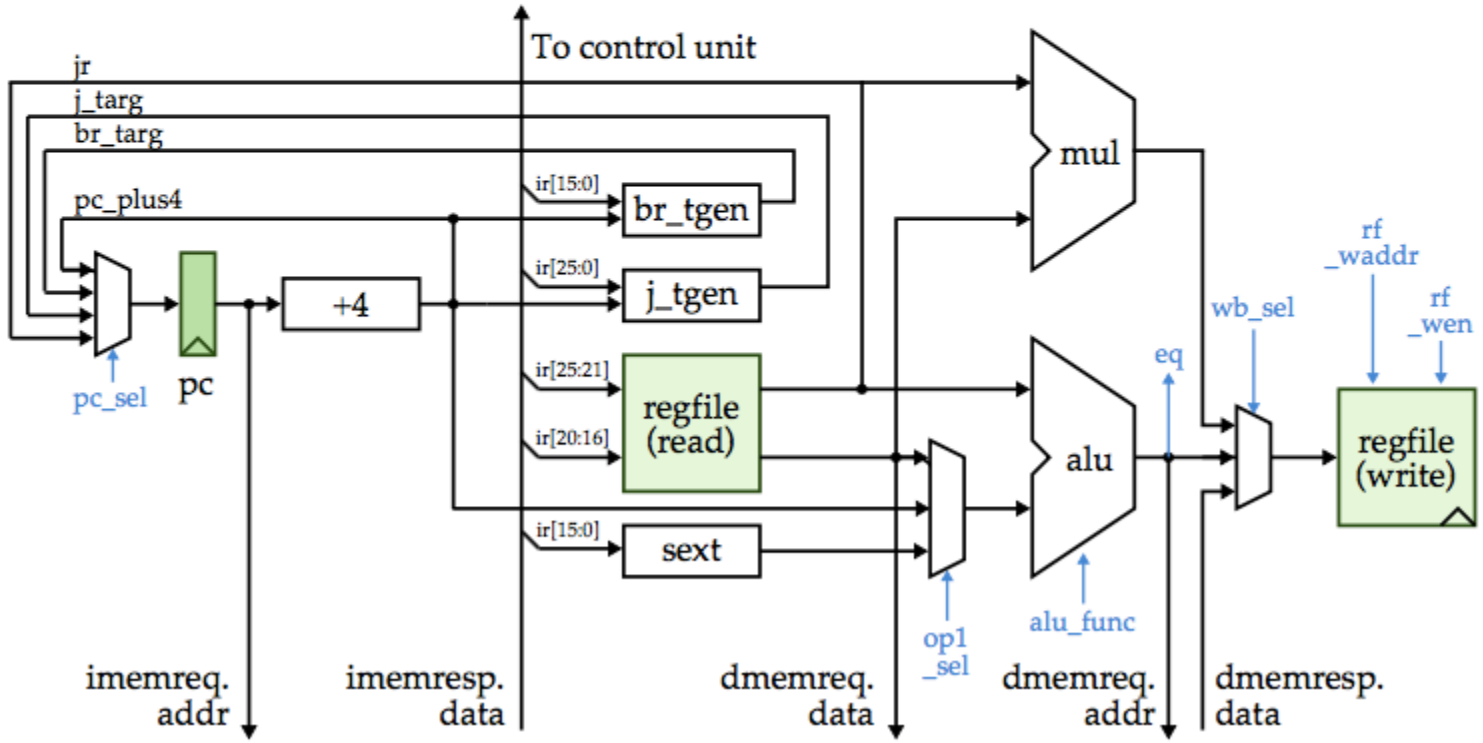


Single-Cycle Processor Control Unit

inst	pc sel	op1 sel	alu func	wb sel	rf waddr	rf wen	imem req val	dmem req val
addu	pc+4	rf	+	alu	rd	1	1	0
addiu								
mul	pc+4	rf	×	mul	rd	1	1	0
lw	pc+4	sext	+	mem	rt	1	1	1
sw								
j	j_targ	–	–	–	–	0	1	0
jal								
jr	jr	–	–	–	–	0	1	0
bne								
lw.ai								

Estimating Cycle Time—Longest Critical Path

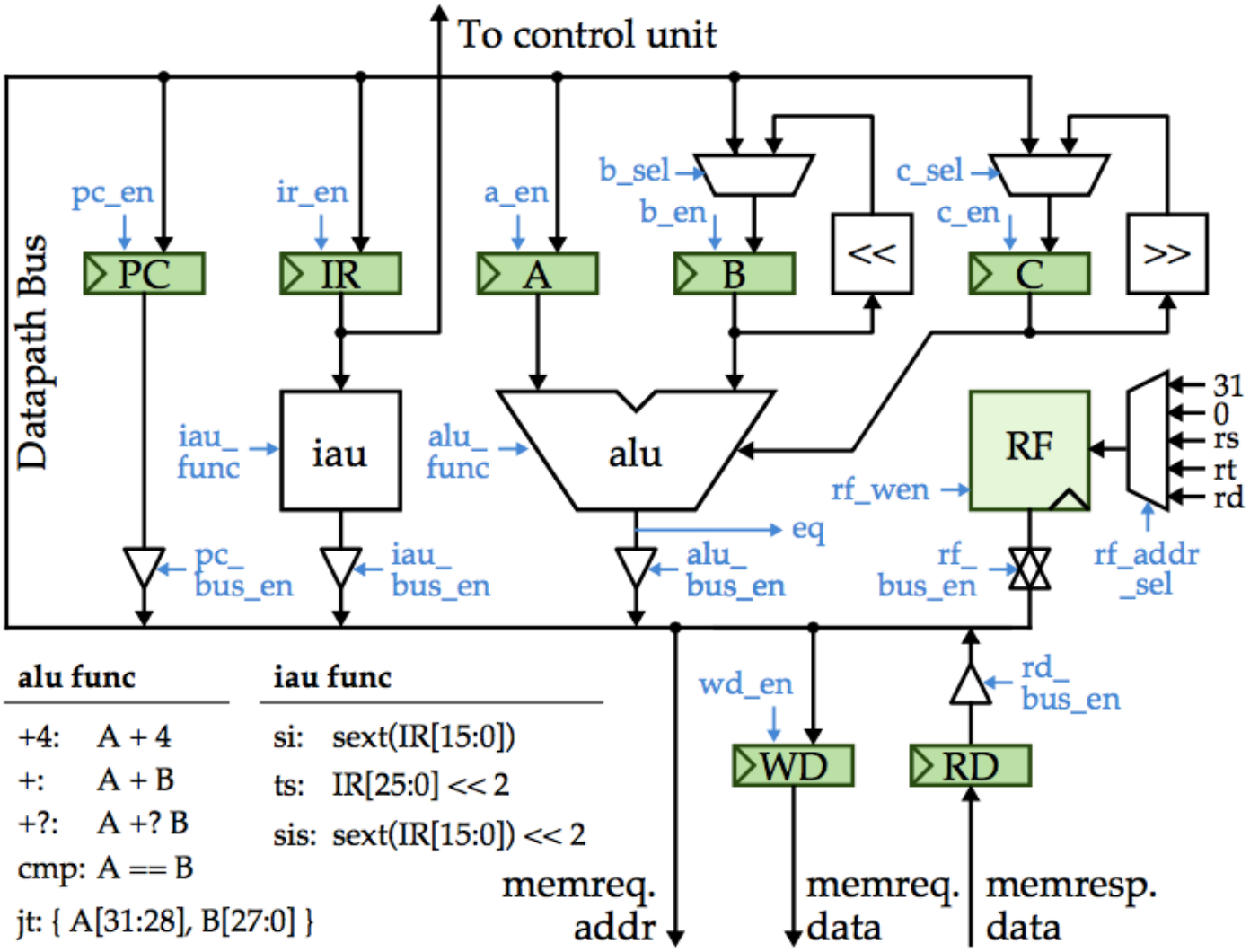
There are many paths through the design that start at a state element and end at a state element. The "critical path" is the longest path across all of these paths. We can usually use a simple first-order static timing estimate to estimate the cycle time (i.e., the clock period and thus also the clock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



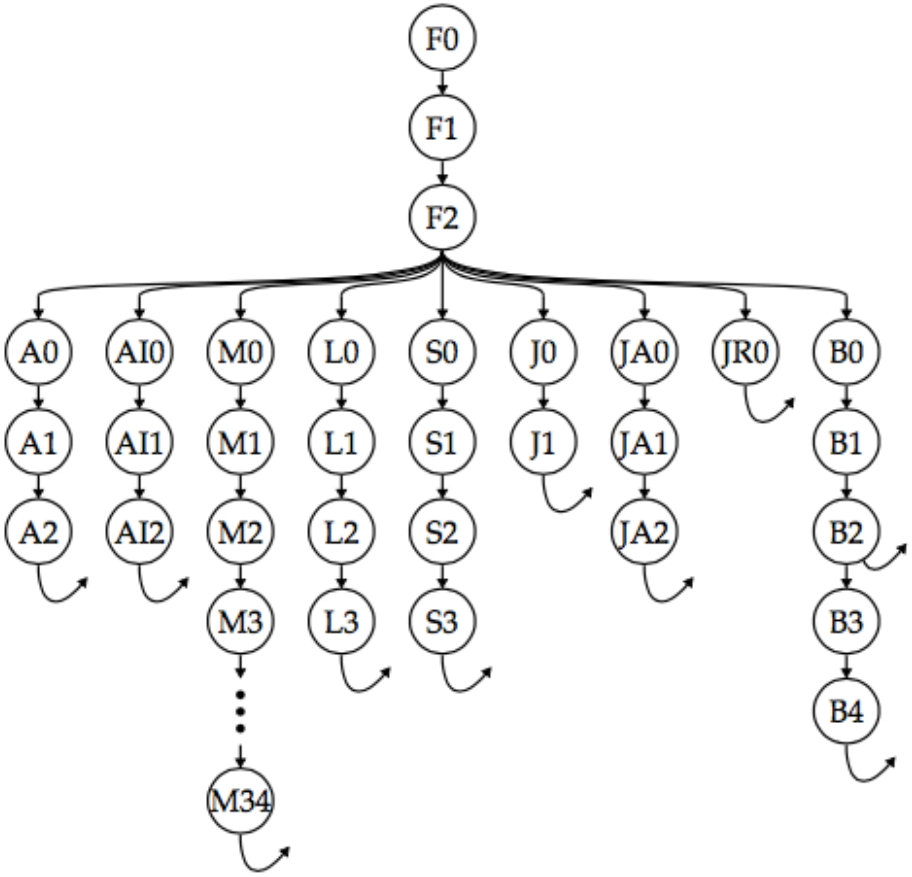
Quiz: Adding a New Auto-Incrementing Load Instruction

`lw.ai rt, offset(rs)`

$$R[\texttt{rt}] \leftarrow M[\, R[\texttt{rs}] + \text{sext}(\texttt{offset})\,];\; R[\texttt{rs}] \leftarrow R[\texttt{rs}] + 4$$



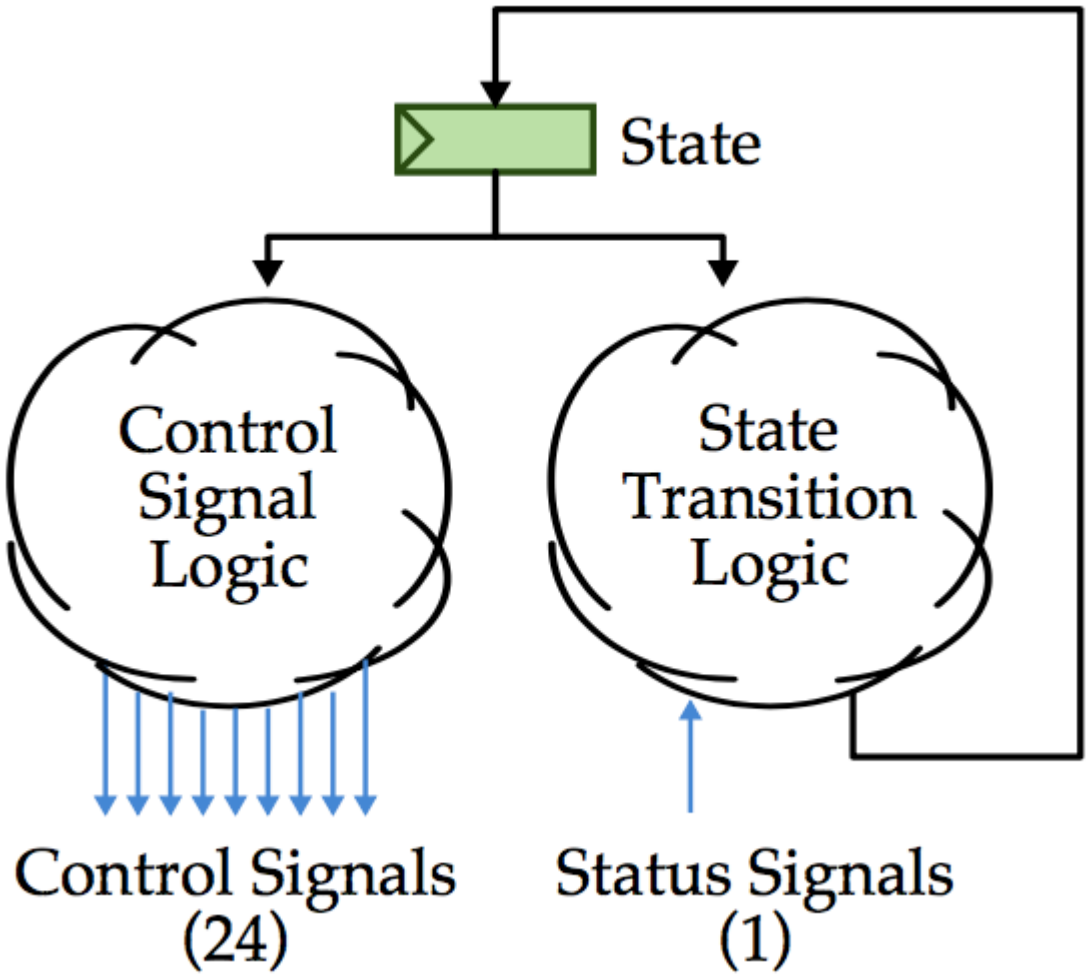
FSM Processor Control Unit



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



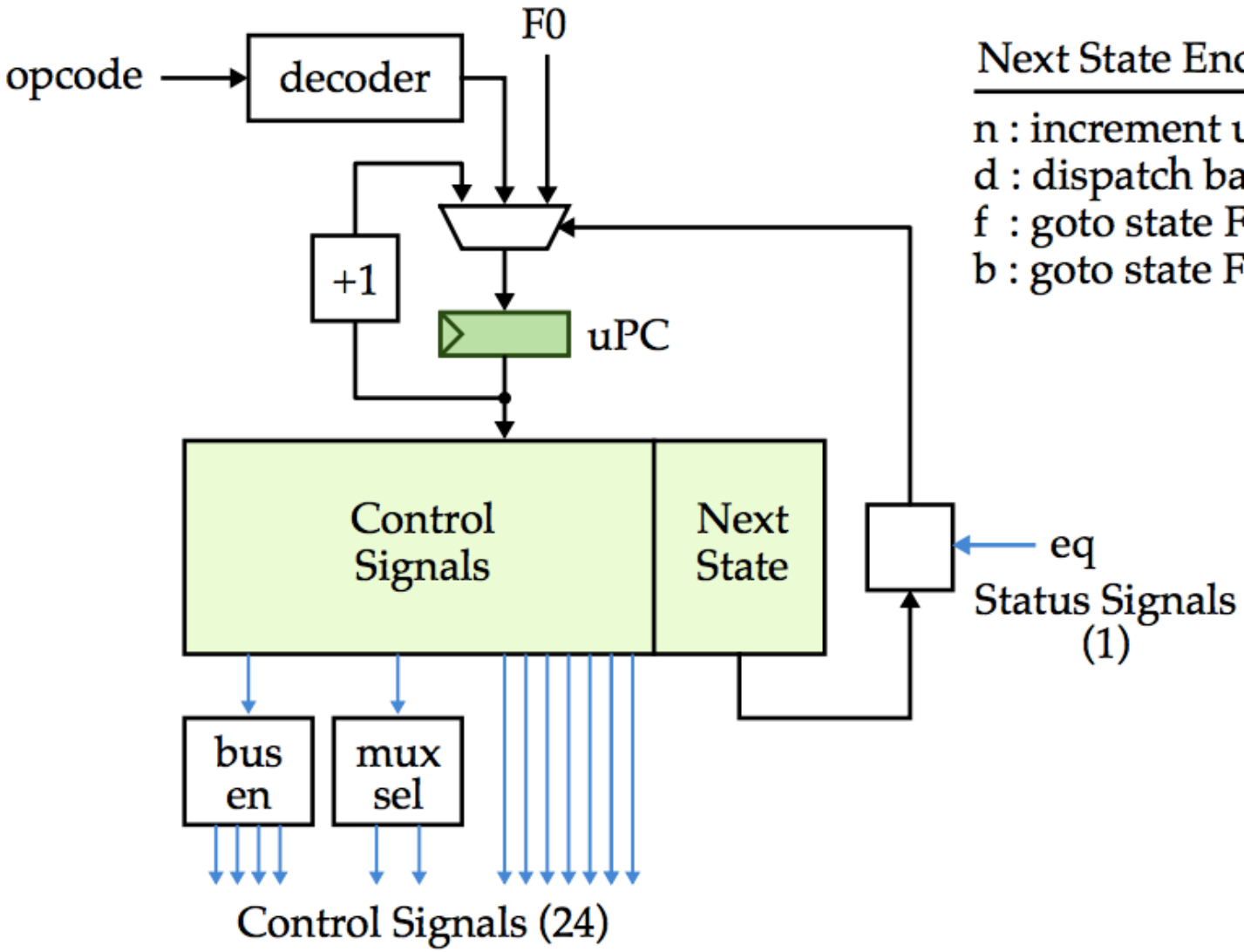
Control Signal Output Table

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

A0: A ← RF[rs]
A1: B ← RF[rt]
A2: RF[rd] ← A + B; goto F0

	Bus Enables					Register Enables						Mux		Func		RF		MReq	
state	pc	iau	alu	rf	rd	pc	ir	a	b	c	wd	b	c	iau	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	1	0	0	0	–	–	–	+4	–	0	0	–
A0																			
A1																			
A2																			

Vertically Microcoded FSM



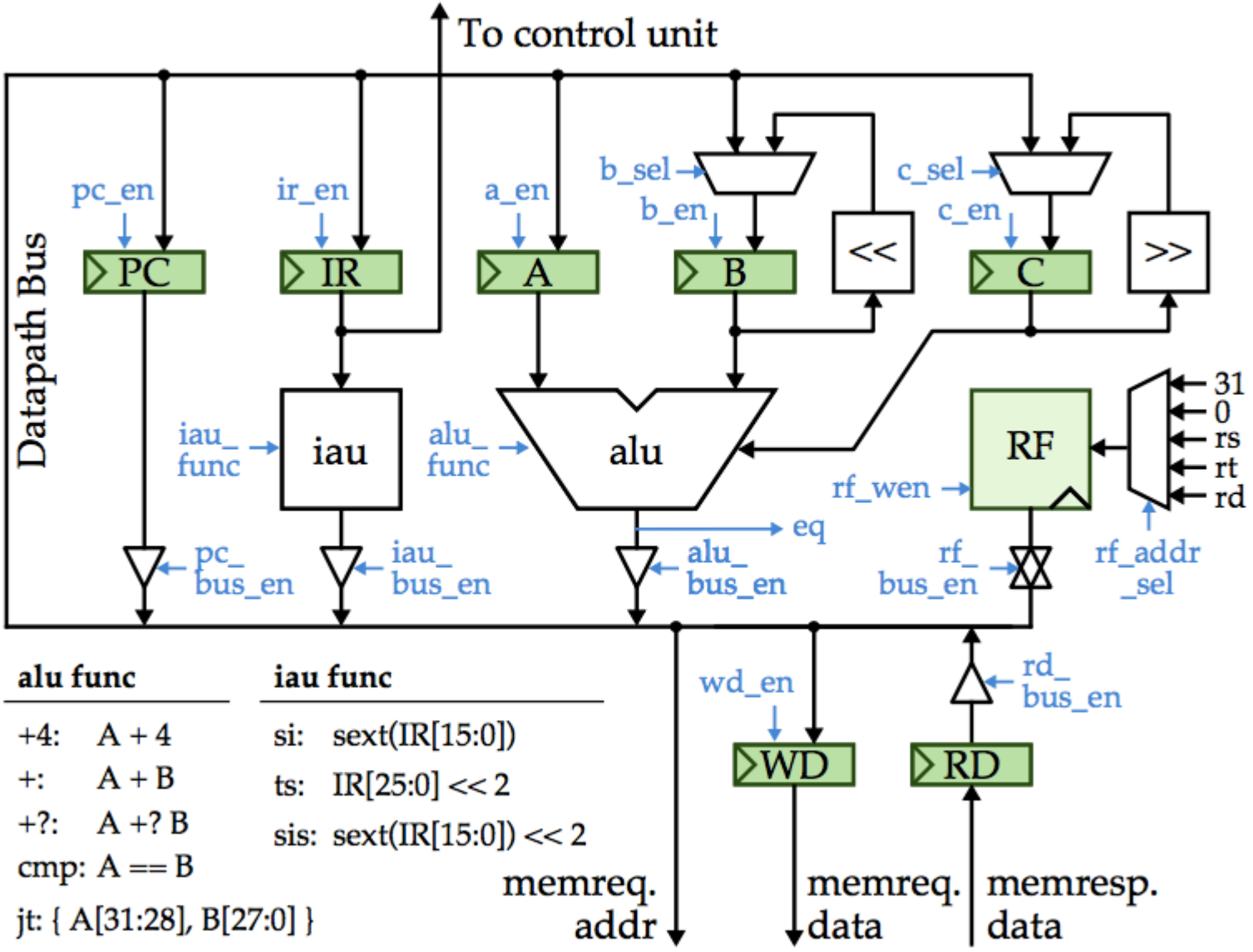
Next State Encoding

n : increment uPC by one
d : dispatch based on opcode
f : goto state F0
b : goto state F0 if A == B

Vertically Microcoded FSM

- Use memory array to encode control logic and state transition logic
  - called control state
  - more flexible than random logic
- Enable 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
  - enable in-field updates
- Read-write control stores can simplify diagnostics and microcode patches

Estimating Cycle Time

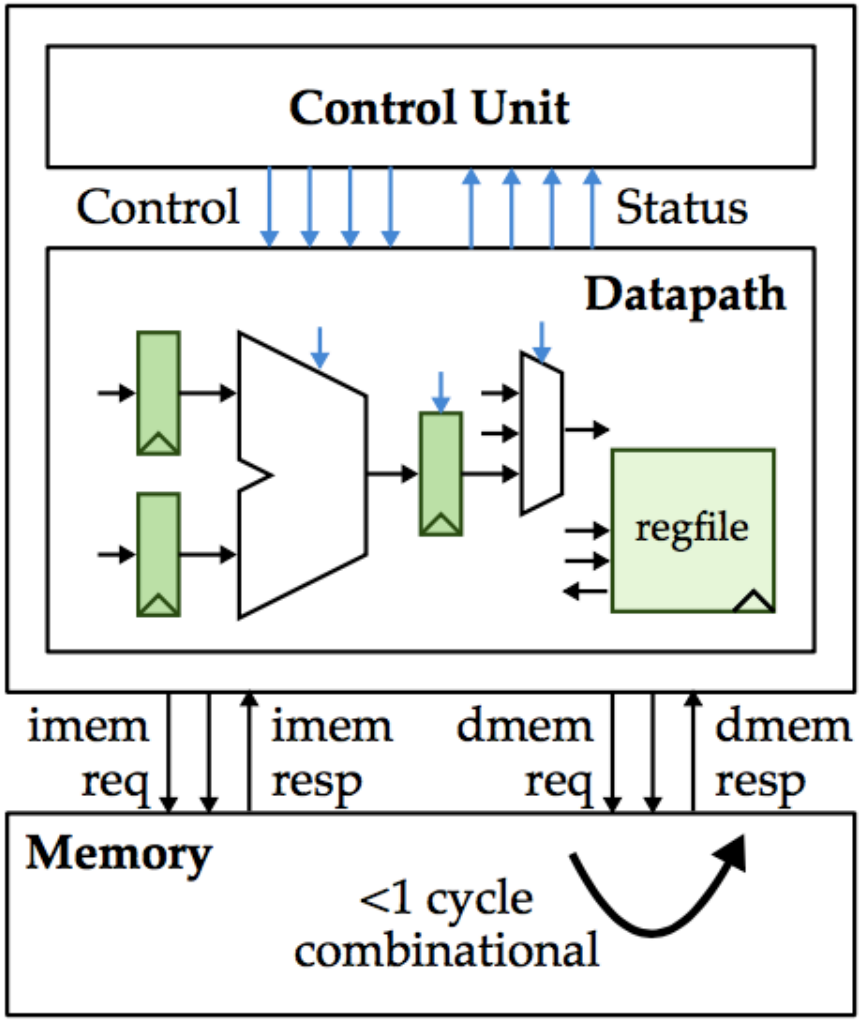


Microarchitecture	CPI	Cycle Time
Single-Cycle Processor	1	long
FSM Processor	>1	short
Pipelined Processor	≈1	short

PARCv1 Pipelined Processor

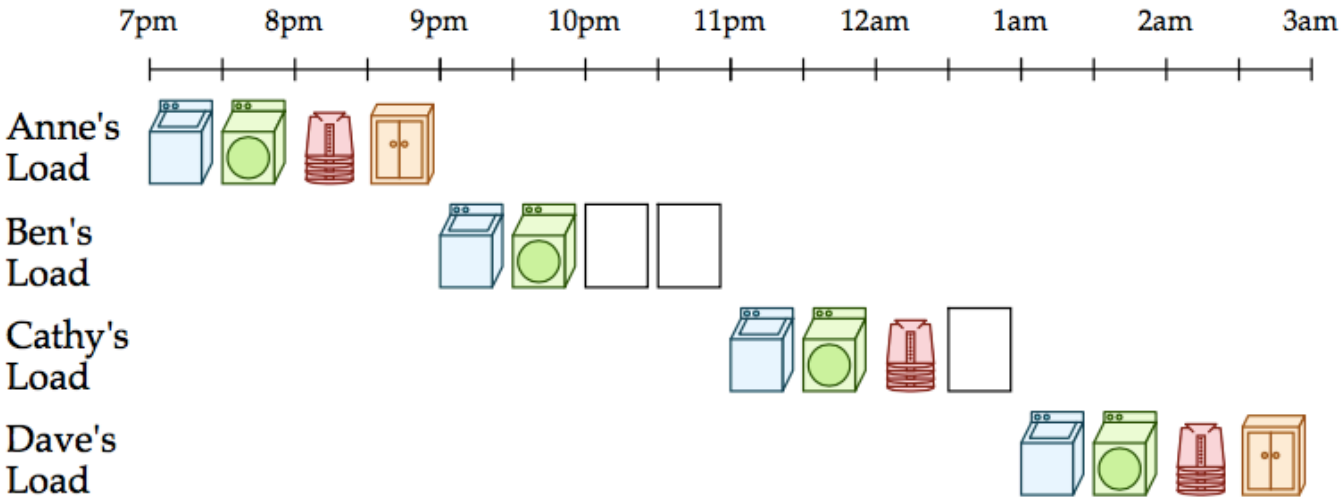
Technology Constraints

- 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

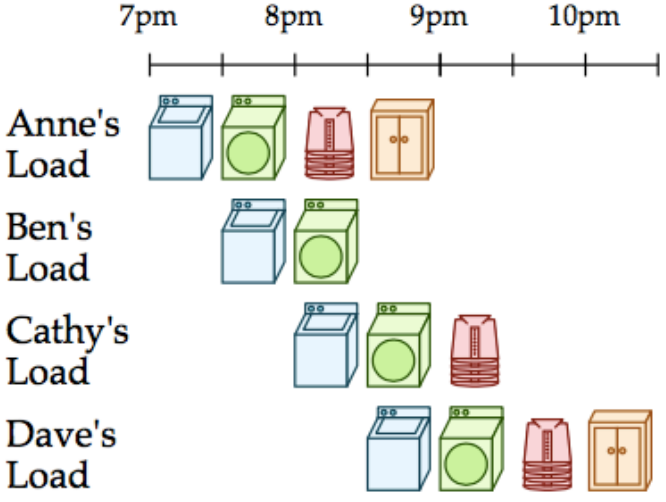


High-Level Idea for Pipelined Processors

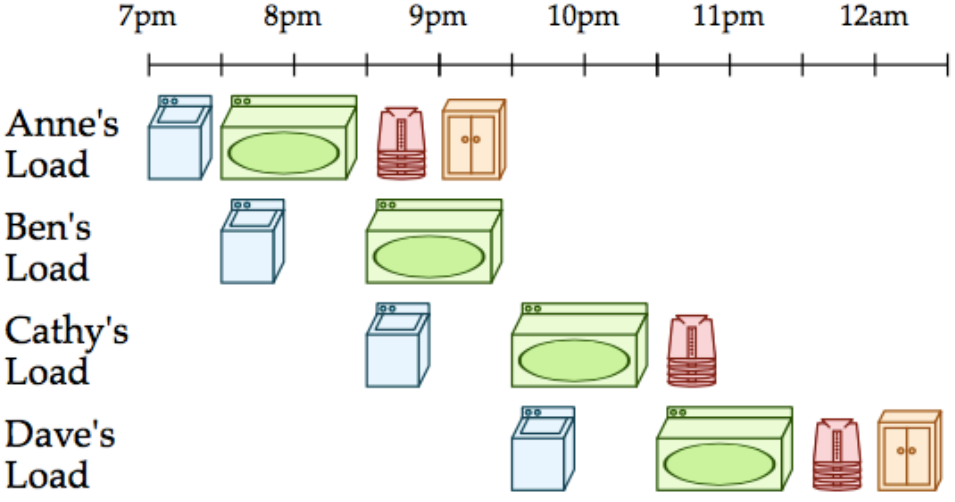
Fixed Time-Slot Laundry



Pipelined Laundry



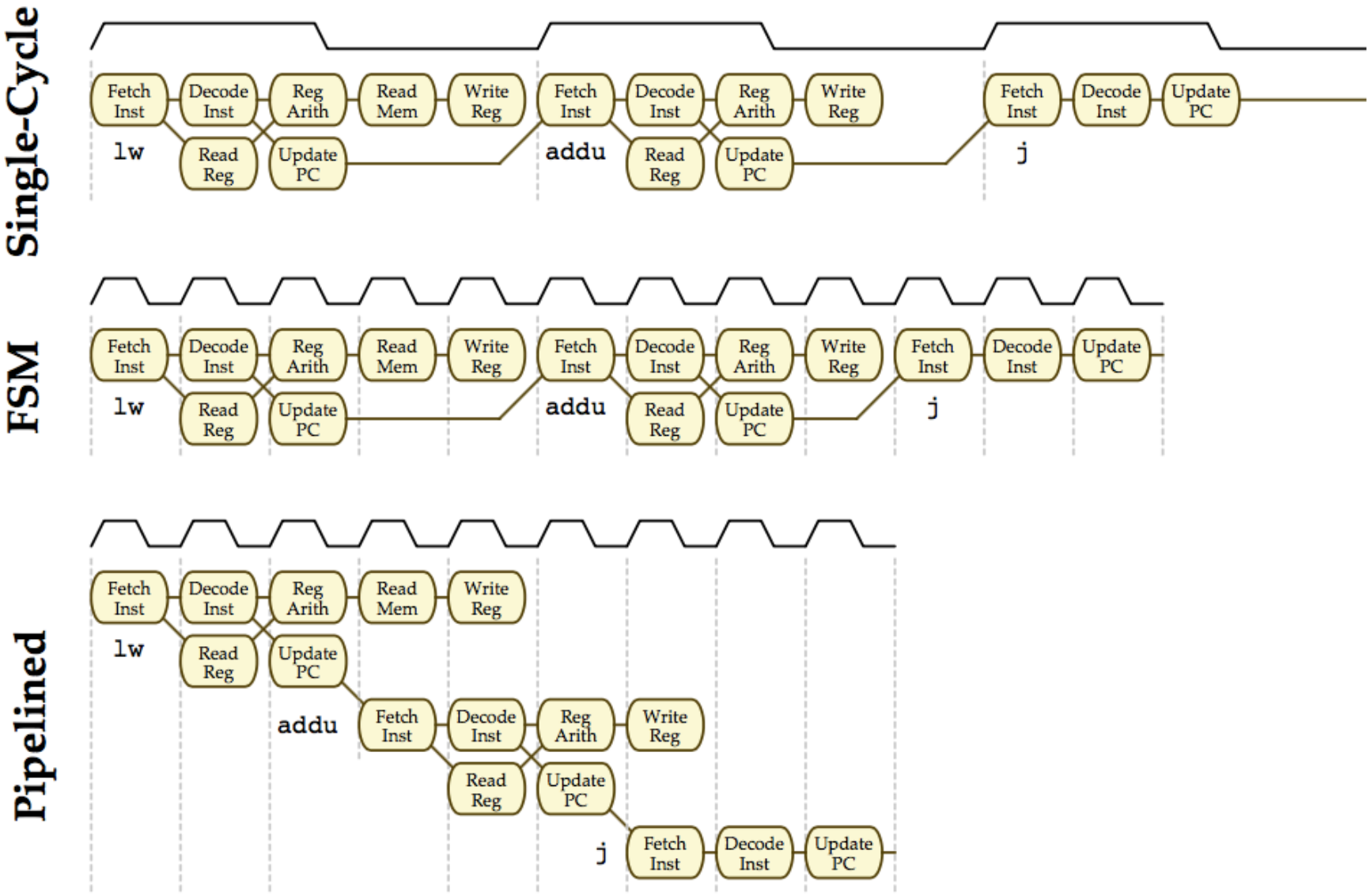
Pipelined Laundry with Slow Dryers



High-Level Idea for Pipelined Processors

- Key pipeline traits
  - multiple transactions operate simultaneously using different resources
  - pipelining does not help the transaction latency
  - pipelining does help the transaction throughput
  - potential speed up is proportional to the number of pipelined stages
  - potential speed up is limited by the slowest pipeline stage
  - potential speed up is reduced by time to fill the pipeline

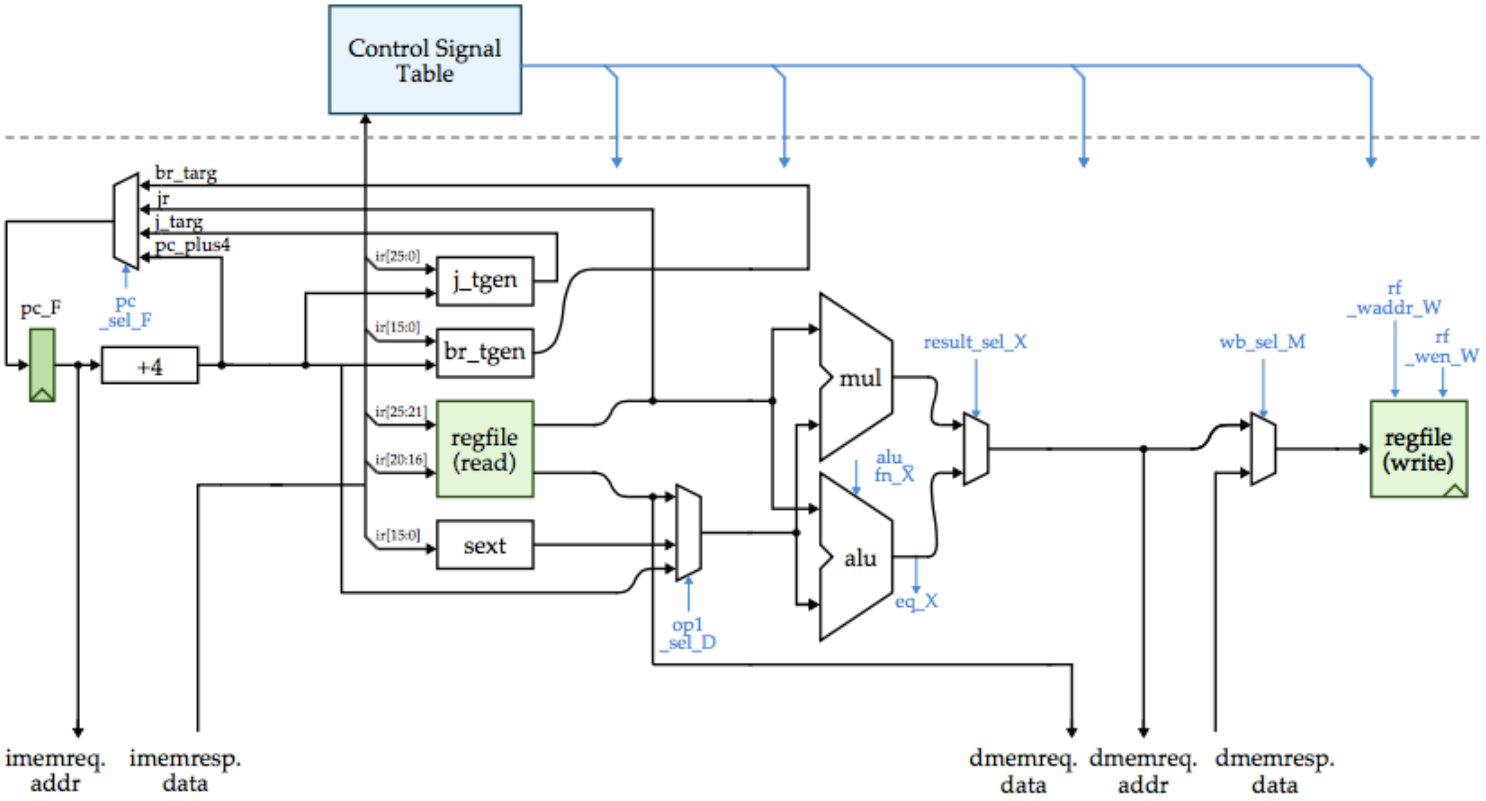
High-Level Idea for Pipelined Processors



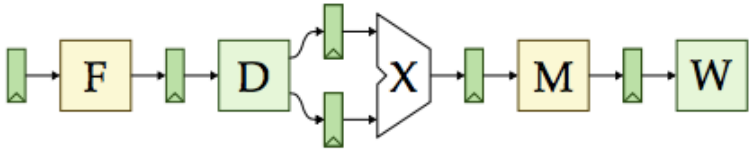
Pipelined Processor Datapath and Control Unit

- 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

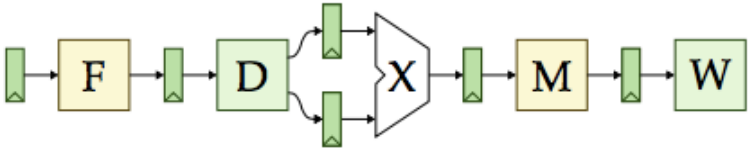
Pipelined Processor Datapath and Control Unit



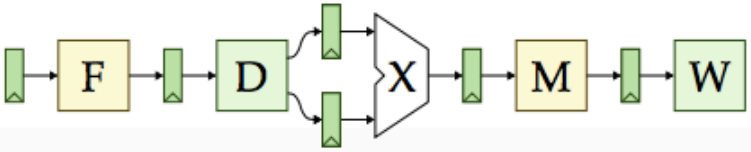

```
addiu r1, r2, 1
```




```
addiu r3, r4, 1
```



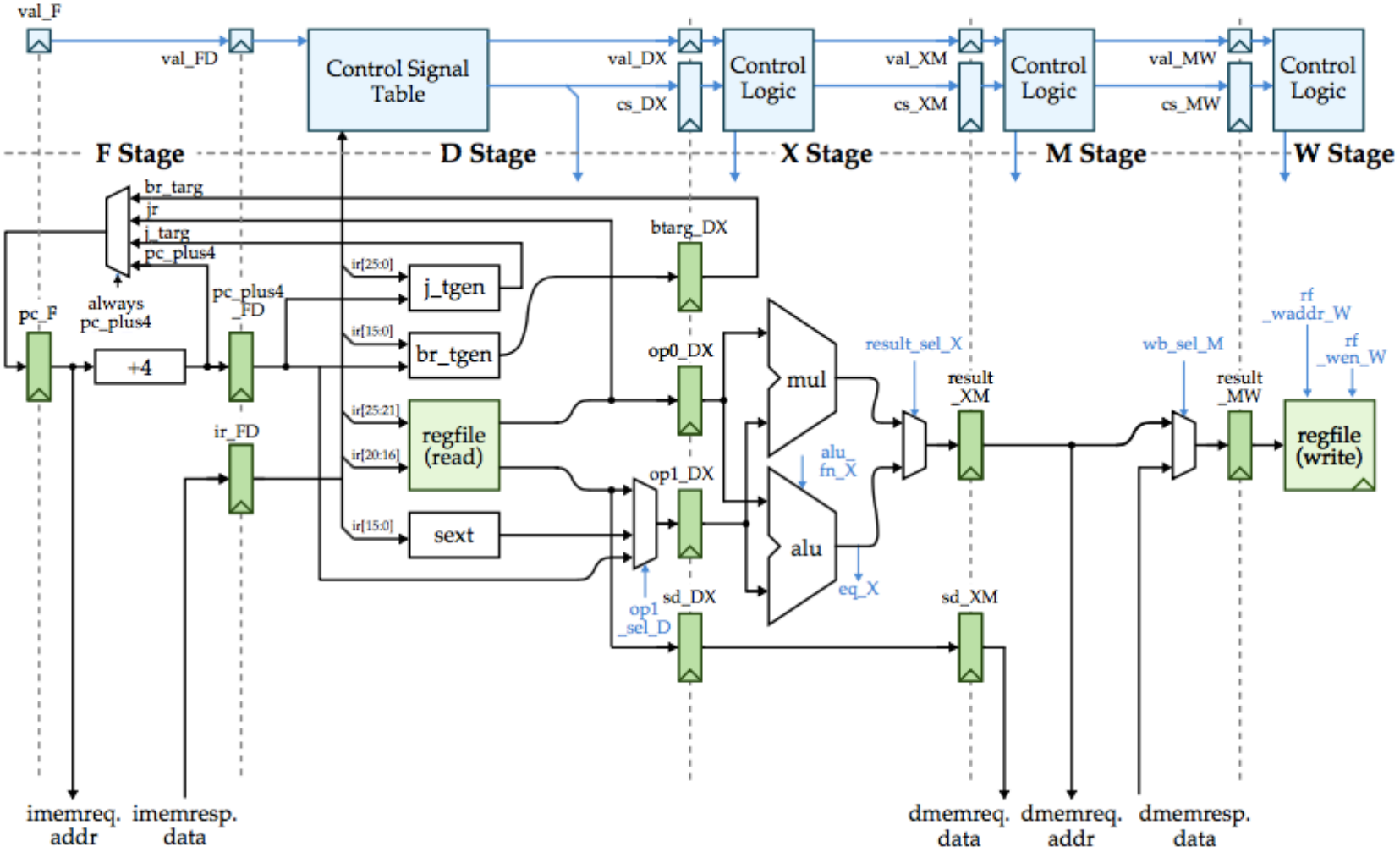

```
addiu r5, r6, 1
```



Quiz: Adding a New Auto-Incrementing Load Instruction

```
lw.ai rt, imm(rs)
```

$$R[rt] \leftarrow M[\, R[rs] + sext(imm)\,];\; R[rs] \leftarrow R[rs] + 4$$



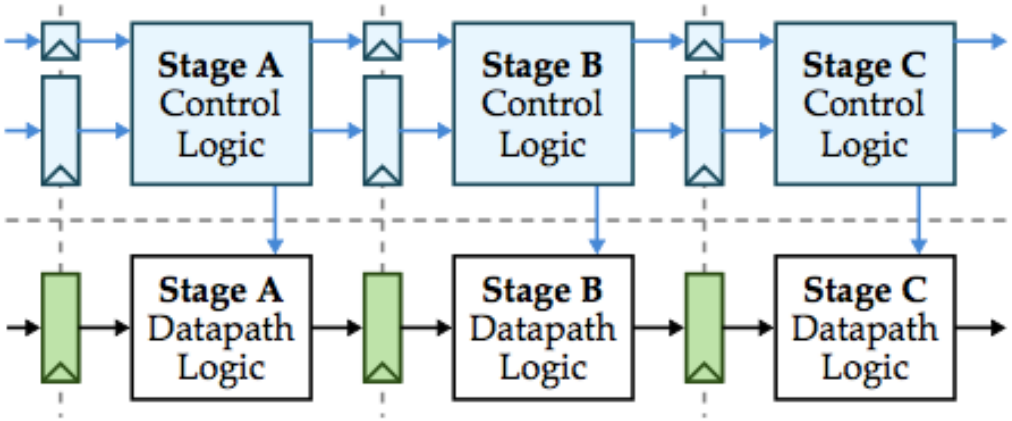
Pipeline Hazards

- 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 one
- 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

Proposed Solutions: Stalling and Squashing Instructions

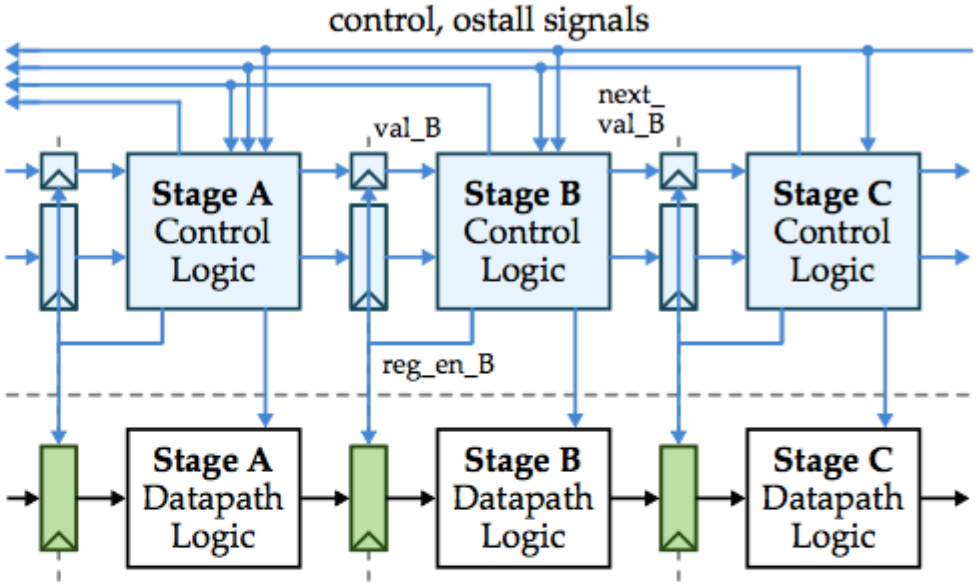
- Stalling
  - an instruction originates a stall due to a hazard, causing all instructions earlier in the pipeline to also stall. When the hazard is resolved, the instruction no longer needs to stall and the pipeline starts flowing again.
- Squashing
  - an instruction originates a squash due to a hazard, 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
```

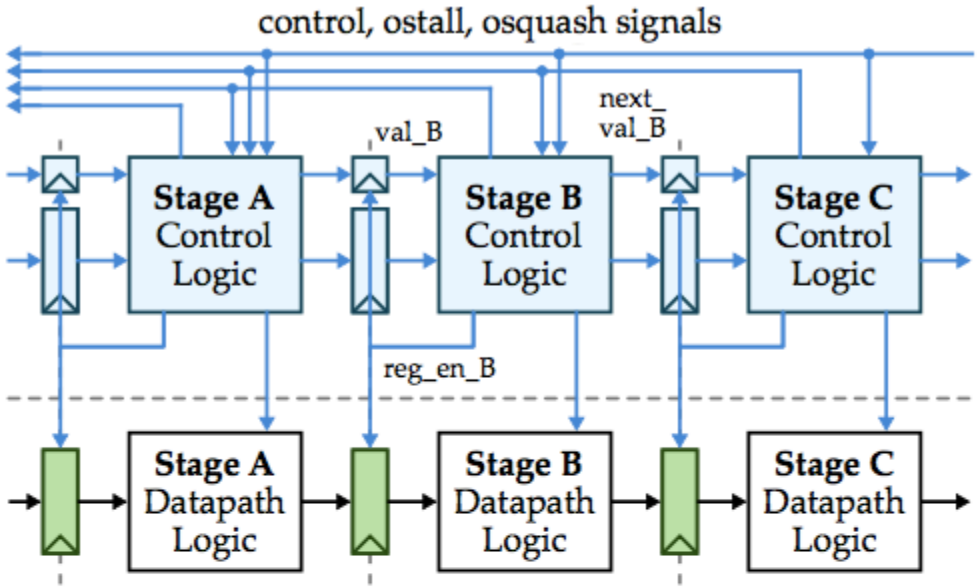
Control Logic with Stalling and No Squashing




```
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

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

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

ostall_B   = val_B && !squash_B && ( ostall_hazard1_B || ostall_hazard2_B )

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

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

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

Control Logic with Stalling and Squashing

`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`.
`ostall_B`	A squash takes priority, since a squashed transaction is invalid and thus it should not originate a stall.
`stall_B`	A squash takes priority, since a squashed transaction is invalid and thus it should not actually stall.
`osquash_B`	Originating squash due to hazards detected in B stage. A squash takes priority, since a squashed transaction is invalid and thus it should not originate a squash. A stall also takes priority, since a stalling transactions should not originate a squash.
`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.

Questions?

Comments?

Discussion?

Acknowledgement

Cornell University, ECE 4750