CS 61C: Great Ideas in Computer Architecture

MIPS CPU Control, Pipelining

Instructor: Alan Christopher

Agenda

- Quick Datapath Review
- Control Implementation
- Administrivia
- Clocking Methodology
- Pipelined Execution
- Pipelined Datapath

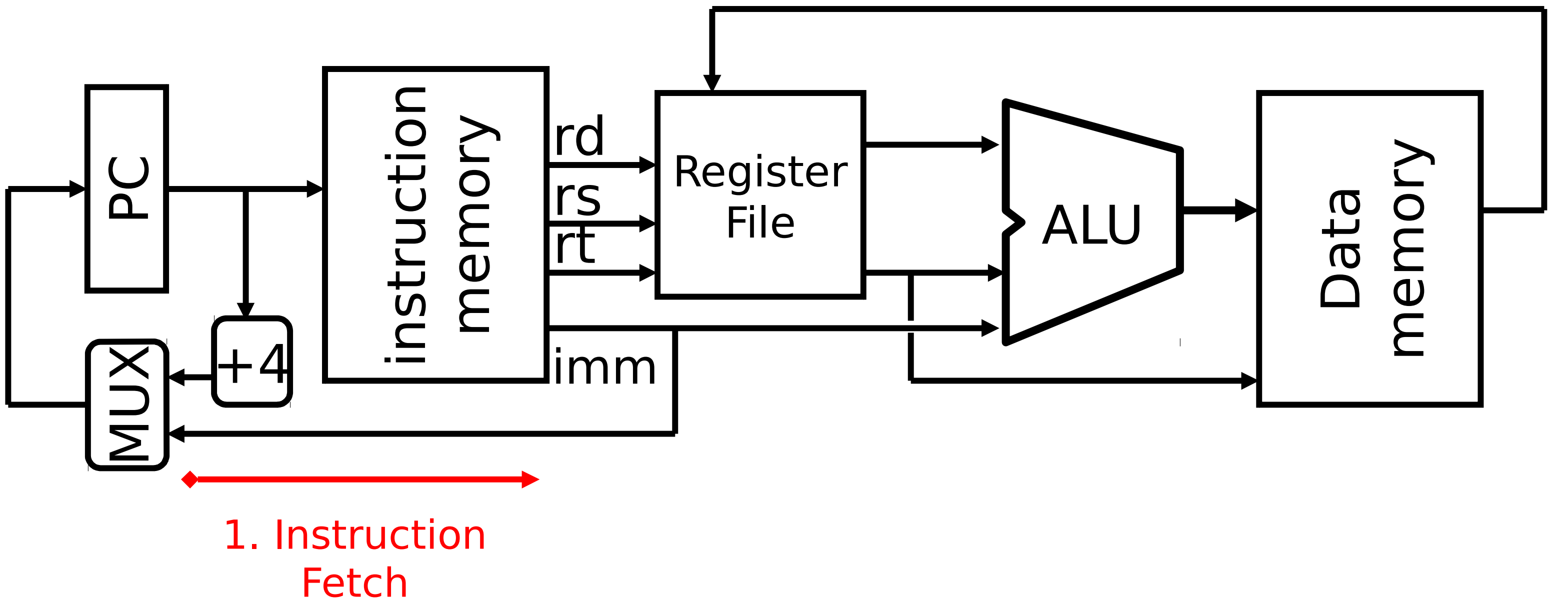
Datapath Review

- Part of the processor; the *hardware* necessary to perform *all* operations required
  - Depends on exact ISA, RTL of instructions

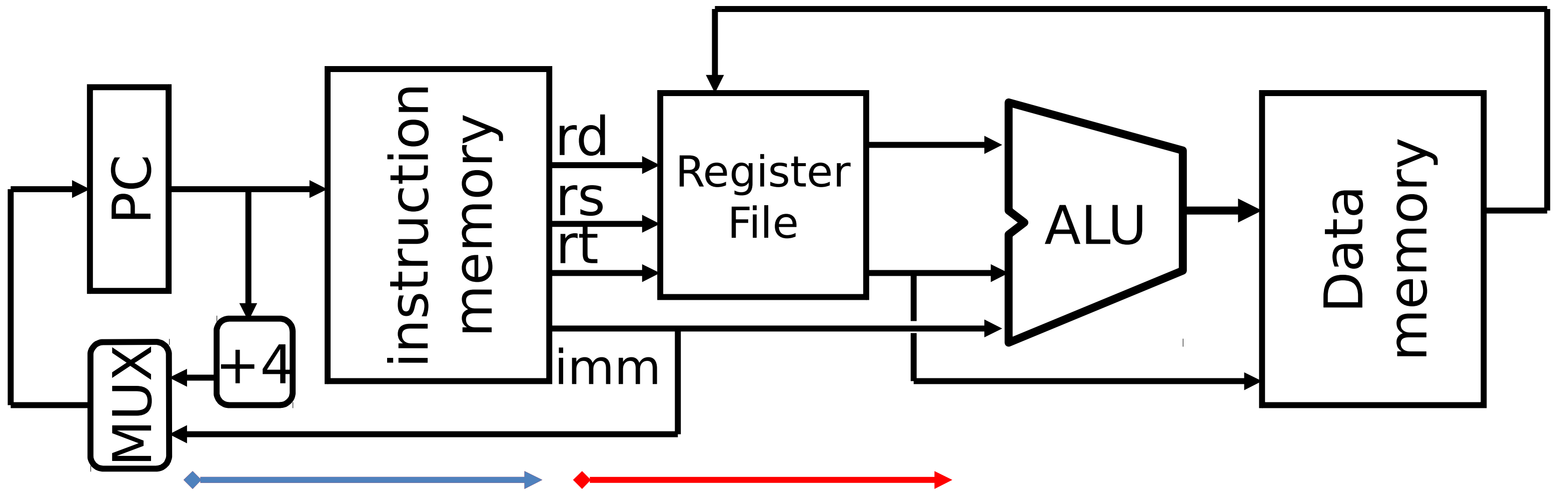
Datapath Review

- Part of the processor; the *hardware* necessary to perform *all* operations required
  - Depends on exact ISA, RTL of instructions
- Major components:
  - PC and Instruction Memory
  - Register File (RegFile holds registers)
  - Extender (sign/zero extend)
  - ALU for operations (on two operands)
  - Data Memory

Five Stages of the Datapath



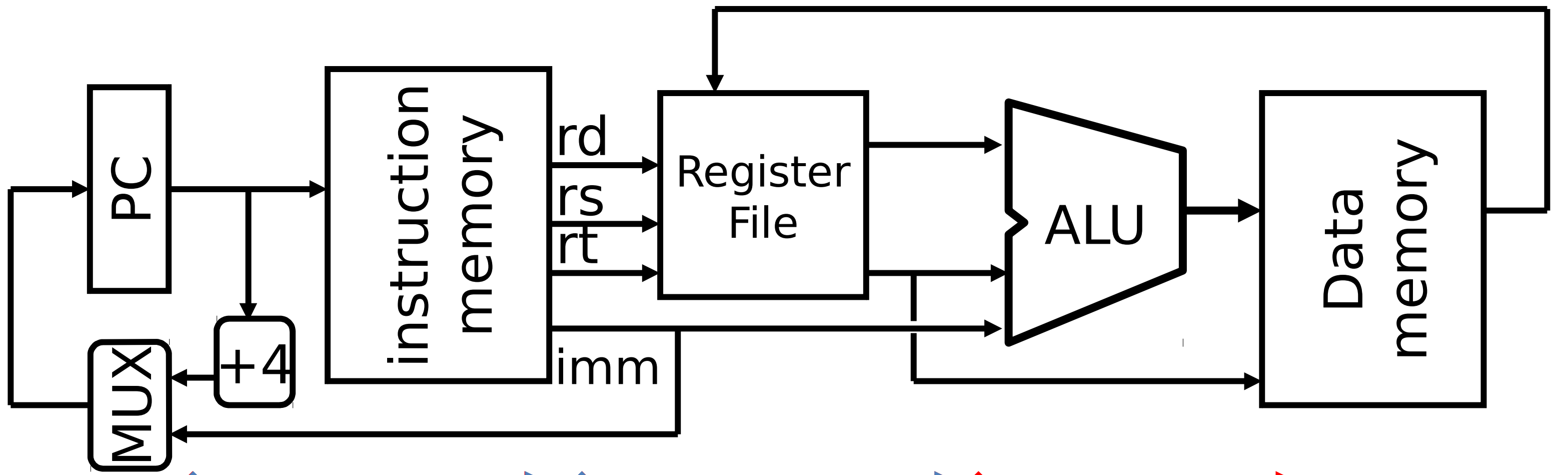
Five Stages of the Datapath



1. Instruction Fetch

2. Decode/ Register Read

Five Stages of the Datapath

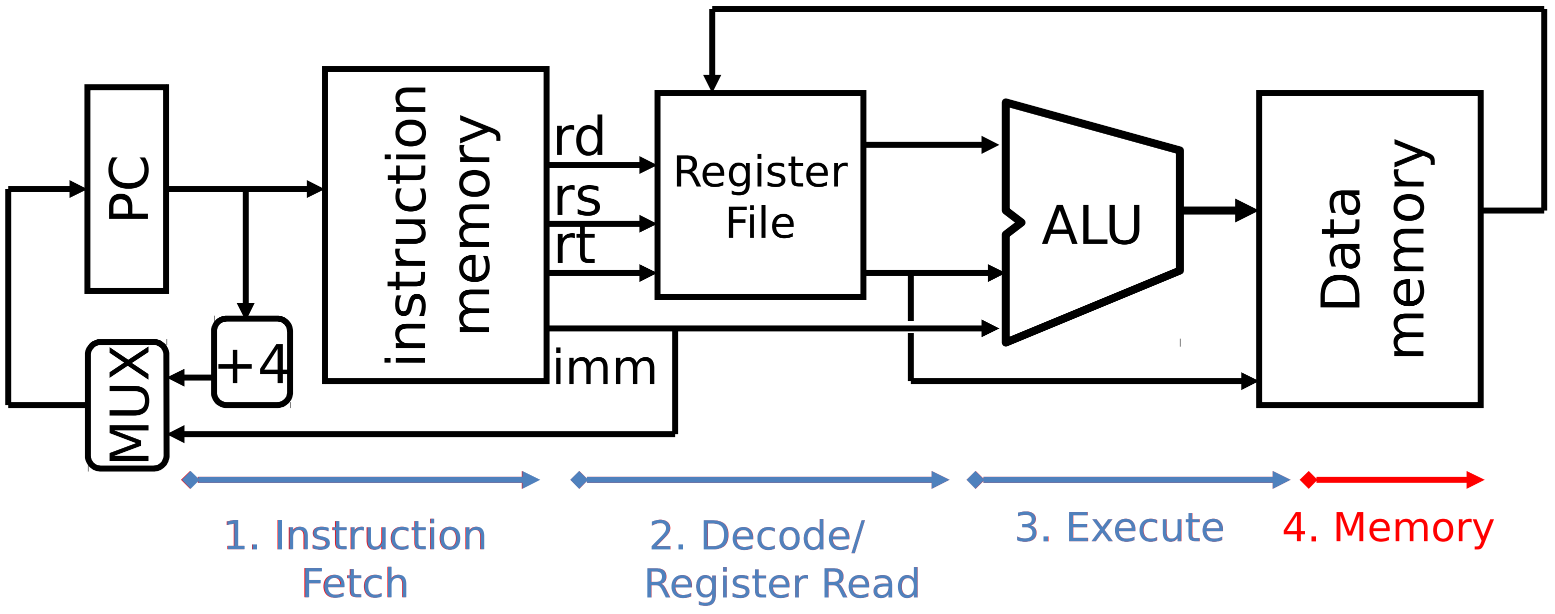


1. Instruction Fetch

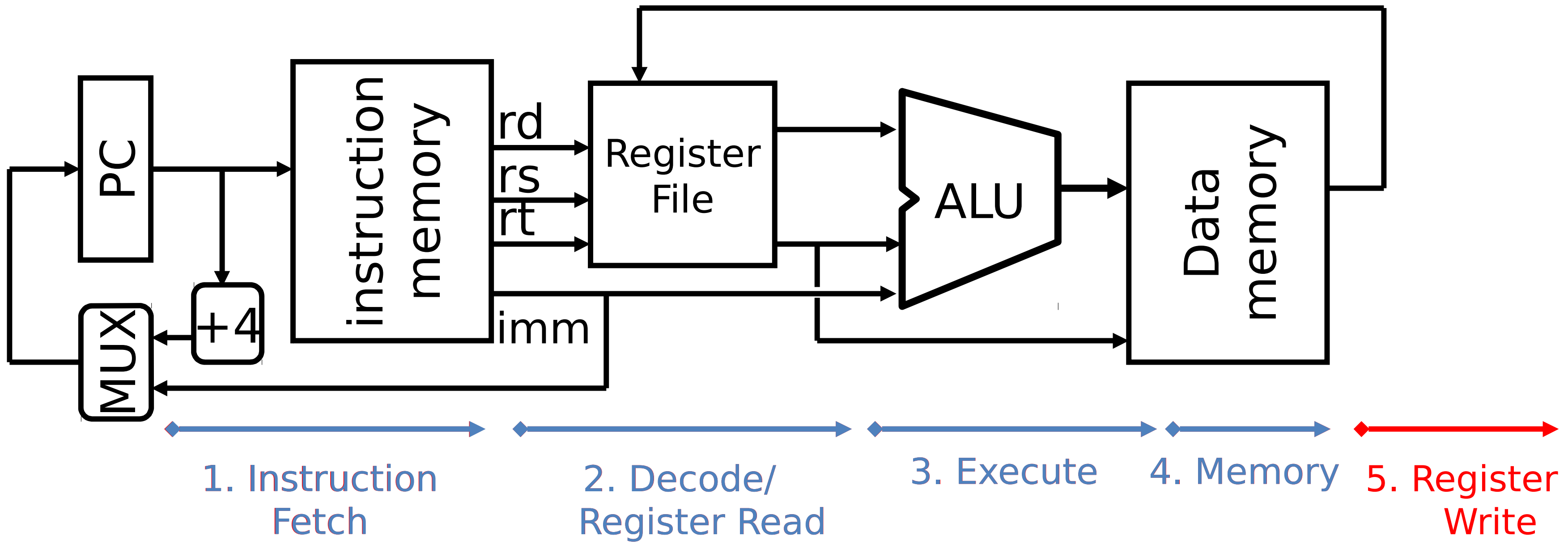
2. Decode/ Register Read

3. Execute

Five Stages of the Datapath



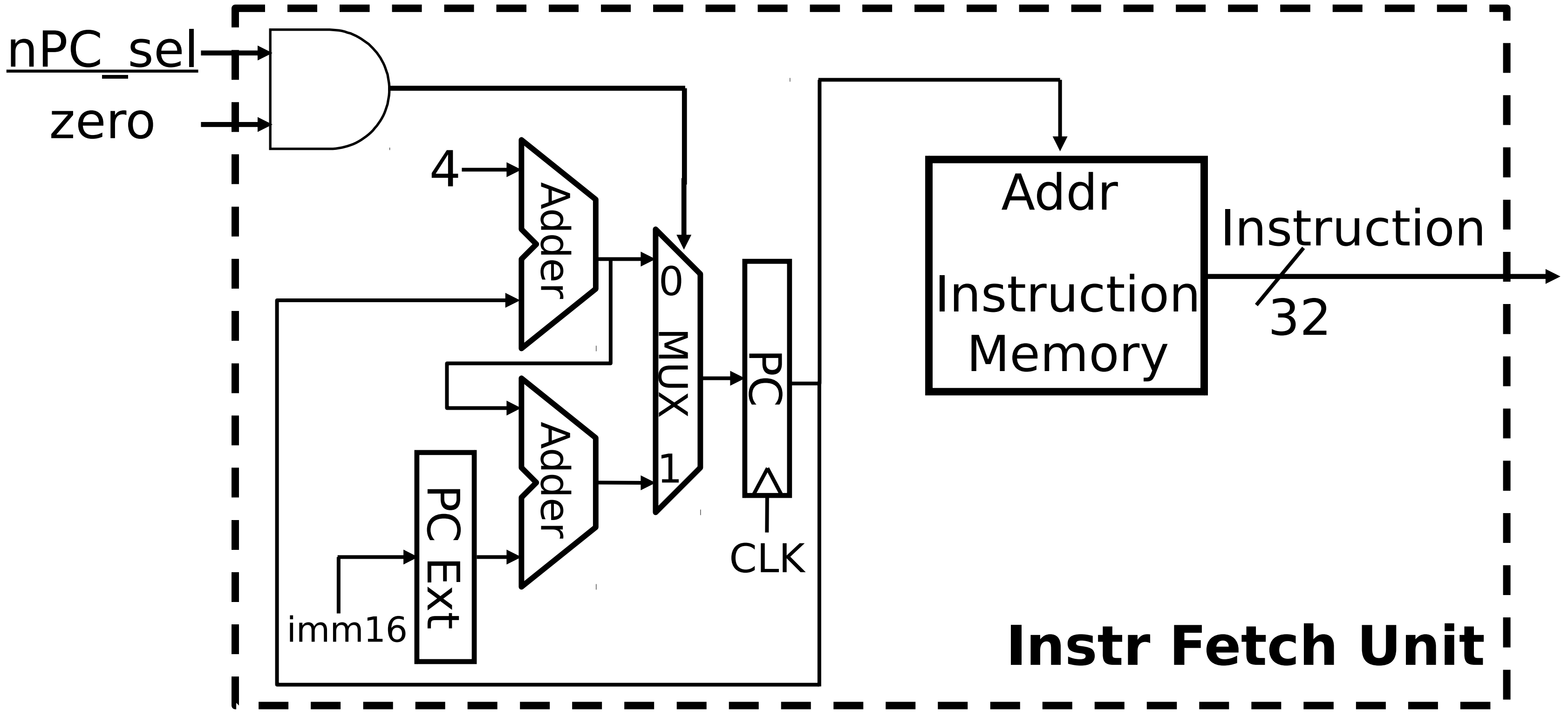
Five Stages of the Datapath



Datapath and Control

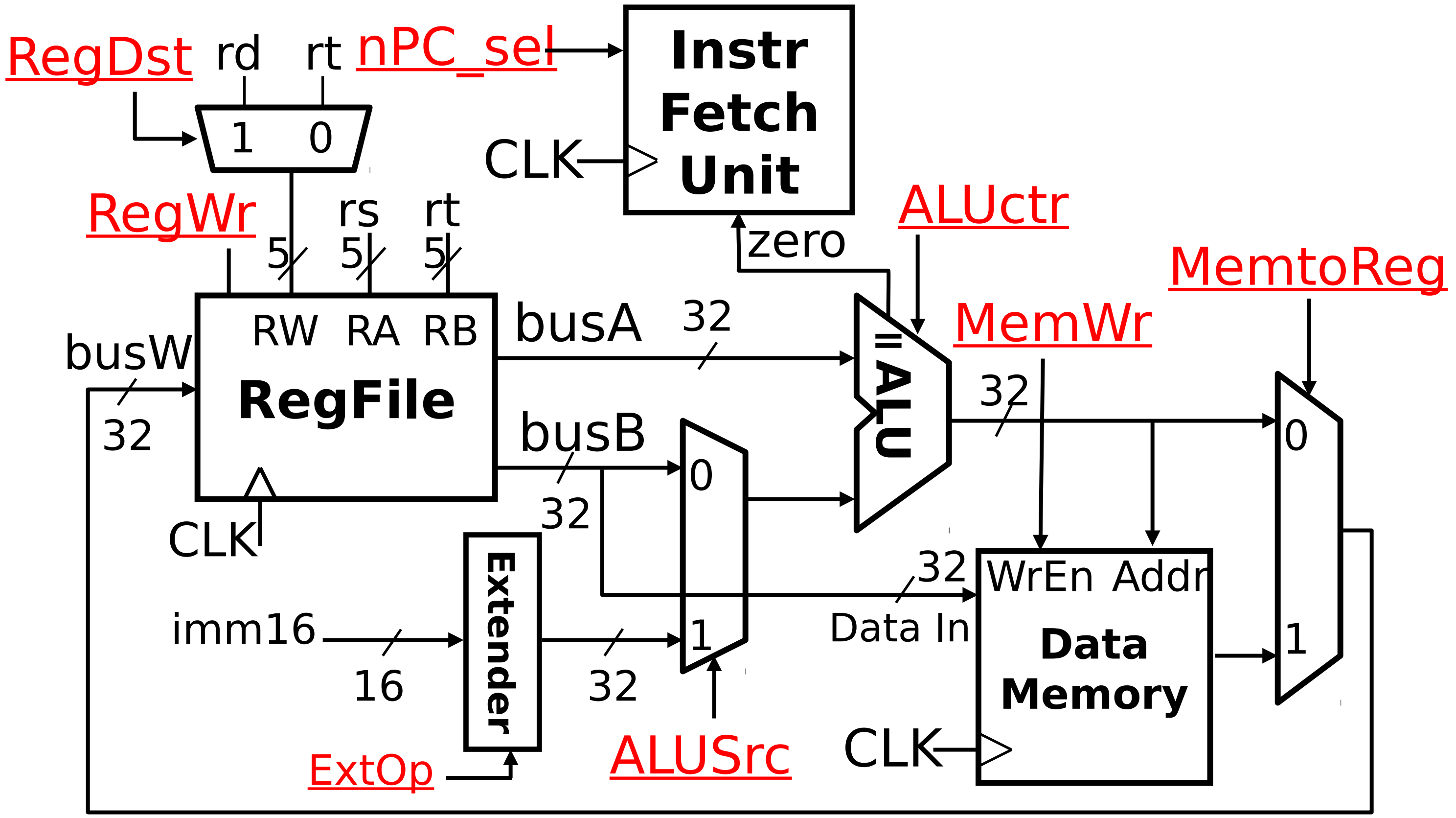
- Route parts of datapath based on ISA needs
  - Add MUXes to select from multiple inputs
  - Add *control signals* for component inputs and MUXes
- Analyze control signals
  - How wide does each one need to be?
  - For each instruction, assign appropriate value for correct routing

MIPS-lite Instruction Fetch

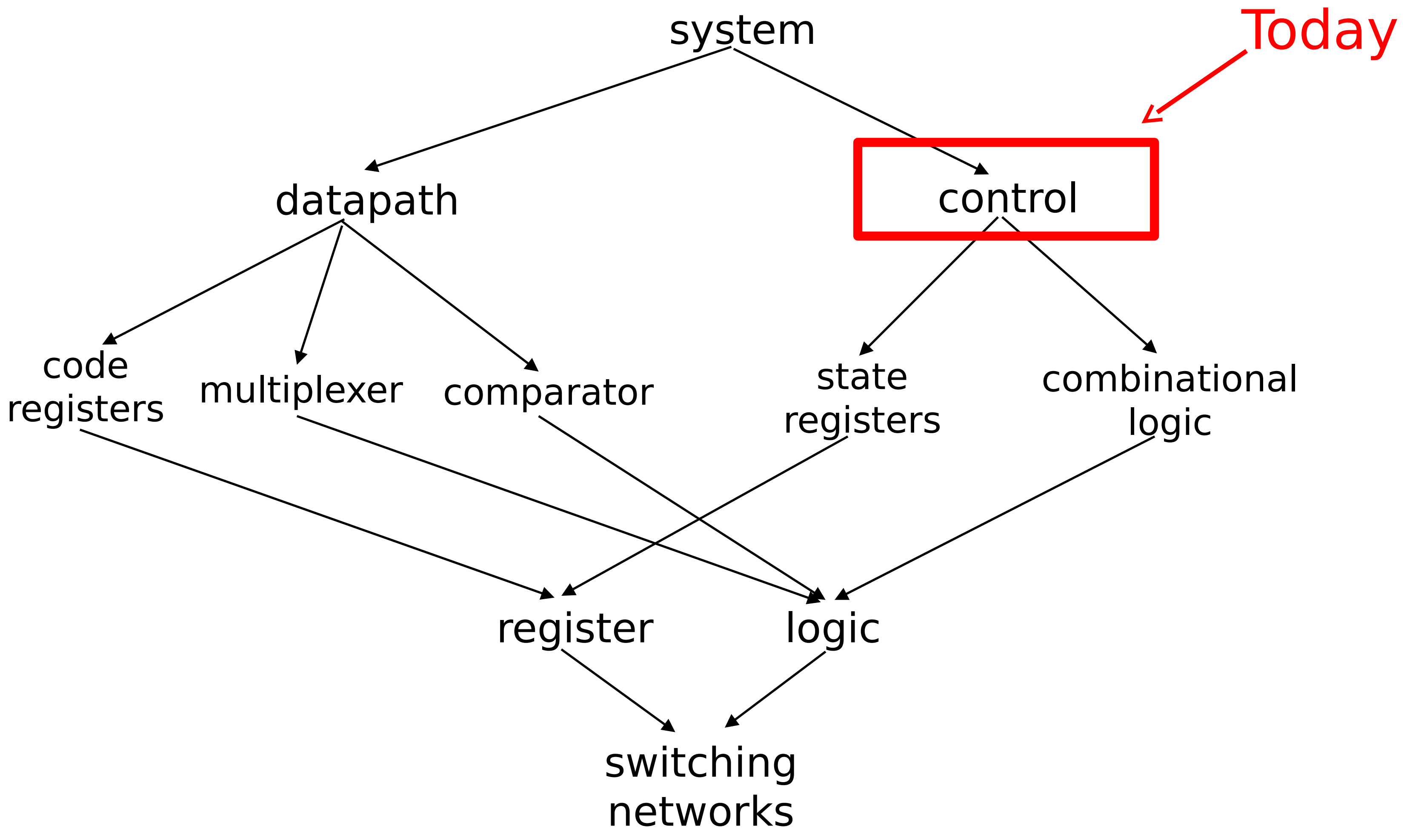


MIPS-lite Datapath Control Signals

ExtOp:	0 → “zero”; 1 → “sign”	**MemWr:**	1 → write memory
ALUsrc:	0 → busB; 1 → imm16	**MemtoReg:**	0 → ALU; 1 → Mem
ALUctr:	“ADD”, “SUB”, “OR”	**RegDst:**	0 → “rt”; 1 → “rd”
nPC_sel:	0 → +4; 1 → branch	**RegWr:**	1 → write register



Hardware Design Hierarchy

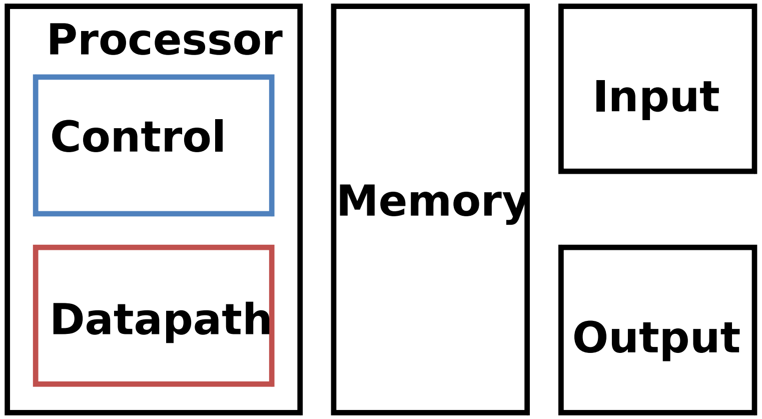


Agenda

- Quick Datapath Review
- Control Implementation
- Administrivia
- Clocking Methodology
- Pipelined Execution
- Pipelined Datapath

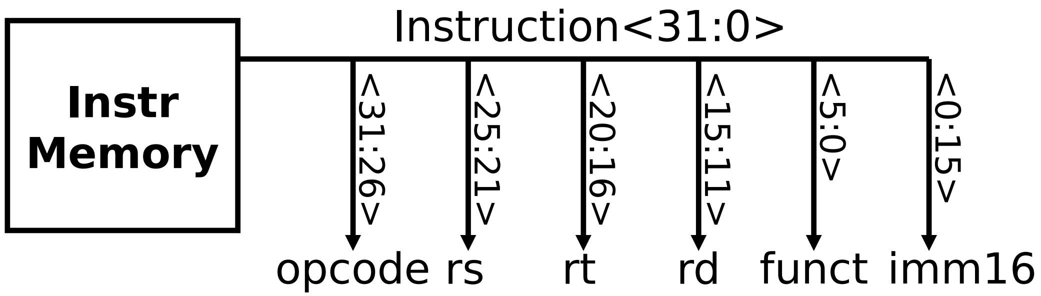
Processor Design Process

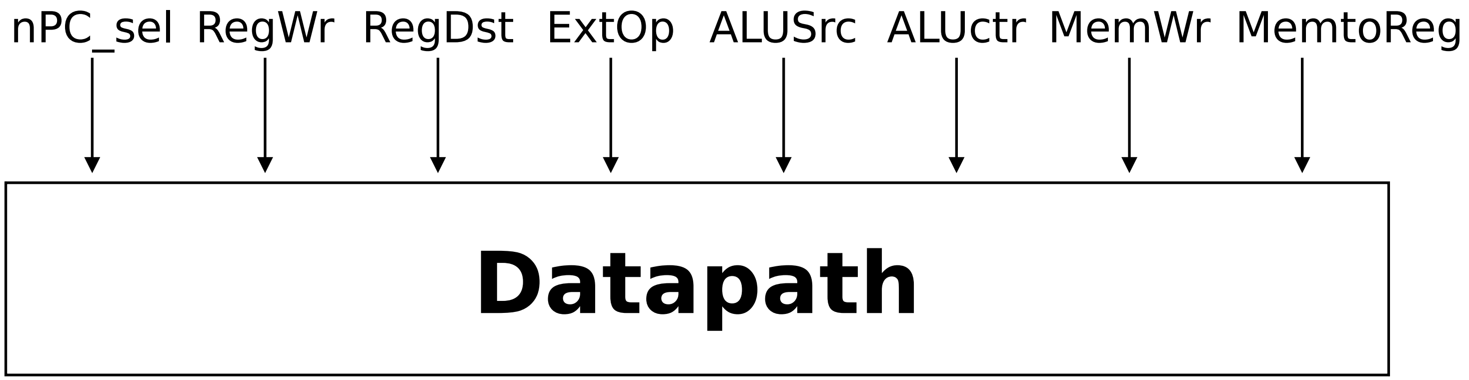
- Five steps to design a processor:
  1. Analyze instruction set → datapath requirements
  2. Select set of datapath components & establish clock methodology
  3. Assemble datapath meeting the requirements
  4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer
  5. Assemble the control logic
     - Formulate Logic Equations
     - Design Circuits



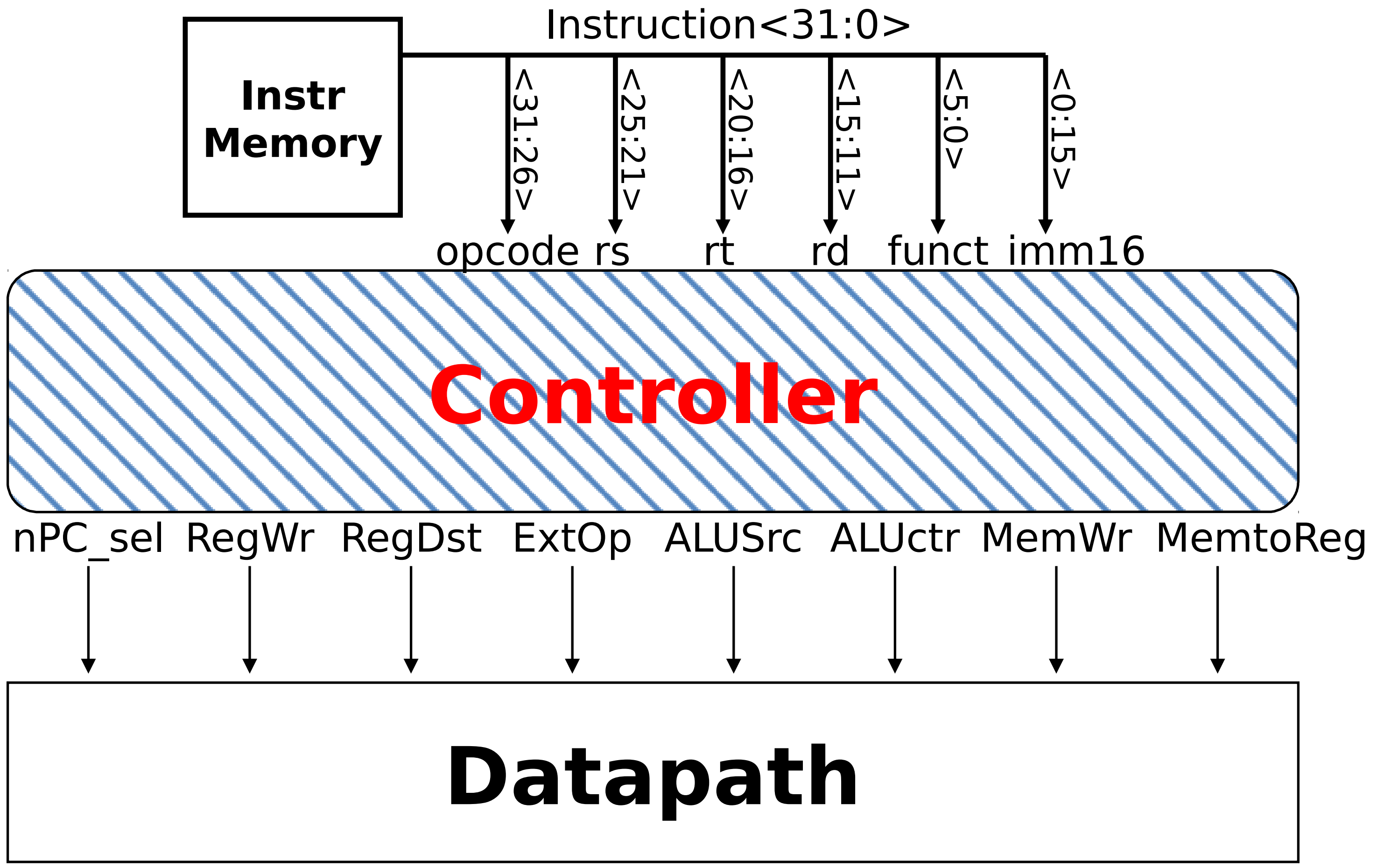
Now

Purpose of Control





Purpose of Control



MIPS-lite Instruction RTL

```
Instr  Register Transfer Language
addu   R[rd]←R[rs]+R[rt]; PC←PC+4
subu   R[rd]←R[rs]–R[rt]; PC←PC+4
ori    R[rt]←R[rs]+zero_ext(imm16); PC←PC+4
lw     R[rt]←MEM[R[rs]+sign_ext(imm16)];
       PC←PC+4
sw     MEM[R[rs]+sign_ext(imm16)]←R[rs];
       PC←PC+4
beq    if(R[rs]==R[rt])
          then PC←PC+4+[sign_ext(imm16)||00]
          else PC←PC+4
```

MIPS-lite Control Signals (1/2)

Instr	**Control Signals**
addu	ALUsrc=RegB, ALUctr="ADD", RegDst=rd, RegWr, nPC_sel="+4"
subu	ALUsrc=RegB, ALUctr="SUB", RegDst=rd, RegWr, nPC_sel="+4"
ori	ALUsrc=Imm, ALUctr="OR", RegDst=rt, RegWr, ExtOp="Zero", nPC_sel="+4"
lw	ALUsrc=Imm, ALUctr="ADD", RegDst=rt, RegWr, ExtOp="Sign", MemtoReg, nPC_sel="+4"
sw	ALUsrc=Imm, ALUctr="ADD", MemWr, ExtOp="Sign", nPC_sel="+4"
beq	ALUsrc=RegB, ALUctr="SUB", nPC_sel="Br"

MIPS-lite Control Signals (2/2)

See MIPS Green Sheet

func	10 0000	10 0010	n/a			
op	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100
	add	sub	ori	lw	sw	beq
RegDst	1	1	0	0	X	X
ALUSrc	0	0	1	1	1	0
MemtoReg	0	0	0	1	X	X
RegWrite	1	1	1	1	0	0
MemWrite	0	0	0	0	1	0
nPC_sel	0	0	0	0	0	1
ExtOp	X	X	0	1	1	X
ALUctr[1:0]	Add	Subtract	Or	Add	Add	Subtract

Control Signals

All Supported Instructions

- Now how do we implement this table with CL?

Generating Boolean Expressions

- **Idea #1:** Treat instruction names as Boolean variables!

Generating Boolean Expressions

- **Idea #1:** Treat instruction names as Boolean variables!
  - `opcode` and `funct` bits are available to us
  - Use gates to generate signals that are 1 when it is a particular instruction and 0 otherwise

Generating Boolean Expressions

- **Idea #1:** Treat instruction names as Boolean variables!
  - `opcode` and `funct` bits are available to us
  - Use gates to generate signals that are 1 when it is a particular instruction and 0 otherwise
- Examples:

```
beq = op[5]’•op[4]’•op[3]’•op[2]•op[1]’•op[0]’

Rtype = op[5]’•op[4]’•op[3]’•op[2]’•op[1]’•op[0]’

add = Rtype•funct[5]•funct[4]’•funct[3]’
        •funct[2]’•funct[1]’•funct[0]’
```

Generating Boolean Expressions

- **Idea #2:** Use instruction variables to generate control signals
  - Make each control signal the combination of all instructions that need that signal to be a 1

Generating Boolean Expressions

- **Idea #2:** Use instruction variables to generate control signals
  - Make each control signal the combination of all instructions that need that signal to be a 1
- Examples:
  - `MemWrite = sw`
  - `RegWrite = add + sub + ori + lw`

$\left.\begin{matrix} \\ \\ \end{matrix}\right\}$ Read from row of table

Generating Boolean Expressions

- **Idea #2:** Use instruction variables to generate control signals
  - Make each control signal the combination of all instructions that need that signal to be a 1
- Examples:
  - `MemWrite = sw`
  - `RegWrite = add + sub + ori + lw`
- What about don’t cares (X’s)?

} Read from row of table

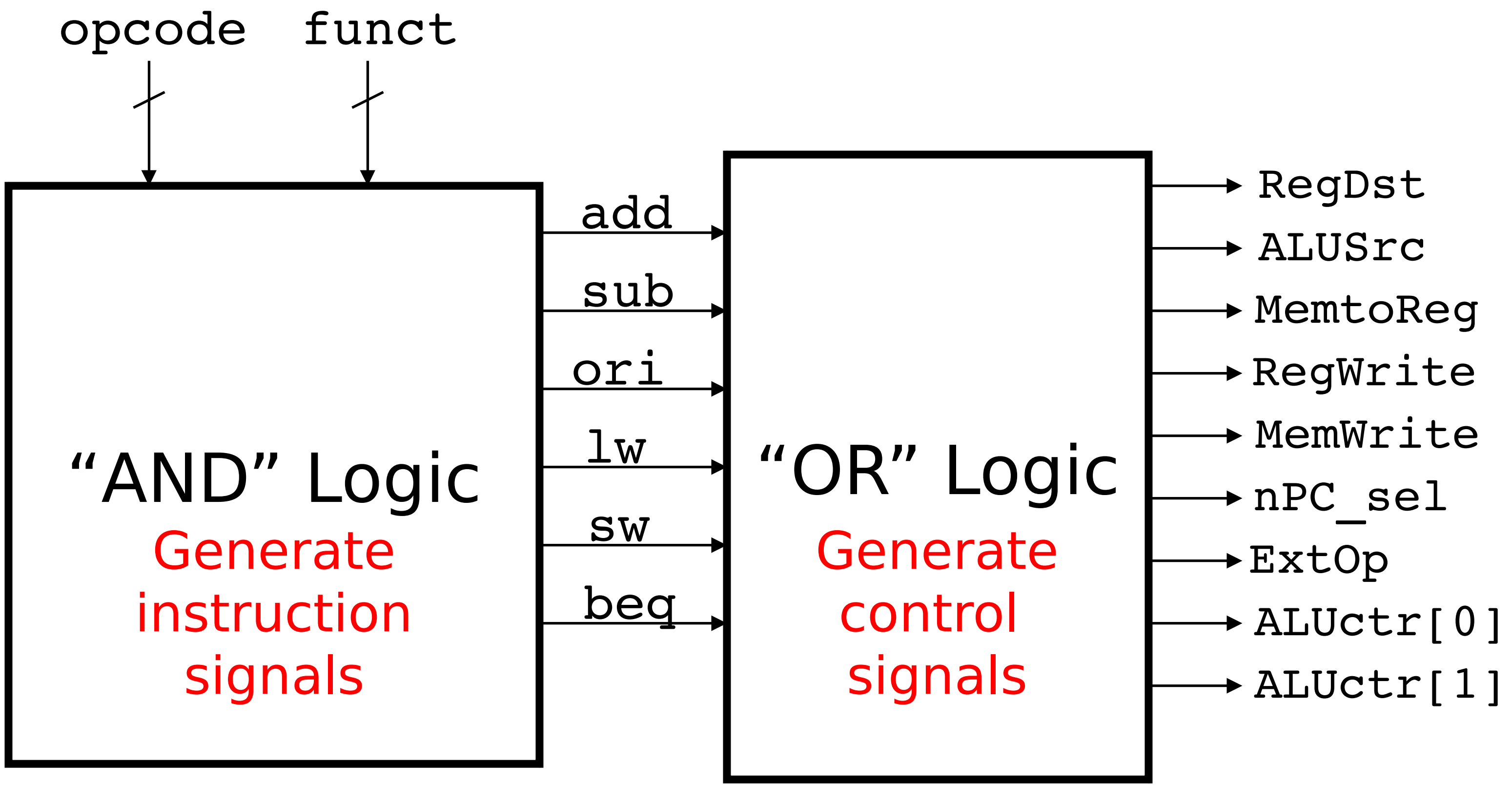
Generating Boolean Expressions

- **Idea #2:** Use instruction variables to generate control signals
  - Make each control signal the combination of all instructions that need that signal to be a 1
- Examples:
  - `MemWrite = sw`
  - `RegWrite = add + sub + ori + lw`

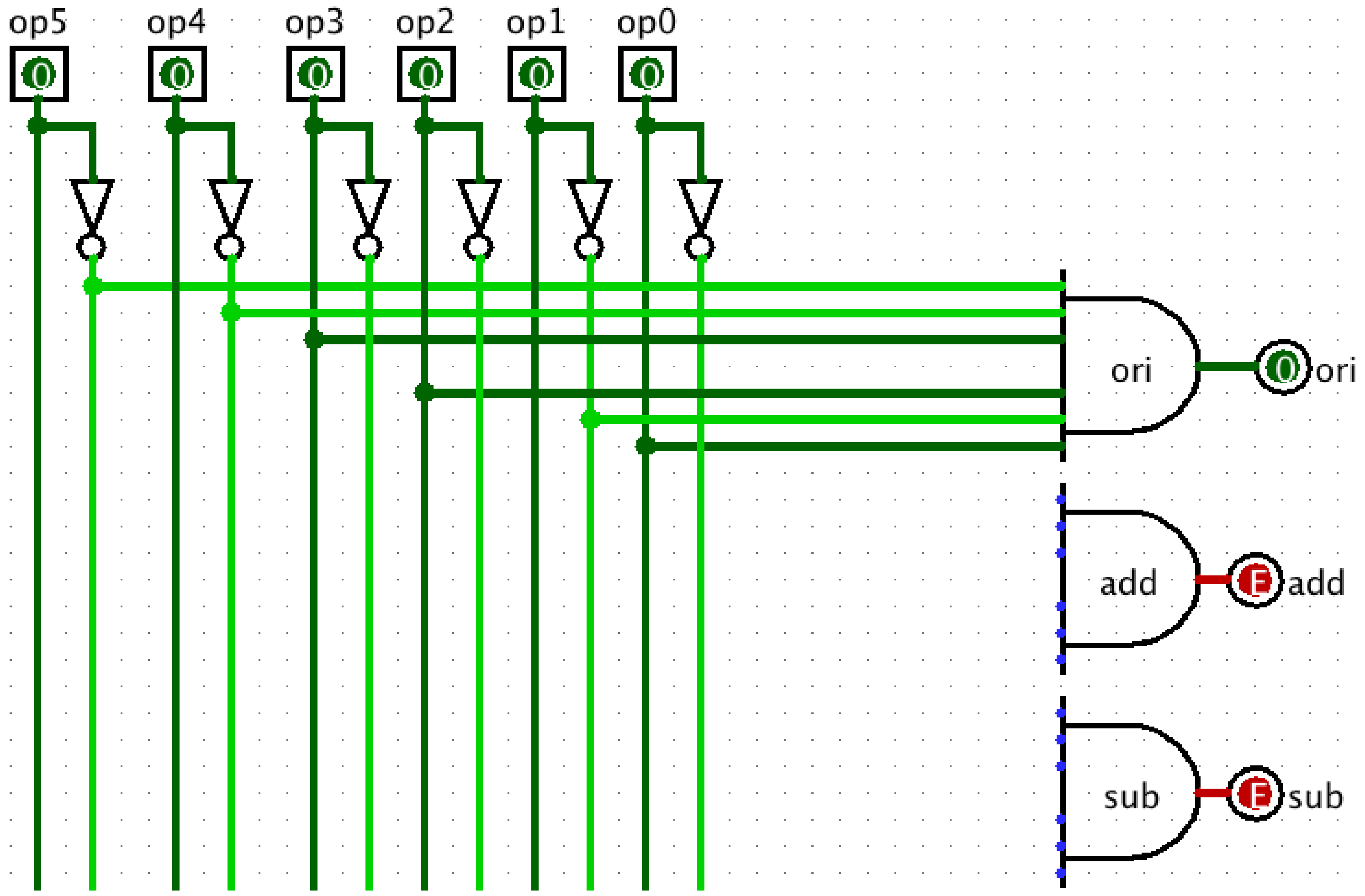
  Read from row of table
- What about don’t cares (X’s)?
  - Want simpler expressions; set to 0!

Controller Implementation

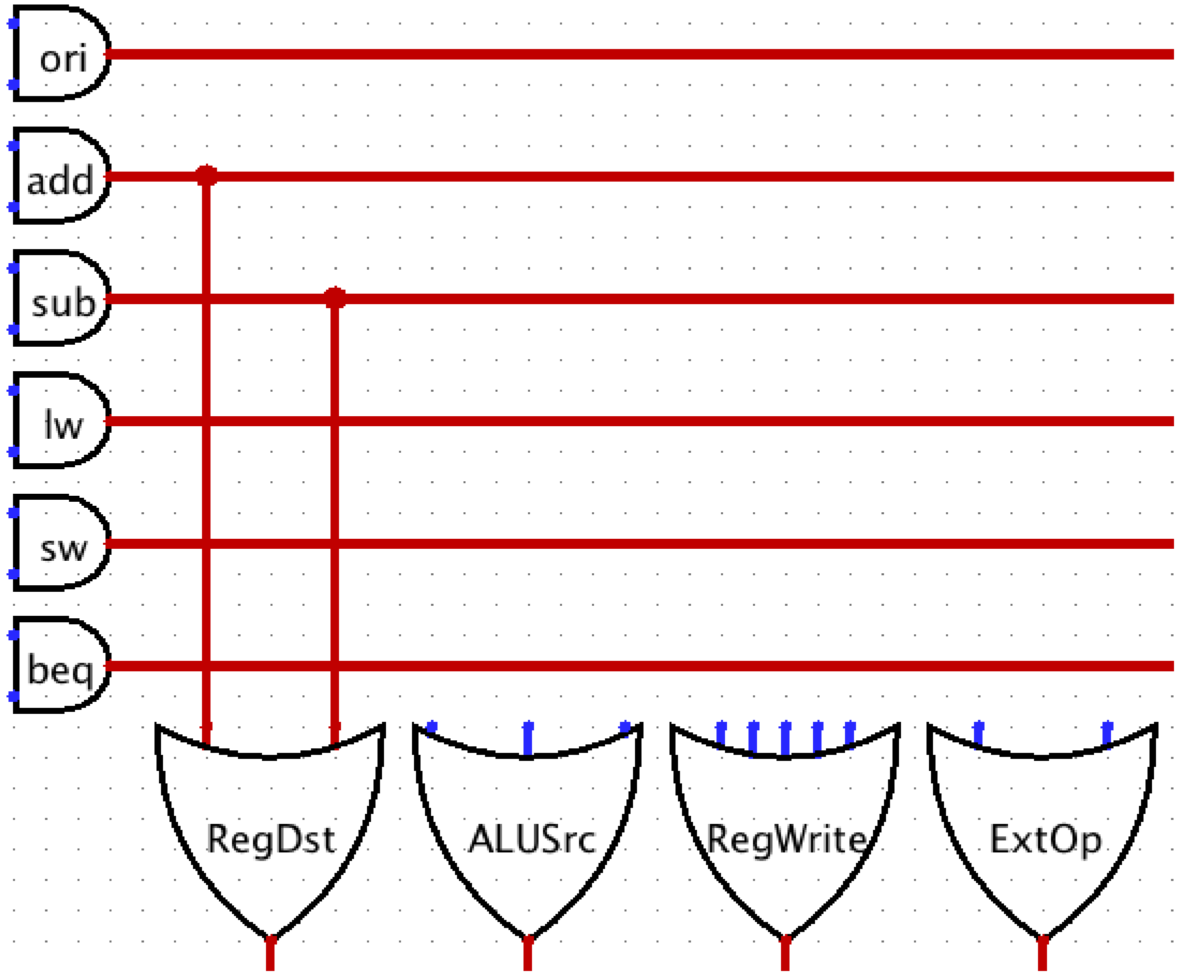
- Use these two ideas to design controller



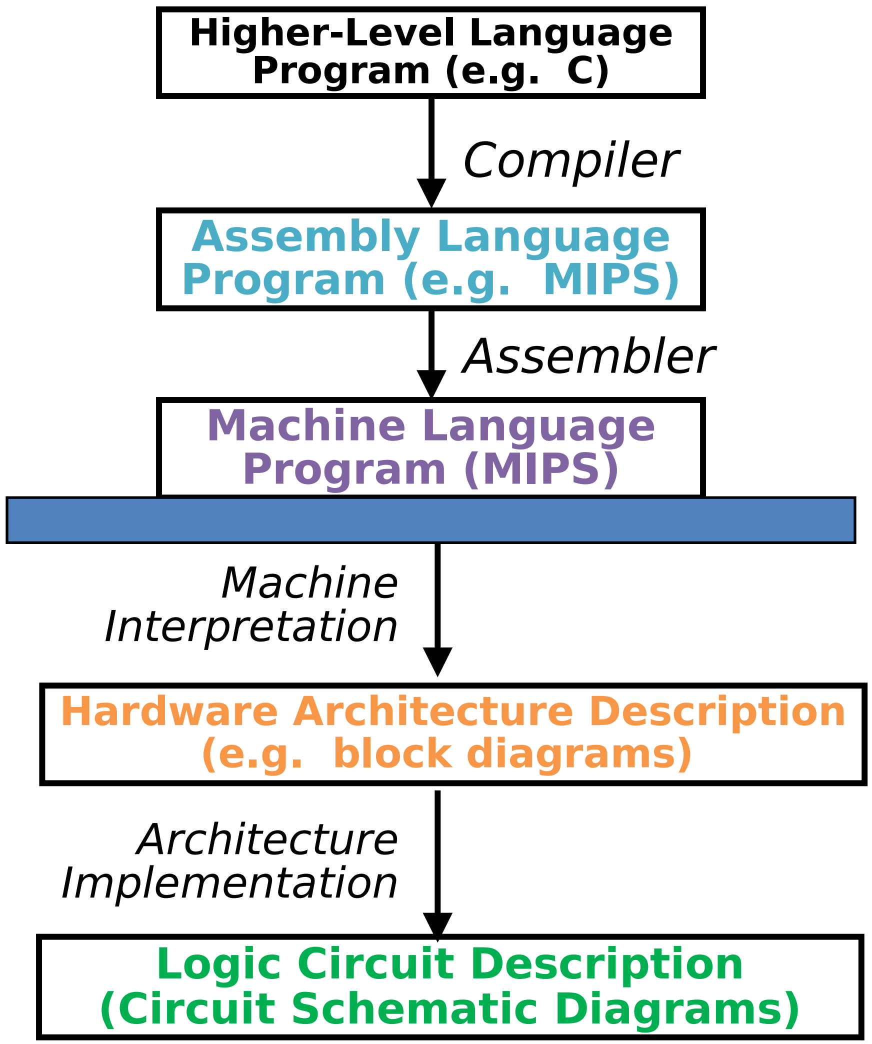
AND Control Logic in Logisim



OR Control Logic in Logisim



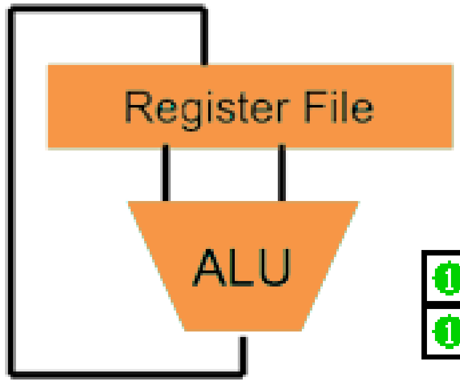
Great Idea #1: Levels of Representation/Interpretation

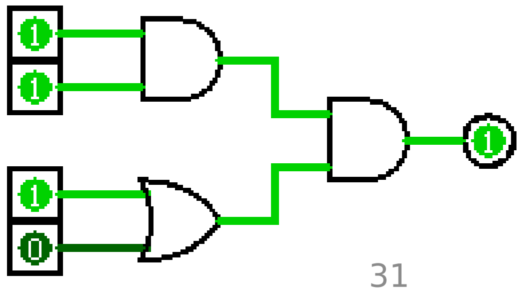

Higher-Level Language Program (e.g. C)
Compiler
Assembly Language Program (e.g. MIPS)
Assembler
Machine Language Program (MIPS)
Machine Interpretation
Hardware Architecture Description (e.g. block diagrams)
Architecture Implementation
Logic Circuit Description (Circuit Schematic Diagrams)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw    $t0, 0($2)
lw    $t1, 4($2)
sw    $t1, 0($2)
sw    $t0, 4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```





Great Idea #1: ...vels of Representation/I...rpretation

CALL HOME, WE'VE MADE HARDWARE/SOFTWARE CONTACT!!!

Higher-Level Language Program (e.g. C)

Compiler

Assembly Language Program (e.g. MIPS)

Assembler

Machine Language Program (MIPS)

Machine Interpretation

Hardware Architecture Description (e.g. block diagrams)

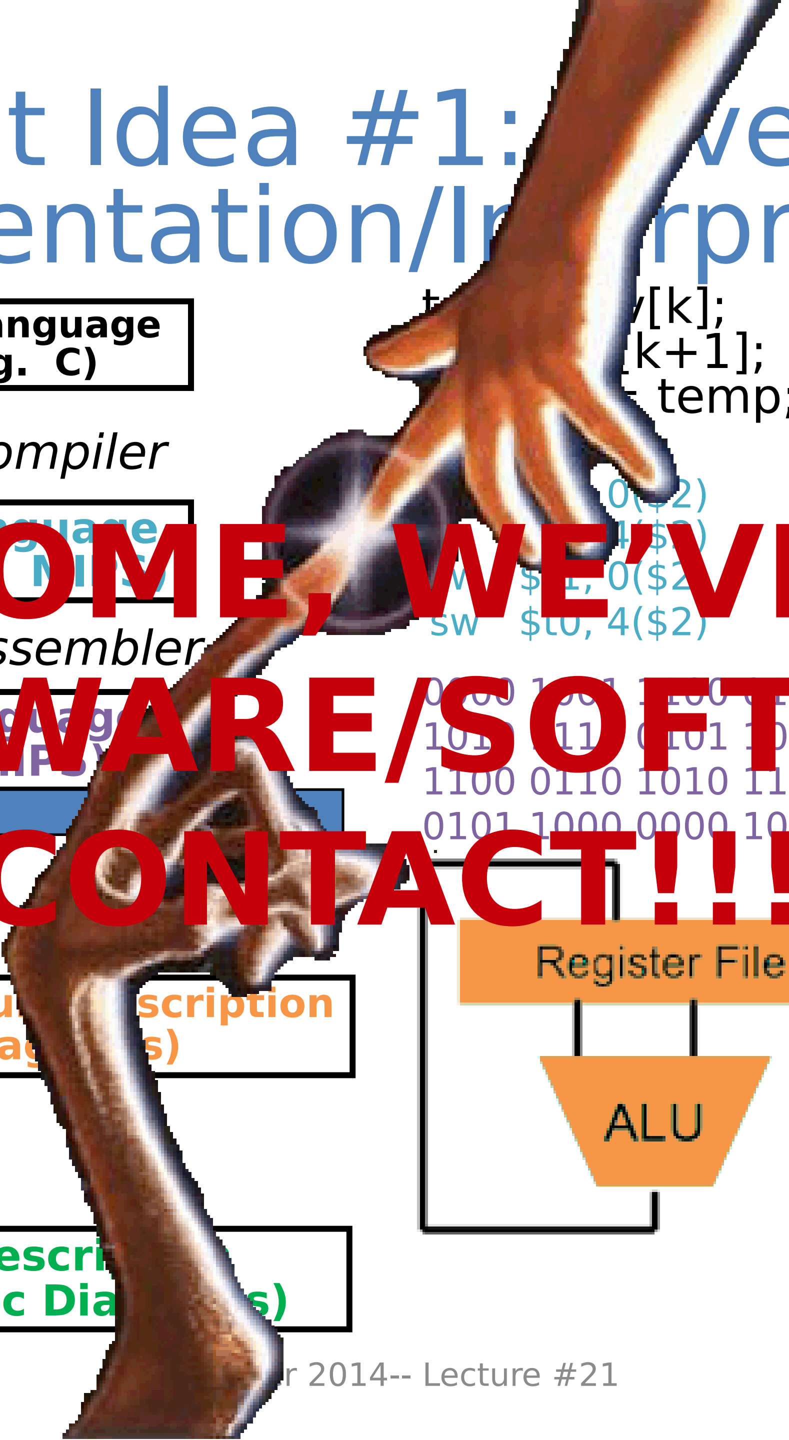
Architecture Implementation

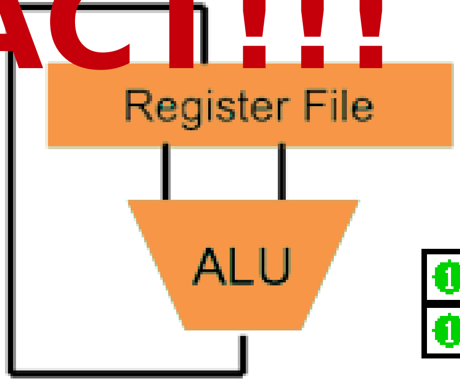
Logic Circuit Description (Circuit Schematic Diagrams)

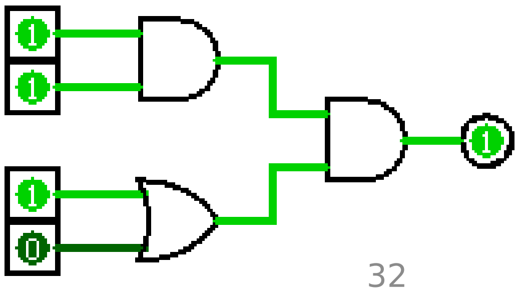
```
t        v[k];
         [k+1];
         = temp;
```

```
           0($2)
           4($2)
sw   $t1, 0($2
sw   $t0, 4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```







Question: Are the following statements TRUE or FALSE? Assume use of the AND-OR controller design.

1) Adding a new *instruction* will NOT require changing any of your existing control logic. (new logic OK though)

2) Adding a new *control signal* will NOT require changing any of your existing control logic. (new logic OK though)

	1	**2**
(B)	**F**	**F**
(G)	**F**	**T**
(P)	**T**	**F**
(Y)	**T**	**T**

Question: Are the following statements TRUE or FALSE? Assume use of the AND-OR controller design.

1) Adding a new *instruction* will NOT require changing any of your existing control logic. (new logic OK though)

2) Adding a new *control signal* will NOT require changing any of your existing control logic. (new logic OK though)

	1	**2**
(B)	**F**	**F**
(G)	**F**	**T**
(P)	**T**	**F**
(Y)	**T**	**T**

Agenda

- Quick Datapath Review
- Control Implementation
- Administrivia
- Clocking Methodology
- Pipelined Execution
- Pipelined Datapath

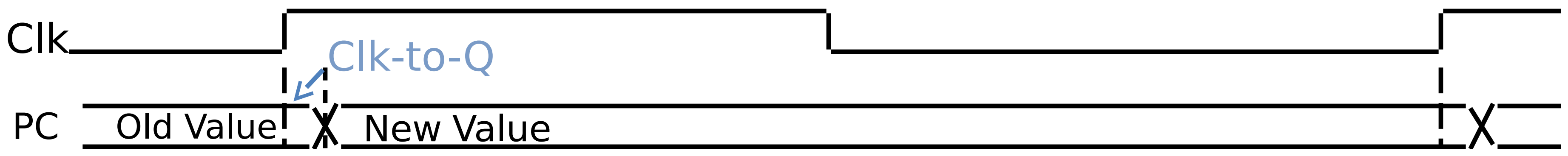
Administrivia

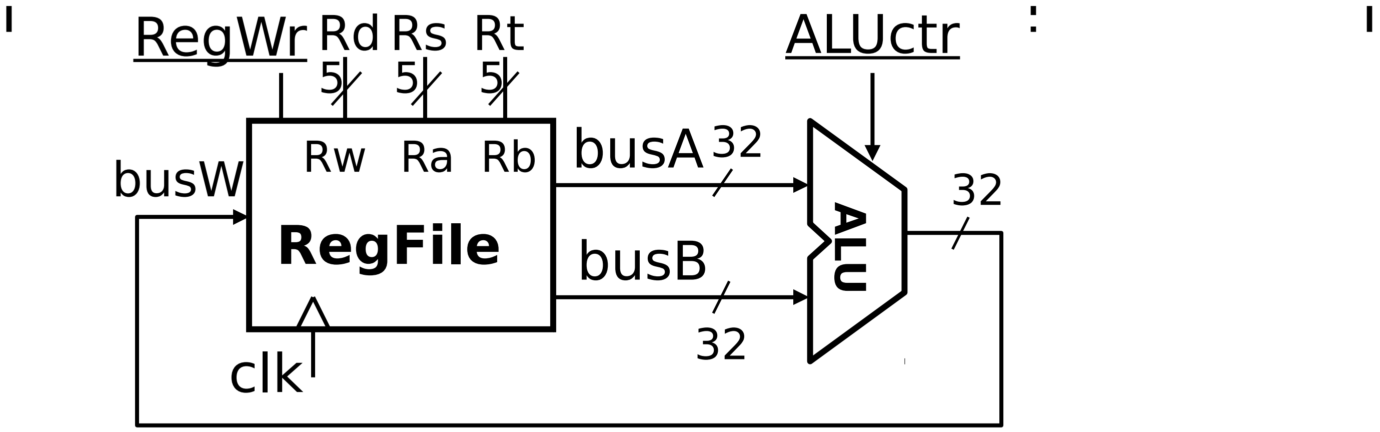
- HW 5 due Thursday
- Project 2 due Sunday
- No lab on Thursday
  - Make up labs encouraged
    - Labs checked off in lab thursday treated as though checked of on tuesday for lateness purposes
- Project 3: Pipelined Processor in Logisim will be released this week

Agenda

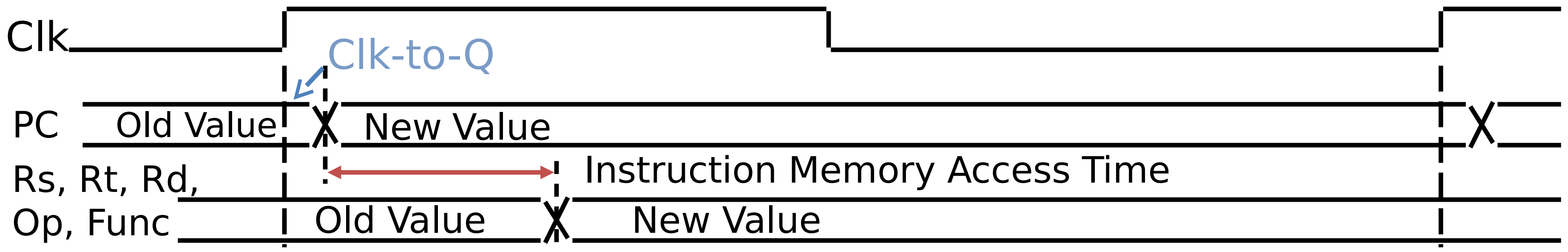
- Quick Datapath Review
- Control Implementation
- Administrivia
- Clocking Methodology
- Pipelined Execution
- Pipelined Datapath

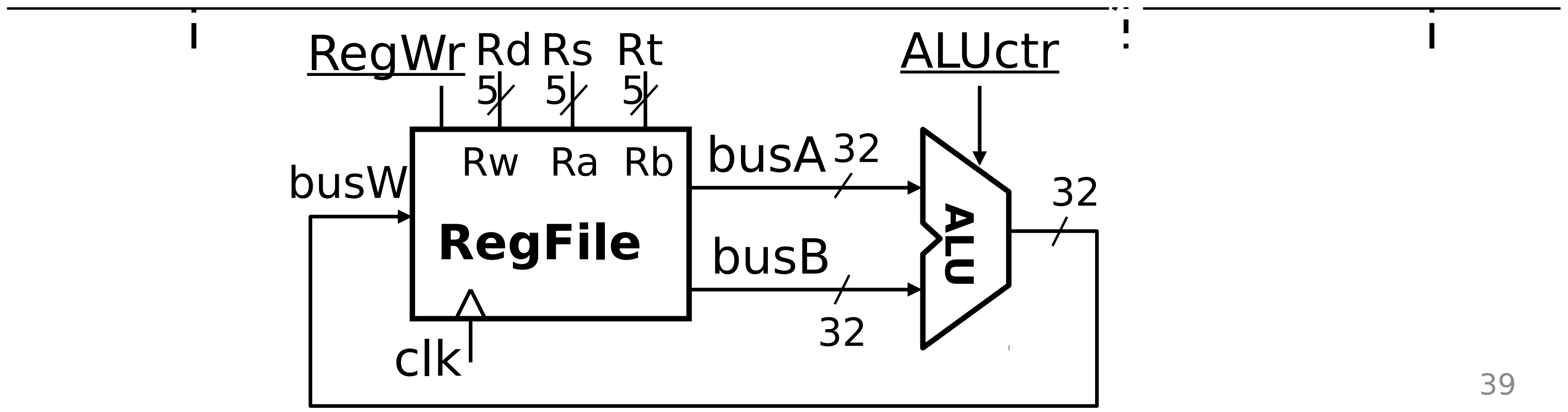
Register-Register Timing: One Complete Cycle for addu



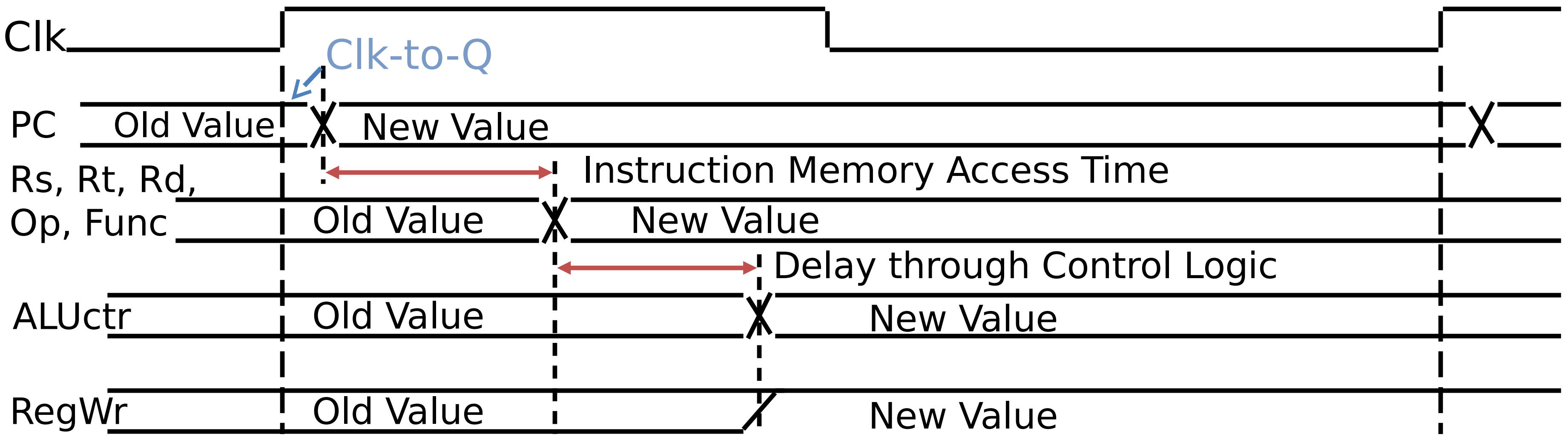


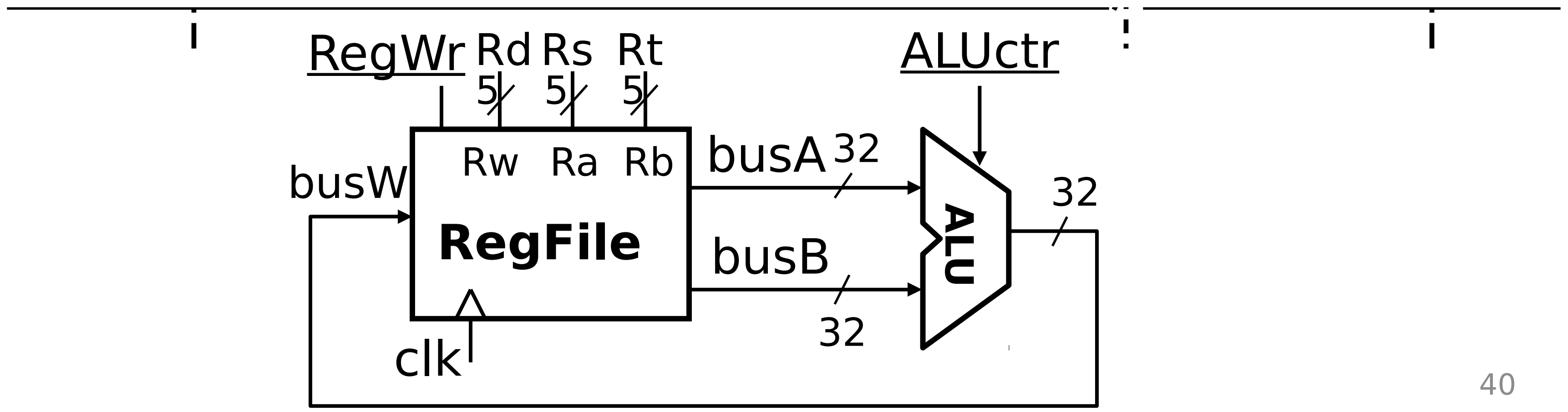
Register-Register Timing: One Complete Cycle for addu



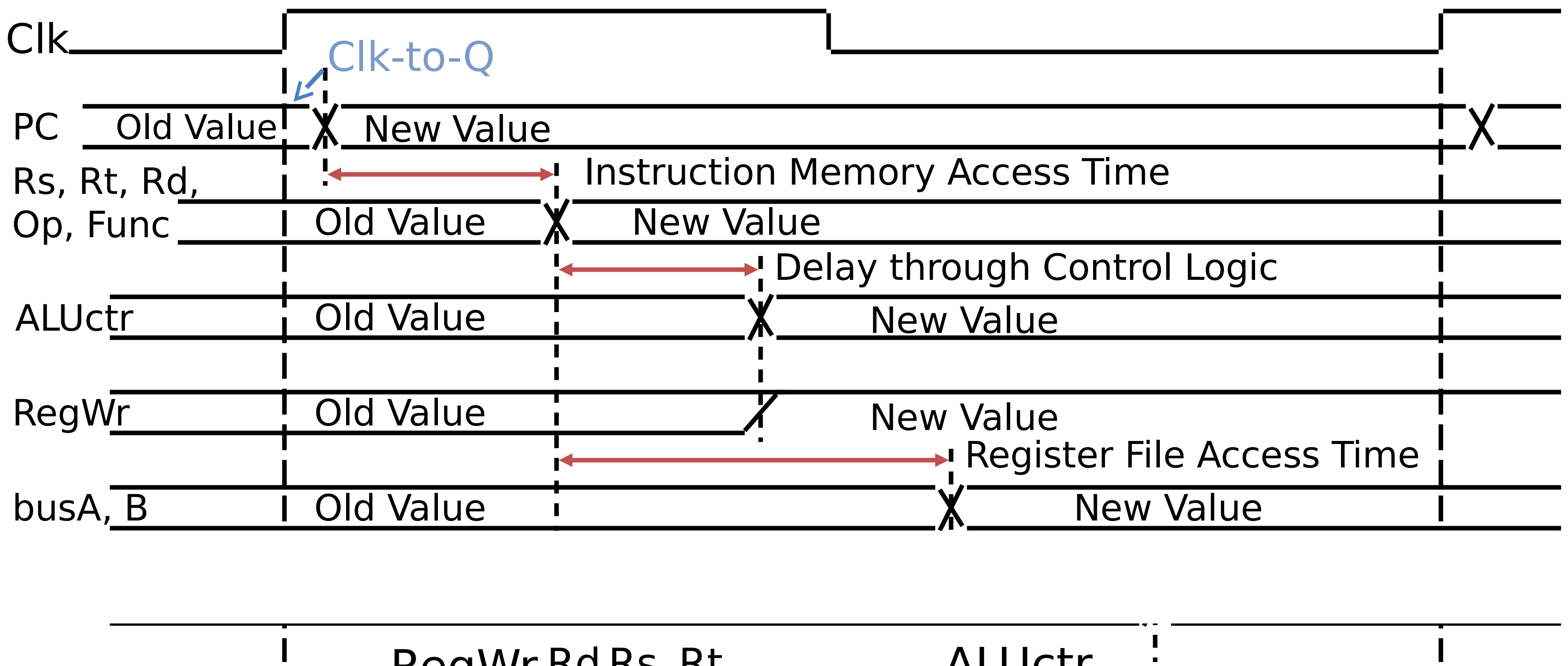


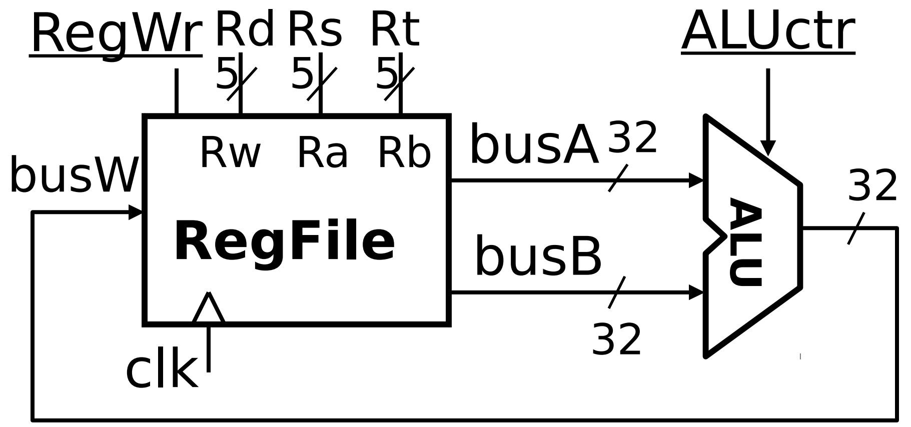
Register-Register Timing: One Complete Cycle for `addu`



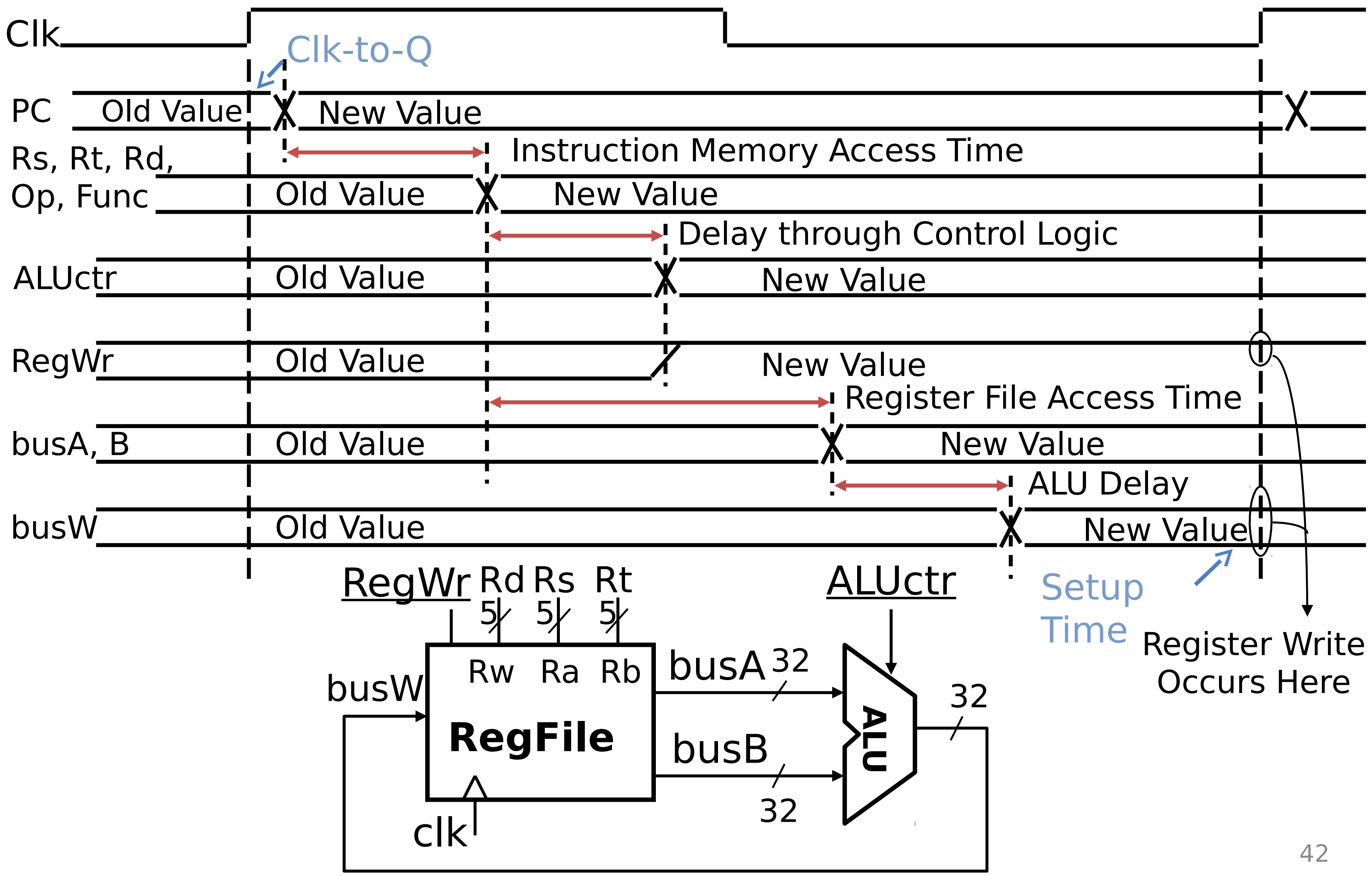


Register-Register Timing: One Complete Cycle for `addu`

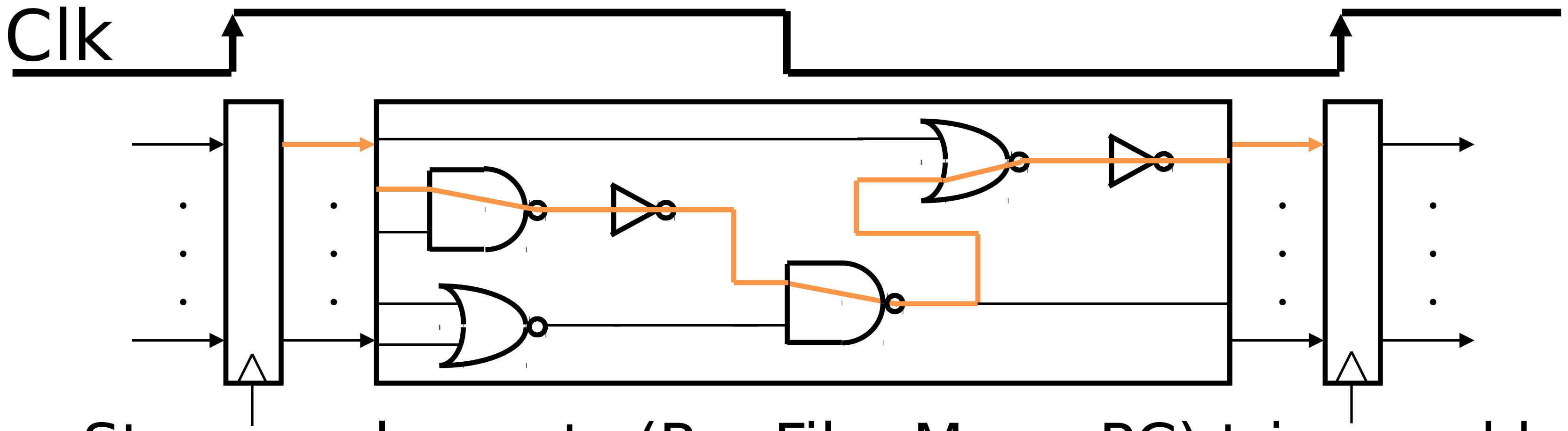




Register-Register Timing: One Complete Cycle for addu

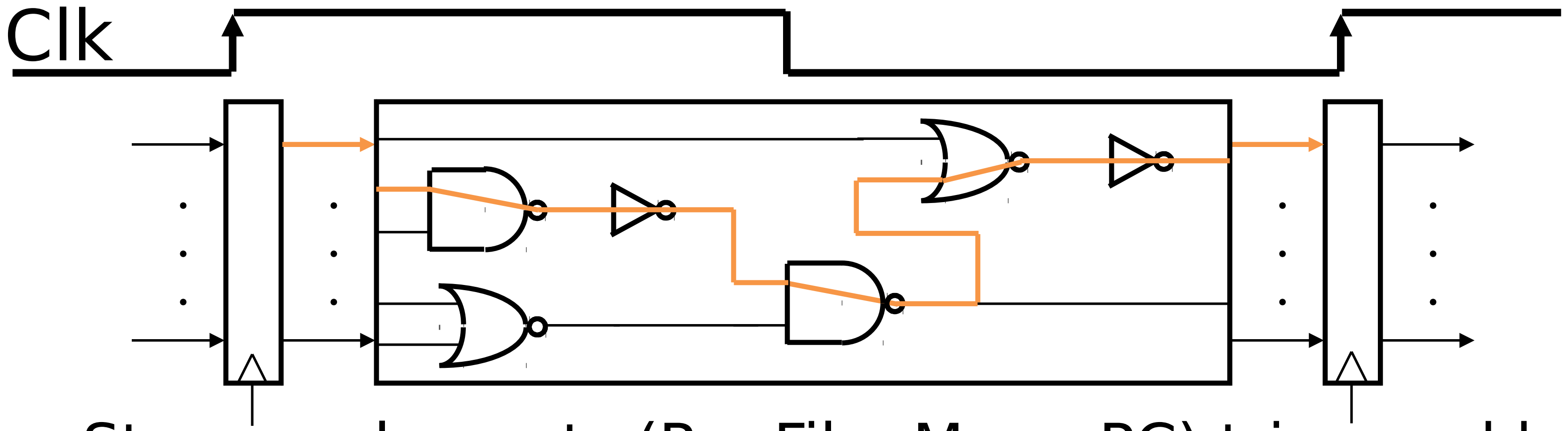


Clocking Methodology



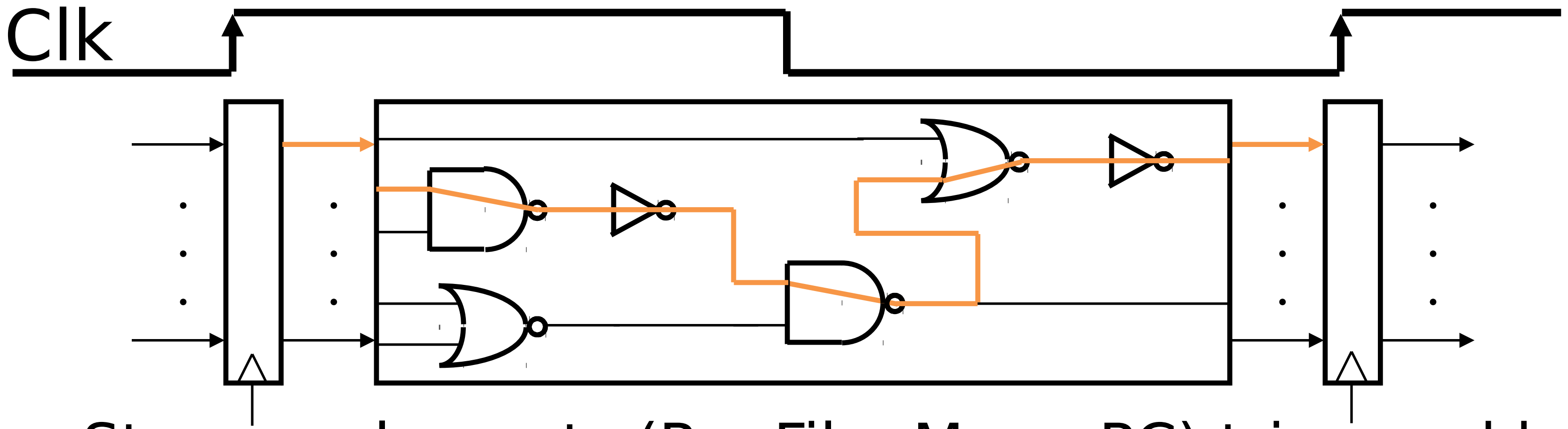
- Storage elements (RegFile, Mem, PC) triggered by same clock

Clocking Methodology



- Storage elements (RegFile, Mem, PC) triggered by same clock
- *Critical path* determines length of clock period
  - This includes CLK-to-Q delay and setup delay

Clocking Methodology



- Storage elements (RegFile, Mem, PC) triggered by same clock
- *Critical path* determines length of clock period
  - This includes CLK-to-Q delay and setup delay
- So far we have built a *single cycle CPU* – entire instructions are executed in 1 clock cycle
  - Up next: pipelining to execute instructions in 5 clock cycles

Single Cycle Performance

- Assume time for actions are 100ps for register read or write; 200ps for other events
- Minimum clock period is?

Instr	**Instr fetch**	**Register read**	**ALU op**	**Memory access**	**Register write**	**Total time**
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Single Cycle Performance

- Assume time for actions are 100ps for register read or write; 200ps for other events
- Minimum clock period is?

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Single Cycle Performance

- Assume time for actions are 100ps for register read or write; 200ps for other events
- Minimum clock period is?

Instr	**Instr fetch**	**Register read**	**ALU op**	**Memory access**	**Register write**	**Total time**
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

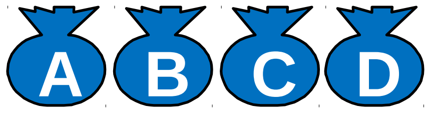
- What can we do to improve clock rate?
- Will this improve performance as well?
  - Want increased clock rate to mean faster programs

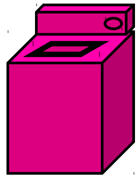
Agenda

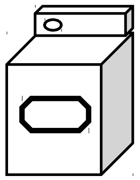
- Quick Datapath Review
- Control Implementation
- Administrivia
- Clocking Methodology
- Pipelined Execution
- Pipelined Datapath

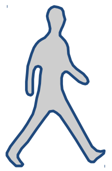
Pipeline Analogy: Doing Laundry

- Ann, Brian, Cathy, and Dave each have one load of clothes to wash, dry, fold, and put away
  - Washer takes 30 minutes
  - Dryer takes 30 minutes
  - "Folder" takes 30 minutes
  - "Stasher" takes 30 minutes to put clothes into drawers

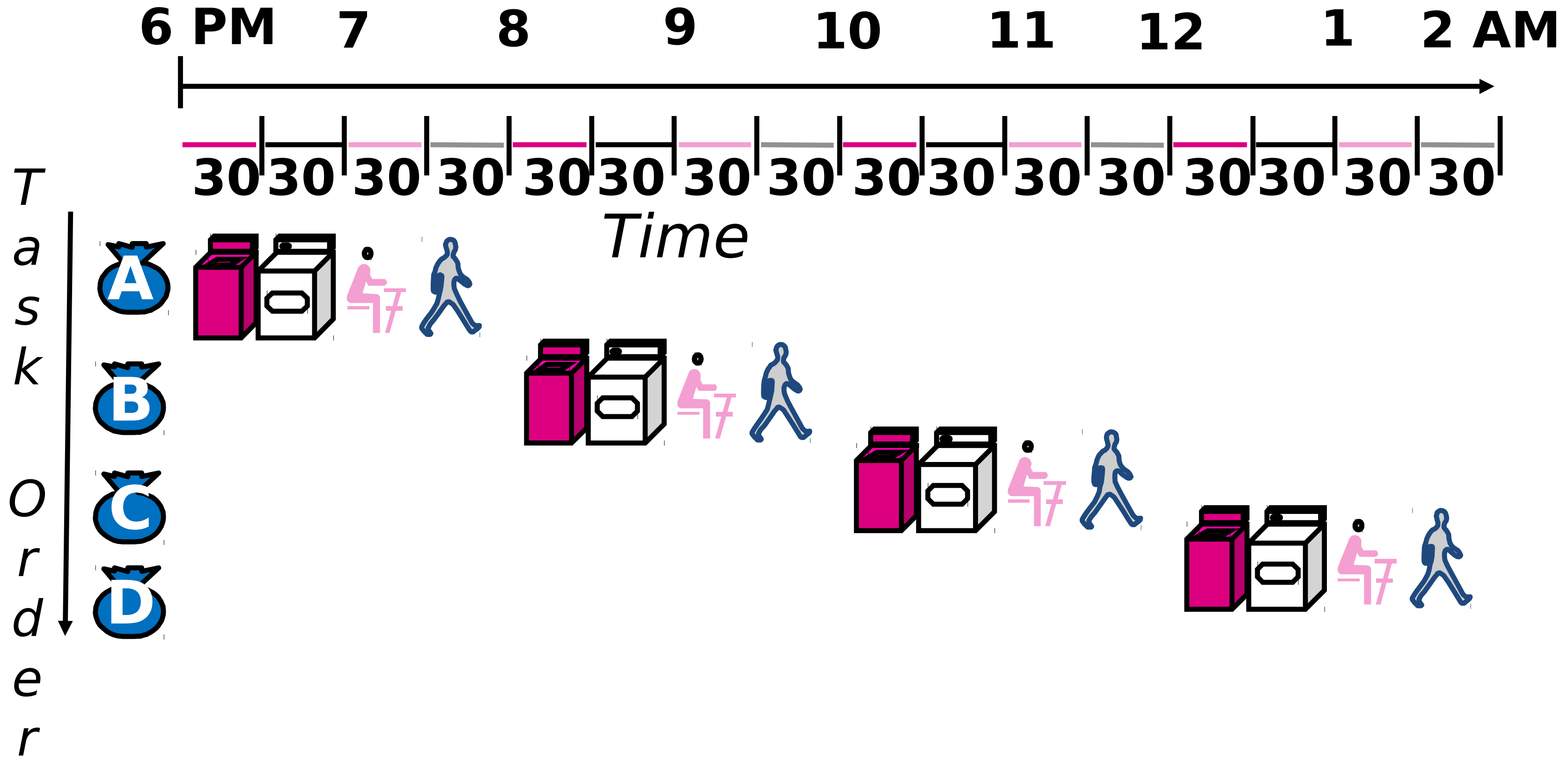




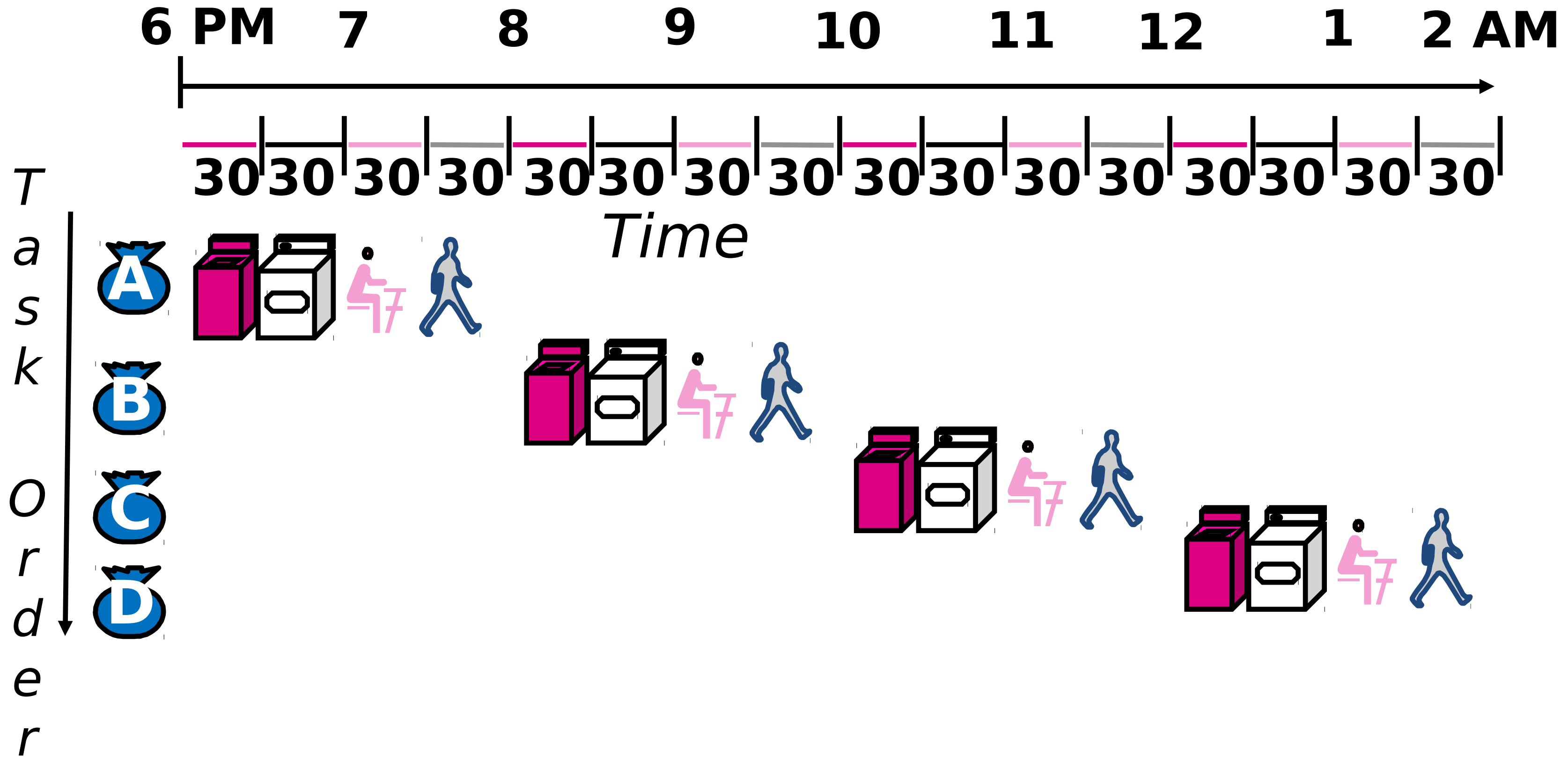




Sequential Laundry

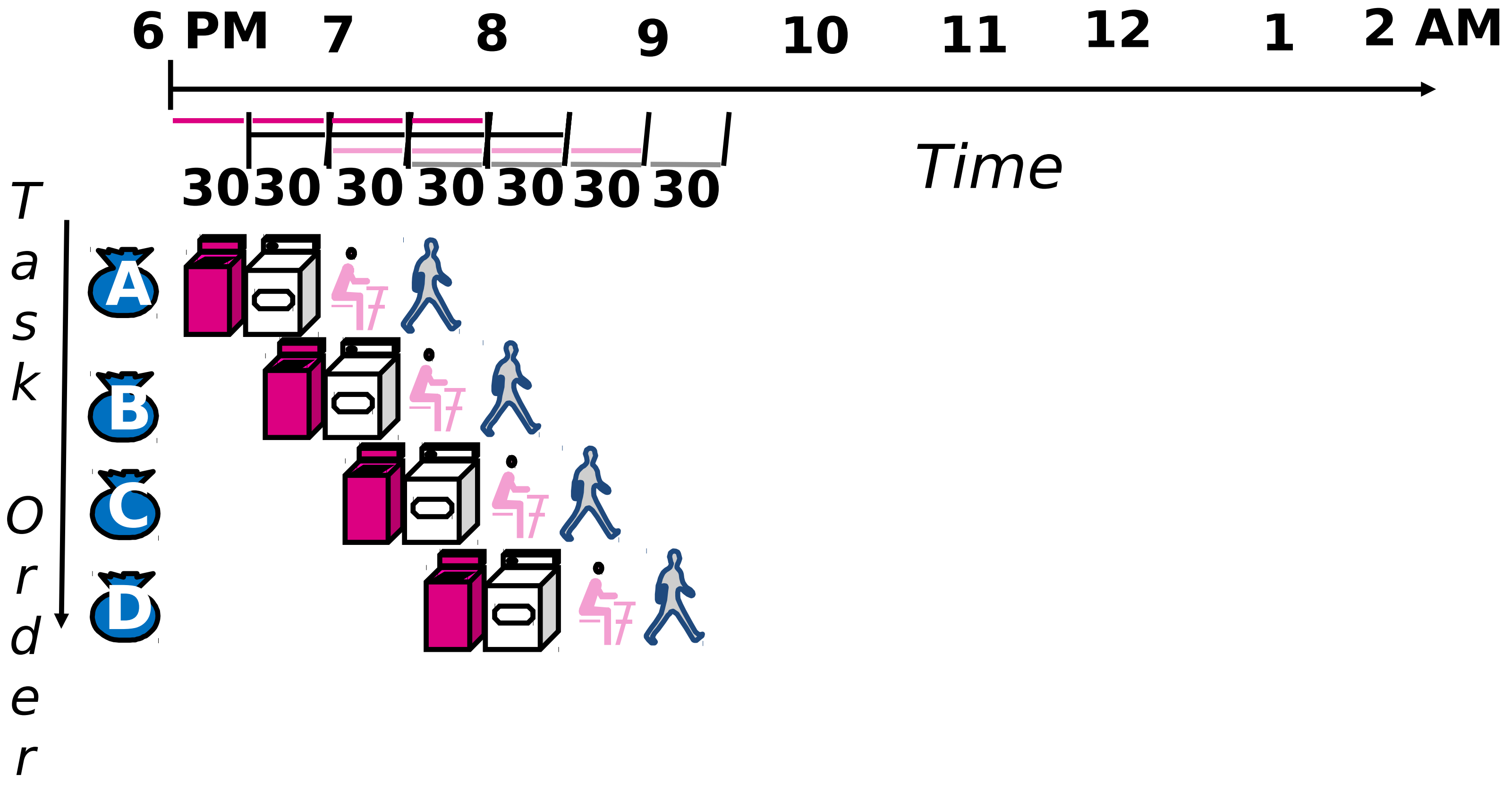


Sequential Laundry

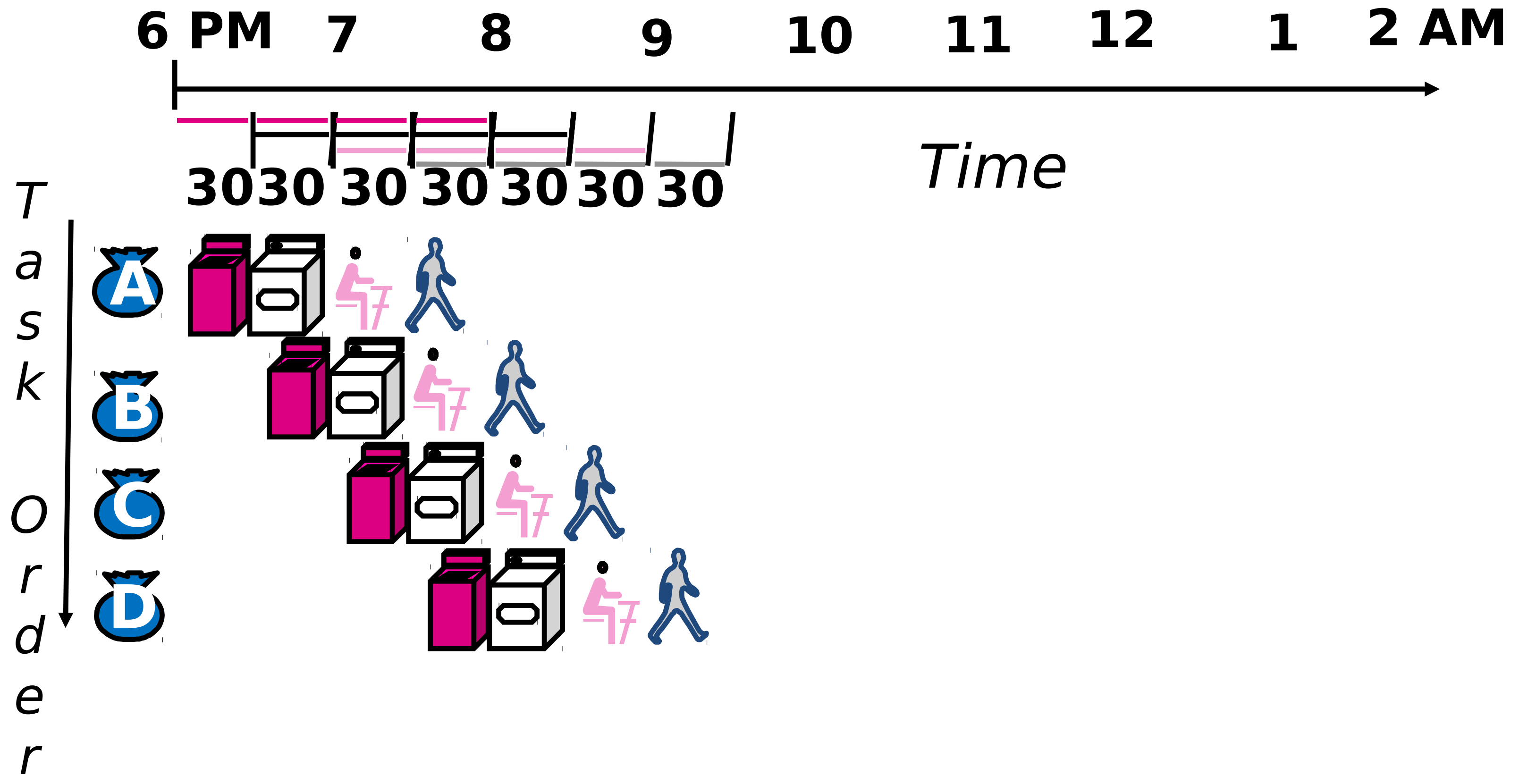


- Sequential laundry takes 8 hours for 4 loads

Pipelined Laundry

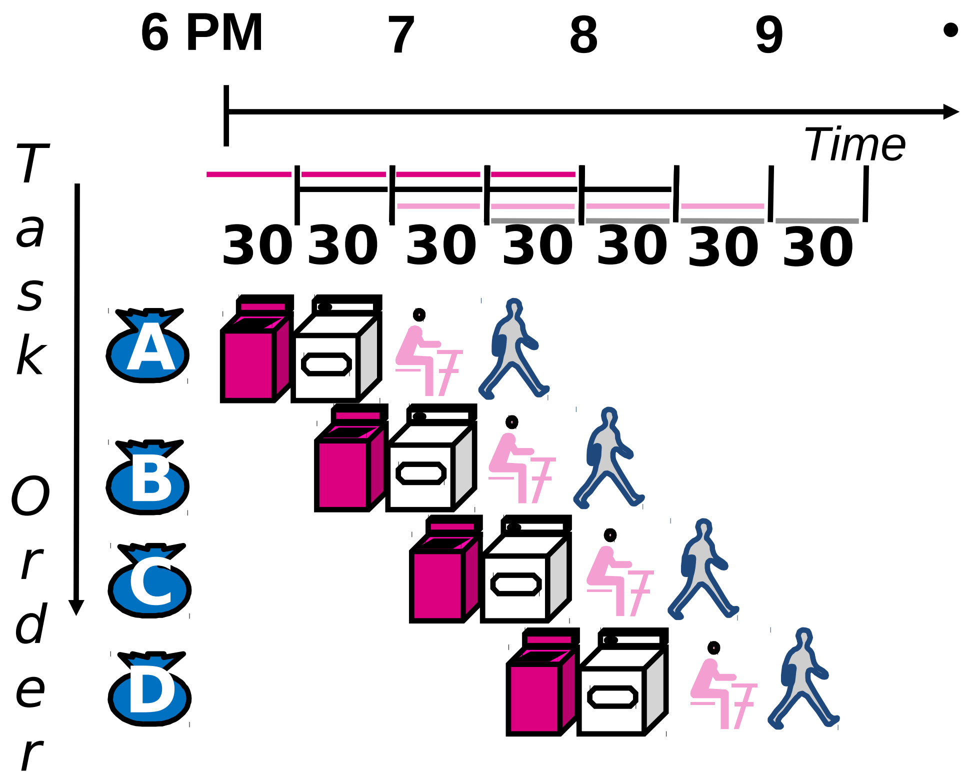


Pipelined Laundry



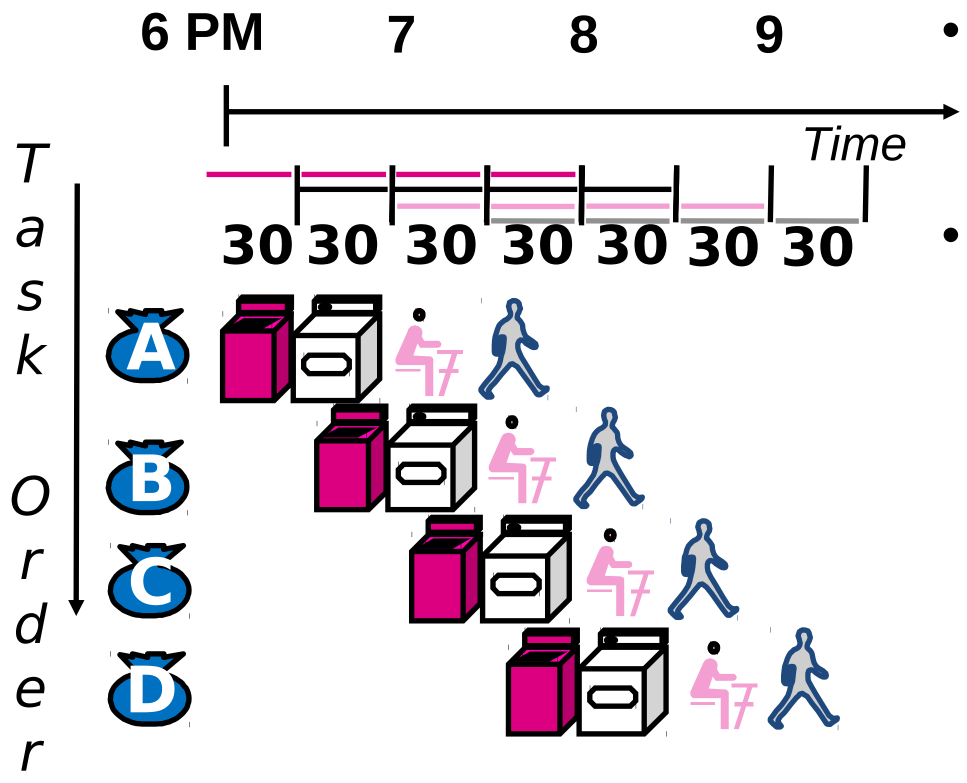
- *Pipelined laundry takes 3.5 hours for 4 loads!*

Pipelining Lessons (1/2)



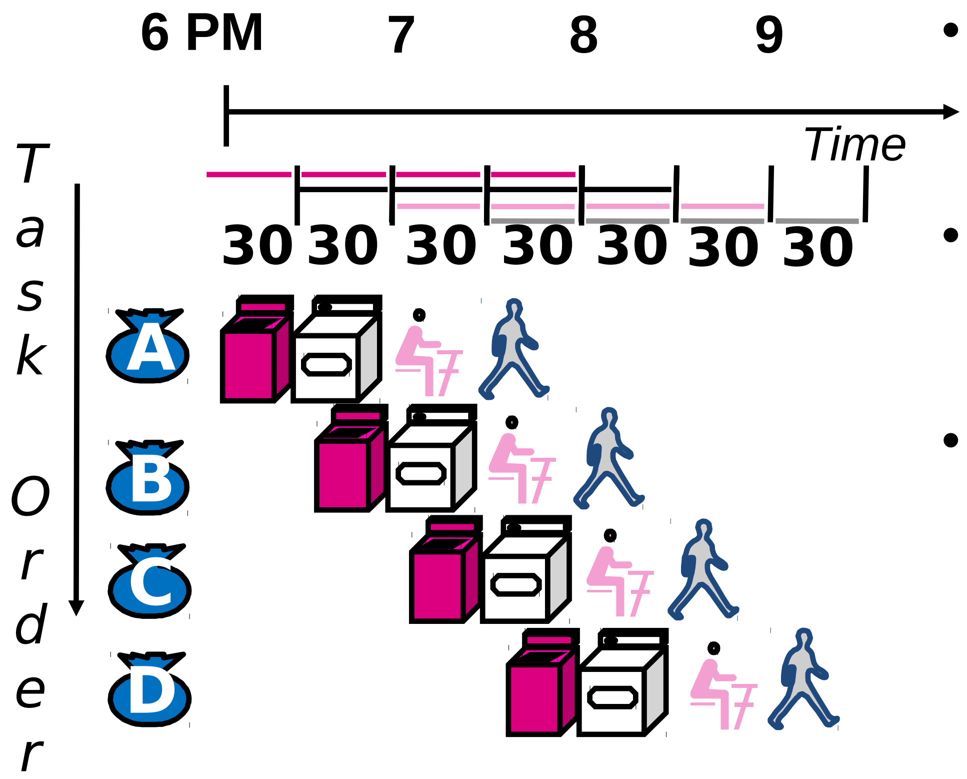
- Pipelining doesn’t help *latency* of single task, just *throughput* of entire workload

Pipelining Lessons (1/2)



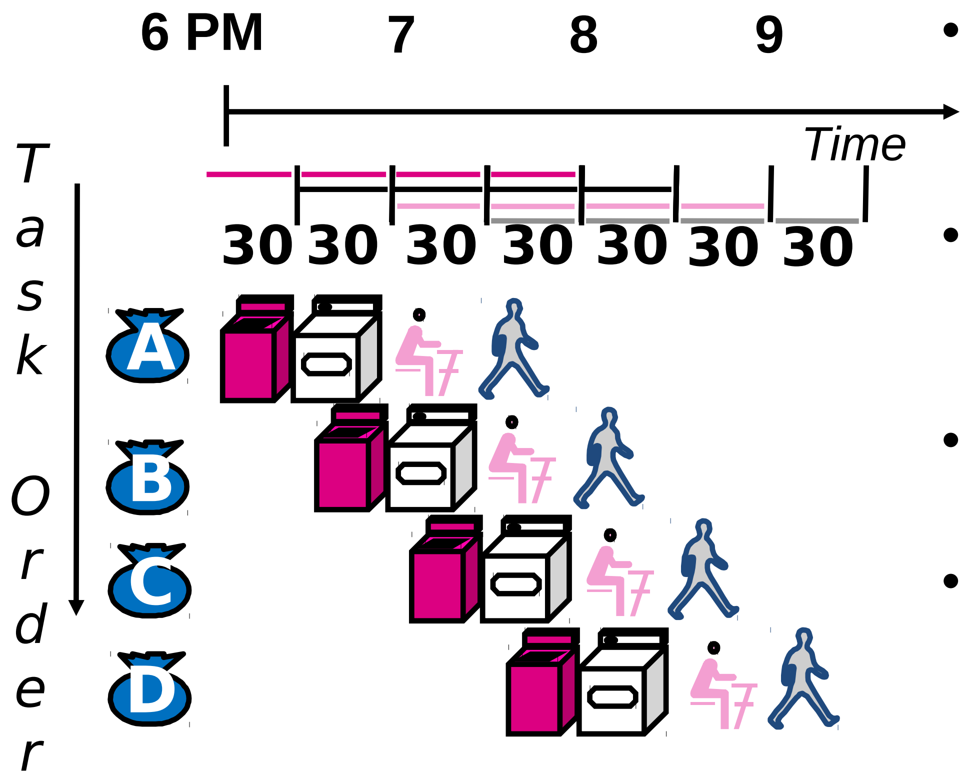
- Pipelining doesn't help *latency* of single task, just *throughput* of entire workload
- *Multiple* tasks operating simultaneously using different resources

Pipelining Lessons (1/2)



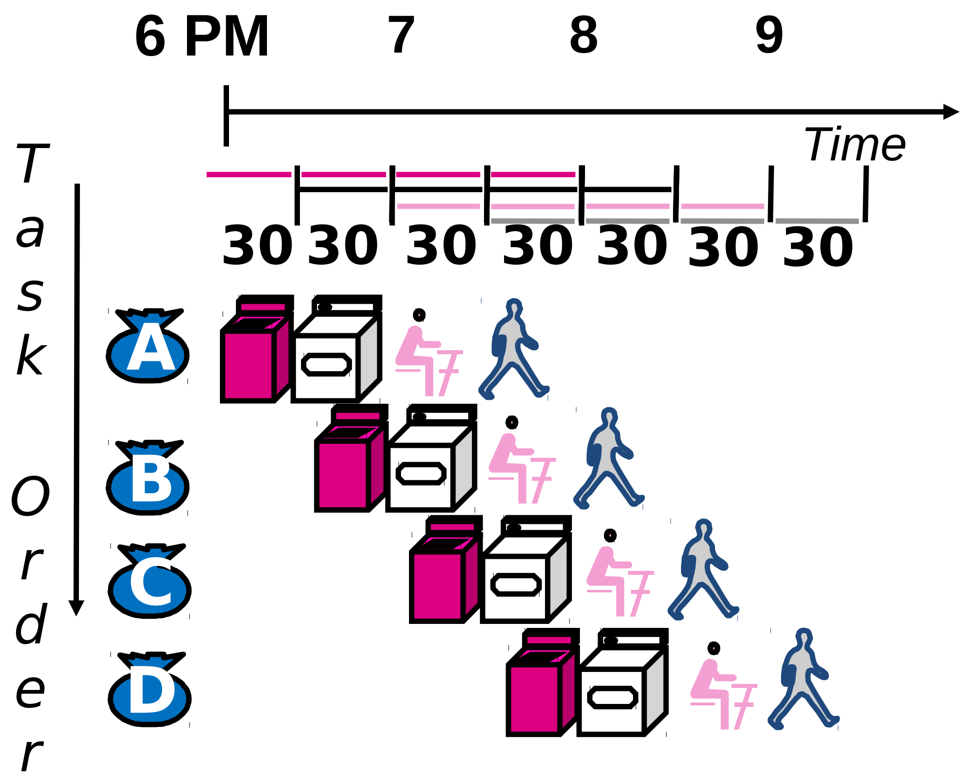
- Pipelining doesn't help *latency* of single task, just *throughput* of entire workload
- *Multiple* tasks operating simultaneously using different resources
- Potential speedup = number of pipeline stages

Pipelining Lessons (1/2)



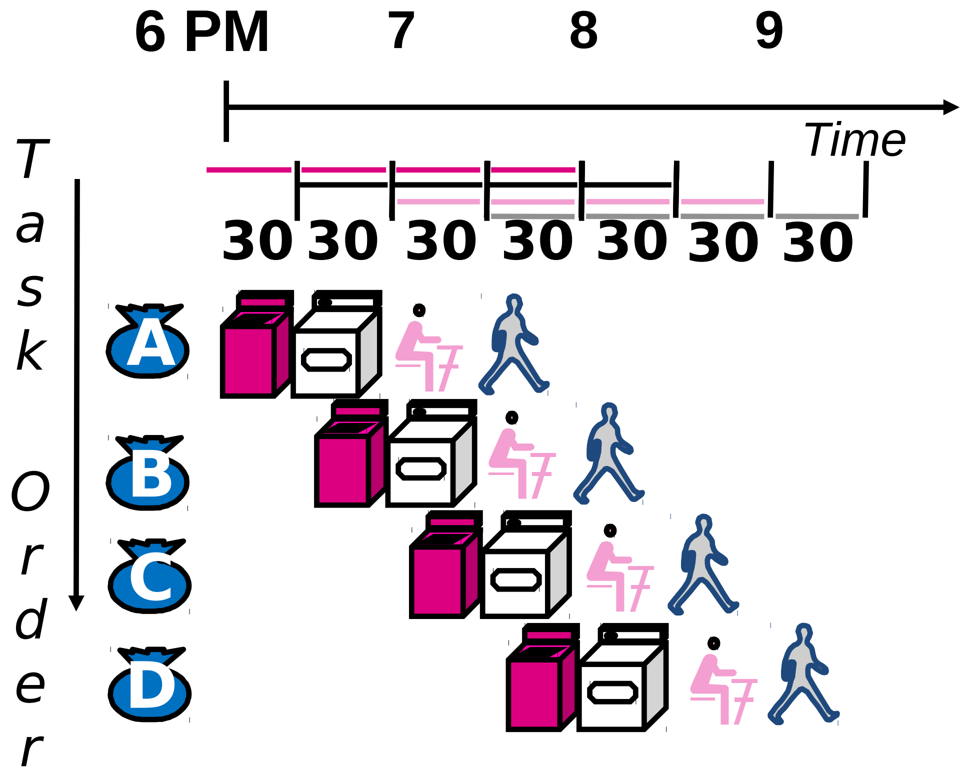
- Pipelining doesn't help *latency* of single task, just *throughput* of entire workload
- *Multiple* tasks operating simultaneously using different resources
- Potential speedup = number of pipeline stages
- Speedup reduced by time to *fill* and *drain* the pipeline: 8 hours/3.5 hours or 2.3X v. potential 4X in this example

Pipelining Lessons (2/2)



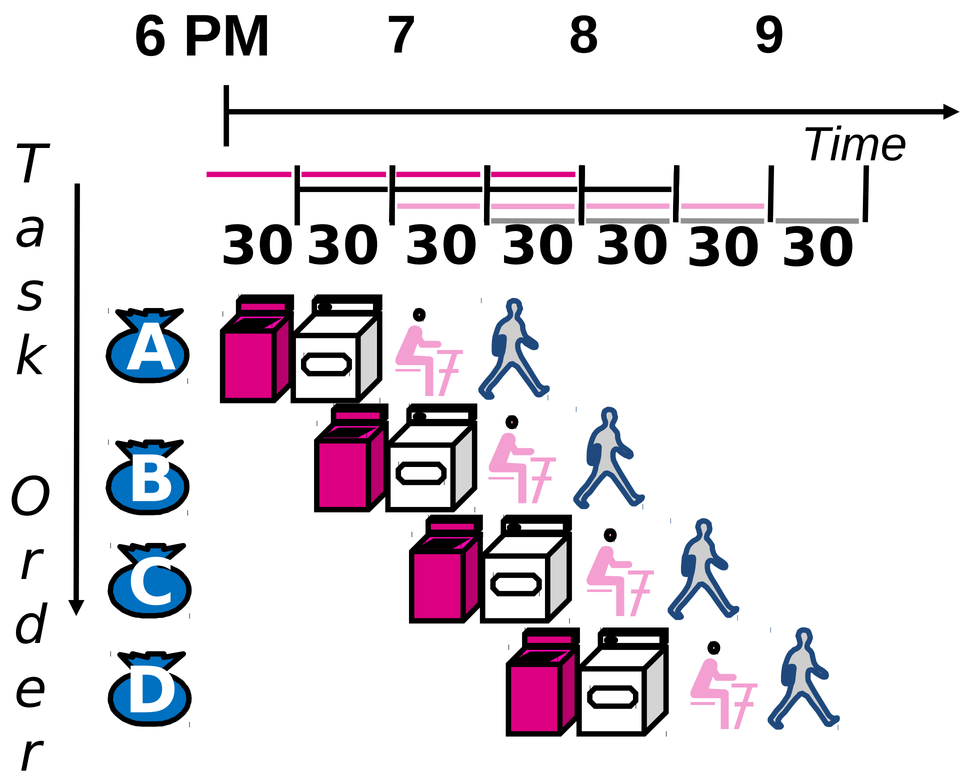
- Suppose new Washer takes 20 minutes, new Stasher takes 20 minutes. How much faster is pipeline?

Pipelining Lessons (2/2)



- Suppose new Washer takes 20 minutes, new Stasher takes 20 minutes. How much faster is pipeline?
  - Pipeline rate limited by *slowest* pipeline stage

Pipelining Lessons (2/2)



- Suppose new Washer takes 20 minutes, new Stasher takes 20 minutes. How much faster is pipeline?
  - Pipeline rate limited by *slowest* pipeline stage
  - Unbalanced lengths of pipeline stages reduces speedup

Agenda

- Quick Datapath Review
- Control Implementation
- Administrivia
- Clocking Methodology
- Pipelined Execution
- Pipelined Datapath

Recall: 5 Stages of MIPS Datapath

1) IF: Instruction Fetch, Increment PC

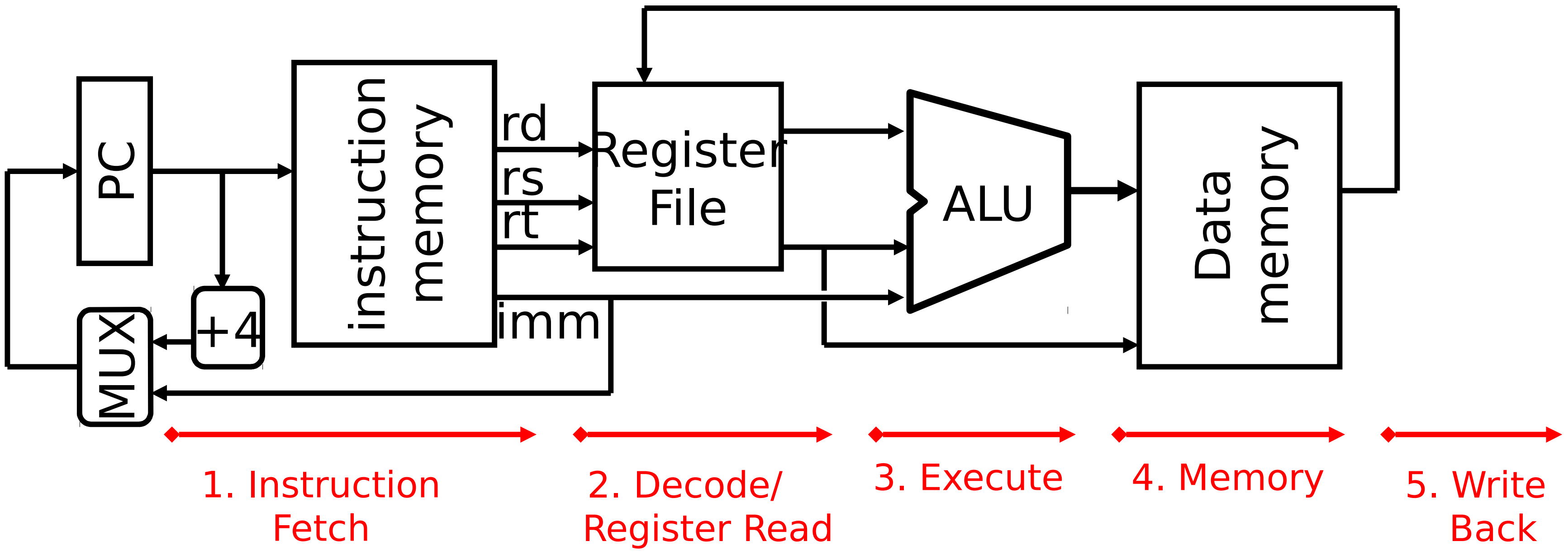
2) ID: Instruction Decode, Read Registers

3) EX: Execution (ALU)
   Load/Store: Calculate Address
   Others: Perform Operation

4) MEM:
   Load: Read Data from Memory
   Store: Write Data to Memory

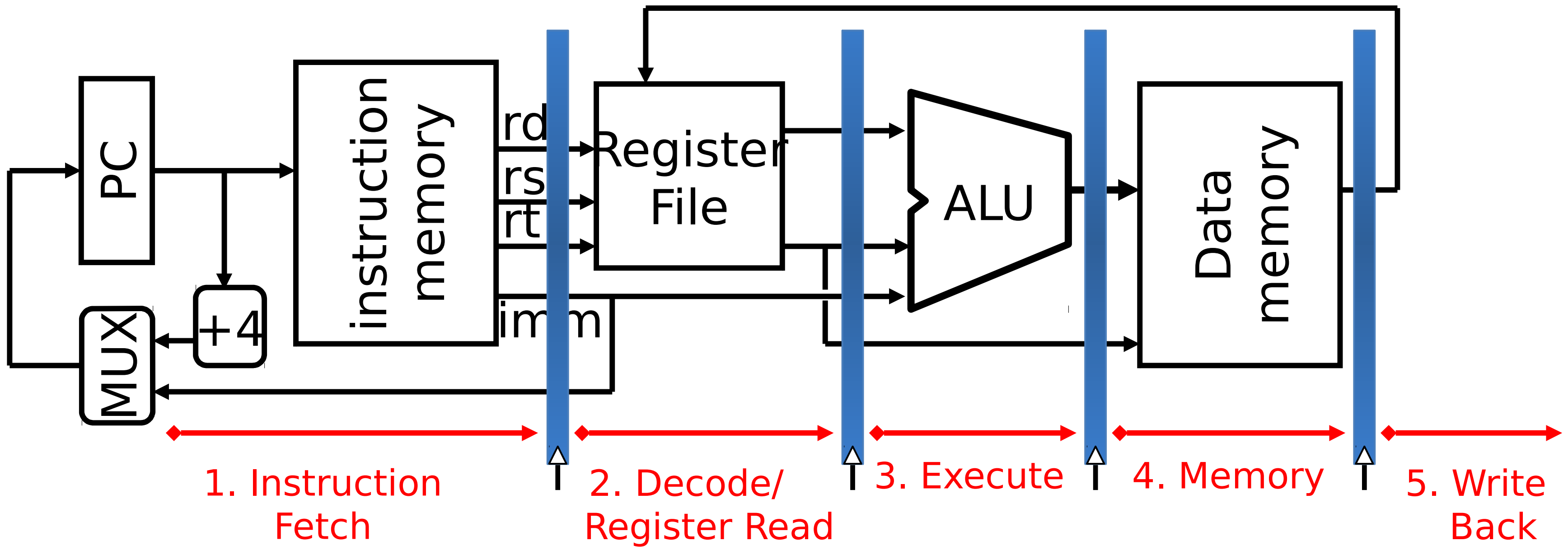
5) WB: Write Data Back to Register

Pipelined Datapath



- Add registers between stages
  - Hold information produced in previous cycle

Pipelined Datapath



- Add registers between stages
  - Hold information produced in previous cycle
- 5 stage pipeline
  - Clock rate *potentially* 5x faster

Pipelining Changes

- Registers affect flow of information
  - Name registers for adjacent stages (e.g. IF/ID)

Pipelining Changes

- Registers affect flow of information
  - Name registers for adjacent stages (e.g. IF/ID)
  - Registers *separate* the information between stages
    - You can still pass information *around* registers

Pipelining Changes

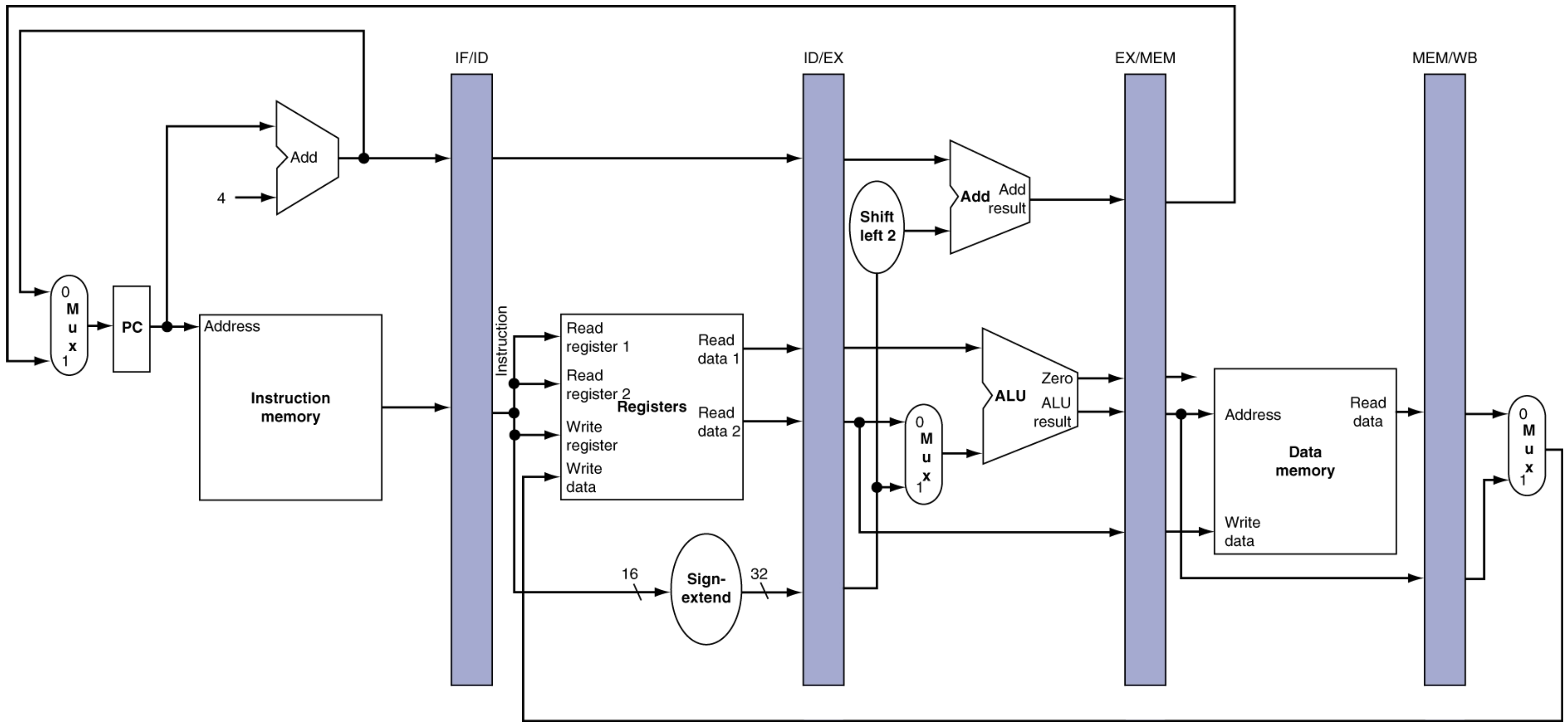
- Registers affect flow of information
  - Name registers for adjacent stages (e.g. IF/ID)
  - Registers *separate* the information between stages
    - You can still pass information *around* registers
  - At any instance of time, each stage working on a *different* instruction!

Pipelining Changes

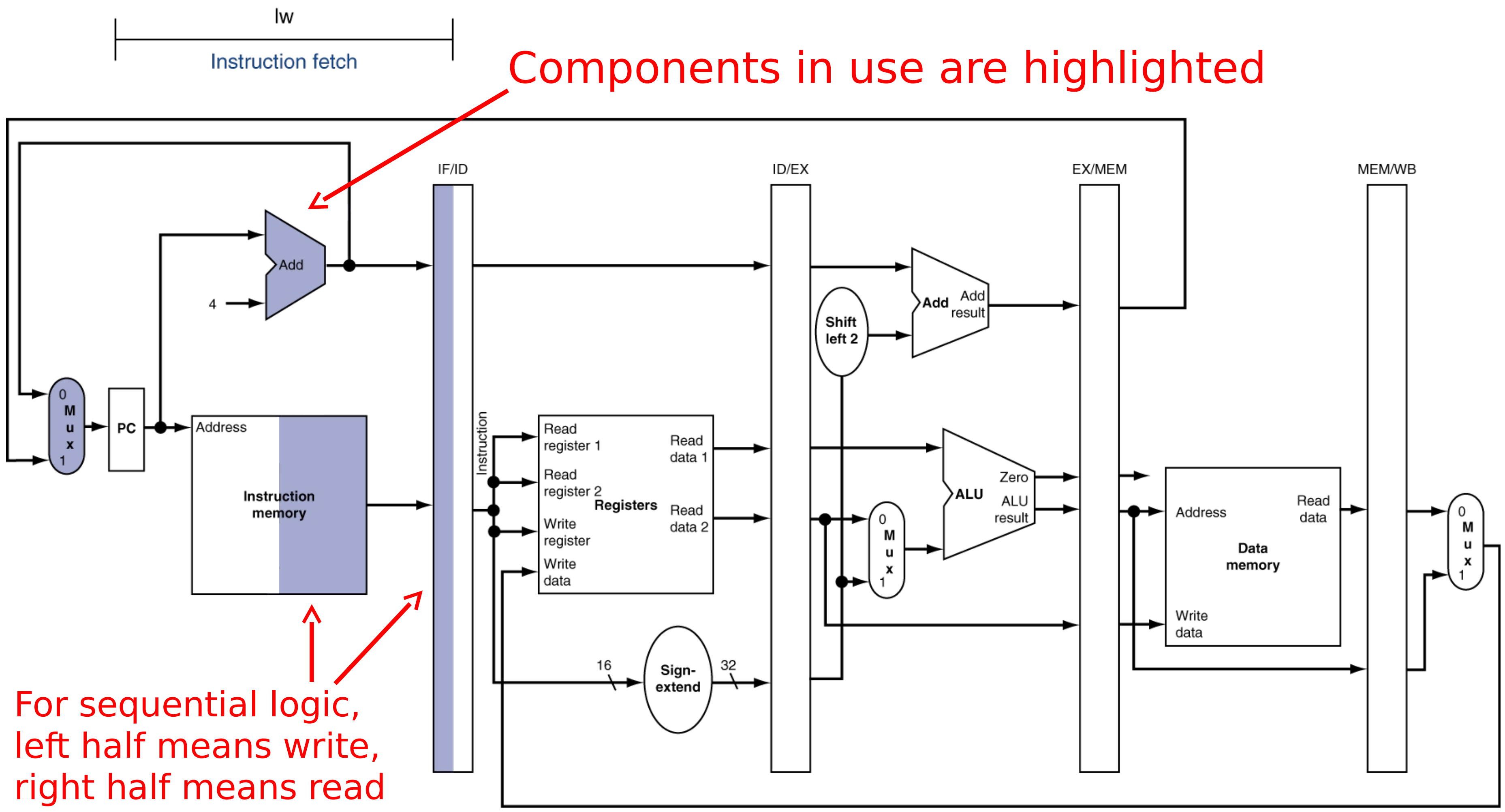
- Registers affect flow of information
  - Name registers for adjacent stages (e.g. IF/ID)
  - Registers *separate* the information between stages
    - You can still pass information *around* registers
  - At any instance of time, each stage working on a *different* instruction!
- Will need to re-examine placement of wires and hardware in datapath

More Detailed Pipeline

- Examine flow through pipeline for `lw`



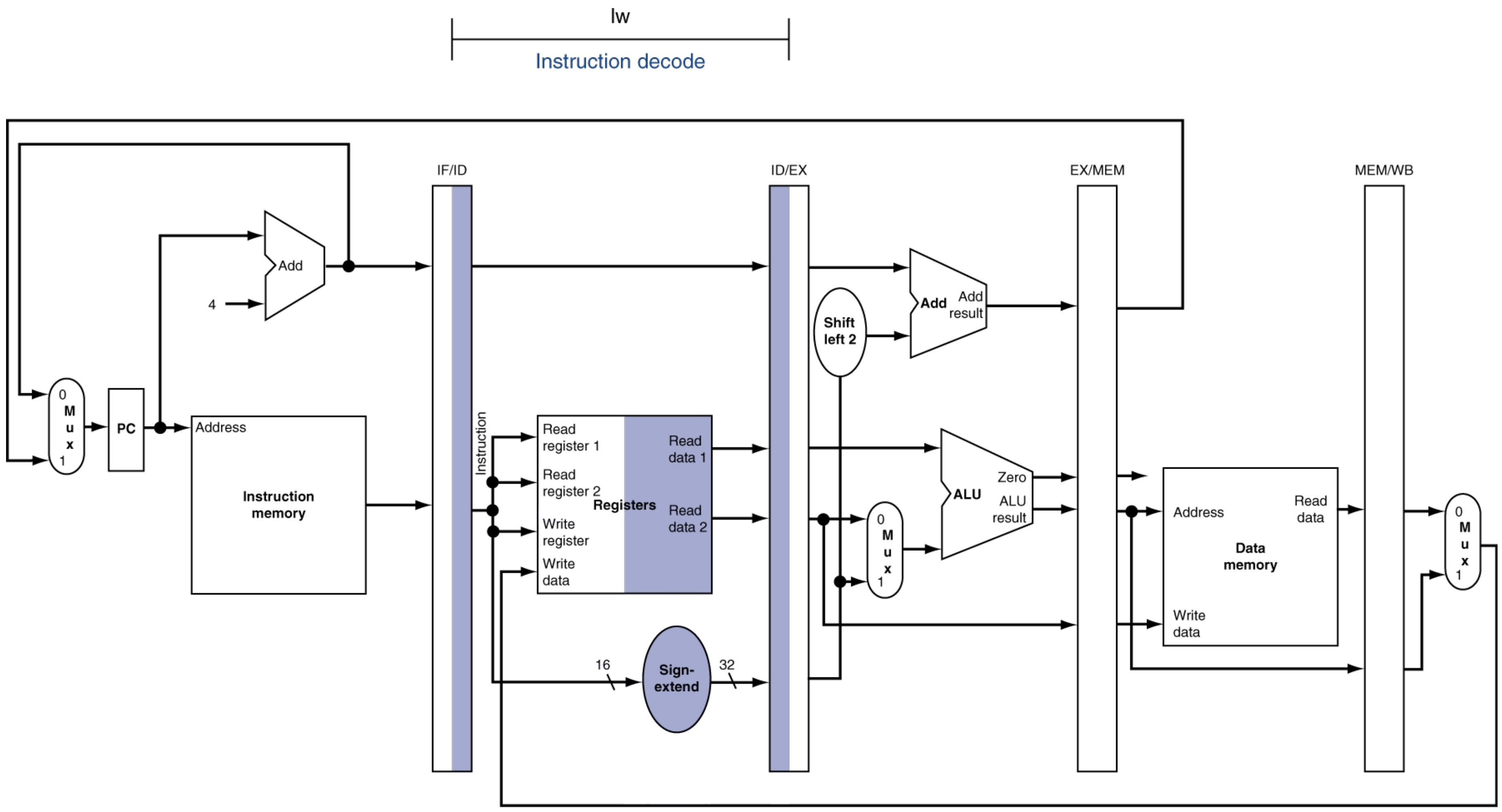
Instruction Fetch (IF) for Load



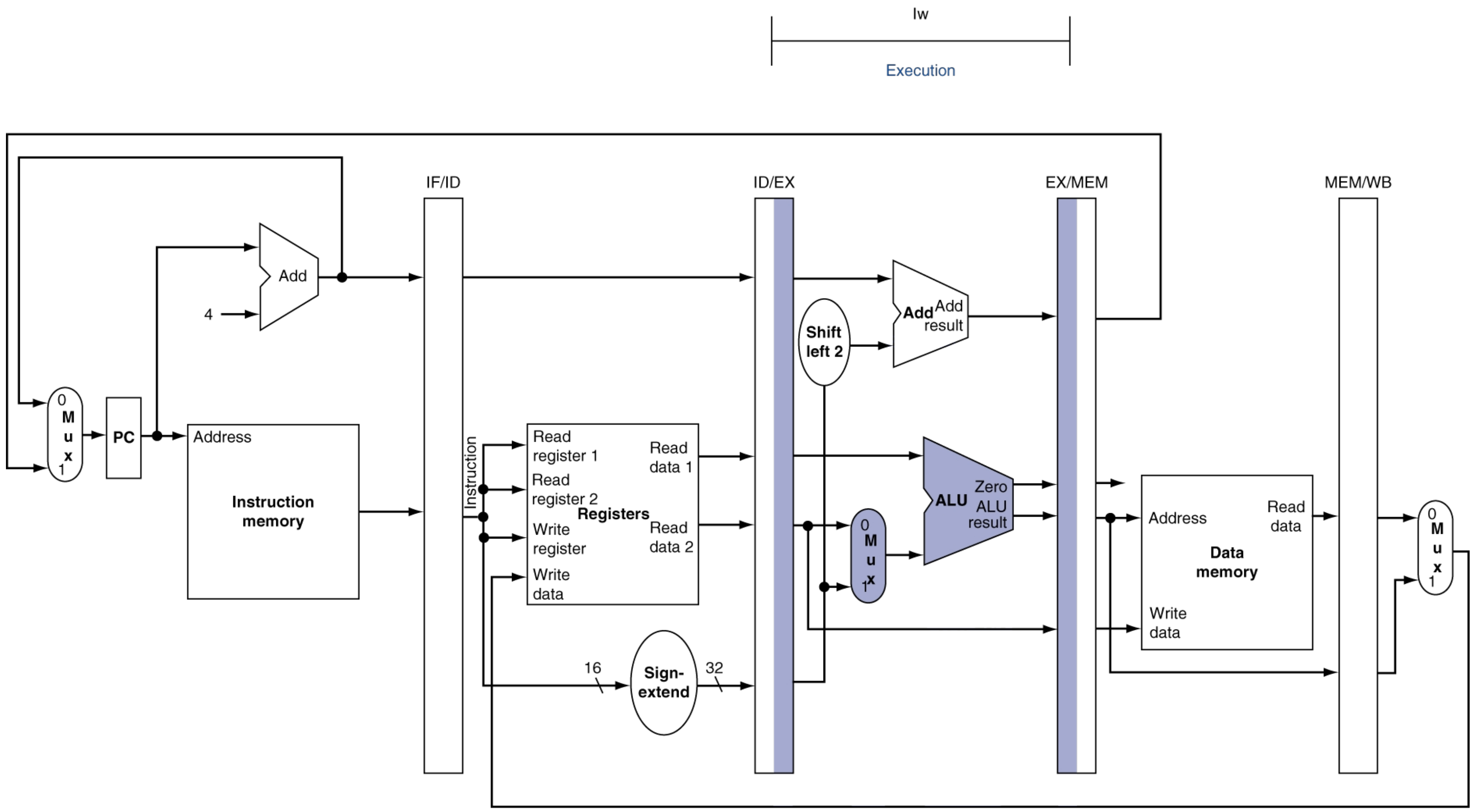
Components in use are highlighted

For sequential logic,
left half means write,
right half means read

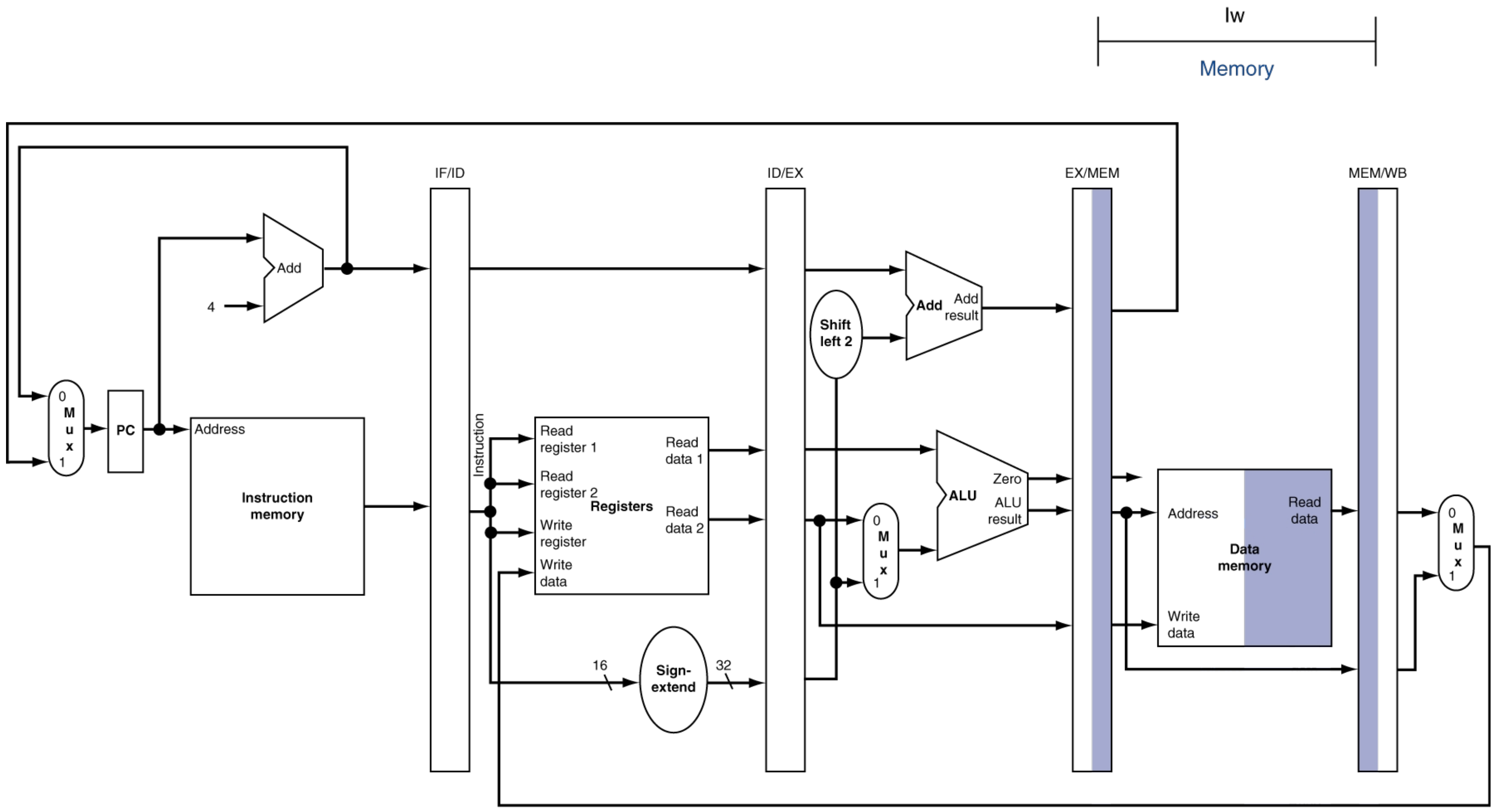
Instruction Decode (ID) for Load



Execute (EX) for Load

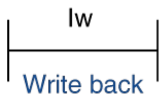


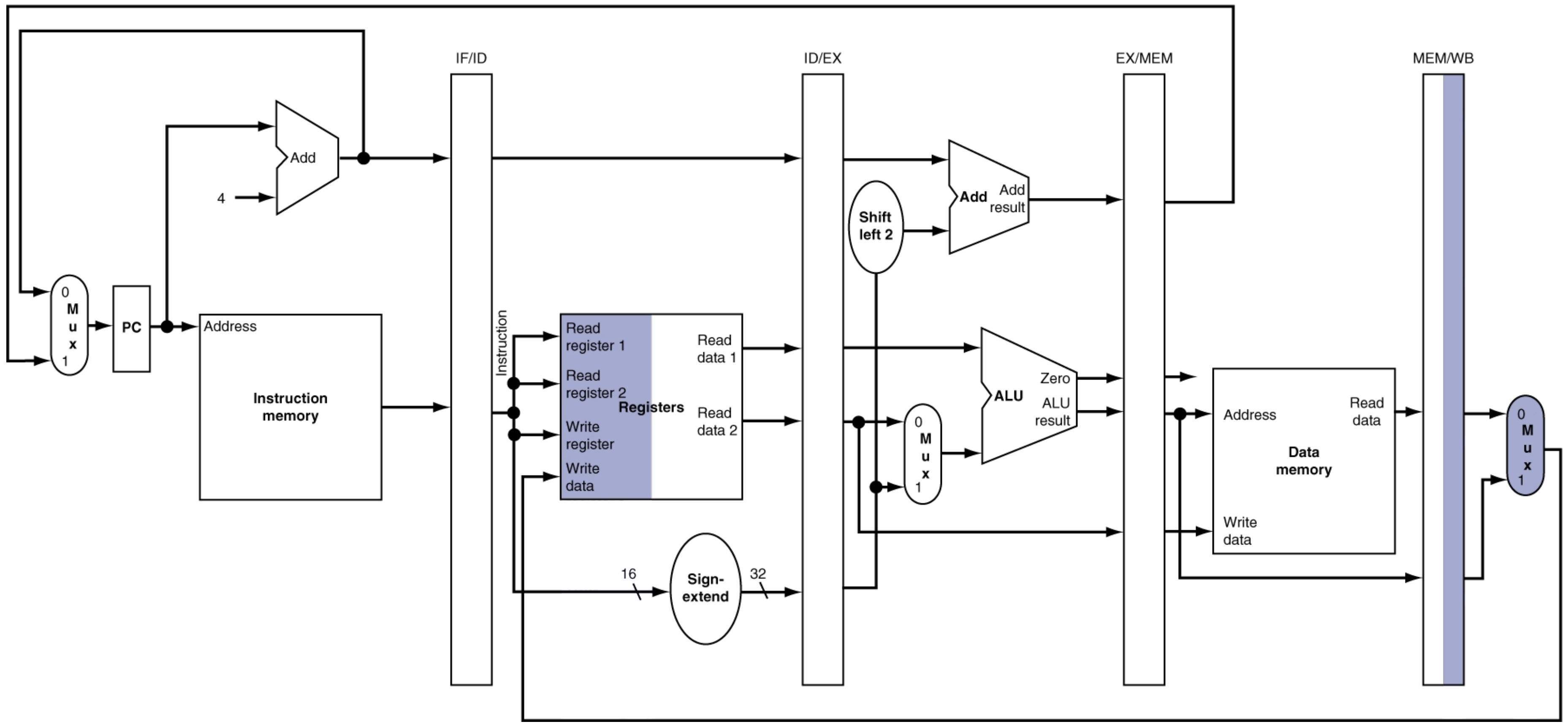
Memory (MEM) for Load



Write Back (WB) for Load

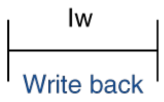
There's something wrong here! (Can you spot it?)

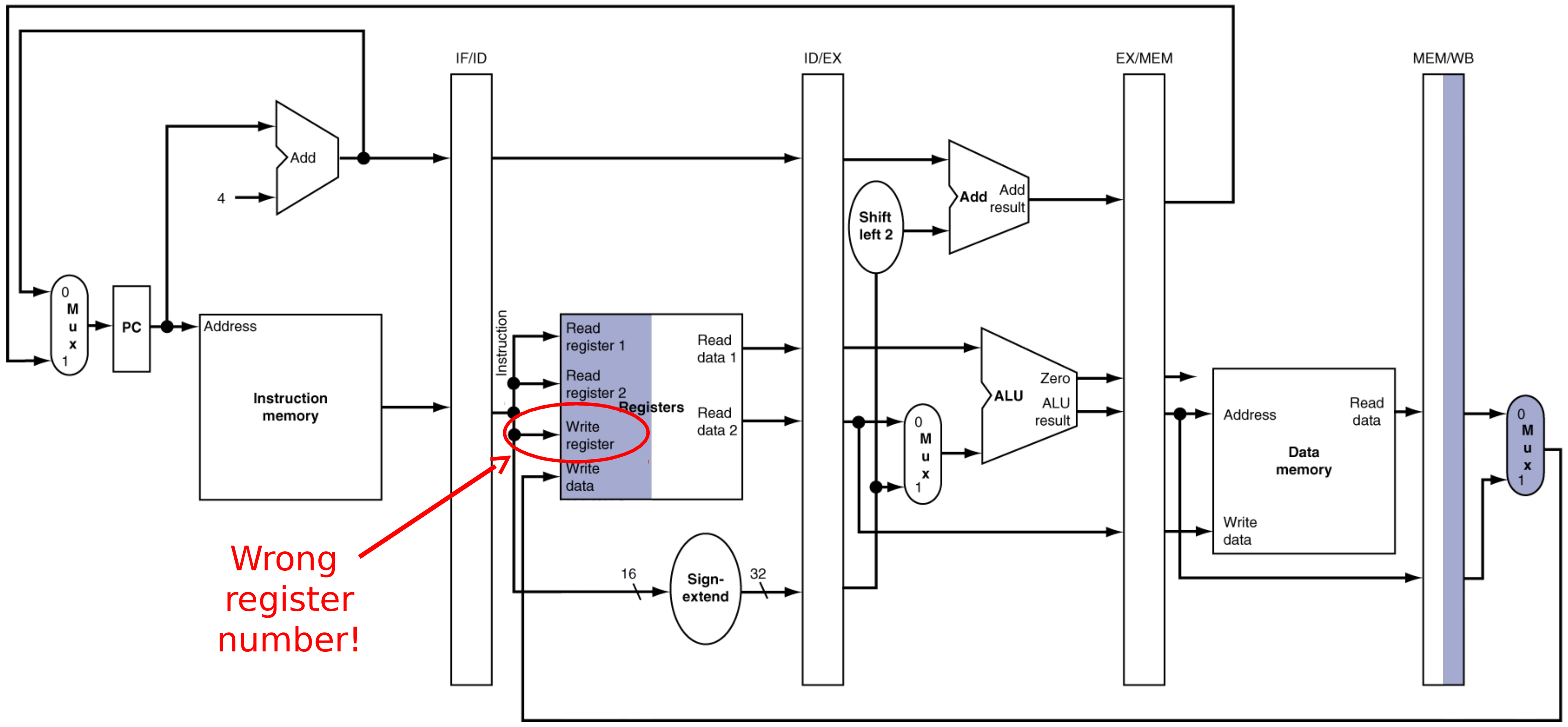




Write Back (WB) for Load

There's something wrong here! (Can you spot it?)

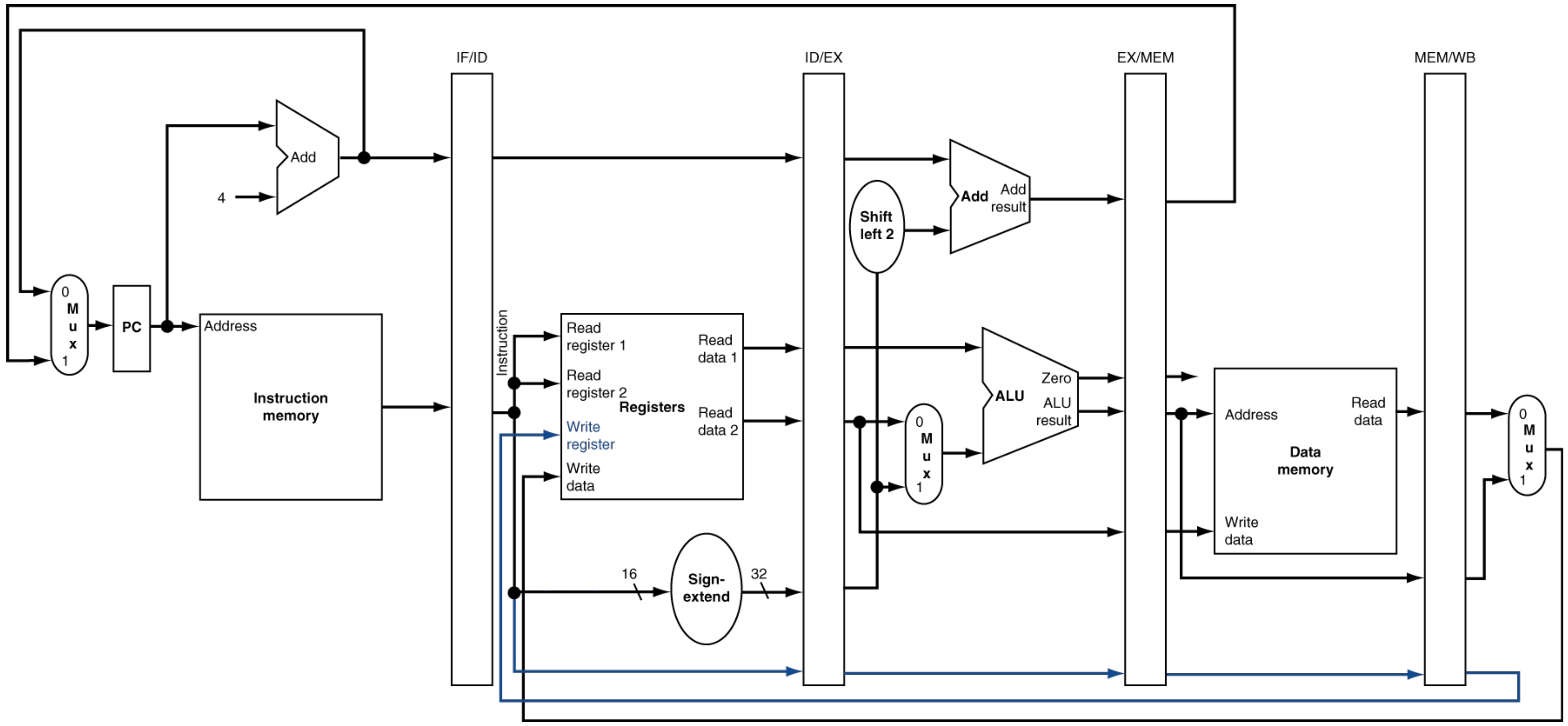




Wrong register number!

Corrected Datapath

- Now any instruction that writes to a register will work properly



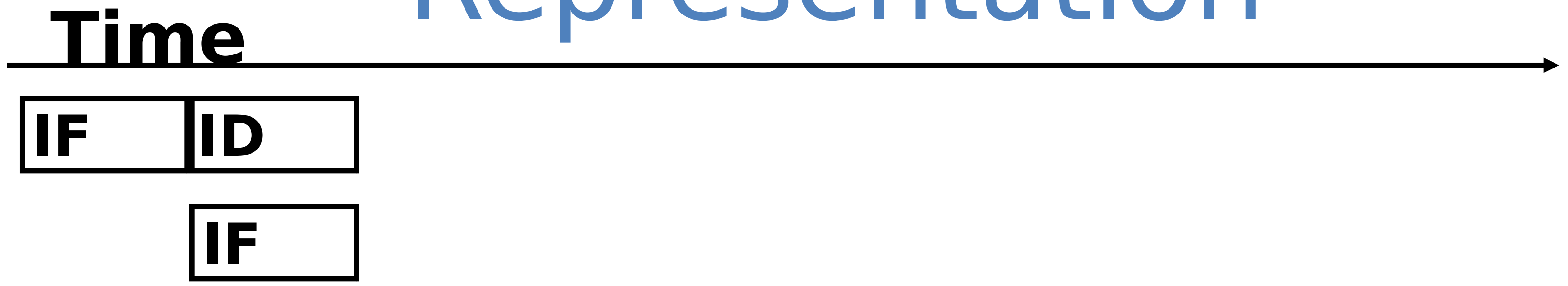
Technology Break

Agenda

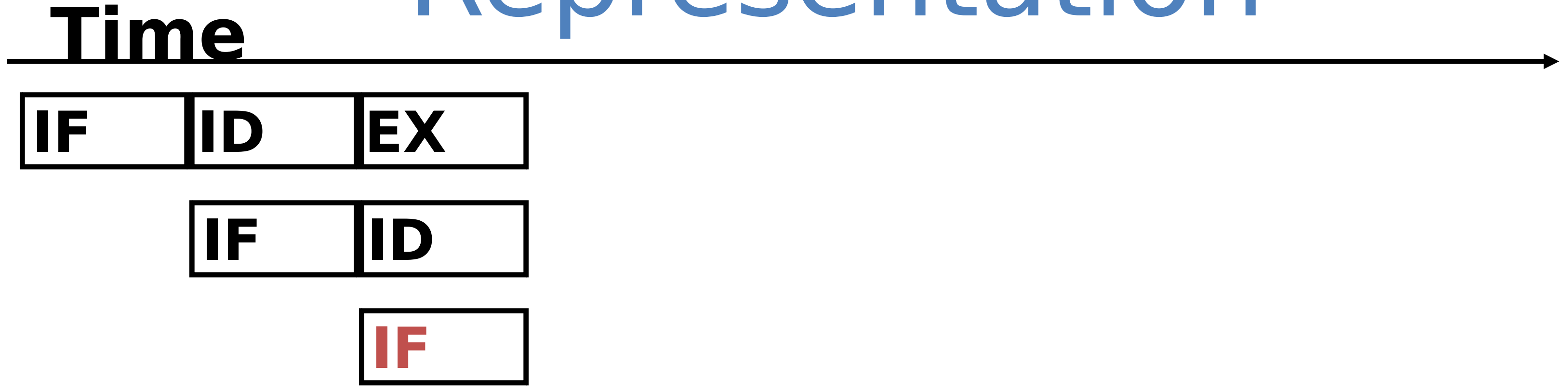
- Quick Datapath Review
- Control Implementation
- Administrivia
- Clocking Methodology
- Pipelined Execution
- Pipelined Datapath (Continued)

Pipelined Execution Representation

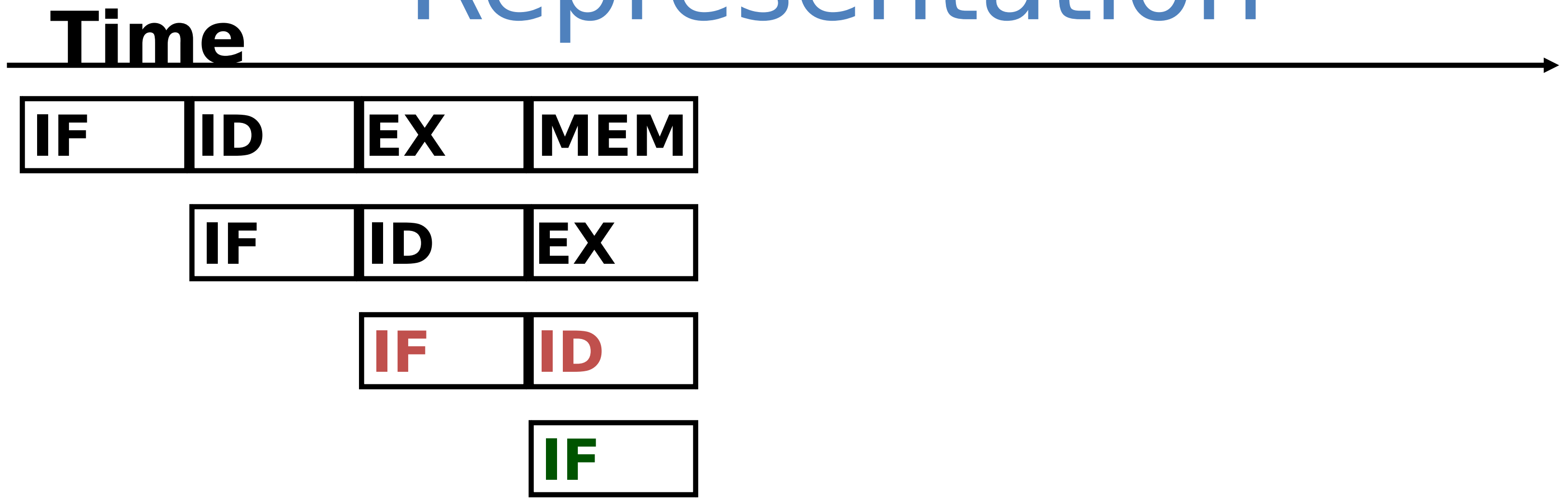
Pipelined Execution Representation



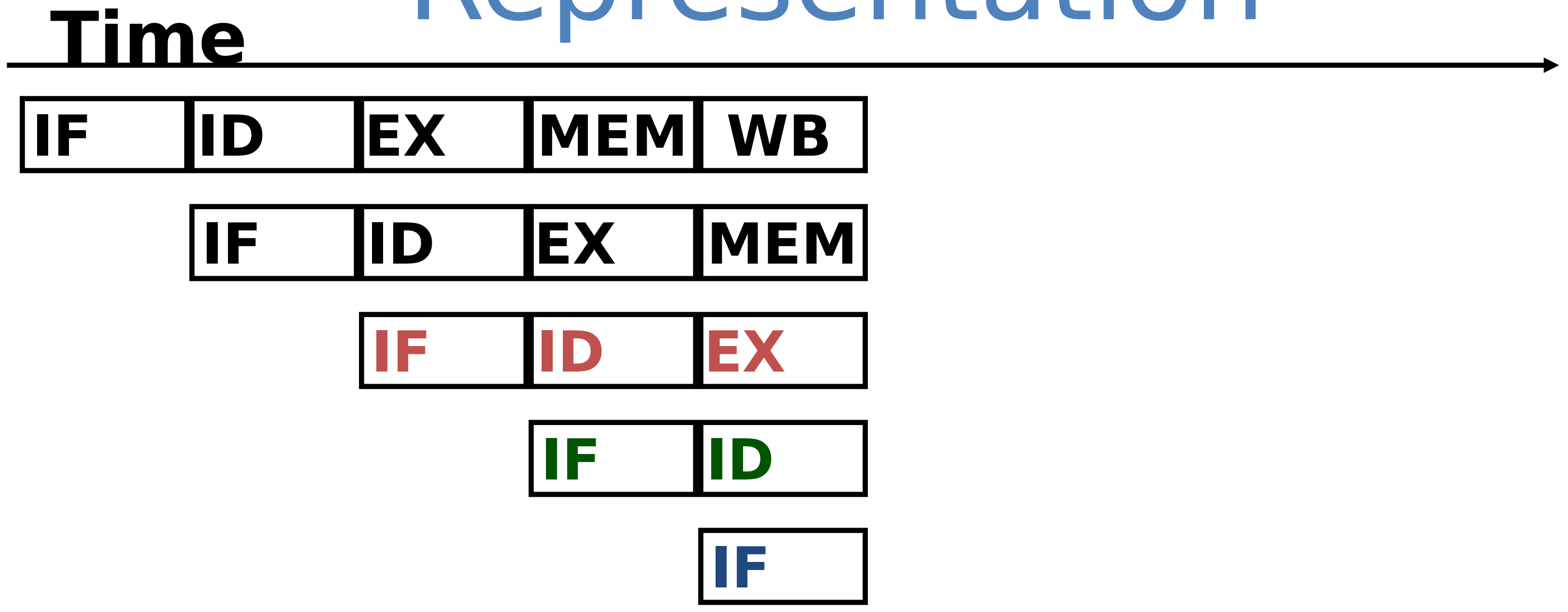
Pipelined Execution Representation



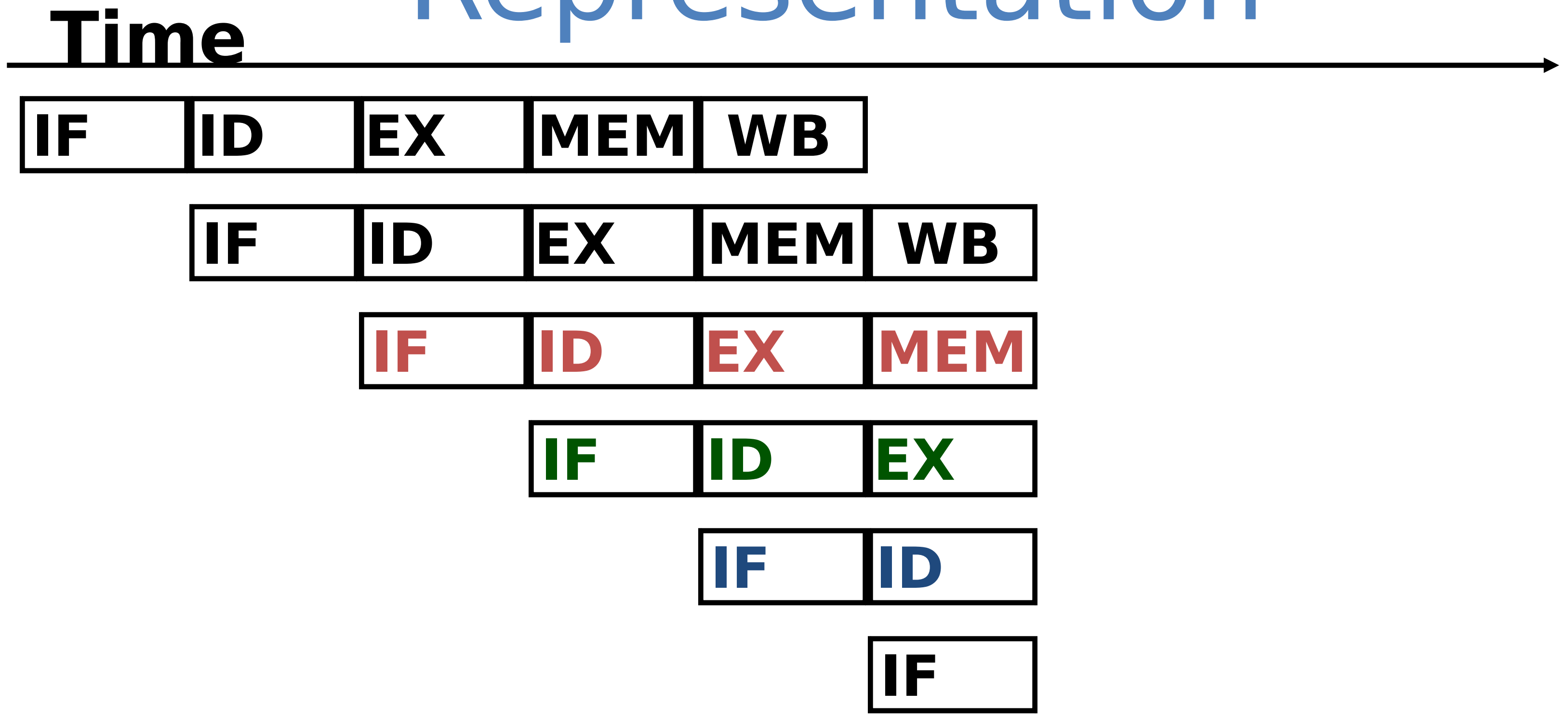
Pipelined Execution Representation



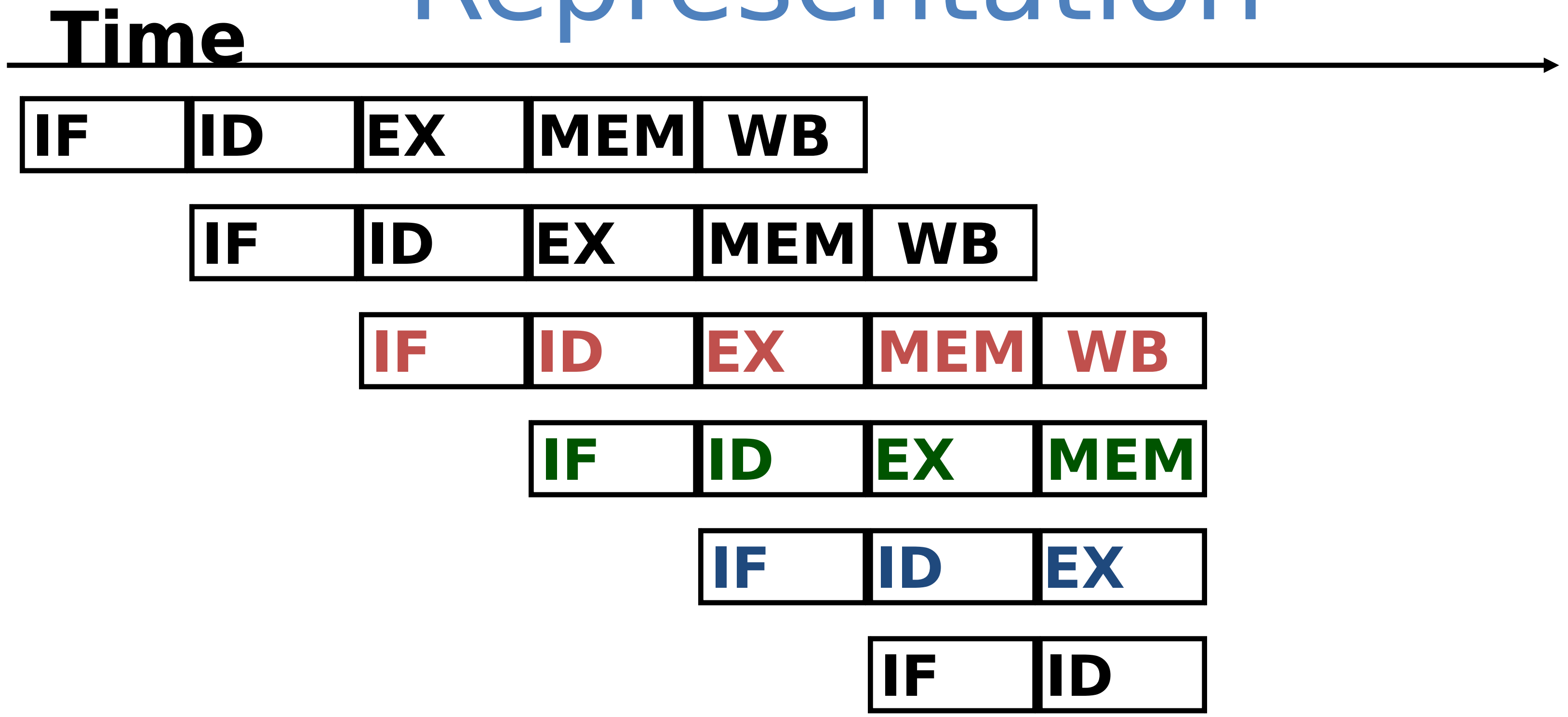
Pipelined Execution Representation



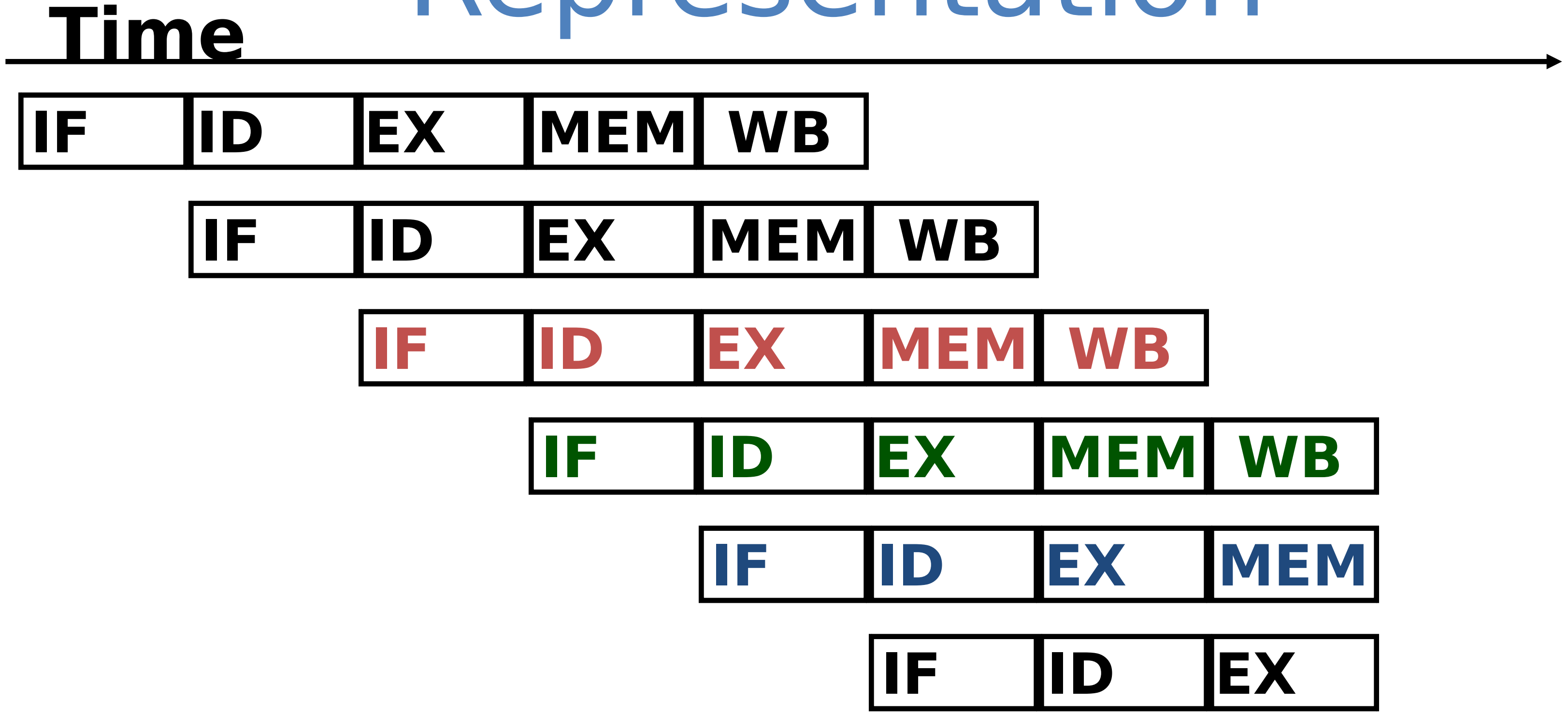
Pipelined Execution Representation



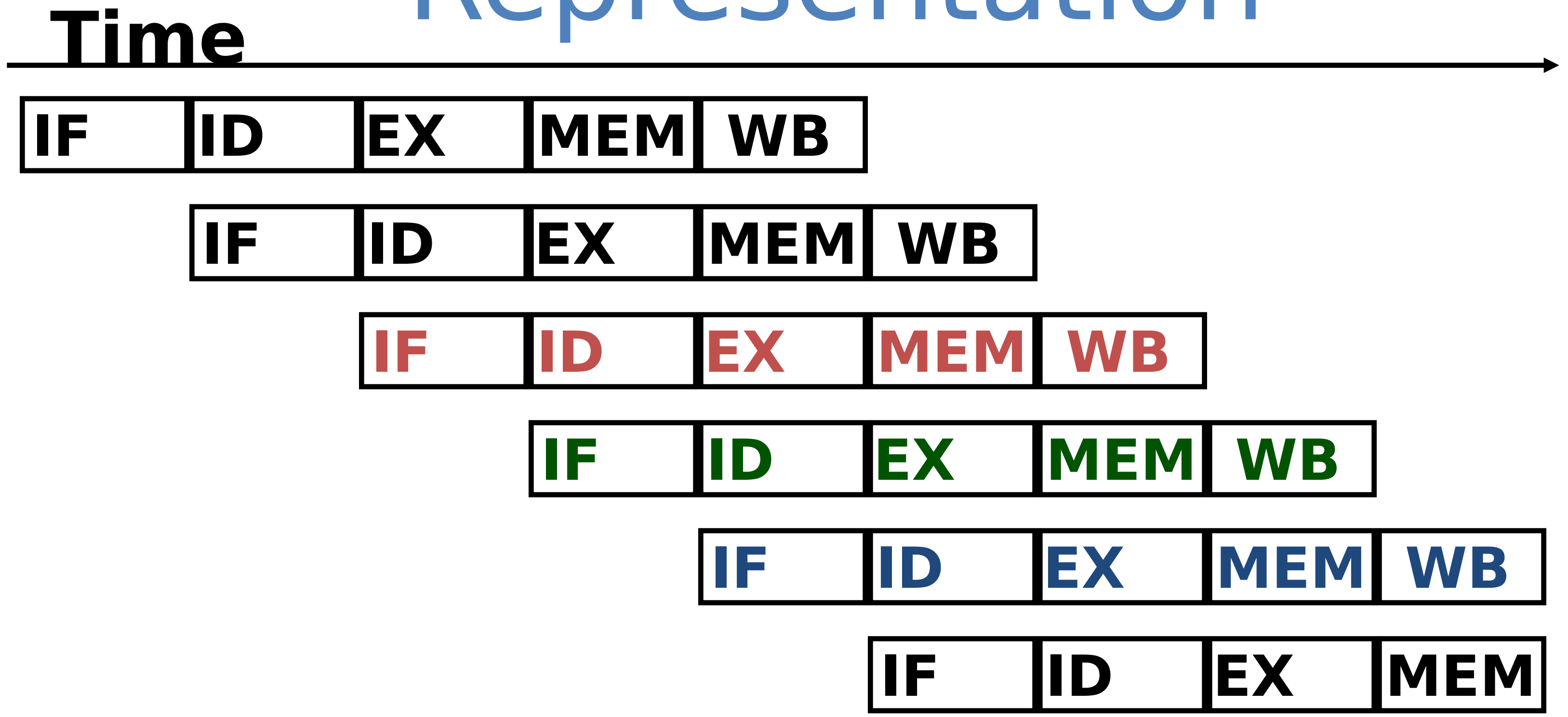
Pipelined Execution Representation



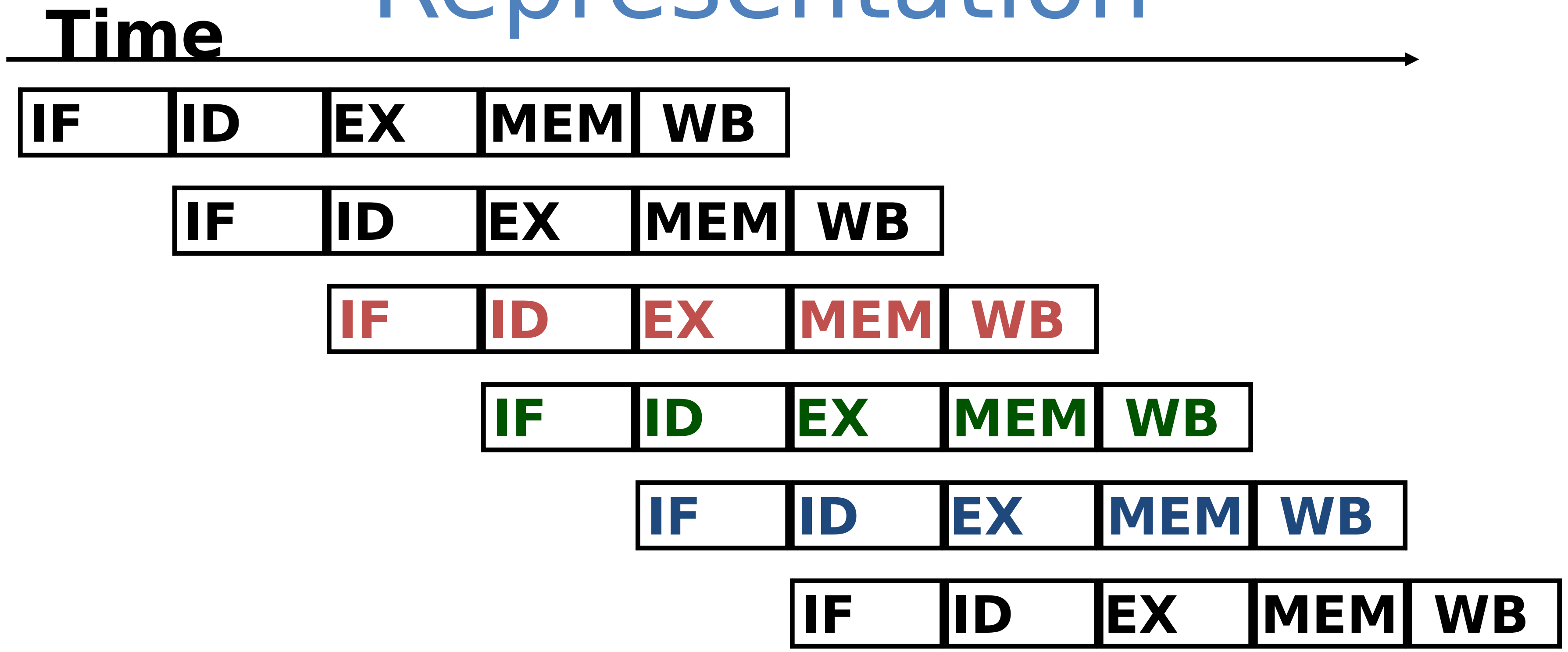
Pipelined Execution Representation



Pipelined Execution Representation

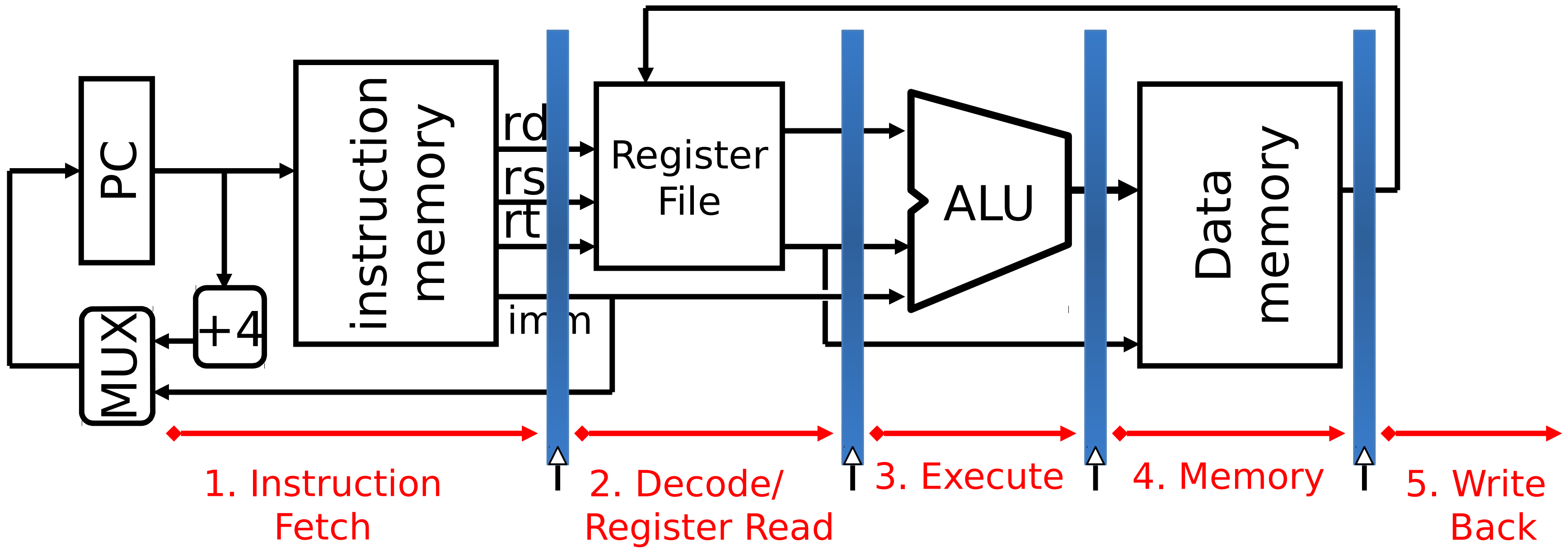


Pipelined Execution Representation

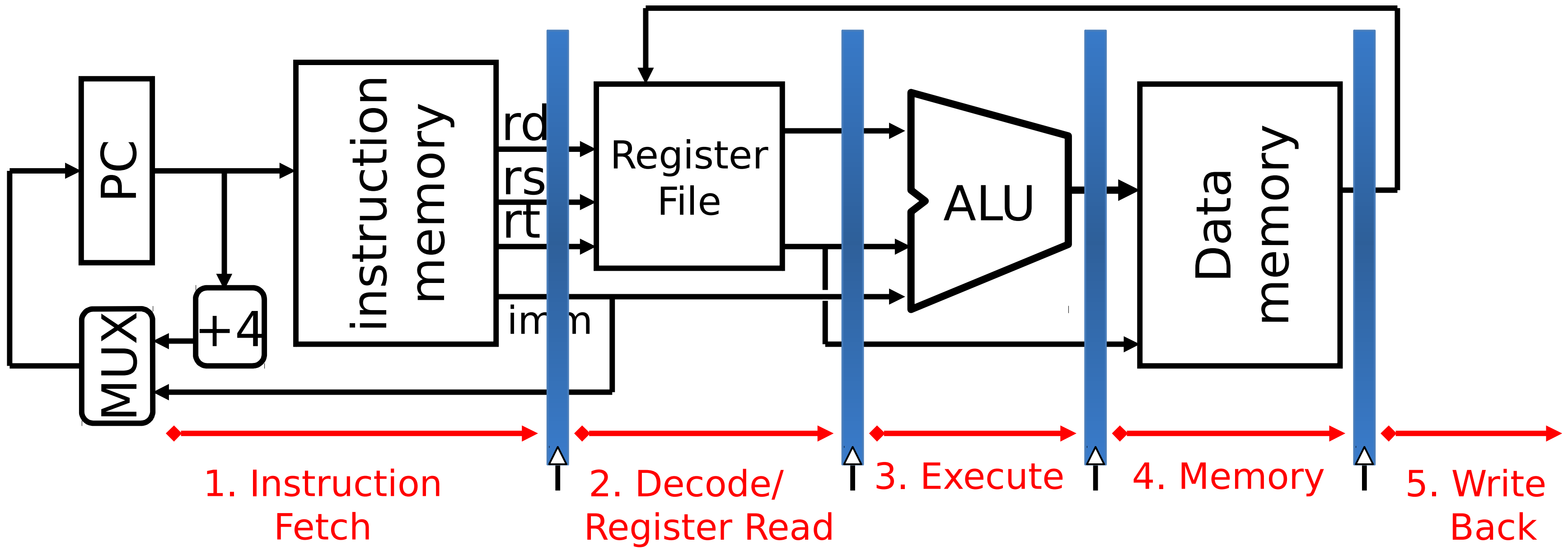


- Every instruction must take same number of steps, so some will idle
  - e.g. MEM stage for any arithmetic instruction

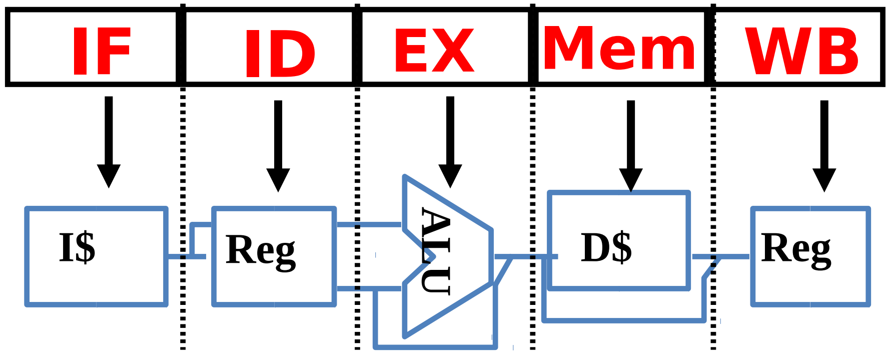
Graphical Pipeline Diagrams



Graphical Pipeline Diagrams

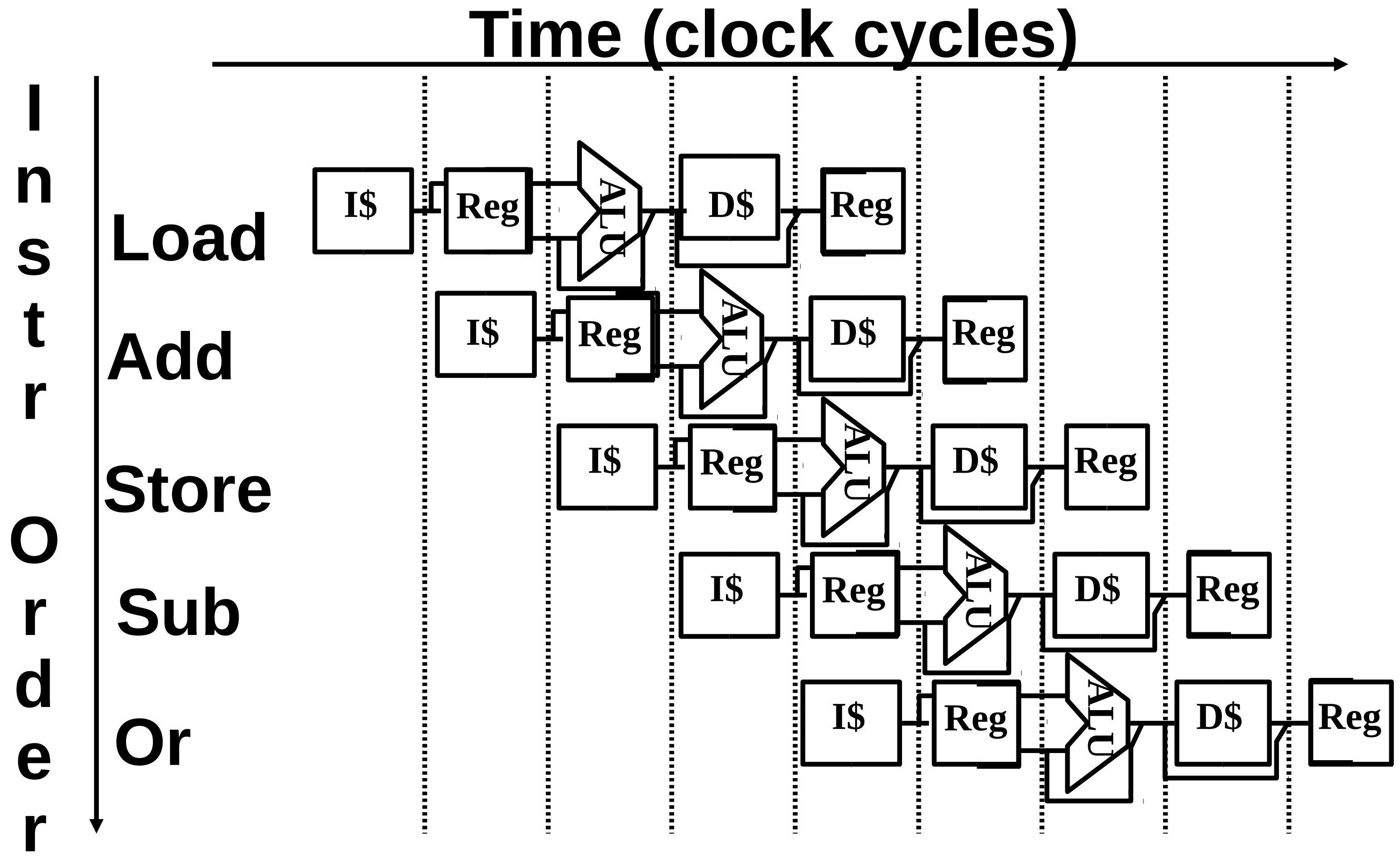


- Use datapath figure below to represent pipeline:



Graphical Pipeline Representation

- RegFile: right half is read, left half is write



Instruction Level Parallelism (ILP)

- Pipelining allows us to execute parts of multiple instructions at the same time using the same hardware!
  - This is known as *instruction level parallelism*
- **Recall:** Types of parallelism
  - DLP: same operation on lots of data (SIMD)
  - TLP: executing multiple threads “simultaneously” (OpenMP)

Pipeline Performance (1/3)

- Use T_c ("time between completion of instructions") to measure speedup
  - $$T_{c,\text{pipelined}} \geq \frac{T_{c,\text{single-cycle}}}{\text{Number of stages}}$$

Pipeline Performance (1/3)

- Use T_c ("time between completion of instructions") to measure speedup
  - $T_{c,\text{pipelined}} \geq \dfrac{T_{c,\text{single-cycle}}}{\text{Number of stages}}$
  - Equality only achieved if stages are *balanced* (i.e. take the same amount of time) and register timing costs are negligible
- If not balanced, speedup is reduced

Pipeline Performance (1/3)

- Use T_c ("time between completion of instructions") to measure speedup
  - $$T_{c,\text{pipelined}} \geq \frac{T_{c,\text{single-cycle}}}{\text{Number of stages}}$$
  - Equality only achieved if stages are *balanced* (i.e. take the same amount of time) and register timing costs are negligible
- If not balanced, speedup is reduced
- Speedup due to increased *throughput*
  - *Latency* for each instruction does not decrease

Pipeline Performance (2/3)

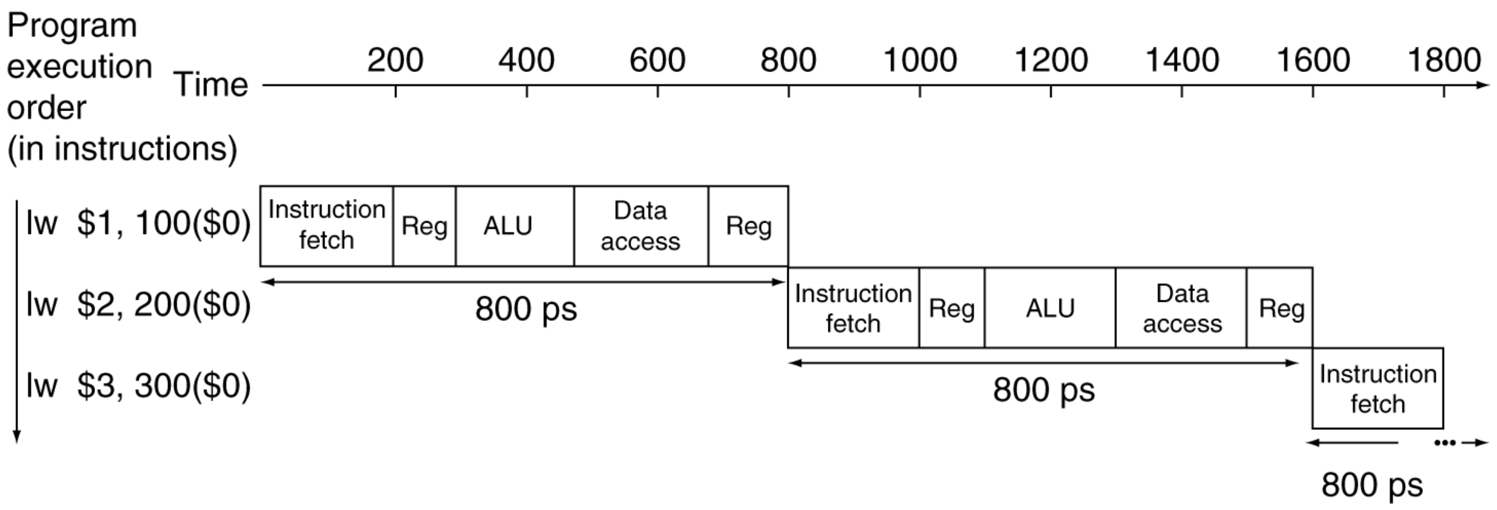
- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages

Instr	**Instr fetch**	**Register read**	**ALU op**	**Memory access**	**Register write**	**Total time**
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

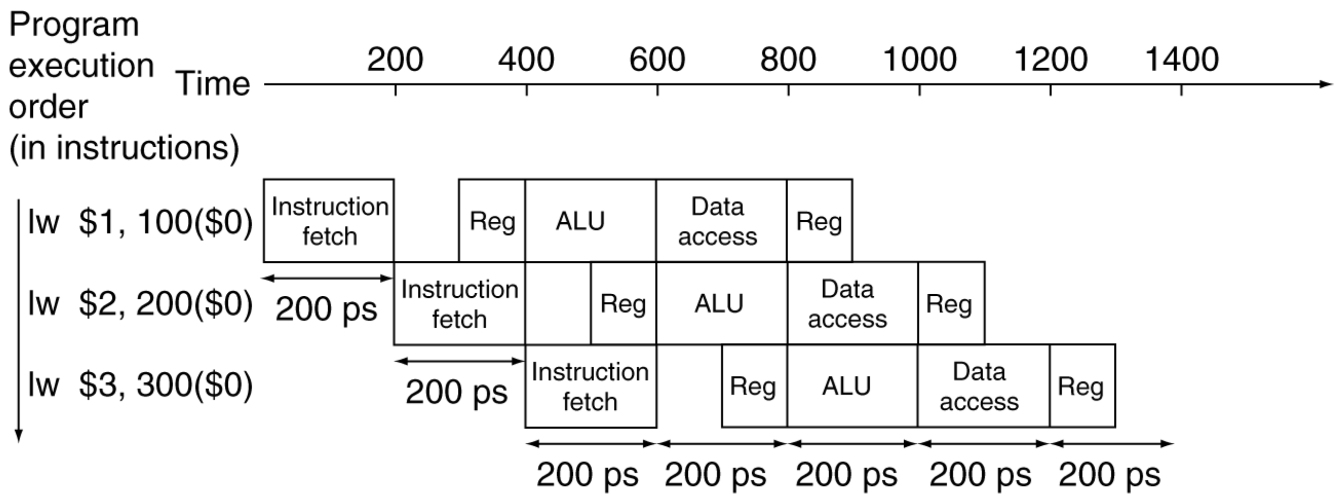
- What is pipelined clock rate?
  - Compare pipelined datapath with single-cycle datapath

Pipeline Performance (3/3)

Single-cycle

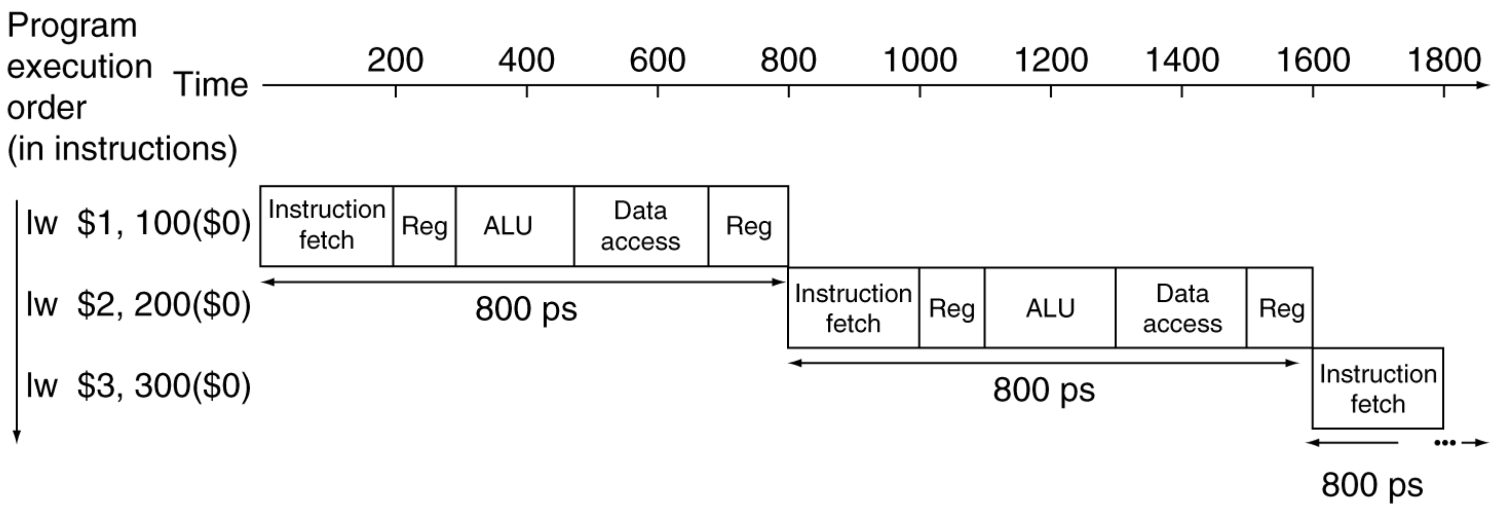


Pipelined

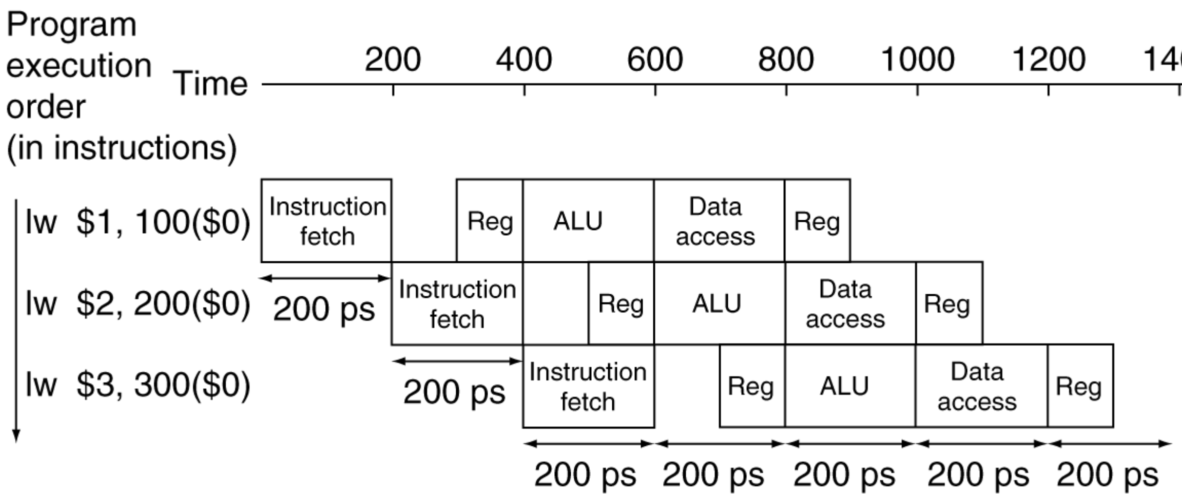


Pipeline Performance (3/3)

Single-cycle
T_c = 800 ps



Pipelined
T_c = 200 ps



Pipelining and ISA Design

- MIPS Instruction Set designed for pipelining!

Pipelining and ISA Design

- MIPS Instruction Set designed for pipelining!
- All instructions are 32-bits
  - Easier to fetch and decode in one cycle

Pipelining and ISA Design

- MIPS Instruction Set designed for pipelining!
- All instructions are 32-bits
  - Easier to fetch and decode in one cycle
- Few and regular instruction formats, 2 source register fields always in same place
  - Can decode and read registers in one step

Pipelining and ISA Design

- MIPS Instruction Set designed for pipelining!
- All instructions are 32-bits
  - Easier to fetch and decode in one cycle
- Few and regular instruction formats, 2 source register fields always in same place
  - Can decode and read registers in one step
- Memory operands only in Loads and Stores
  - Can calculate address 3rd stage, access memory 4th stage

Pipelining and ISA Design

- MIPS Instruction Set designed for pipelining!
- All instructions are 32-bits
  - Easier to fetch and decode in one cycle
- Few and regular instruction formats, 2 source register fields always in same place
  - Can decode and read registers in one step
- Memory operands only in Loads and Stores
  - Can calculate address 3rd stage, access memory 4th stage
- Alignment of memory operands
  - Memory access takes only one cycle

Question: Assume the stage times shown below. Suppose we remove loads and stores from our ISA. Consider going from a single-cycle implementation to a 4-stage pipelined version.

Instr Fetch	Reg Read	ALU Op	Mem Access	Reg Write
200ps	100 ps	200ps	200ps	100 ps

1) The *latency* will be 1.25x slower.
2) The *throughput* will be 3x faster.

	1	**2**
(B)	F	F
(G)	F	T
(P)	T	F
(Y)	T	T

Question: Assume the stage times shown below. Suppose we remove loads and stores from our ISA. Consider going from a single-cycle implementation to a 4-stage pipelined version.

Instr Fetch	Reg Read	ALU Op	Mem Access	Reg Write
200ps	100 ps	200ps	200ps	100 ps

1) The *latency* will be 1.25x slower.
2) The *throughput* will be 3x faster.

	1	2
(B)	F	F
[✓ (G)]	F	T
(P)	T	F
(Y)	T	T

Summary

- Implementing controller for your datapath
  - Take decoded signals from instruction and generate control signals
  - Use “AND” and “OR” Logic scheme
- Pipelining improves performance by exploiting Instruction Level Parallelism
  - 5-stage pipeline for MIPS: IF, ID, EX, MEM, WB
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
  - Be careful of signal passing (*more on this next lecture*)