# Computer Hardware Engineering (IS1200)
# Computer Organization and Components (IS1500)

Spring 2019

Lecture 9: ALU and Single-Cycle Processors

Elias Castegren

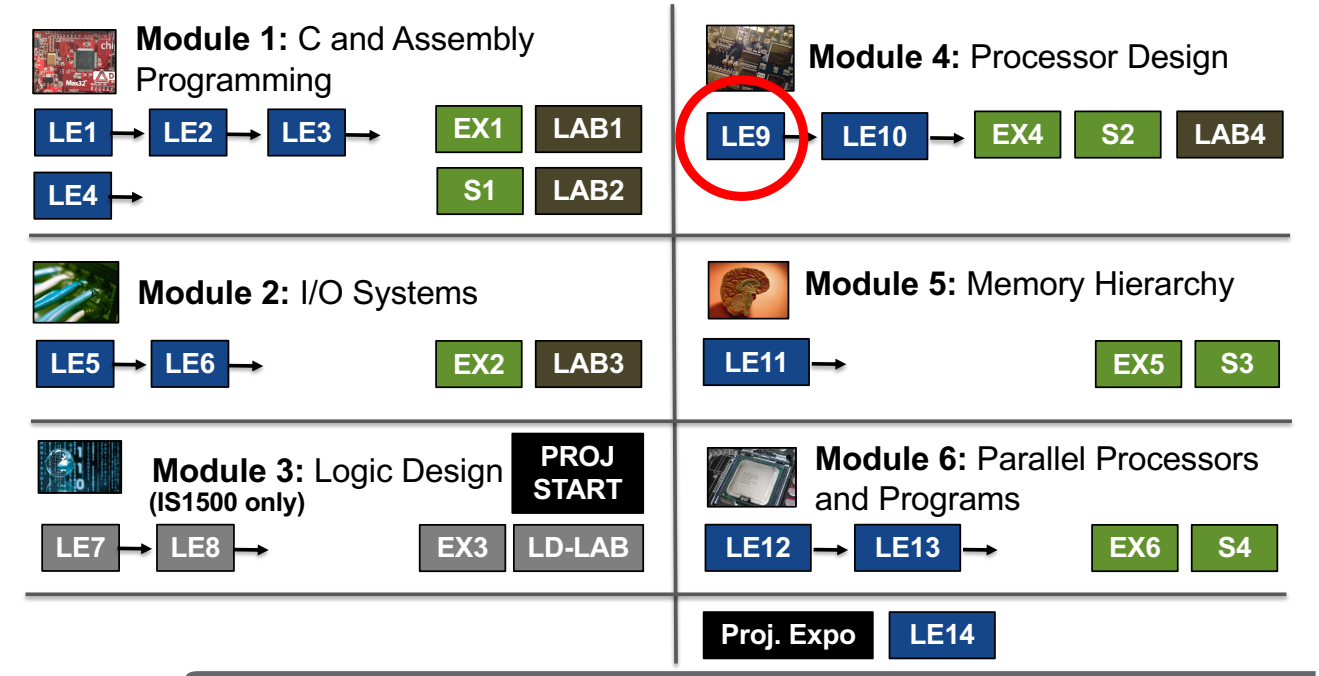
Postdoc, KTH Royal Institute of Technology

Slides by David Broman

Slides version 1.1



## Course Structure



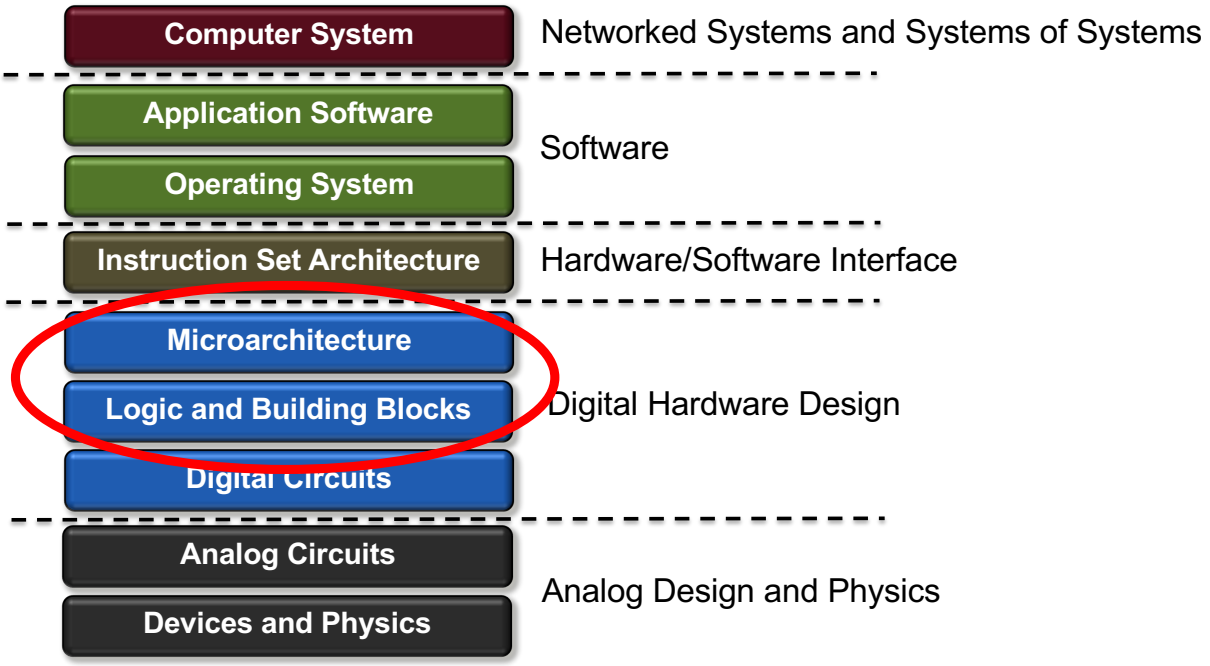

David Broman
dbro@kth.se

**Part I**
Arithmetic Logic Unit

**Part II**
Data Path in a Single-Cycle Processor

**Part III**
Control Unit in a Single-Cycle Processor

# Abstractions in Computer Systems







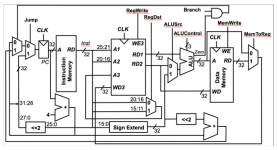

# Agenda

**Part I**

**Arithmetic Logic Unit**

**Part II**

**Data Path in a Single-Cycle Processor**



**Part III**

**Control Unit in a Single-Cycle Processor**

# Part I

# Arithmetic Logic Unit

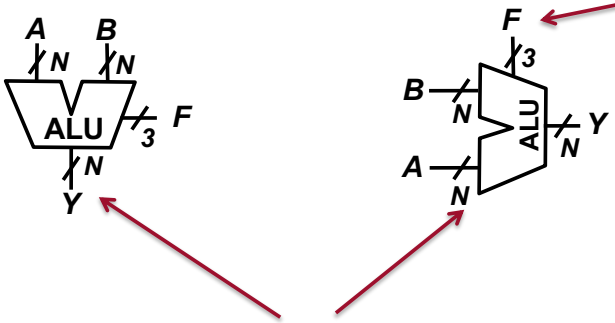
Acknowledgement: The structure and several of the good examples are derived from the book "Digital Design and Computer Architecture" (2013) by D. M. Harris and S. L. Harris.





## Arithmetic Logic Unit (ALU)

An **ALU** saves hardware by combining different arithmetic and logic operations in one single unit/element.




Input ***F*** specifies the function that the ALU should perform

ALUs can have different functions and be designed differently.

ALU symbol: both figures have the same function

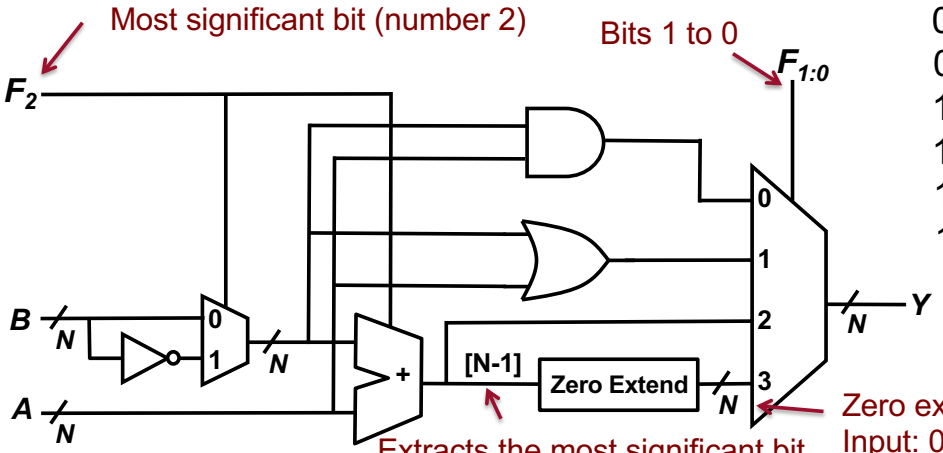
An ALU can also include **output flags**, for instance:

- **Overflow flag** (adder overflowed)
- **Zero flag** (output is zero)

# Arithmetic Logic Unit (ALU)

**Exercise:**
Determine the functional behavior for each value of **F**.

| $F_{2:0}$ | Function |
|---|---|
| 000 | A AND B |
| 001 | A OR B |
| 010 | A + B |
| 011 | not used |
| 100 | A AND !B |
| 101 | A OR !B |
| 110 | A – B |
| 111 | SLT |




Extracts the most significant bit.
Adds zeros for bits N-1 to 1.

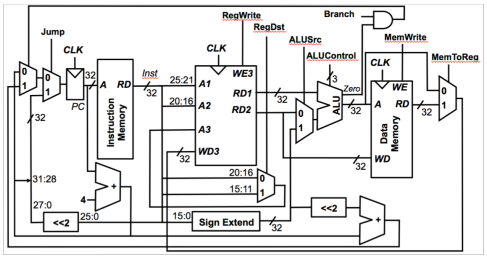
Zero extend examples (N=32):
Input: 0xffff1234. Output: 0x1
Input: 0x00ff676a. Output: 0x0





# Part II

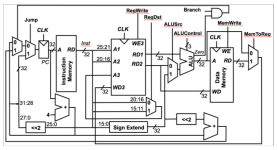
# Data Path in a Single-Cycle Processor




Acknowledgement: The structure and several of the good examples are derived from the book "Digital Design and Computer Architecture" (2013) by D. M. Harris and S. L. Harris.

# Data Path and Control Unit

A processor is typically divided into two parts



**Data Path**

- Operates on a word of data.
- Consists of elements such as registers, memory, ALUs etc.

**Control Unit**

- Gets the current instruction from the data path and tells the data path how to execute the instruction.





# Instructions

In this lecture, we construct a microarchitecture for a subset of a MIPS processor with the following instructions

**R-Type:** `add`, `sub`, `and`, `or`, `slt`

Arithmetic / logic instructions

Memory instructions

**I-Type:** `addi`, `lw`, `sw`, `beq`

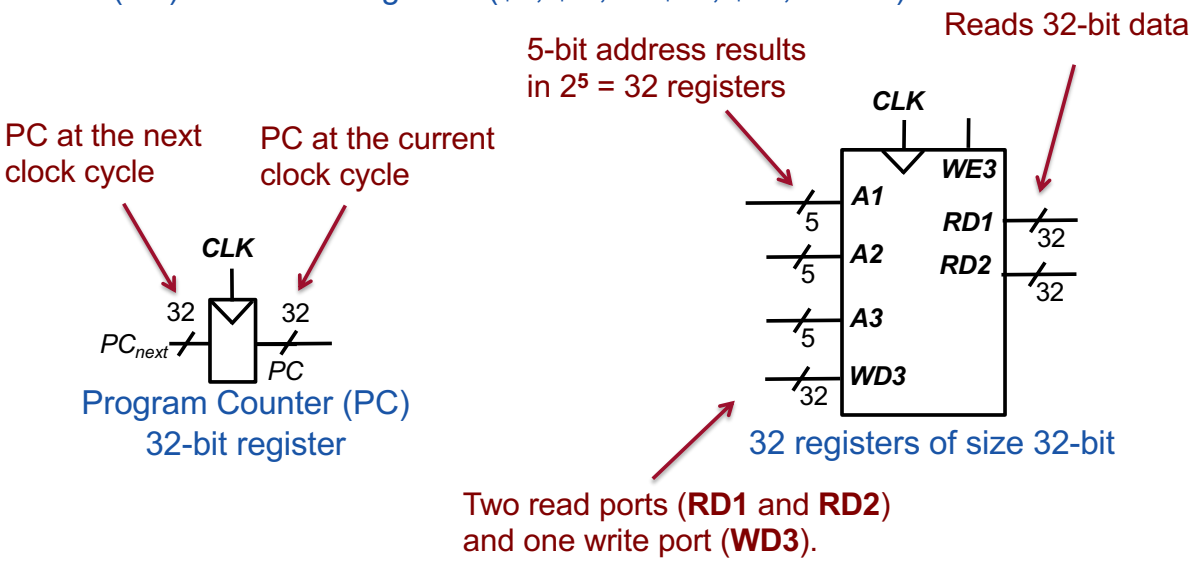
Arithmetic immediate instruction

Branch instructions

**J-Type:** `j`

# State Elements (1/3)
# Program Counter and Register File

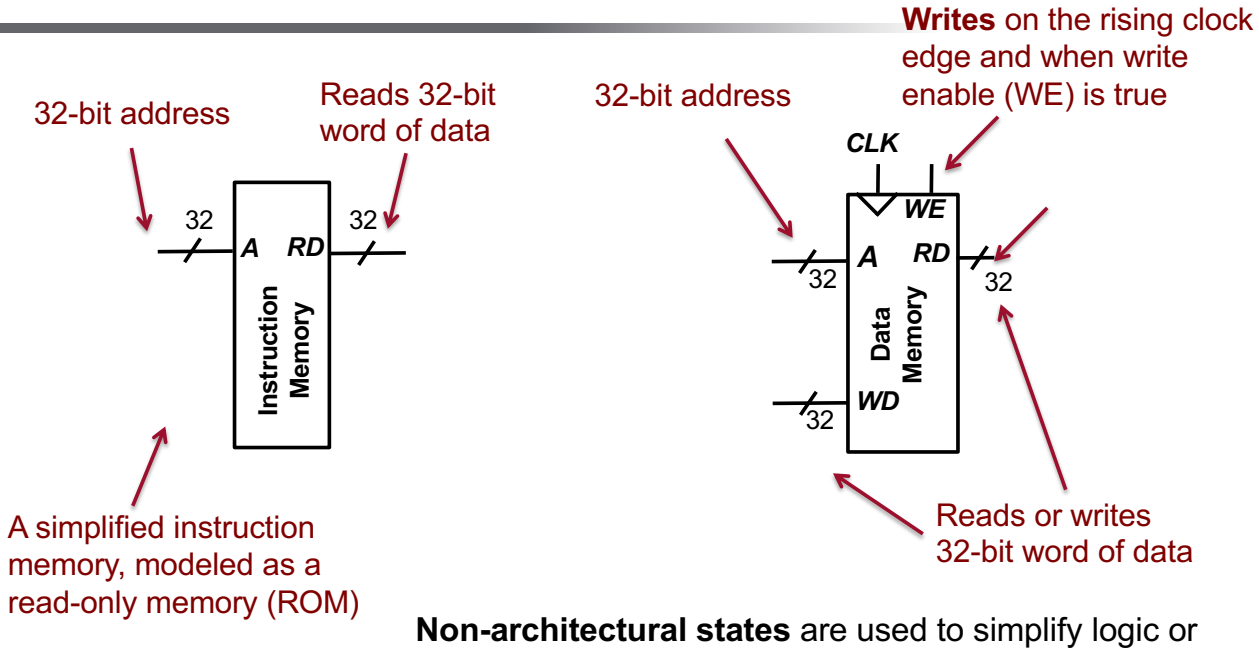
The **architectural states** for this MIPS processor are the program counter (PC) and the 32 registers ($0, $t0, … $s0, $s1, … etc.)








# State Elements (2/3)
# Instructions and Data Memories

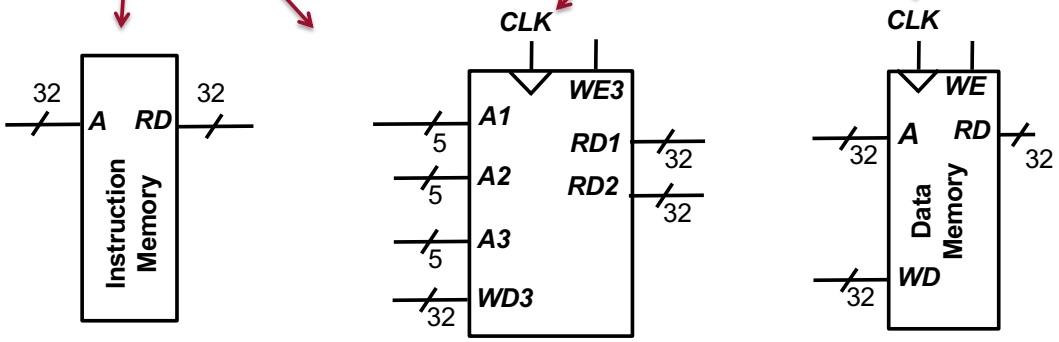



**Non-architectural states** are used to simplify logic or improve performance (introduced in the next lecture).

# State Elements (3/3)
# Reading combinationally, writing at clock edge

All the blocks below **read** *combinationally*: when the address changes, the data on the read port change after some propagation time.
There is no clock involved.

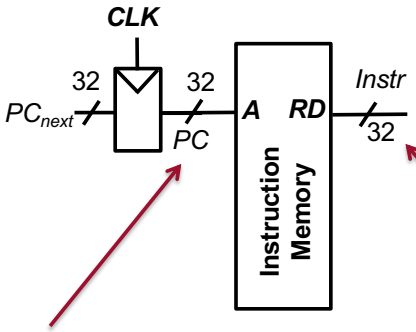
The register file and the data memory **write** at the raising clock edge.







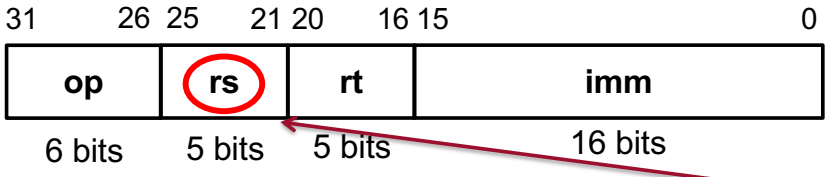

# Read Instruction from the Current PC




First step. Read the instruction at the current PC address.

A 32-bit instruction *Inst* is **fetched**.

# `lw` instruction – Read Base Address



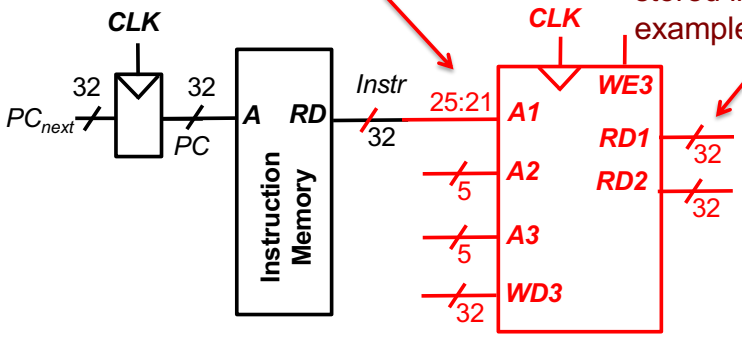

**Example**

```
lw   $s0,4($s1)
```

Base address in **rs**

Read out the base address from the register file. 25:21 cuts out the 5 bits from the instruction.

RD1 has now the address stored in $s1 (in the above example).

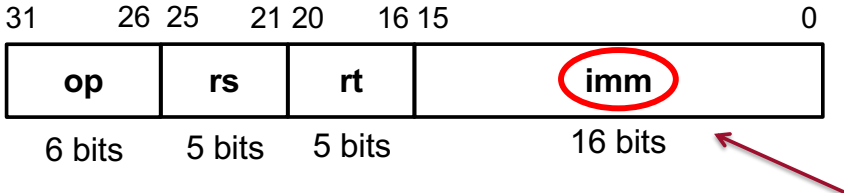



Loads the value from $s1, adds 4, and stores the result in $s0.





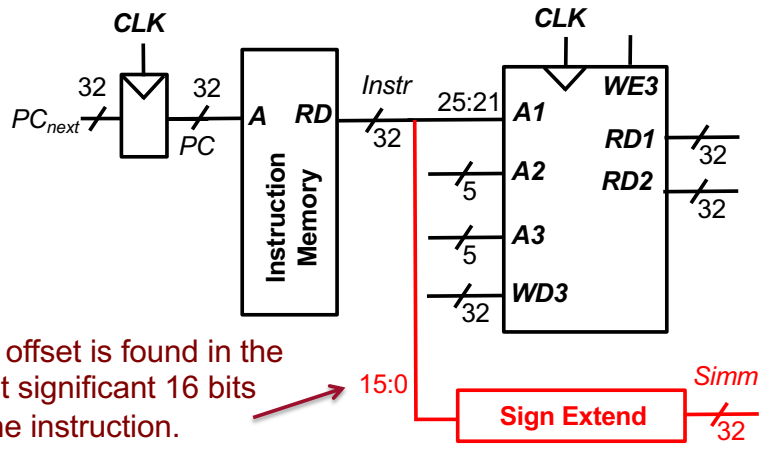
# `lw` instruction – Read Offset




**Example**

```
lw   $s0,4($s1)
```

The offset is stored in the imm field.




The offset is signed. Sign extend to 32 bits.

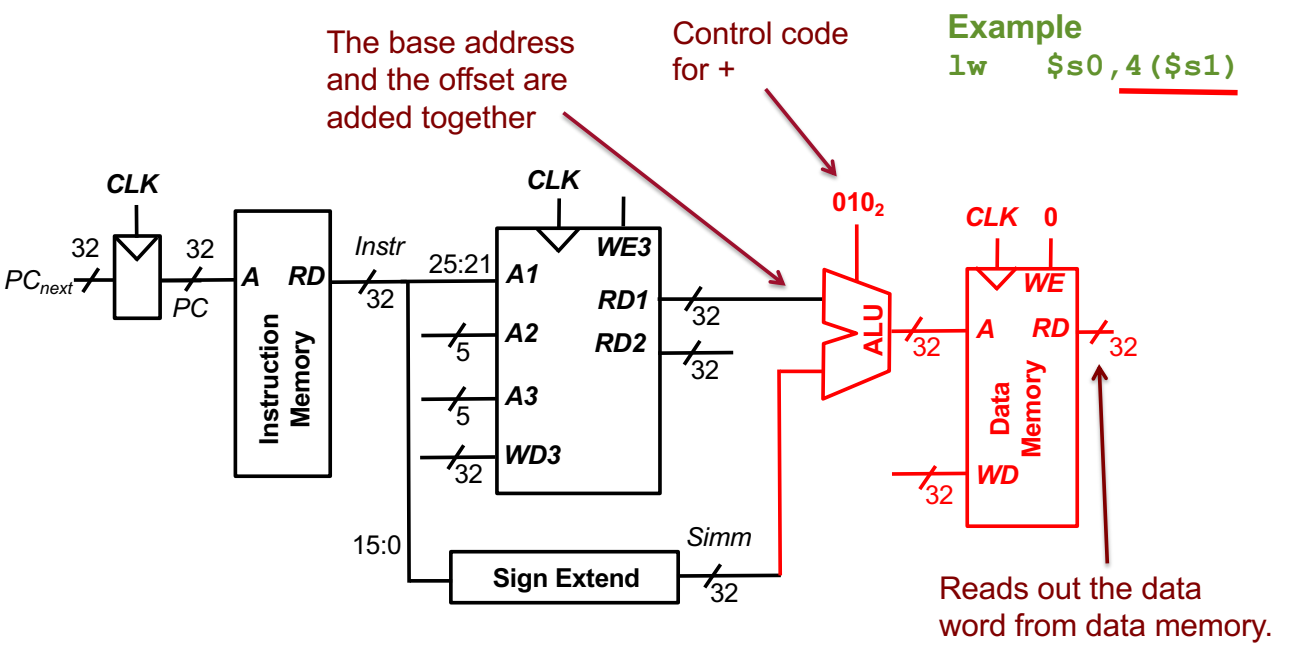
That is:

$Simm_{15:0} = Instr_{15:0}$

$Simm_{31:16} = Instr_{15}$

The offset is found in the least significant 16 bits of the instruction.

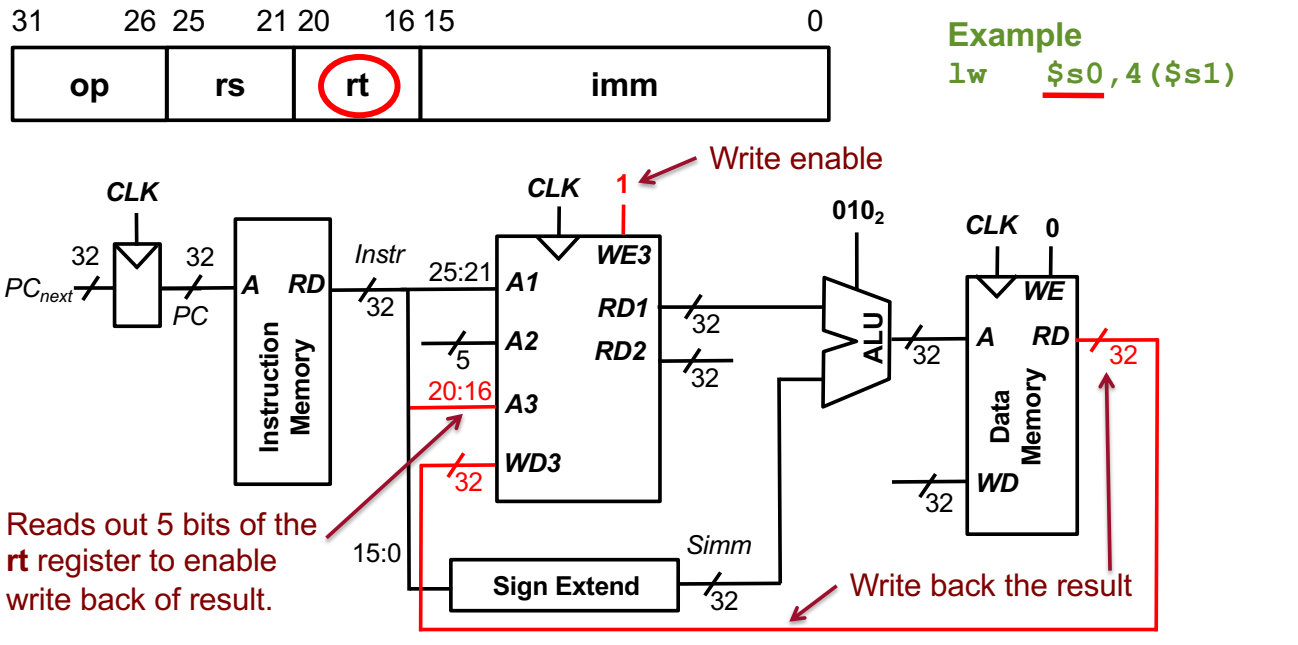
# `lw` instruction – Read Data Word



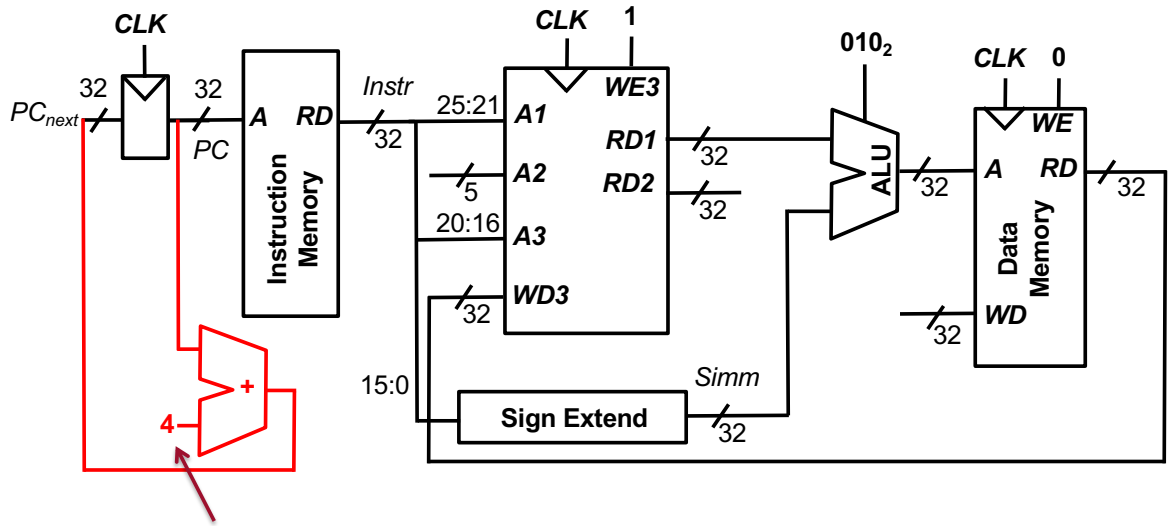





# `lw` instruction – Write Back



# `lw` instruction – Increment PC



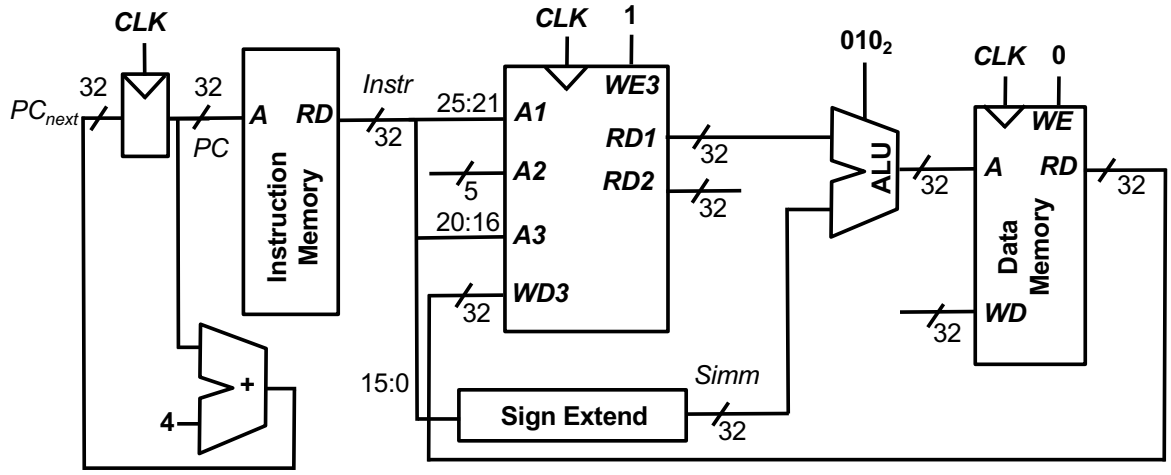

Increment the PC by 4.
(Next instruction is at address PC + 4)

This is the complete data path for the load word (`lw`) instruction.



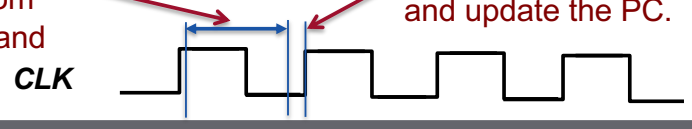


# `lw` instruction – Timing




Combinational logic during clock cycle: read instruction, sign extend, read from register file, perform ALU operation, and read from the data memory.

At the raising clock edge: Write to the register file and update the PC.

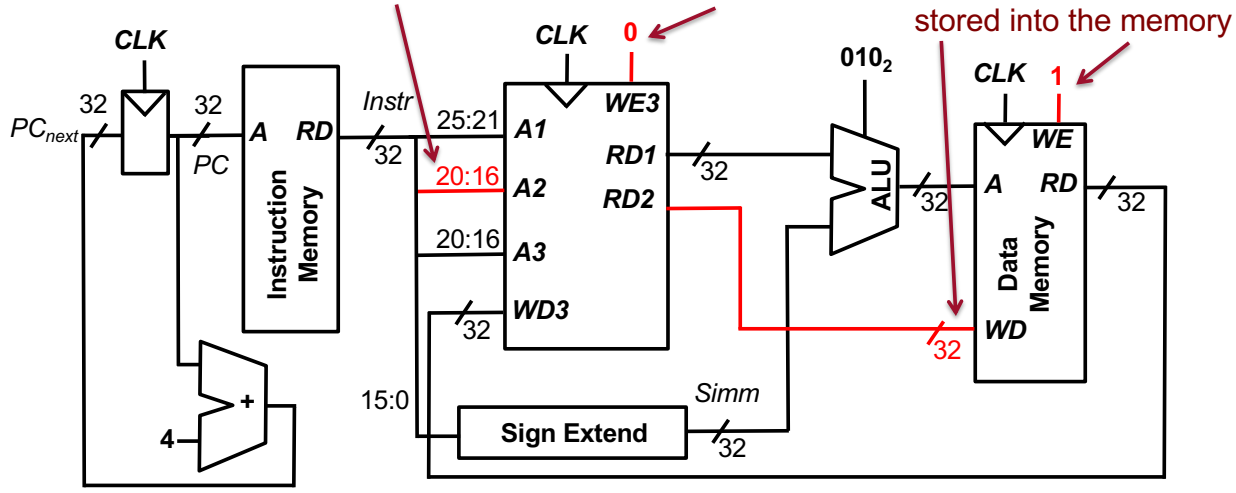


# `sw` instruction

We need to read the base address, read the offset, and compute an address. Good news: **We have already done that!**.

**Example**

```
sw    $s0,4($s1)
```

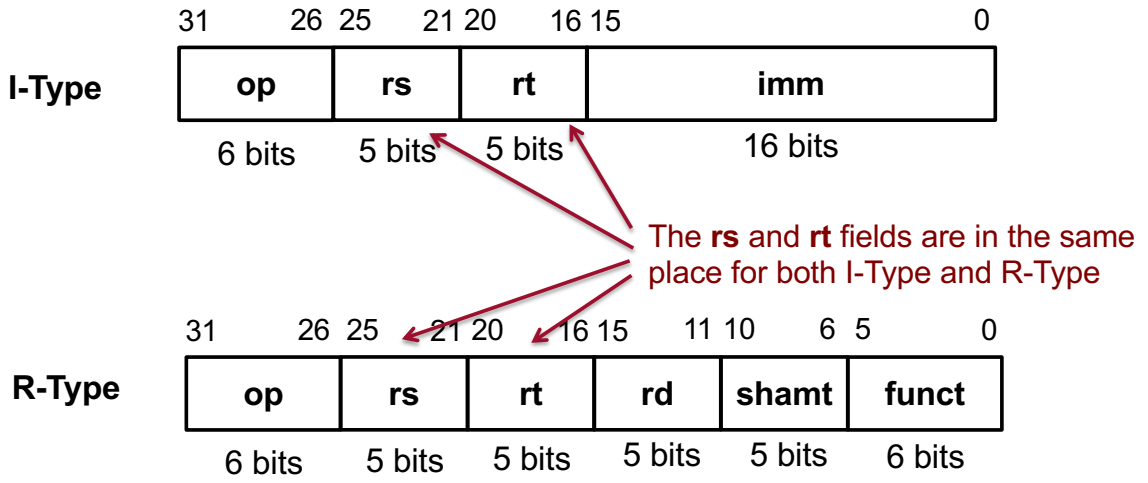







# R-type instructions – Machine Encoding

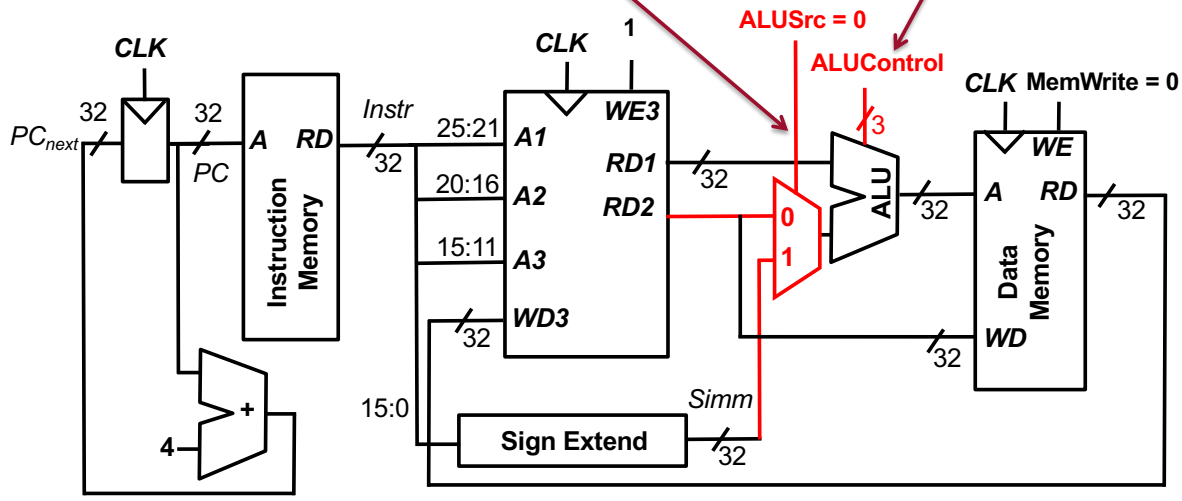
We are now going to handle all R-type instructions the same uniform way. That is, we should handle **`add`**, **`sub`**, **`and`**, **`or`**, and **`slt`**.



# R-type instructions – ALU Usage

We want to send the second operand to the ALU, but still be compatible with the `lw`-instruction.

Different ALU control signals for different instructions.



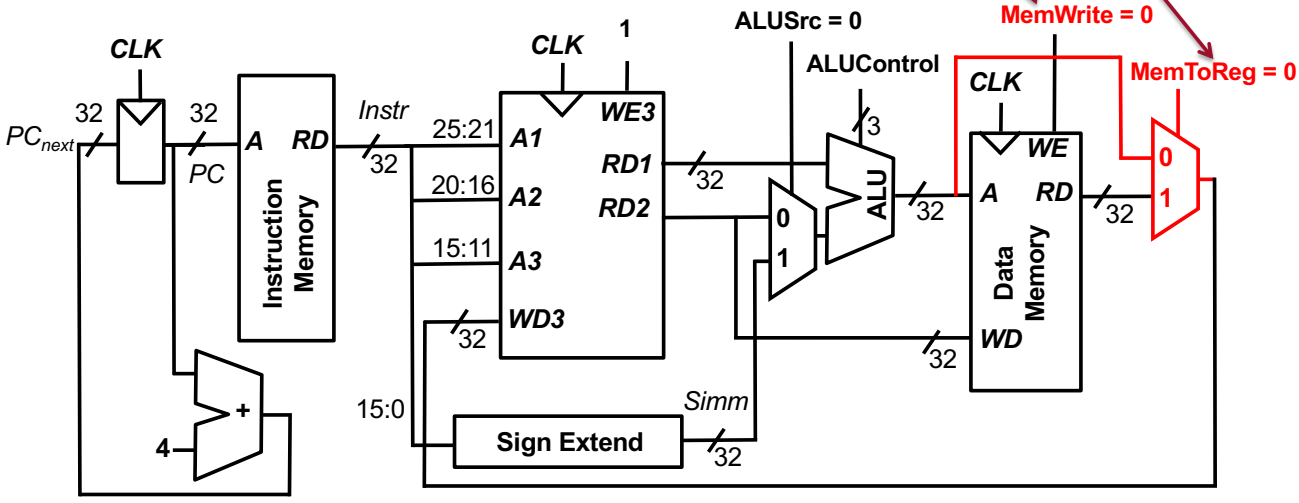




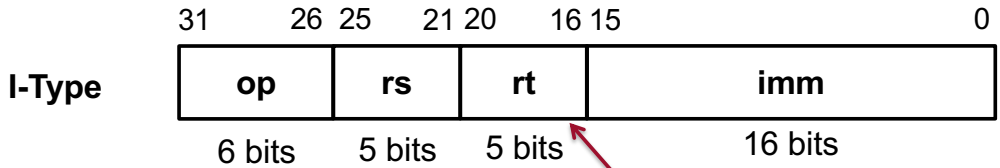

# R-type instructions – Write to Register

R-Type instructions write to registers and not to memory.

Bypass the data memory if an R-Type instruction

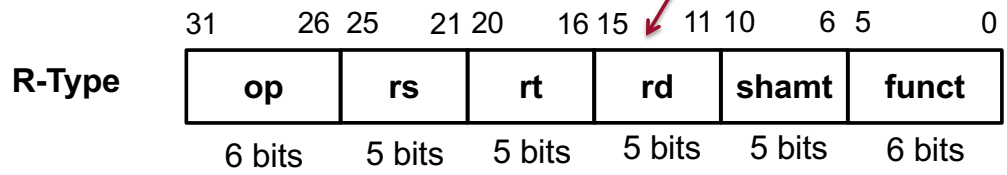


# R-type instructions – Machine Encoding




For the `lw` instruction, the target register is stored in the **rt** field (bits 20:16)

For R-type instructions, the target register is stored in the **rd** field (bits 15:11)



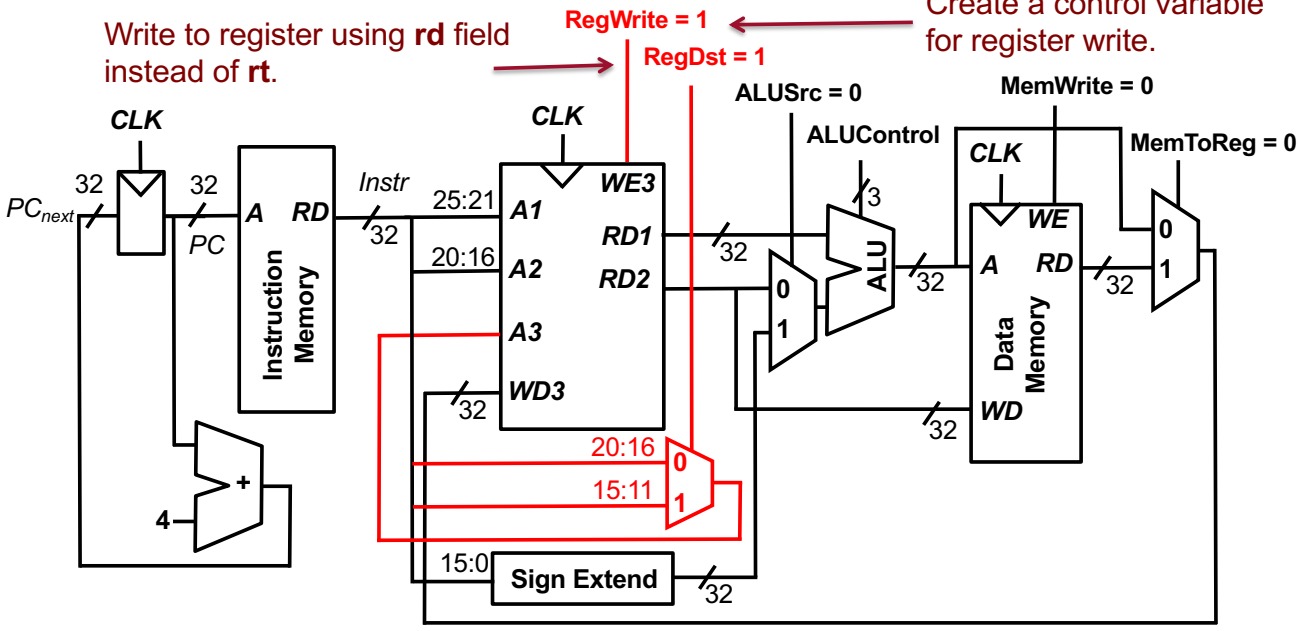





# R-type instructions – Use the rd field

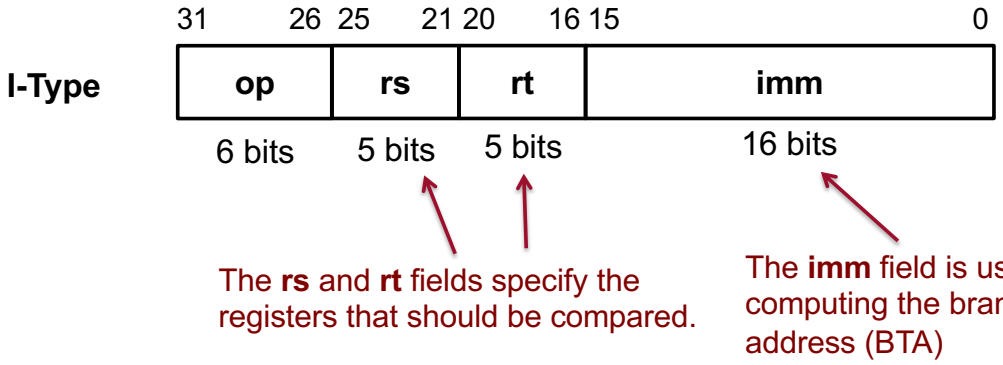
Write to register using **rd** field instead of **rt**.

Create a control variable for register write.



# `beq` instruction – Machine Encoding

Recall that the `beq` instruction is a branch instruction, encoded in the I-Type.




**Recall how to compute the BTA:**
BTA = PC + 4 + imm * 4

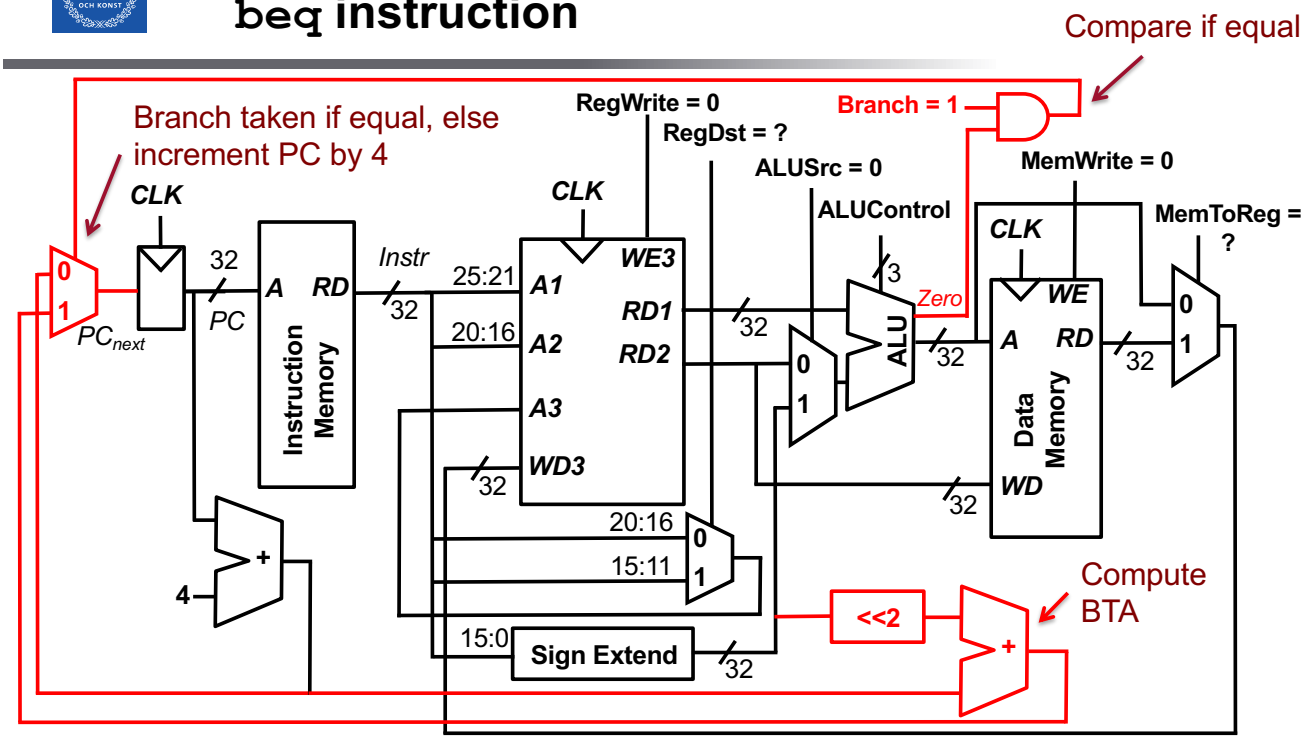
**Example**

```
beq    $s0,$s1,loop
```



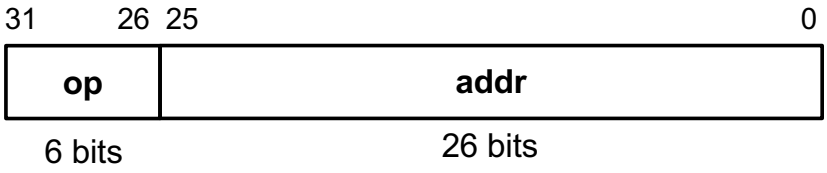


# `beq` instruction



# Pseudo-Direct Addressing (Revisited)

The **J** and **JAL** instructions are encoded using the **J-type**. But, the address is not 32 bits, only 26 bits.



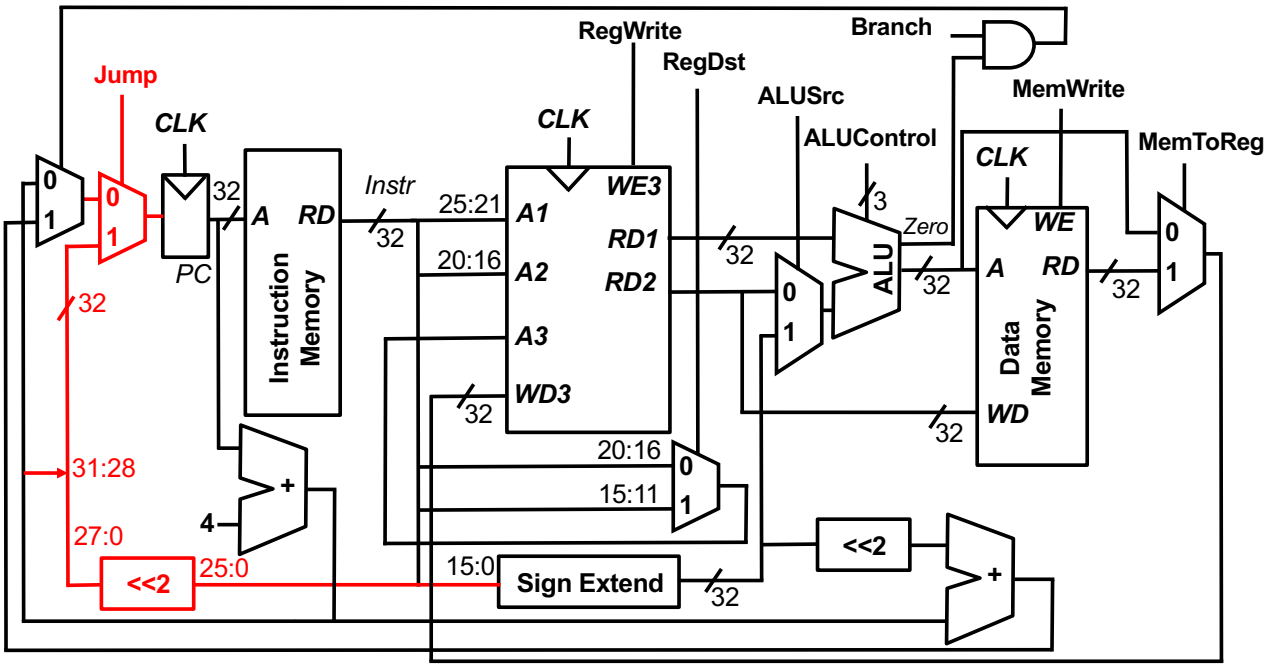

A **32-bit Pseudo-Direct Address** is computed as follows:

- Bits 1 to 0 (least significant) are always zero because word alignment of code.
- Bits 27 to 2 is taken directly from the **addr** field of the machine code instruction.
- Bits 31 to 28 are obtained from the four most significant bits from PC + 4.

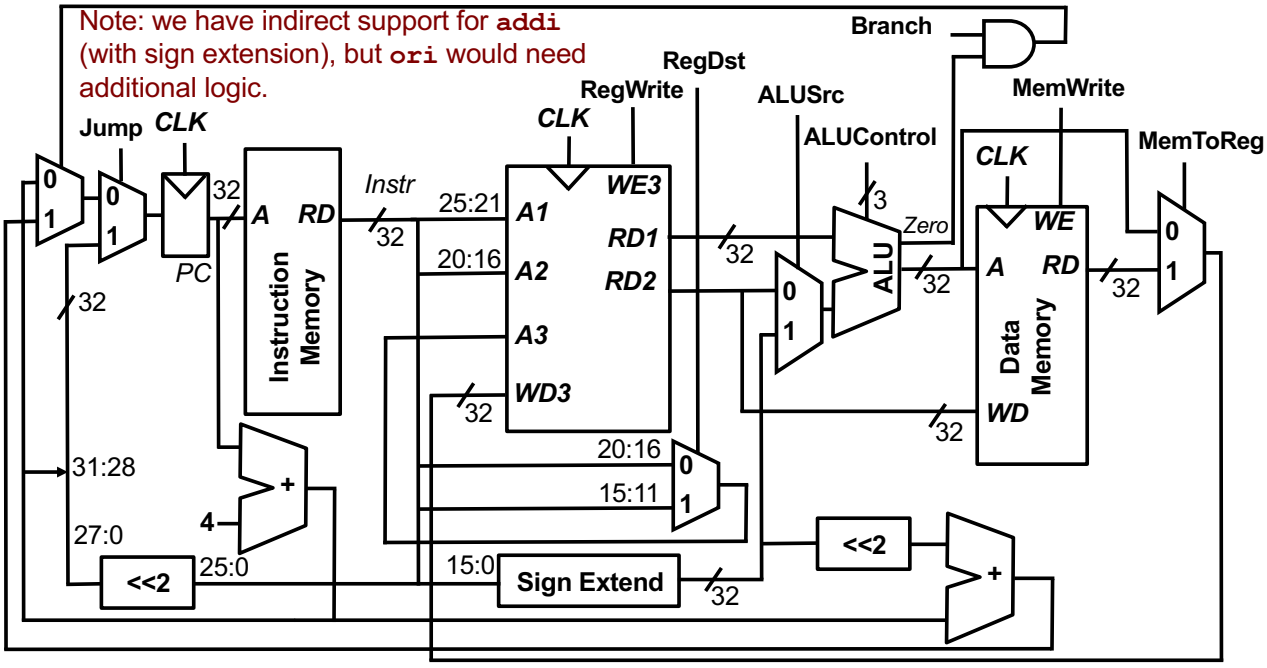




# `j` instruction



# Data Path for Instructions
## `add,sub,and,or,slt,addi,lw,sw,beq,j`





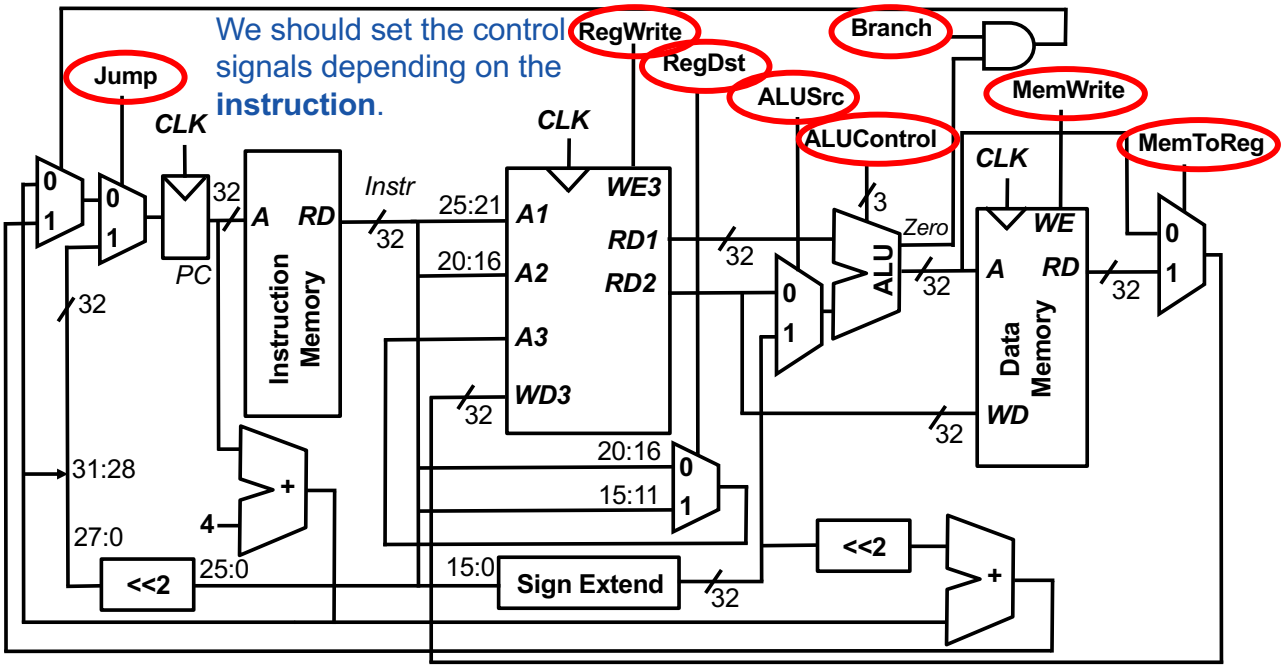



# Part III

# Control Unit in a Single-Cycle Processor

Acknowledgement: The structure and several of the good examples are derived from the book "Digital Design and Computer Architecture" (2013) by D. M. Harris and S. L. Harris.

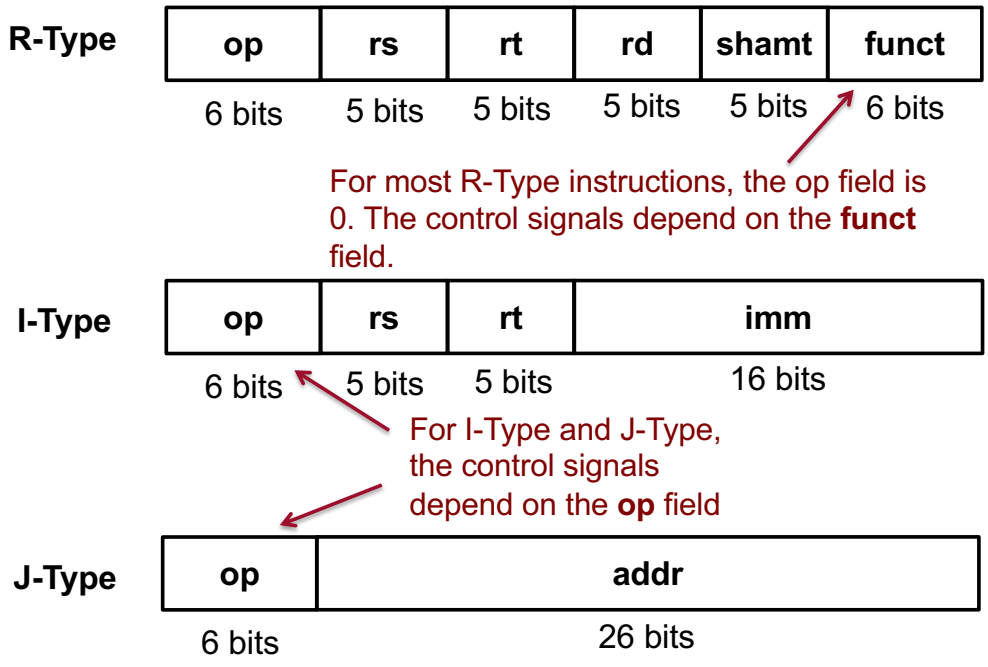
# What to Control?



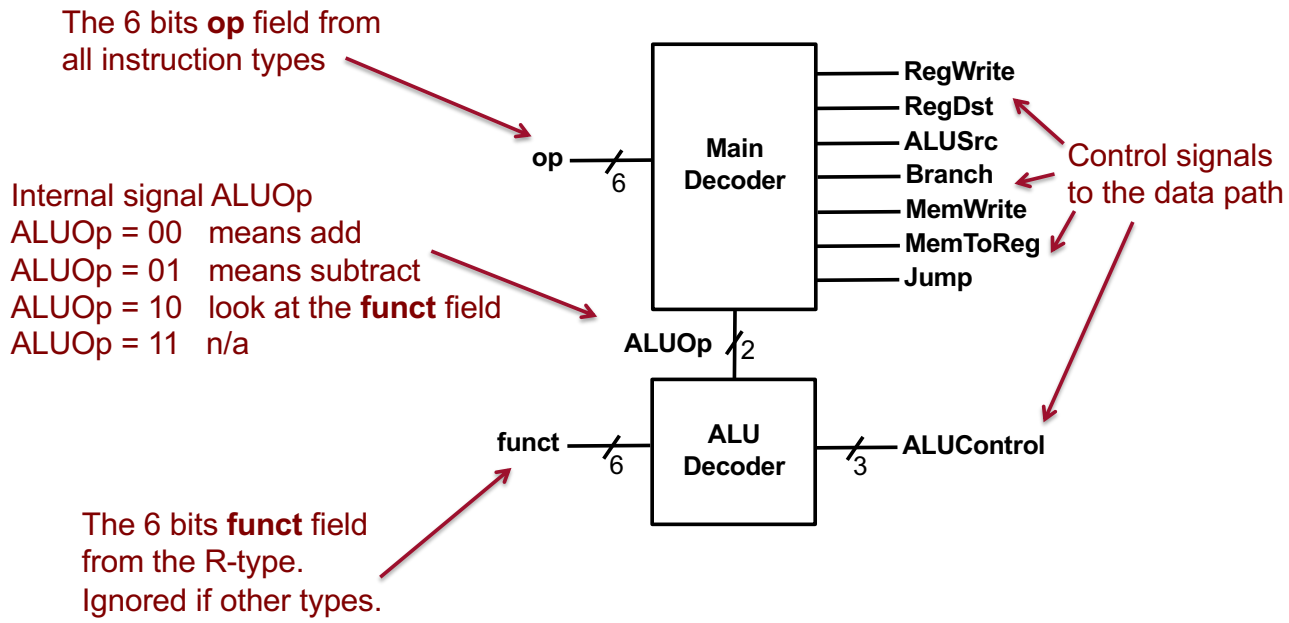





# Control Unit Input: Machine Code



# Control Unit Structure

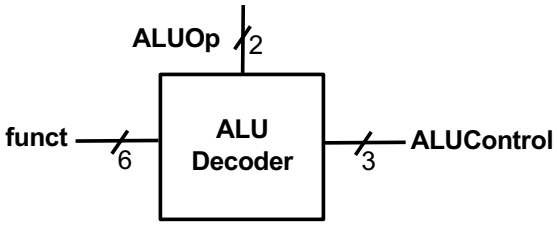







# ALU Decoder

Enough to check one bit (faster decoding)




| ALUOp | funct | ALUControl |
|---|---|---|
| 00 | ? | 010 (add) |
| ?1 | ? | 110 (subtract) |
| 1? | 100000 (add) | 010 (add) |
| 1? | 100010 (sub) | 110 (subtract) |
| 1? | 100100 (and) | 000 (and) |
| 1? | 100101 (or) | 001 (or) |
| 1? | 101010 (slt) | 111 (set less than) |

# Main Decoder

| Instr | op | RegWrite | RegDst | ALUSrc | Branch | MemWrite | MemToReg | Jump | ALUOp |
|---|---|---|---|---|---|---|---|---|---|
| R-Type | 000000 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 10 |
| lw | 100011 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 00 |
| sw | 101011 | 0 | ? | 1 | 0 | 1 | ? | 0 | 00 |
| beq | 000100 | 0 | ? | 0 | 1 | 0 | ? | 0 | 01 |
| addi | 001000 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 00 |
| j | 000010 | 0 | ? | ? | ? | 0 | ? | 1 | ?? |





# Performance Analysis (1/2) General View

How should we analyze the performance of a computer?

- By clock frequency?
- By instructions per program?

$$\text{Execution time (in seconds)} = \#\text{ instructions} \times \frac{\text{clock cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{clock cycle}}$$

Number of instructions in a program (# = number of)

Average **cycles per instruction (CPI)**

Seconds per cycle = **clock period $T_C$**

Determined by programmer or the compiler or both.

Determined by the micro-architecture implementation.

Determined by the critical path in the logic.

**Problem**:
- Your program may have many inputs.
- Not only one specific program might be interesting.

**Solution**:
Use a **benchmark** (a set of programs).
Example: SPEC CPU Benchmark

# Performance Analysis (2/2) Single-Cycle Processor

$$\text{Execution time (in seconds)} = \#\text{ instructions} \times \frac{\text{clock cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{clock cycle}}$$

Number of instructions in a program (# = number of)

Average **cycles per instruction (CPI)**

Seconds per cycle = **clock period $T_c$**.

Determined by programmer or the compiler or both.

Determined by the micro-architecture implementation.

Determined by the critical path in the logic.

Each instruction takes one clock cycle. That is, CPI = 1.

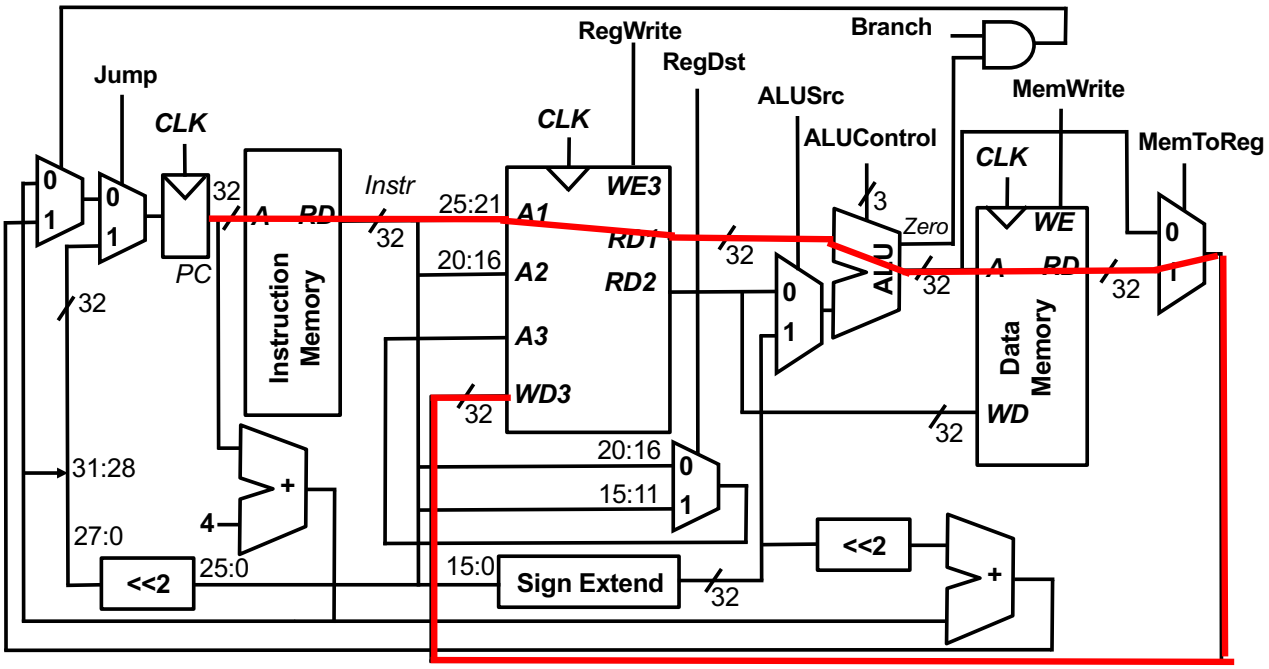
The main problem with this design is the **long critical path.**

The `lw` instruction has longer path than R-Type instructions. However, because of synchronous logic, the clock period is determined by the slowest instruction.





# Critical Path Example: Load Word (`lw`) Instruction



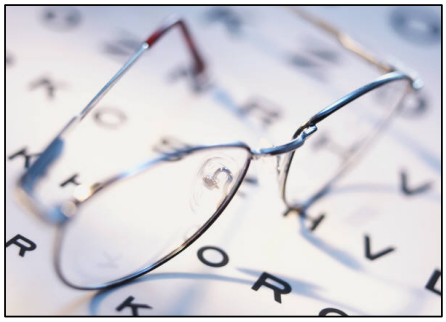
# You can soon stretch your legs…

**…but wait just a second more**





# Reading Guidelines



**Module 4: Processor Design**

Lecture 9: ALU and Single-Cycled Processors
- H&H Chapters 5.2.4, 7.1-7.3.

Lecture 10: Pipelined processors
- H&H Chapters 7.5, 7.8.1-7.8.2, 7.9

**Reading Guidelines**
See the course webpage
for more information.

# Summary

**Some key take away points:**

- The **ALU** performs most of the arithmetic and logic computations in the processor.
- The **data path** consists of sequential logic that performs processing of words in the processor.
- The **control unit** decodes instructions and tells the data path what to do.
- The single-cycle processor has a **long critical path**. We will solve this in the next lecture by introducing a **pipelined processor**.


**Thanks for listening!**

David Broman
dbro@kth.se