EECS150 - Digital Design

Lecture 10- CPU Microarchitecture

Feb 18, 2010

John Wawrzynek

Processor Microarchitecture Introduction

Microarchitecture: how to implement an architecture in hardware

Good examples of how to put principles of digital design to practice.

Introduction to final project.

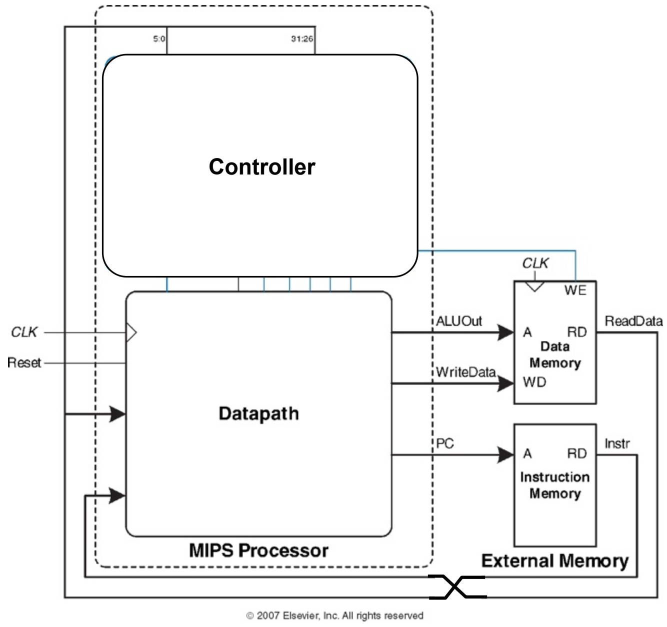
Application Software	programs
Operating Systems	device drivers
Architecture	instructions registers
Micro-architecture	datapaths controllers
Logic	adders memories
Digital Circuits	AND gates NOT gates
Analog Circuits	amplifiers filters
Devices	transistors diodes
Physics	electrons

MIPS Processor Architecture

- For now we consider a subset of MIPS instructions:
  - R-type instructions: `and, or, add, sub, slt`
  - Memory instructions: `lw, sw`
  - Branch instructions: `beq`
- Later we'll add `addi` and `j`

MIPS Micrarchitecture Oganization

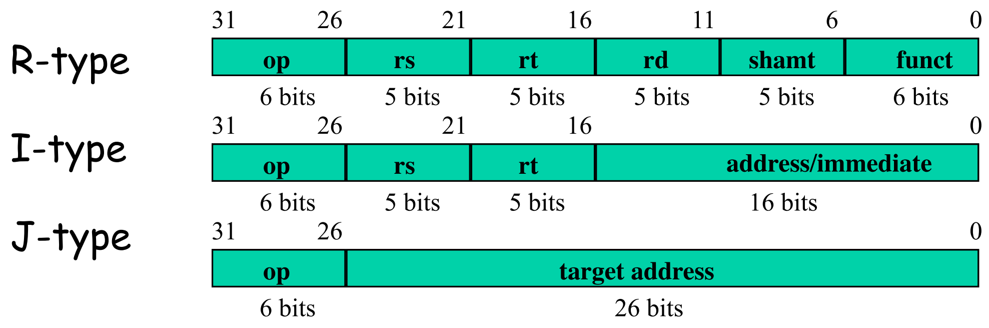
Datapath + Controller + External Memory



How to Design a Processor: step-by-step

1. Analyze instruction set architecture (ISA) ⟹ datapath requirements
   - meaning of each instruction is given by the ***data transfers (register transfers)***
   - datapath must include storage element for ISA registers
   - datapath must support each data transfer
2. Select set of datapath components and establish clocking methodology
3. Assemble datapath meeting requirements
4. Analyze implementation of each instruction to determine setting of control points that effects the data transfer.
5. Assemble the control logic.

Review: The MIPS Instruction

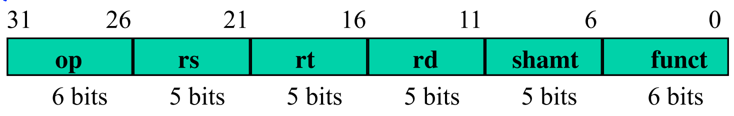


The different fields are:
op: operation ("opcode") of the instruction
rs, rt, rd: the source and destination register specifiers
shamt: shift amount
funct: selects the variant of the operation in the "op" field
address / immediate: address offset or immediate value
target address: target address of jump instruction

Subset for Lecture

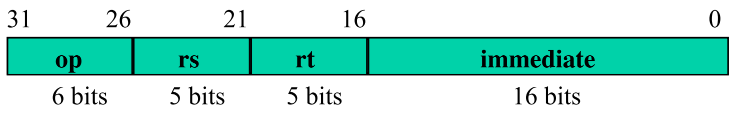
add, sub, or, slt

- `addu rd,rs,rt`
- `subu rd,rs,rt`



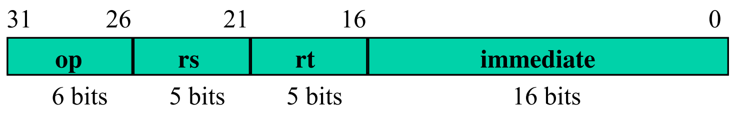
lw, sw

- `lw rt,rs,imm16`
- `sw rt,rs,imm16`



beq

- `beq rs,rt,imm16`



Register Transfer Descriptions

All start with instruction fetch:

{op , rs , rt , rd , shamt , funct} ← IMEM[PC] OR

{op , rs , rt , Imm16} ← IMEM[PC] THEN

inst	Register Transfers	
add	R[rd] ← R[rs] + R[rt];	PC ← PC + 4
sub	R[rd] ← R[rs] – R[rt];	PC ← PC + 4
or	R[rd] ← R[rs] | R[rt];	PC ← PC + 4
slt	R[rd] ← (R[rs] < R[rt]) ? 1 : 0;	PC ← PC + 4
lw	R[rt] ← DMEM[R[rs] + sign_ext(Imm16)];	PC ← PC + 4
sw	DMEM[R[rs] + sign_ext(Imm16)] ← R[rt];	PC ← PC + 4
beq	if (R[rs] == R[rt]) then PC ← PC + 4 + {sign_ext(Imm16), 00} else PC ← PC + 4	

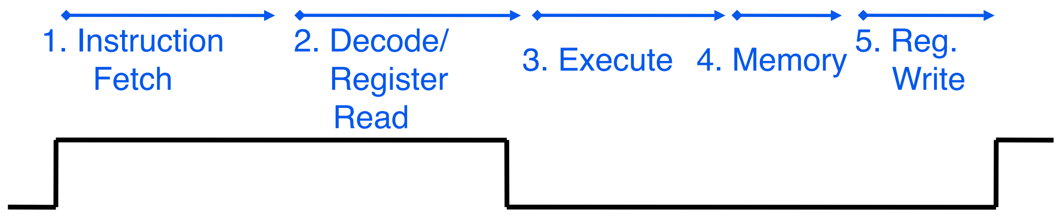
Microarchitecture

Multiple implementations for a single architecture:

- Single-cycle
  - Each instruction executes in a single clock cycle.
- Multicycle
  - Each instruction is broken up into a series of shorter steps with one step per clock cycle.
- Pipelined
  - Each instruction is broken up into a series of steps with one step per clock cycle
  - Multiple instructions execute at once.

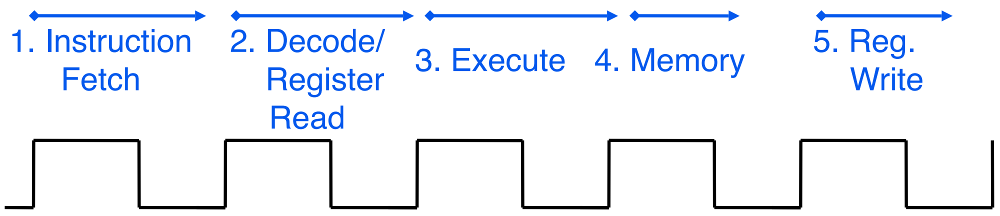
CPU clocking (1/2)

- Single Cycle CPU: All stages of an instruction are completed within one ***long*** clock cycle.
  - The clock cycle is made sufficient long to allow each instruction to complete all stages without interruption and within one cycle.



CPU clocking (2/2)

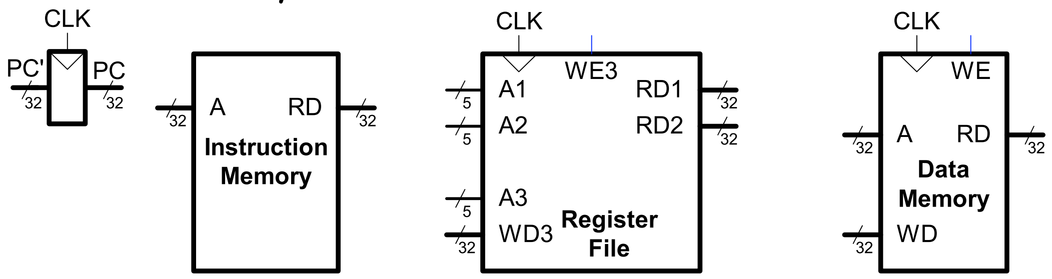
- Multiple-cycle CPU: Only one stage of instruction per clock cycle.
  - The clock is made as long as the slowest stage.



Several significant advantages over single cycle execution: Unused stages in a particular instruction can be skipped OR instructions can be pipelined (overlapped).

MIPS State Elements

- Determines everything about the execution status of a processor:
  - PC register
  - 32 registers
  - Memory



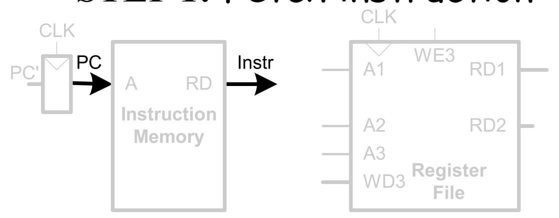
Note: for these state elements, clock is used for write but not for read (asynchronous read, synchronous write).

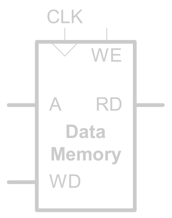
Single-Cycle Datapath: `lw` fetch

- First consider executing `lw`

R[rt] ← DMEM[R[rs] + sign_ext(Imm16)]

- **STEP 1:** Fetch instruction

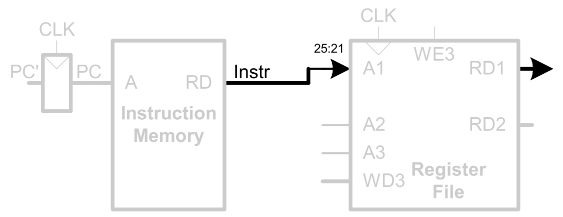


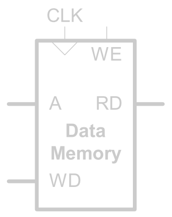


Single-Cycle Datapath: `lw` register read

R[rt] ← DMEM[R[rs] + sign_ext(Imm16)]

- **STEP 2:** Read source operands from register file

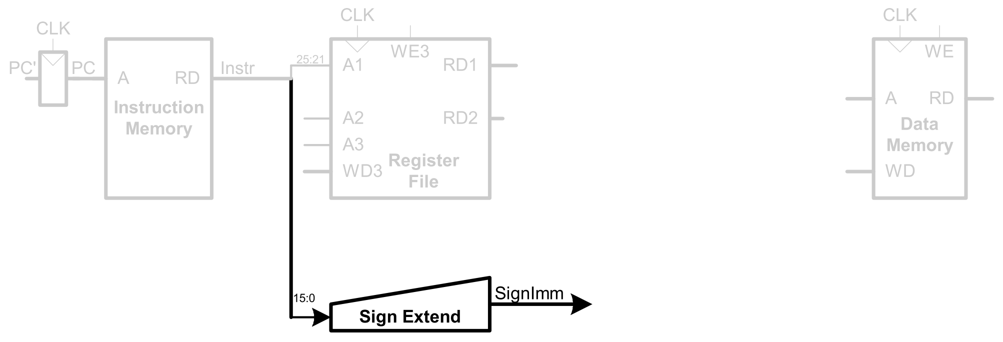




Single-Cycle Datapath: `lw` immediate

R[rt] ← DMEM[R[rs] + sign_ext(Imm16)]

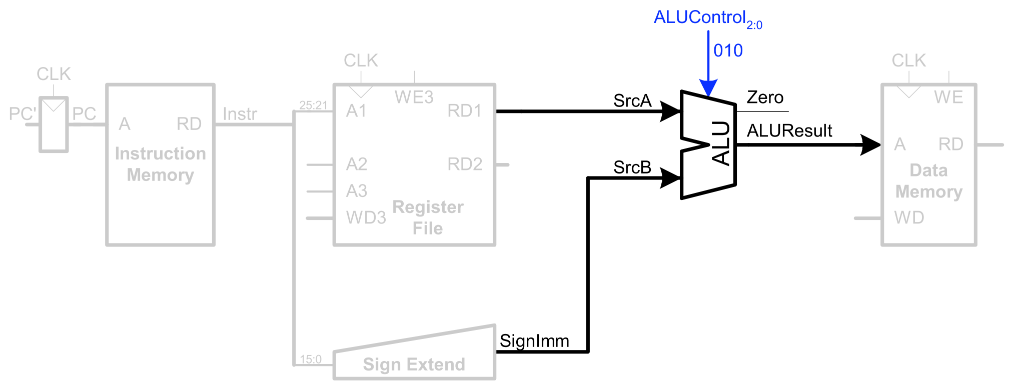
- **STEP 3:** Sign-extend the immediate



Single-Cycle Datapath: `lw` address

R[rt] ← DMEM[R[rs] + sign_ext(Imm16)]

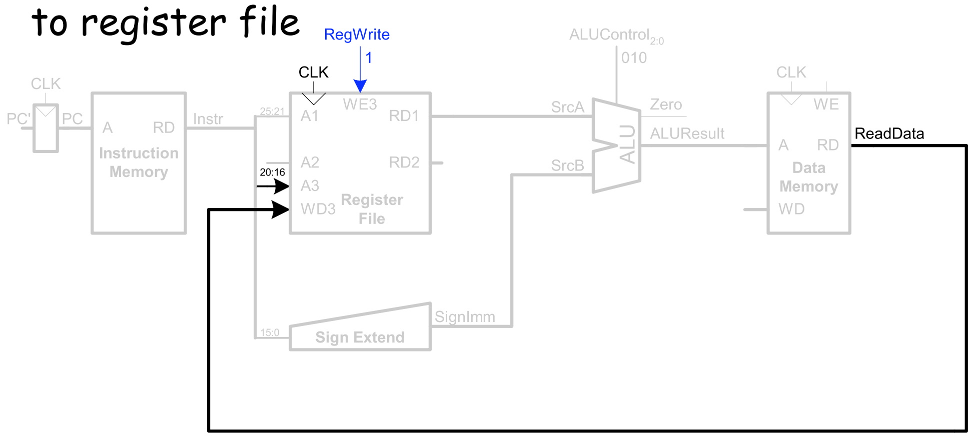
- **STEP 4:** Compute the memory address



Single-Cycle Datapath: `lw` memory read

R[rt] ← DMEM[R[rs] + sign_ext(Imm16)]

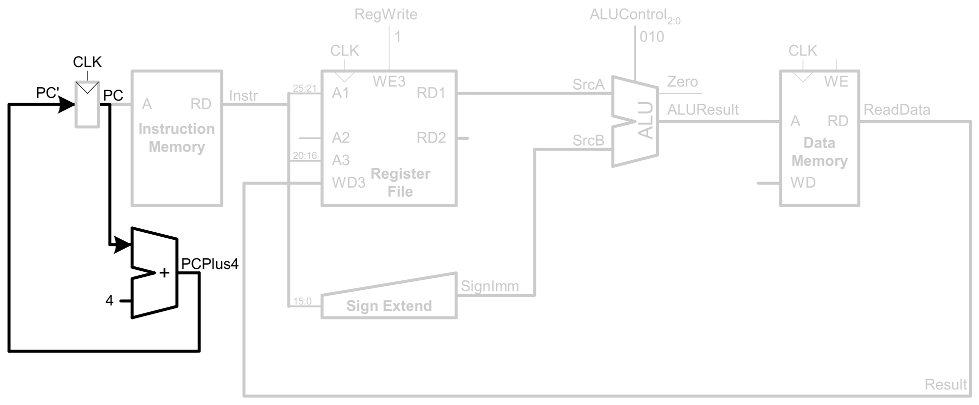
- **STEP 5:** Read data from memory and write it back to register file



Single-Cycle Datapath: `lw` PC increment

- **STEP 6:** Determine the address of the next instruction

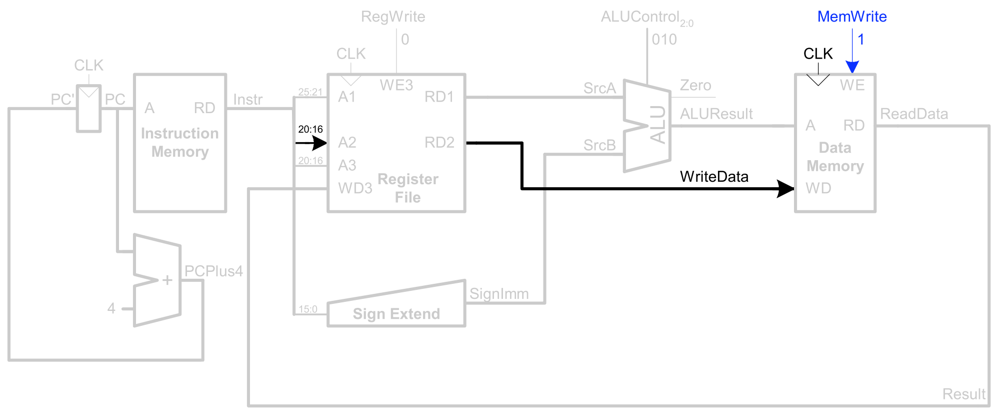
PC ← PC + 4



Single-Cycle Datapath: `sw`

DMEM[R[rs] + sign_ext(Imm16)] ← R[rt]

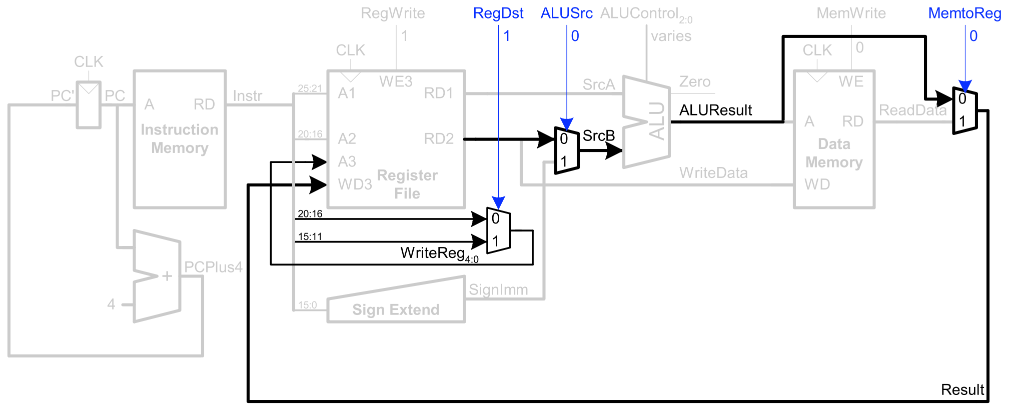
- Write data in `rt` to memory



Single-Cycle Datapath: R-type instructions

- Read from `rs` and `rt`
- Write *ALUResult* to register file
- Write to `rd` (instead of `rt`)

R[rd] ← R[rs] op R[rt]

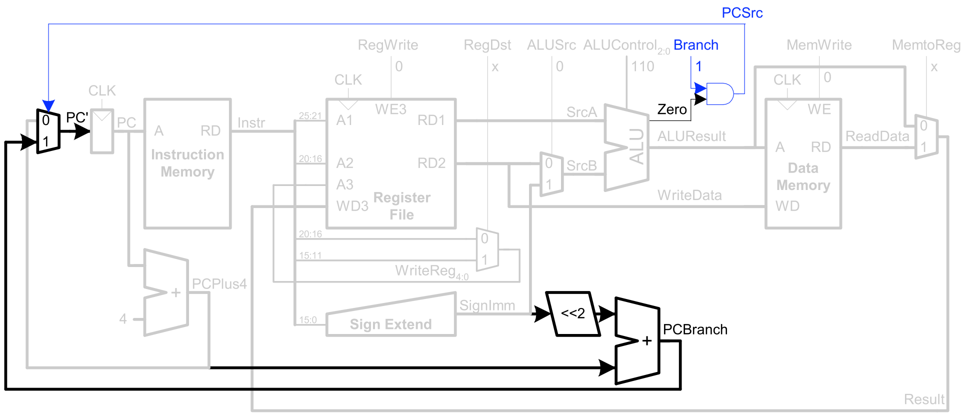


Single-Cycle Datapath: beq

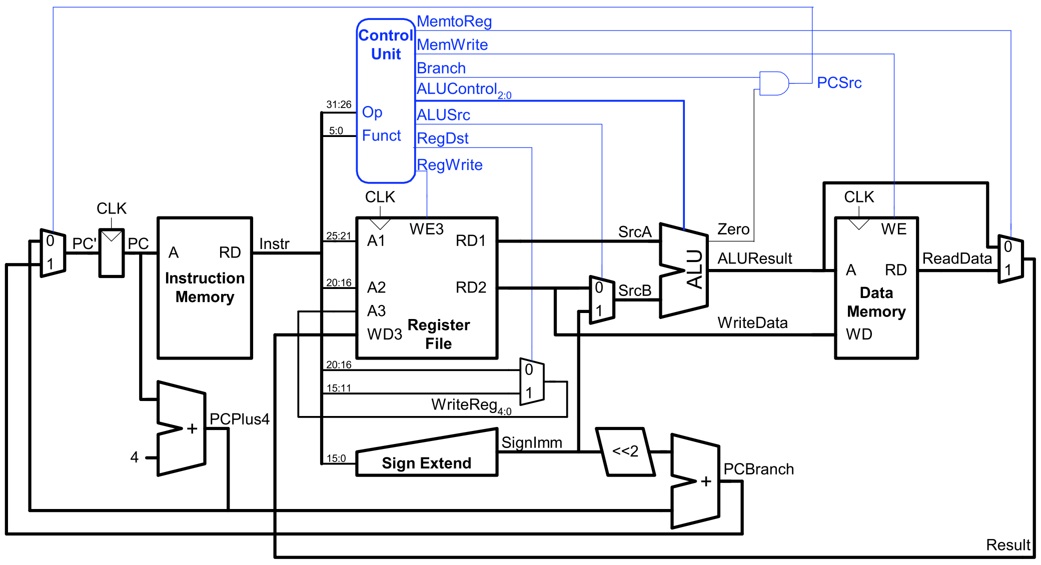
if (R[rs] == R[rt]) then PC ← PC + 4 + {sign_ext(Imm16), 00}

- Determine whether values in `rs` and `rt` are equal
- Calculate branch target address:

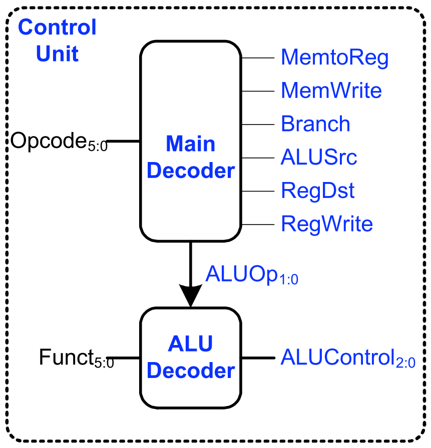
  BTA = (sign-extended immediate << 2) + (PC+4)



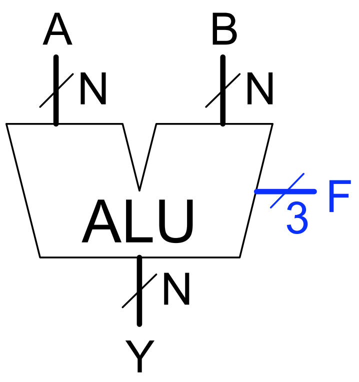
Complete Single-Cycle Processor



Control Unit



Review: ALU



$F_{2:0}$	Function
0	A & B
1	A | B
10	A + B
11	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

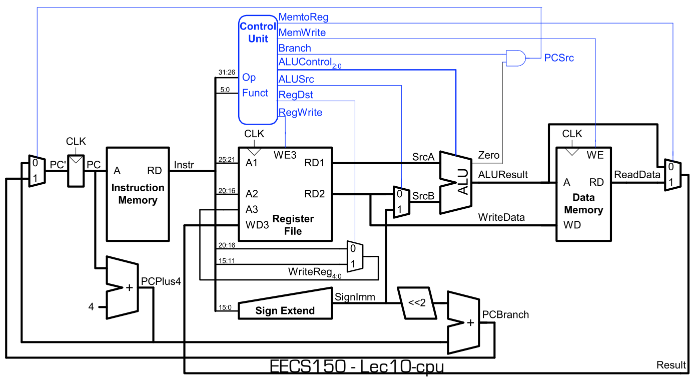
Control Unit: ALU Decoder

$ALUOp_{1:0}$	Meaning
0	Add
1	Subtract
10	Look at Funct
11	Not Used

$ALUOp_{1:0}$	Funct	$ALUControl_{2:0}$
0	X	010 (Add)
X1	X	110 (Subtract)
1X	100000 (add)	010 (Add)
1X	100010 (sub)	110 (Subtract)
1X	100100 (and)	000 (And)
1X	100101 (or)	001 (Or)
1X	101010 (slt)	111 (SLT)

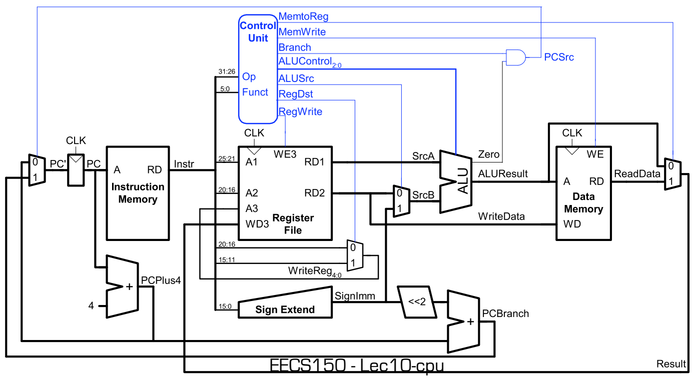
Control Unit: Main Decoder

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$
R-type	0							
lw	1E+05							
sw	1E+05							
beq	100							

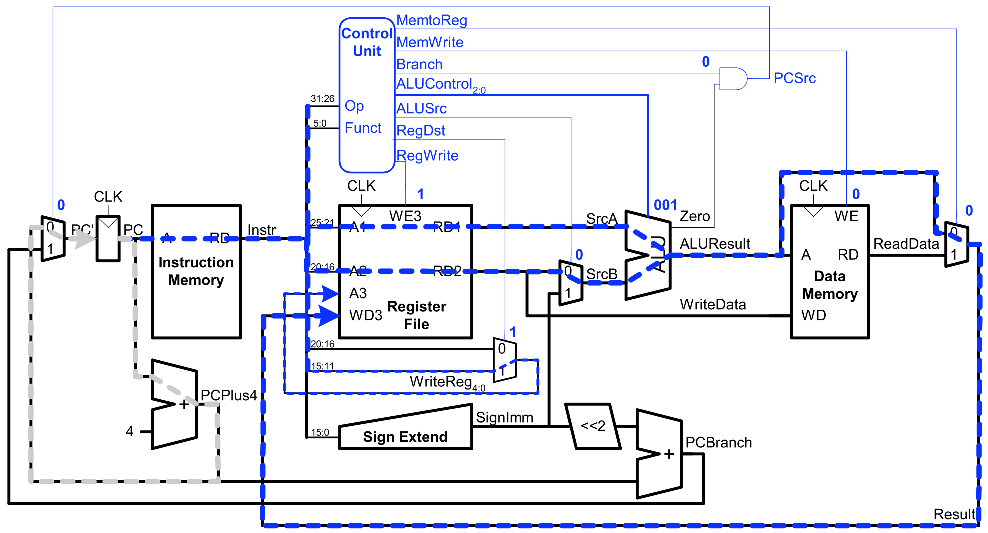


Control Unit: Main Decoder

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$
R-type	0	1	1	0	0	0	0	10
`lw`	1E+05	1	0	1	0	0	0	0
`sw`	1E+05	0	X	1	0	1	X	0
`beq`	100	0	X	0	1	0	X	1

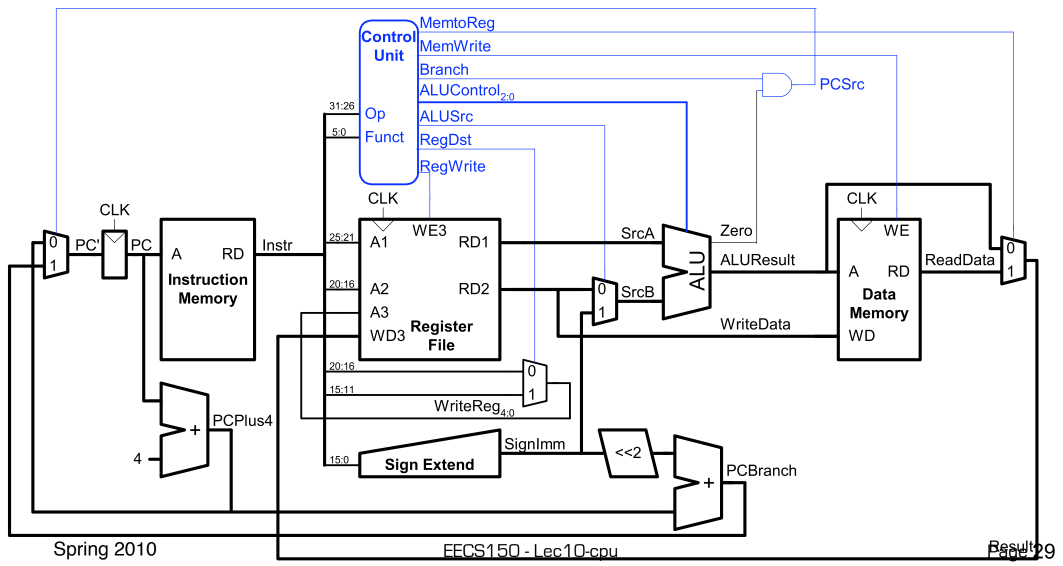


Single-Cycle Datapath Example: `or`



Extended Functionality: addi

- No change to datapath



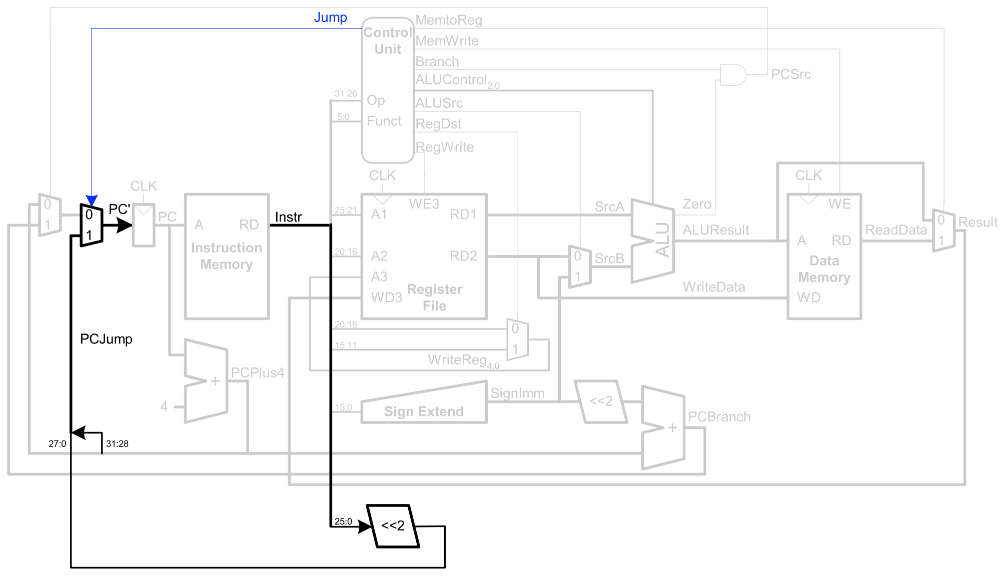
Control Unit: addi

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$
R-type	0	1	1	0	0	0	0	10
lw	1E+05	1	0	1	0	0	1	0
sw	1E+05	0	X	1	0	1	X	0
beq	100	0	X	0	1	0	X	1
addi	1000							

Control Unit: `addi`

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$
R-type	0	1	1	0	0	0	0	10
`lw`	1E+05	1	0	1	0	0	1	0
`sw`	1E+05	0	X	1	0	1	X	0
`beq`	100	0	X	0	1	0	X	1
`addi`	1000	1	0	1	0	0	0	0

Extended Functionality: `j`



Control Unit: Main Decoder

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$	Jump
R-type	0	1	1	0	0	0	0	10	0
`lw`	1E+05	1	0	1	0	0	1	0	0
`sw`	1E+05	0	X	1	0	1	X	0	0
`beq`	100	0	X	0	1	0	X	1	0
`j`	100								

Control Unit: Main Decoder

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$	Jump
R-type	0	1	1	0	0	0	0	10	0
`lw`	1E+05	1	0	1	0	0	1	0	0
`sw`	1E+05	0	X	1	0	1	X	0	0
`beq`	100	0	X	0	1	0	X	1	0
`j`	100	0	X	X	X	0	X	XX	1

Review: Processor Performance

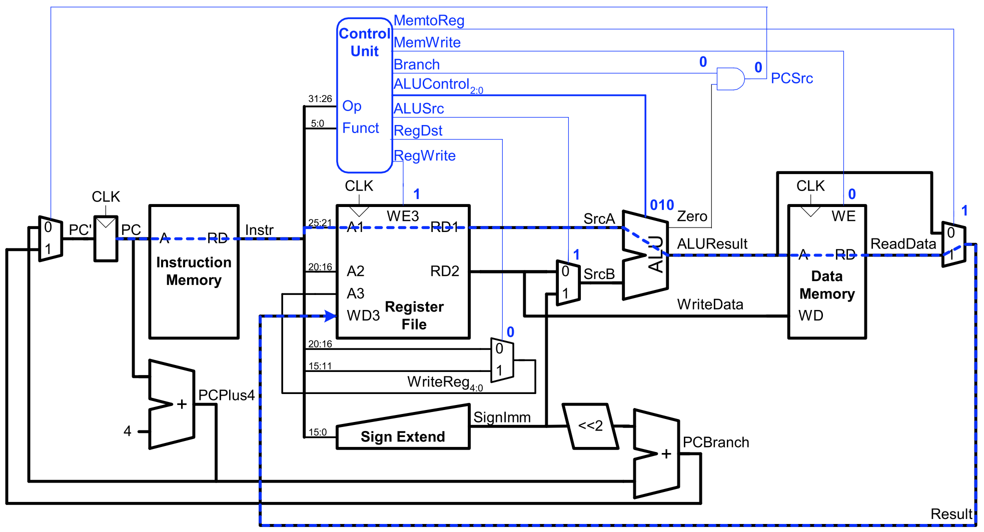
Program Execution Time

= (# instructions)(cycles/instruction)(seconds/cycle)

= # instructions x CPI x T_C

Single-Cycle Performance

- T_C is limited by the critical path (`lw`)



Single-Cycle Performance

- Single-cycle critical path:
  $T_c = t_{pcq\_PC} + t_{\text{mem}} + \max(t_{RF\text{read}}, t_{sext} + t_{\text{mux}}) + t_{\text{ALU}} + t_{\text{mem}} + t_{\text{mux}} + t_{RF\text{setup}}$
- In most implementations, limiting paths are:
  - memory, ALU, register file.
  - $T_c = t_{pcq\_PC} + 2t_{\text{mem}} + t_{RF\text{read}} + t_{\text{mux}} + t_{\text{ALU}} + t_{RF\text{setup}}$

Single-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	$t_{RF\text{read}}$	150
Register file setup	$t_{RF\text{setup}}$	20

$T_c =$

Single-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	$t_{RF\text{read}}$	150
Register file setup	$t_{RF\text{setup}}$	20

$$\begin{aligned} T_c &= t_{pcq\_PC} + 2t_{\text{mem}} + t_{RF\text{read}} + t_{\text{mux}} + t_{\text{ALU}} + t_{RF\text{setup}} \\ &= [30 + 2(250) + 150 + 25 + 200 + 20] \text{ ps} \\ &= 925 \text{ ps} \end{aligned}$$

Single-Cycle Performance Example

- For a program with 100 billion instructions executing on a single-cycle MIPS processor,

Execution Time =

Single-Cycle Performance Example

- For a program with 100 billion instructions executing on a single-cycle MIPS processor,

Execution Time = # instructions x CPI x T_C

$= (100 \times 10^9)(1)(925 \times 10^{-12}\ \mathrm{s})$

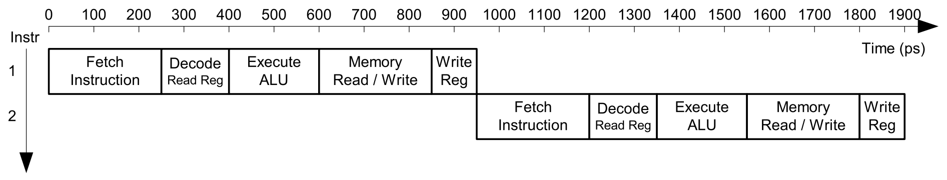
$= 92.5$ seconds

Pipelined MIPS Processor

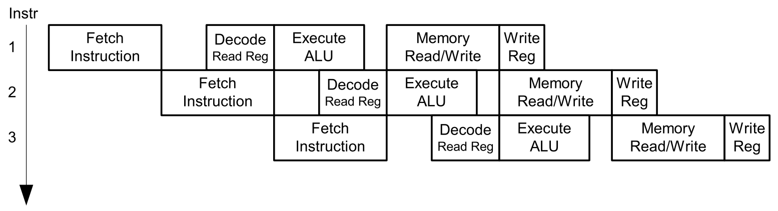
- Temporal parallelism
- Divide single-cycle processor into 5 stages:
  - Fetch
  - Decode
  - Execute
  - Memory
  - Writeback
- Add pipeline registers between stages

Single-Cycle vs. Pipelined Performance

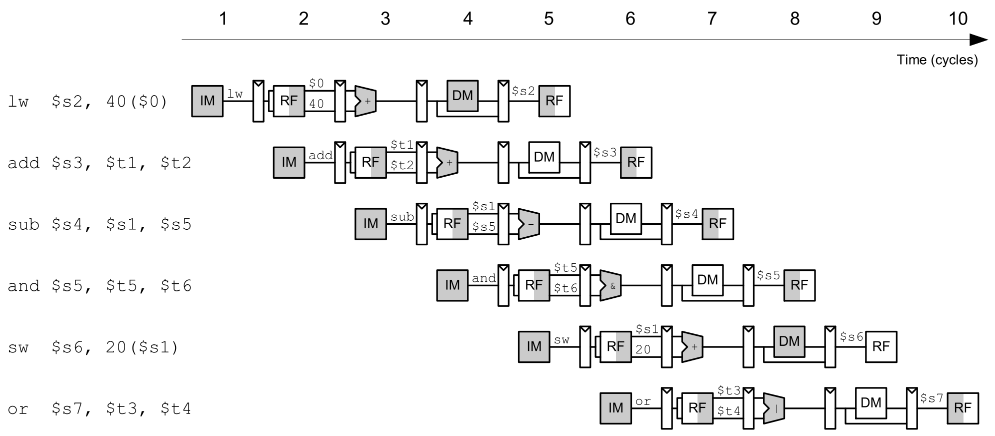
Single-Cycle



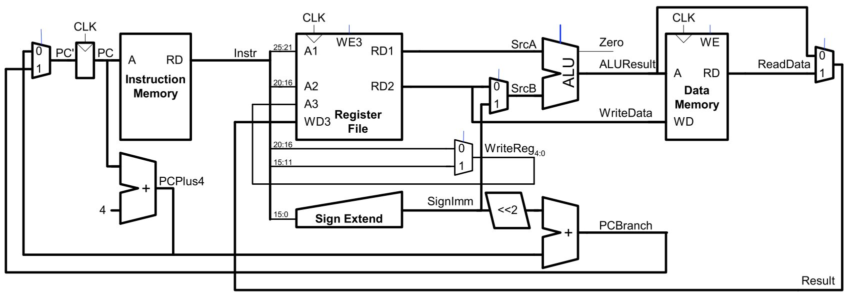
Pipelined

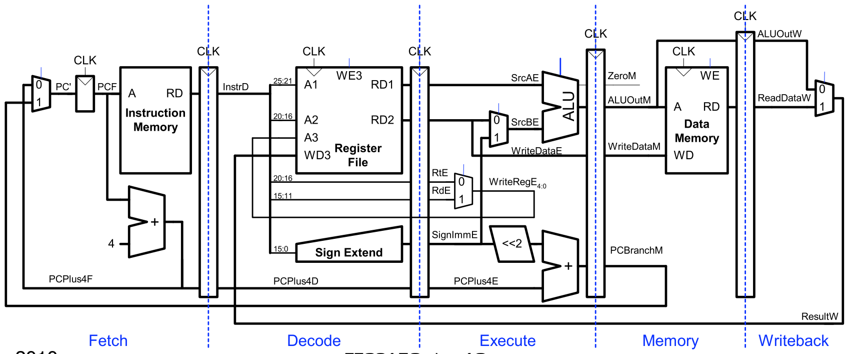


Pipelining Abstraction



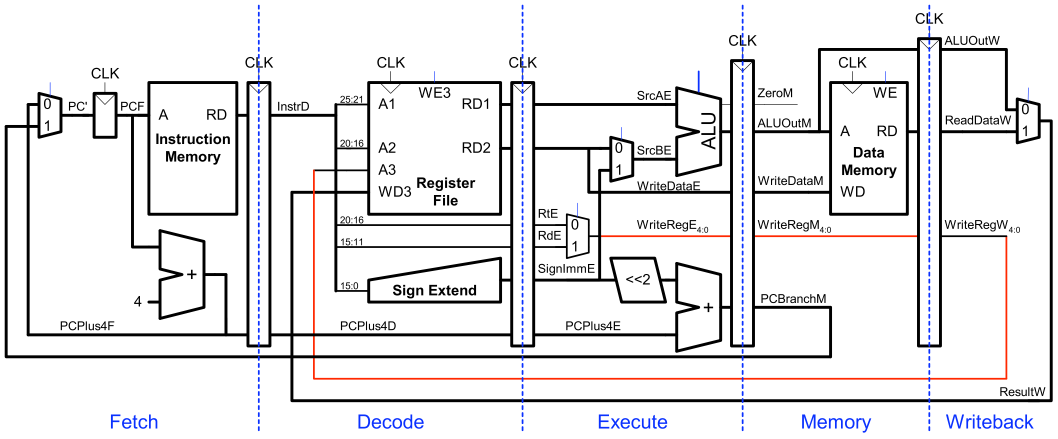
Single-Cycle and Pipelined Datapath



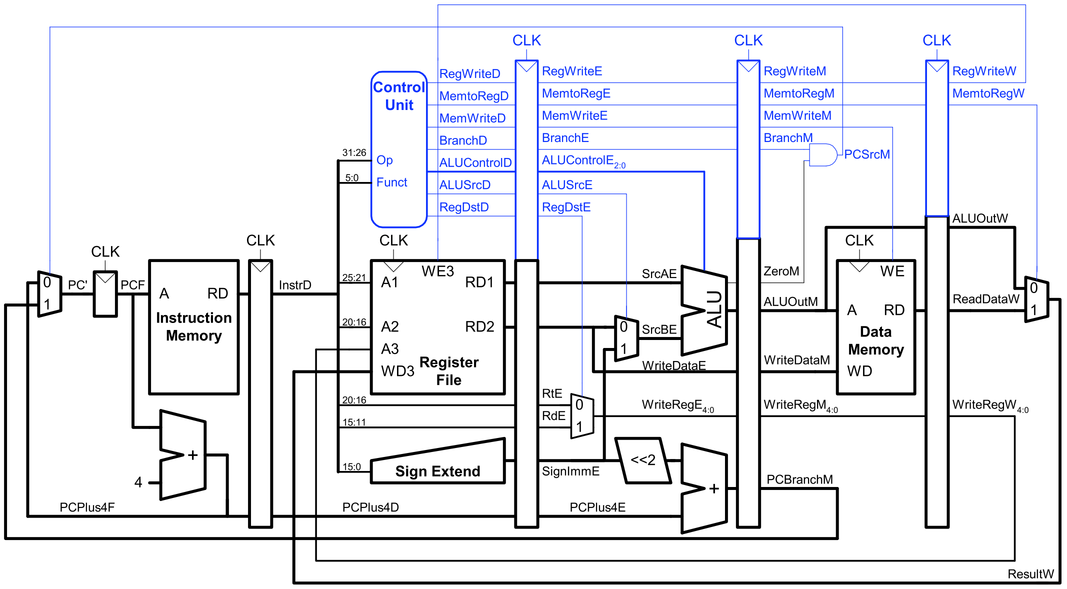


Corrected Pipelined Datapath

- WriteReg must arrive at the same time as Result



Pipelined Control



Same control unit as single-cycle processor

Control delayed to proper pipeline stage

Pipeline Hazards

- Occurs when an instruction depends on results from previous instruction that hasn't completed.
- Types of hazards:
  - **Data hazard:** register value not written back to register file yet
  - **Control hazard:** next instruction not decided yet (caused by branches)