Computer Architecture

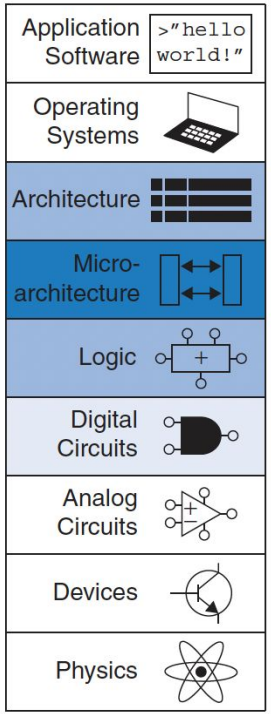
Processor: Part 1

Single-cycle microarchitectures

Agenda

- The big picture
  - Architecture state and instruction set
  - Design process
  - MIPS microarchitectures
- Single-cycle processor

The big picture



- MIPS

- Combinational logic
- Sequential logic
- ALU
- FSM

We have learned how to build circuits for some given combinational and sequential logic.

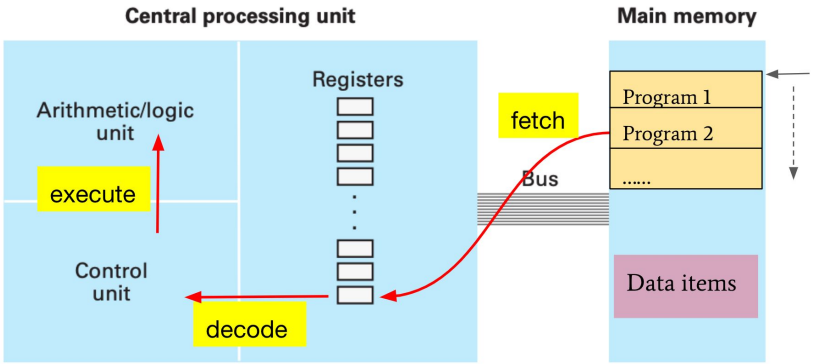
We also know the MIPS architecture. Now it is time to implement MIPS with the digital logic.

The design that connects MIPS to digital logic is called **microarchitecture**.

Architectural sate and instruction set

A computer architecture is defined by its **instruction set** and **architectural state**:

- the instruction set defines what a processor can do;
- the architectural state defines the processor's current status as it carries out instructions.



To run a program,

- the computer executes the instruction and data in registers currently;
- then, the next instruction is fetched into registers. The machine will go on to process the instruction (and data) in the registers.

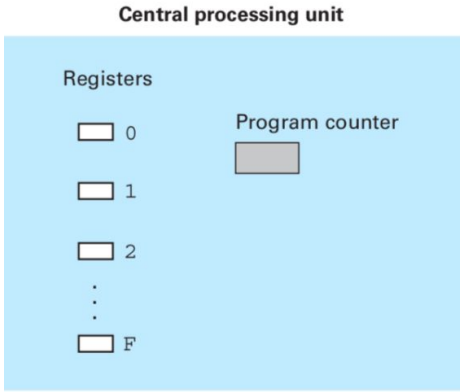
What devices should we use to represents the states in this flow of execution?

MIPS Architectural State

The **architectural state** for MIPS processor consist of:

- The value of the program counter
- The values of the 32 registers

Based on the current architectural state, the processor executes a particular instruction with a particular set of data to produce a new architectural state.

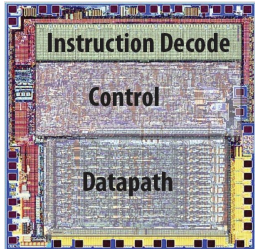


MIPS instruction set

In order to keep the microarchitecture easy to understand, we will only consider a subset of the MIPS instruction set:

- R-type arithmetic/logic instructions: `add, sub, and, or, slt`
- Memory instructions: `lw, sw`
- Branches: `beq`

Data path and control



A microarchitecture is typically divided into two interacting parts:

- **Datapath**: It is a collection of functional units and connections within a processor that carry out data processing operations specified by the instruction set. It operates on words of data.
  - It contains structures like **memories**, **registers**, **ALU**, and **multiplexers**
  - We use a 32-bit datapath since MIPS is a 32-bit architecture.
- **Control**: it receives the current instruction from the datapath and tells the datapath how to execute an instruction.
  - It produces signals like the multiplexer select, register enable, and memory write to control the operation of the datapath.

Design process

A good way to design a complex system is to start with hardware containing the **state elements**, then, add blocks of combinational logic between the state elements to compute the new state based on the current state.

In the case of the MIPS processor, we start with

- Program counter
- Instruction memory
- Register file
- Data memory

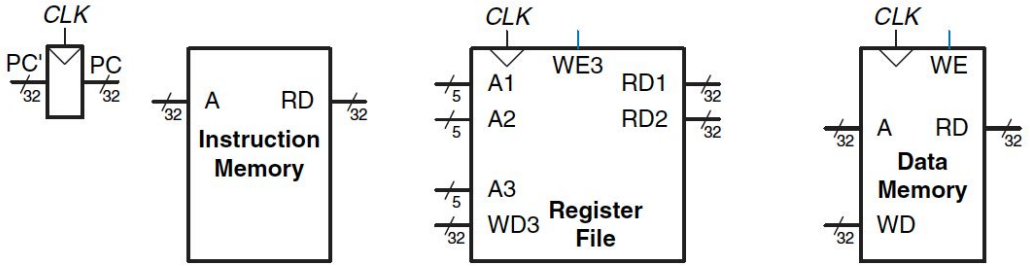
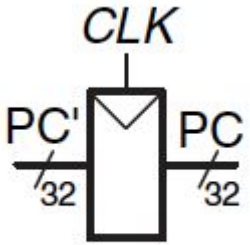


Figure 7.1 State elements of MIPS processor

The program counter

The program counter is an ordinary 32-bit register.

- Its output, ***PC***, points to the current instruction.
- Its inputs, ***PC'***, indicates the address of the next instruction.

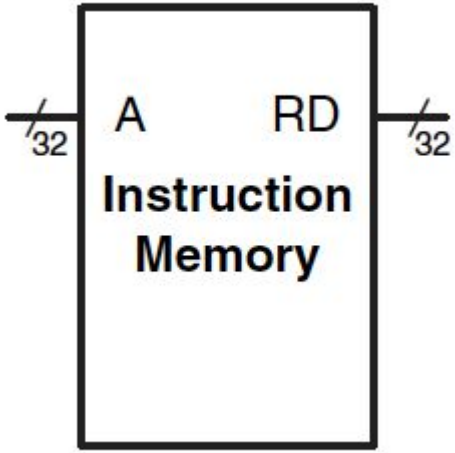


The instruction memory

The instruction memory has a single read port RD.

It takes a 32-bit instruction address input A and reads the 32-bit data (i.e., instruction) from that address onto the read data output RD.

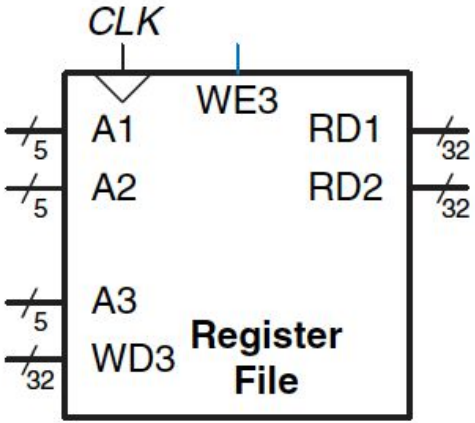
(Note: it is often convenient to partition the overall memory into two small memories, one containing instructions and the other containing data.)



The register file

The 32-element x 32-bit register file has two read ports and one write port:

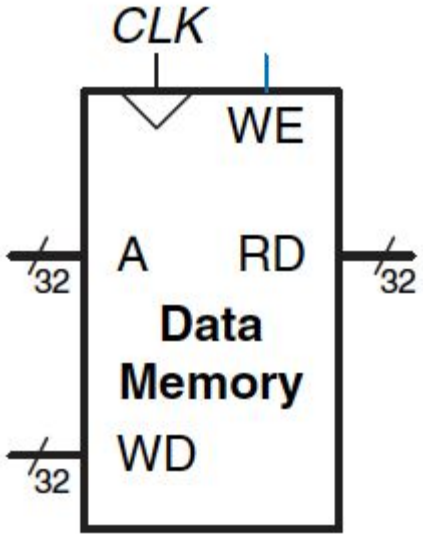
- The two read ports take 5-bit address inputs A1 and A2. each specifying one of 2^5=32 registers are source operands. The register file then provides the two 32-bit register values onto the read data outputs RD1 and RD2.
- The write port takes a 5-bit address input A3; a 32-write data input WD; a write enable input WE3; and a clock. If the write enable WE3 is 1, the register file writes the data into the specified register on the rising edge of the clock.



The data memory

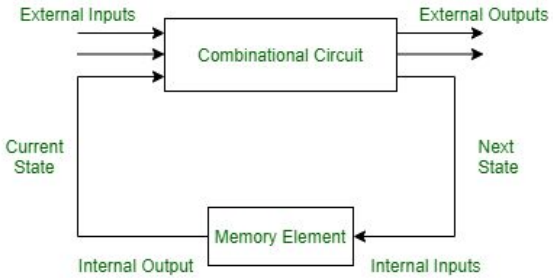
The data memory has a single read/write port:

- If the write enable WE is 1, it writes data WD into address A on the rising edge of the clock.
- If the write enable WE is 0, it reads address A onto RD.



MIPS microarchitectures

Three MIPS microarchitectures with different ways to connect state elements:

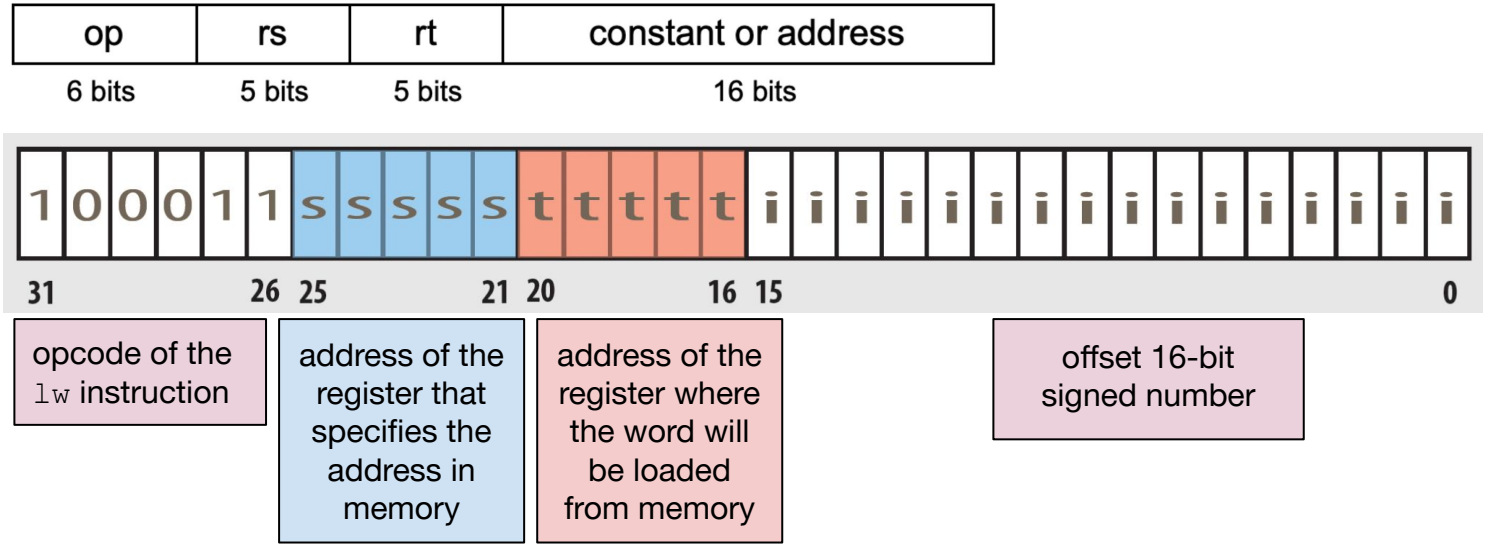


- Single-cycle microarchitecture
  - Executes an entire instruction in each cycle
  - Easy to explain, simple control unit
- Multiple-cycle microarchitecture
  - Executes instructions in a series of shorter cycles
  - Reduces the hardware cost by reusing expensive hardware blocks
  - Executes only one instruction at a time but in several cycles
- Pipelined microarchitecture
  - Can execute several instructions at the same time
  - Improves the throughput significantly
  - All commercial high-performance processors use pipelining today

The single-cycle microarchitecture

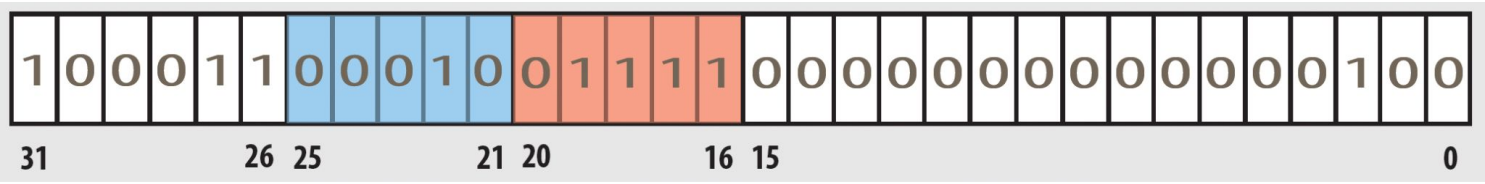
First, we will work out the datapath connections for the `lw` instruction.

- `lw` is an I-format instruction: `lw $t, offset($s)`



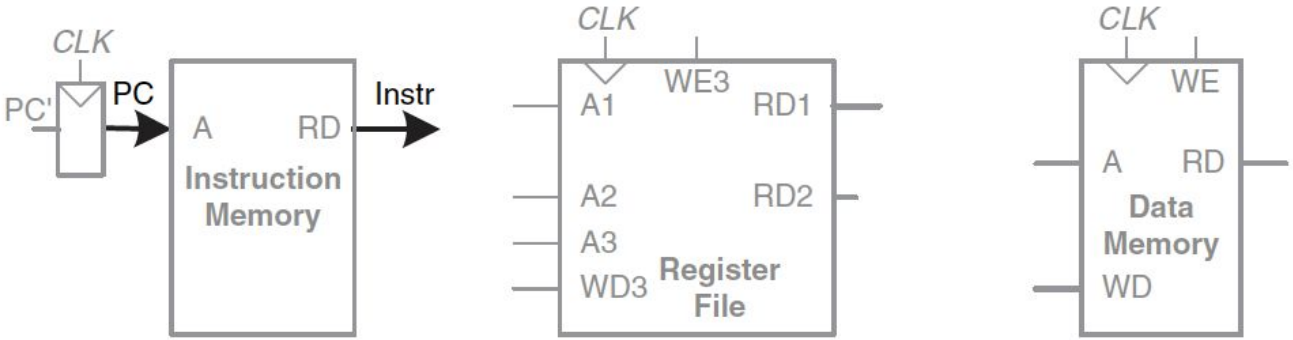
The load word instruction

`lw $15, 4($2)` #loads the value of `Mem[$2+4]` in register `$15`



- It reads the source operand (i.e., the base address), then, adds the immediate number to the base address to obtain the address. Finally, it loads the value from the address and stores it in the target register.

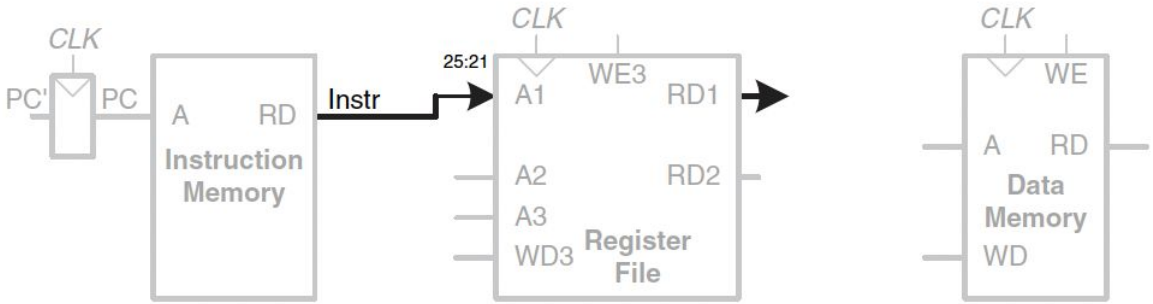
Step 1: fetch instruction from memory



The first step is to read the instructions:

- Connecting ***PC*** to the address port ***A*** of the instruction memory
- The instruction memory reads out (i.e., fetch) the 32-bit instruction labelled ***Instr***

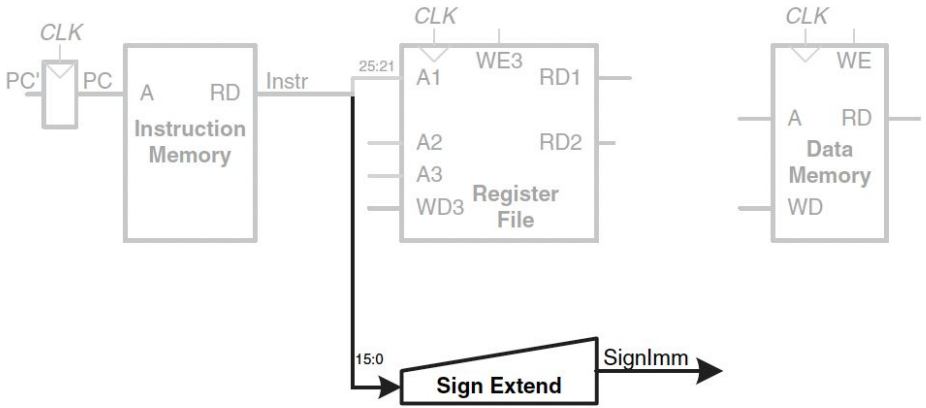
Step2: read source operand from register file



The next step is to read the source register containing the base address.

- The source register is specified in the `rs` field **[25:21]** of the instruction.
- These 5 bits of the instruction are thus connected to the address input of the register file.

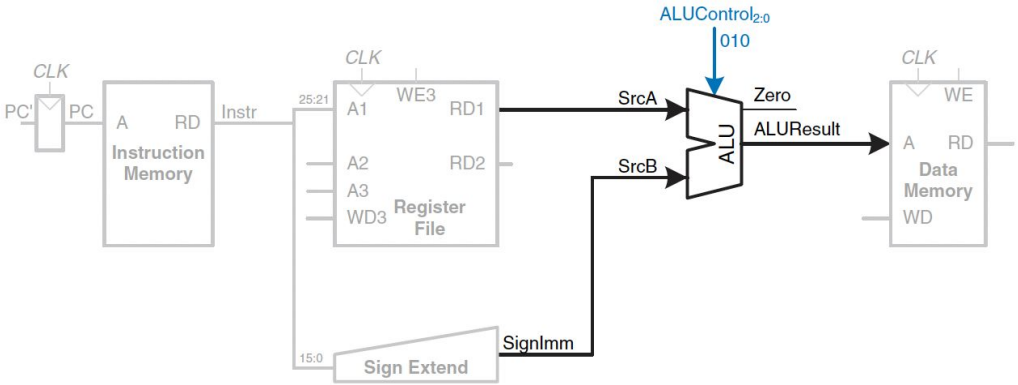
Step 3: sign-extend the immediate



The `lw` instruction needs an offset.

- The offset is stored in the immediate field [15:0] of the instruction
- We needs to sign extend it to a 32-bit integer:
  - ***SignImm[15:0] = Instr[15:0]***; ***SignImm[31:16] = Instr[15]***

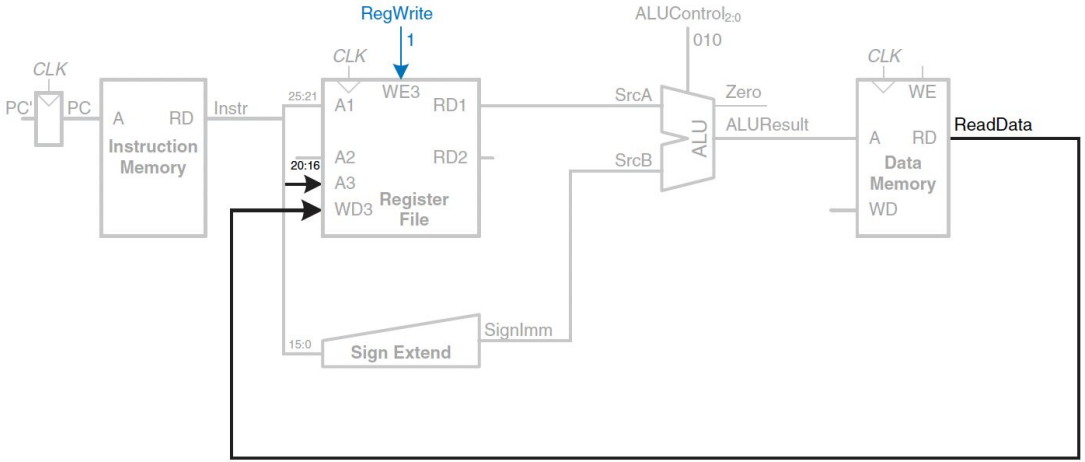
Step 4: compute the memory address



The processor must add the base address to the offset to find the address to read from memory. The addition is performed by the ALU.

- Since the ALU can perform several kinds of operations, the 3-bit ***ALUControl*** signal (010) is needed to specifies the operation: ***addition***.
- The ***Zero*** flag indicates whether ***ALUResult==0***.

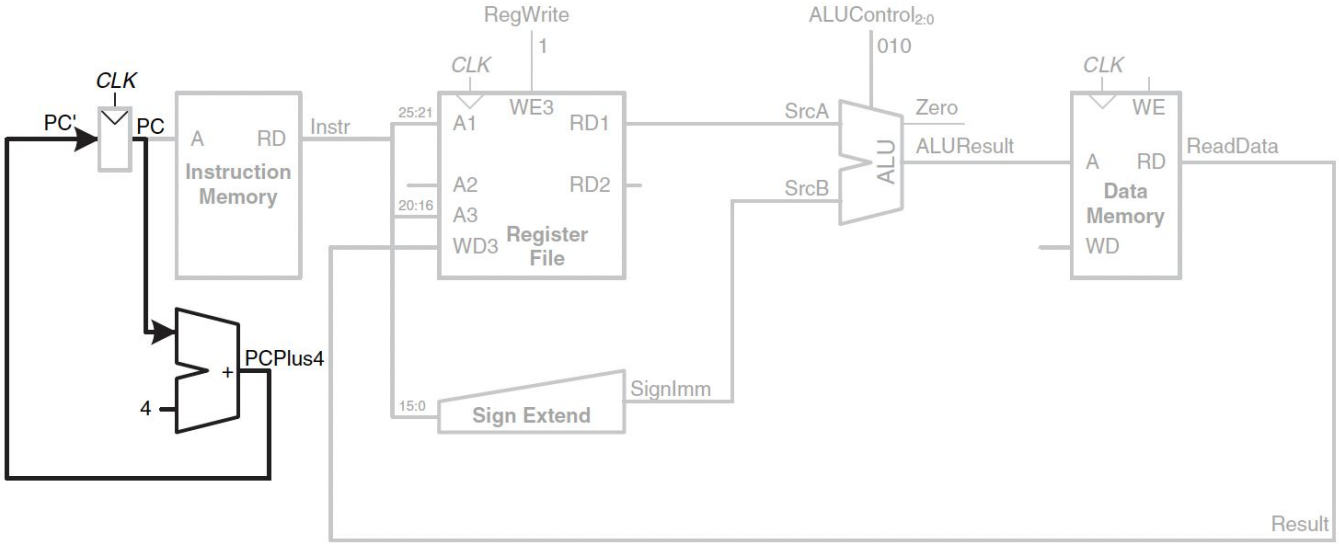
Step 5: write data back to register file



The data is read from the data memory onto the ***ReadData*** bus, and then written back to the destination register at the end of the cycle.

- The destination register of `lw` is specified by the `rt` field [20:16] and thus connected to the address input ***A3*** of the register file.
- The control signal ***RegWrite*** is set to 1 during the instruction `lw` so that the data value is written in the register.
- The write takes place on the rising edge of the clock at the end of the cycle.

Step 6: determine the address of the next instruction

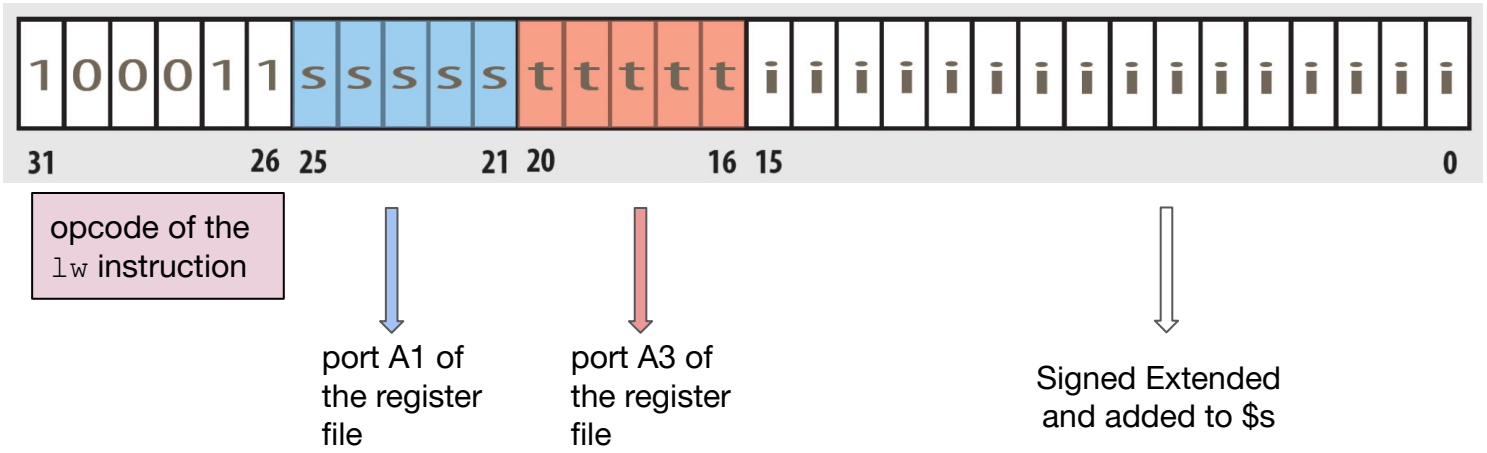


Finally, the processor must compute the address of the next instruction. Because MIPS instructions are 32-bit long = 4 bytes, the next instruction is at ***PC+4***.

- The processor thus uses another ***ALU*** in order to compute the address of the next instruction.
- The address is then written in ***PC*** on the next rising edge of the clock.

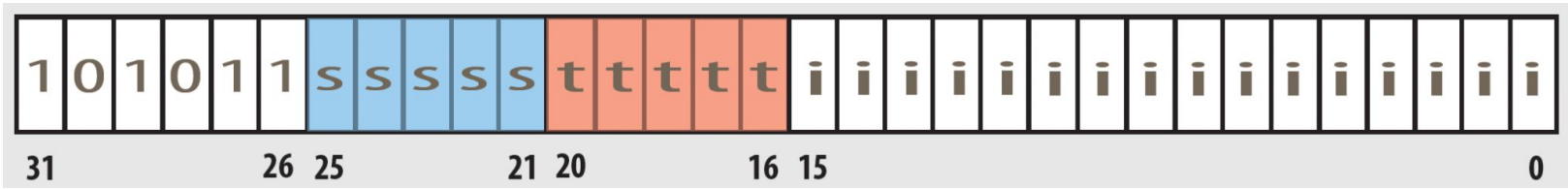
The load word instruction

`lw $t, offset($s)` #loads the value of `Mem[$s+offset]` in register `$t`



The store word instruction

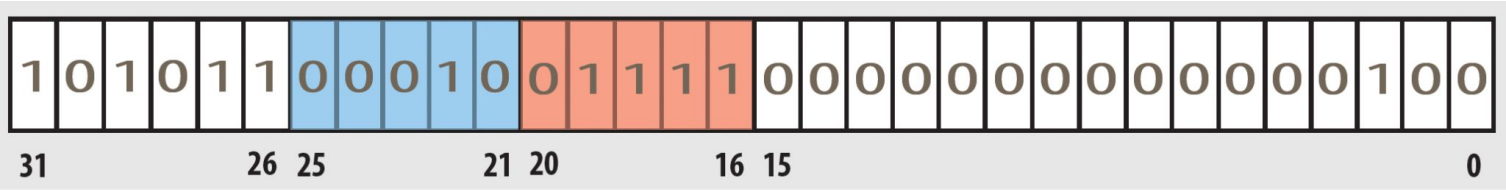
`sw $t, offset($s)` #stores the value of `$t` in the memory address `Mem[$s+offset]`



- It reads the source operand (i.e., `$s`), then, add the immediate number to the base address to obtain the address. Then, it reads the word from `$t`. Finally, it stores the value to the address.

The store word instruction

`sw $15, 4($2)` #store the value in register `$t` into `Mem[$s2+4]`



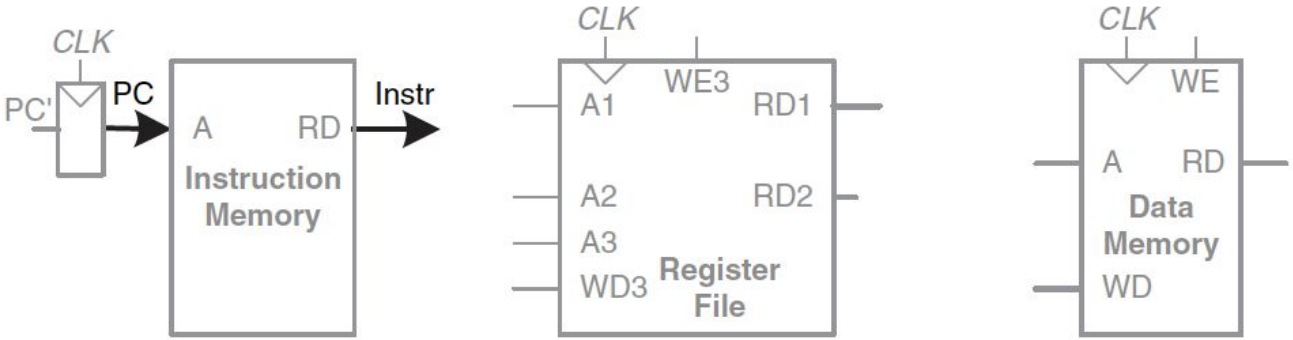
opcode of the `sw` instruction

address of the register 2 which specifies the address in memory

address of the register 15 that contains the word which will be stored from memory

offset 16-bit signed number

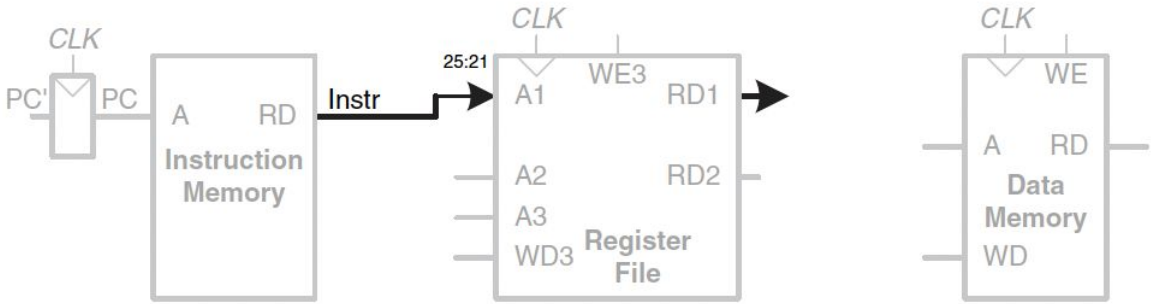
Step 1: fetch instruction from memory



The first step is to read the instructions:

- Connecting ***PC*** to the address port ***A*** of the instruction memory
- The instruction memory reads out (i.e., fetch) the 32-bit instruction labelled ***Instr***

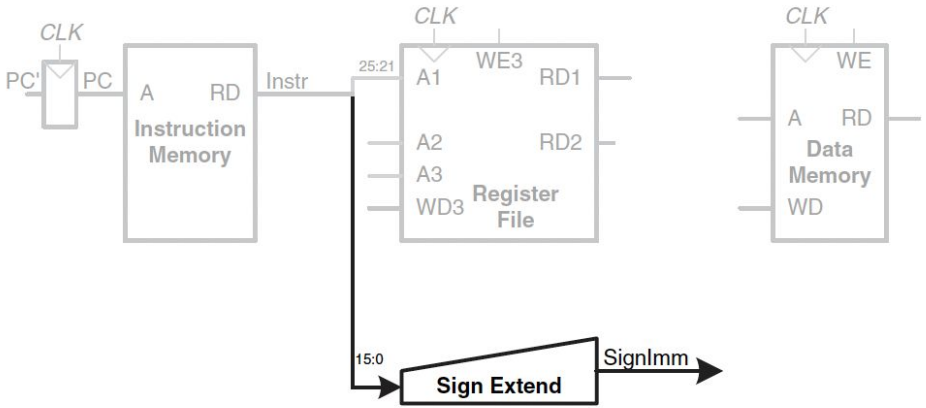
Step 2: read source operand from register file



The next step is to read the source register containing the base address.

- The source register is specified in the ***rs*** field **[25:21]** of the instruction.
- These 5 bits of the instruction are thus connected to the address input of the register file.

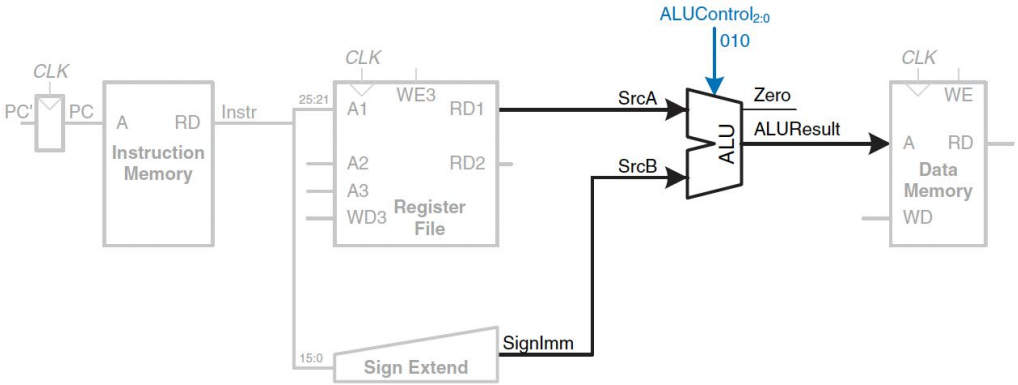
Step 3: sign-extend the immediate



The `sw` instruction needs an offset.

- The offset is stored in the immediate field [15:0] of the instruction
- We needs to sign extend it to a 32-bit integer:
  - ***SignImm[15:0] = Instr[15:0]***; ***SignImm[31:16] = Instr[15]***

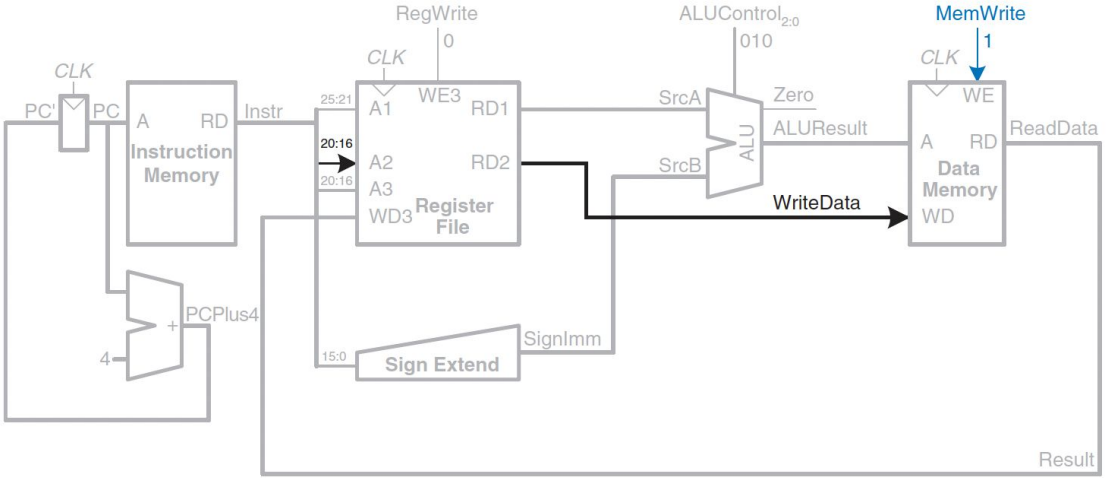
Step 4: compute the memory address



The processor must add the base address to the offset to find the address to read from memory. The addition is performed by the ALU.

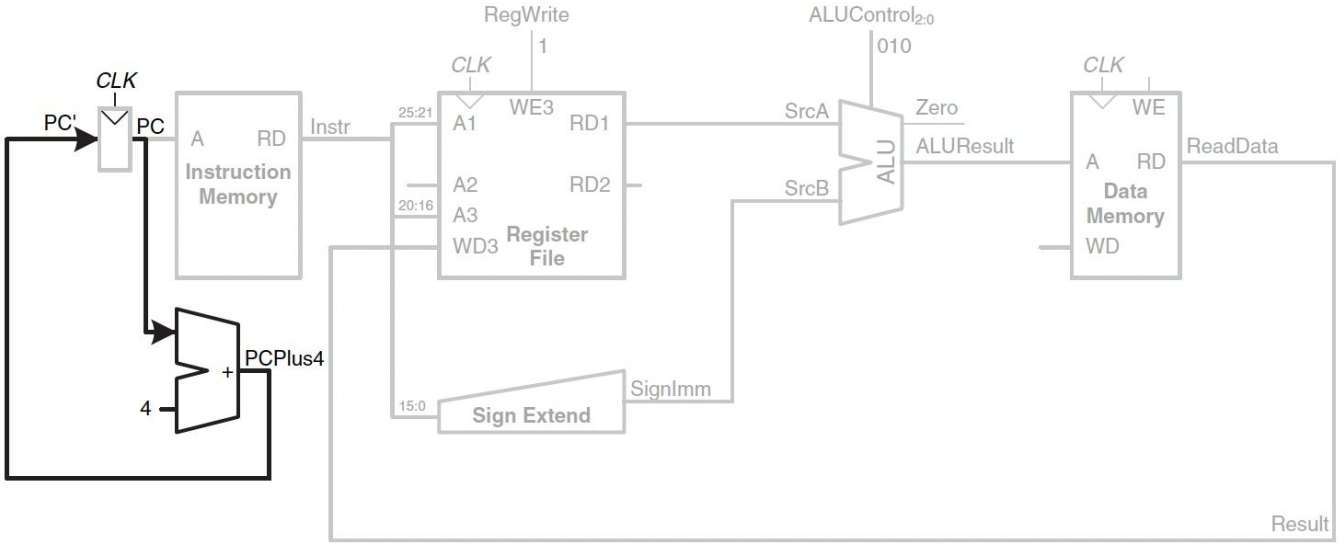
- Since the ALU can perform several kinds of operations, the 3-bit ***ALUControl*** signal (010) is needed to specifies the operation: ***addition***.
- The ***Zero*** flag indicates whether ***ALUResult==0***.

Step 5: write data to memory



- The register whose 32-bit value is transmitted to memory by `sw` is specified by the `rt` field[20:16] which is thus connected to the address input **A2** of the register file.
- The value of the register is then read from the register file onto the *WriteData* bus and connected through the bus to the write data port **WD** of the data memory.
- The control signal *MemWrite* is set to 1 during sw to indicate that the memory cell should be written on the rising edge of the clock at the end of the cycle.

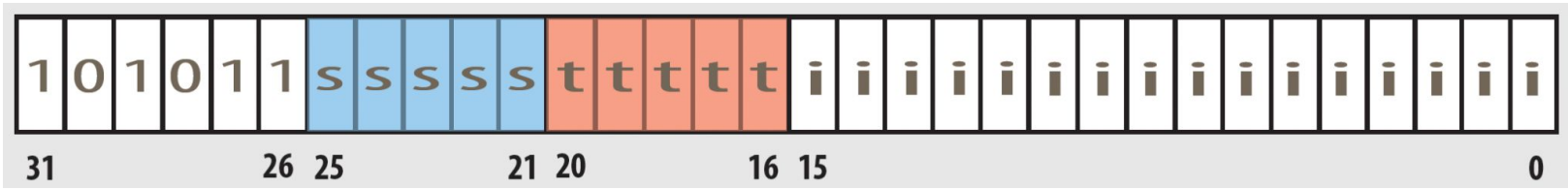
Step 6: determine the address of the next instruction



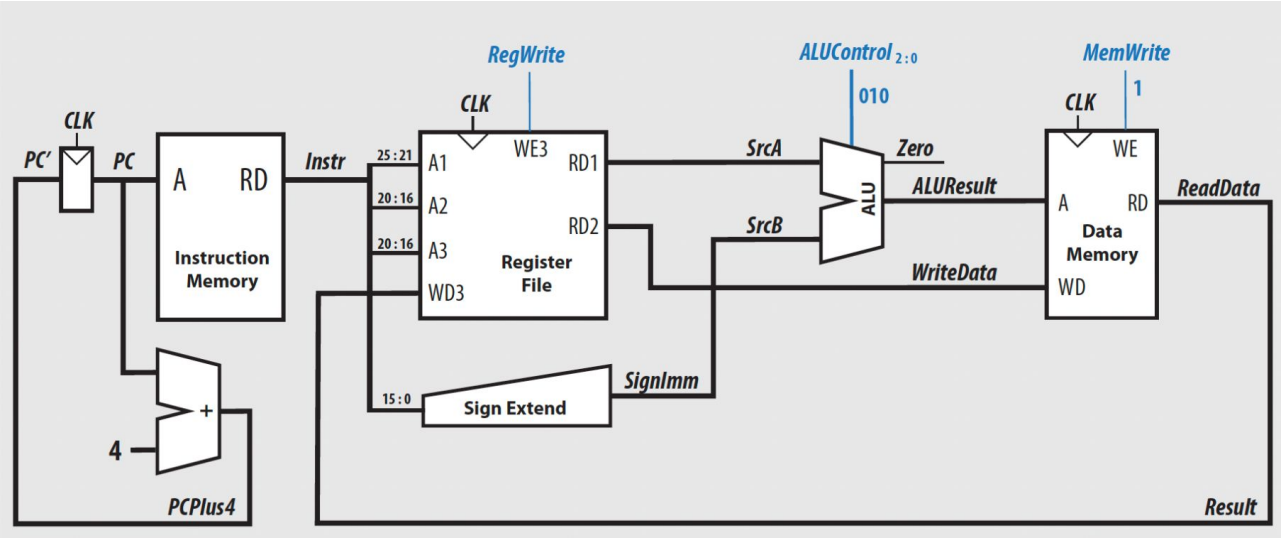
Finally, the processor must compute the address of the next instruction ***PC' = PC+4***.

The store word instruction

`sw $t, offset($s)` #stores the value of `$t` in the memory address `Mem[$s+offset]`

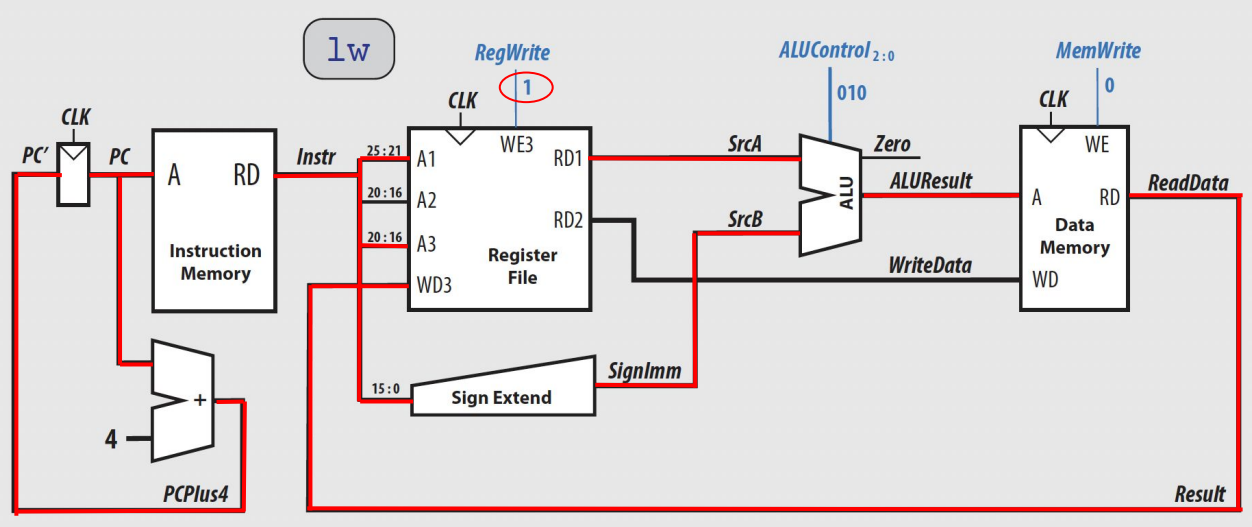


Putting the two instructions together …



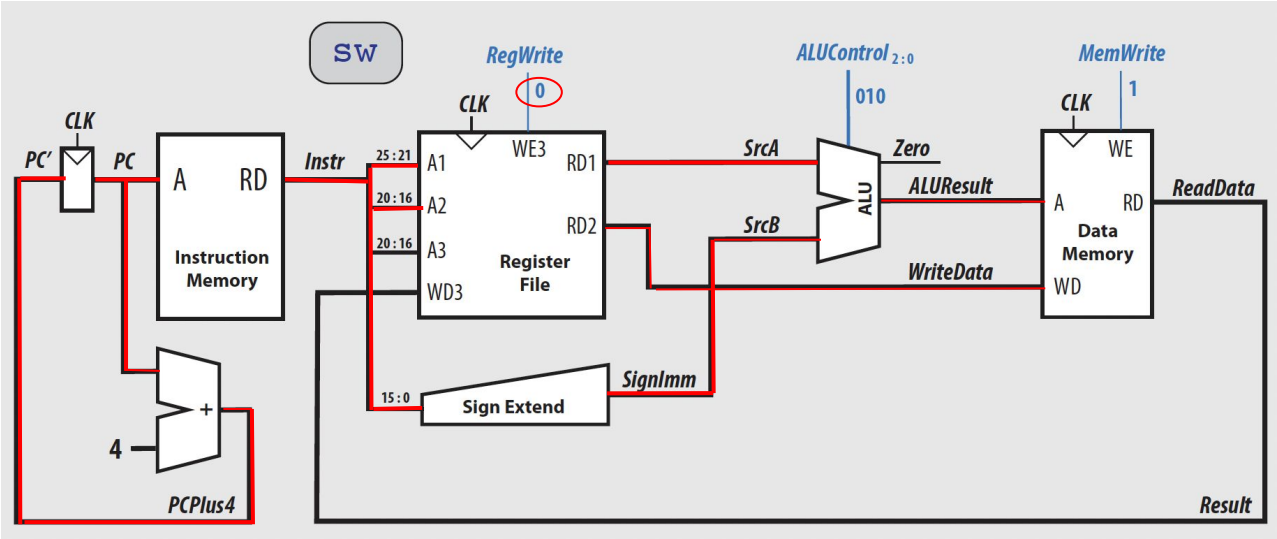
- The control signal ***ReWrite*** is set to **0** during the `sw` instruction to indicate that the data value of ***WD3*** should not be written in the register of address ***A3***.
- Similarly, the control signal ***MemWrite*** is set to **1** during the sw instruction to indicate that the data value ***WD*** should be written at memory address ***A***.

Putting the two instructions together …



- The control signal ***ReWrite*** is set to **1** during the `lw` instruction to indicate that the data value of ***WD3*** should be written in the register of address ***A3***.
- Similarly, the control signal ***MemWrite*** is set to **0** during the `lw` instruction to indicate that the data value ***WD*** should not be written at memory address ***A***.

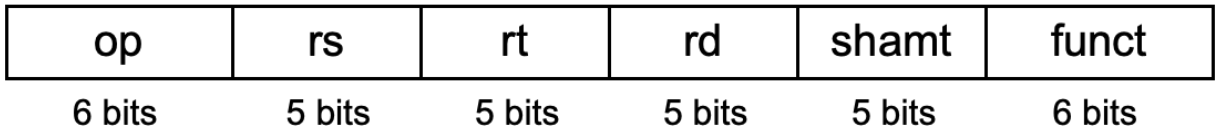
Putting the two instructions together …

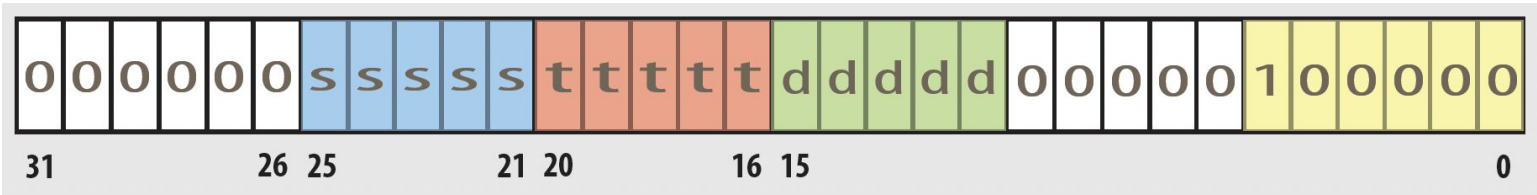


- The control signal ***ReWrite*** is set to **0** during the sw instruction to indicate that the data value of ***WD3*** should not be written in the register of address ***A3***.
- Similarly, the control signal ***MemWrite*** is set to **1** during the sw instruction to indicate that the data value ***WD*** should be written at memory address ***A***.

The add instruction

`add $d, $s, $t` #adds the values of the `$s` and `$t` and stores the result in `$d`





Same opcode for every R-instruction

Address of the first source register

Address of the second source register

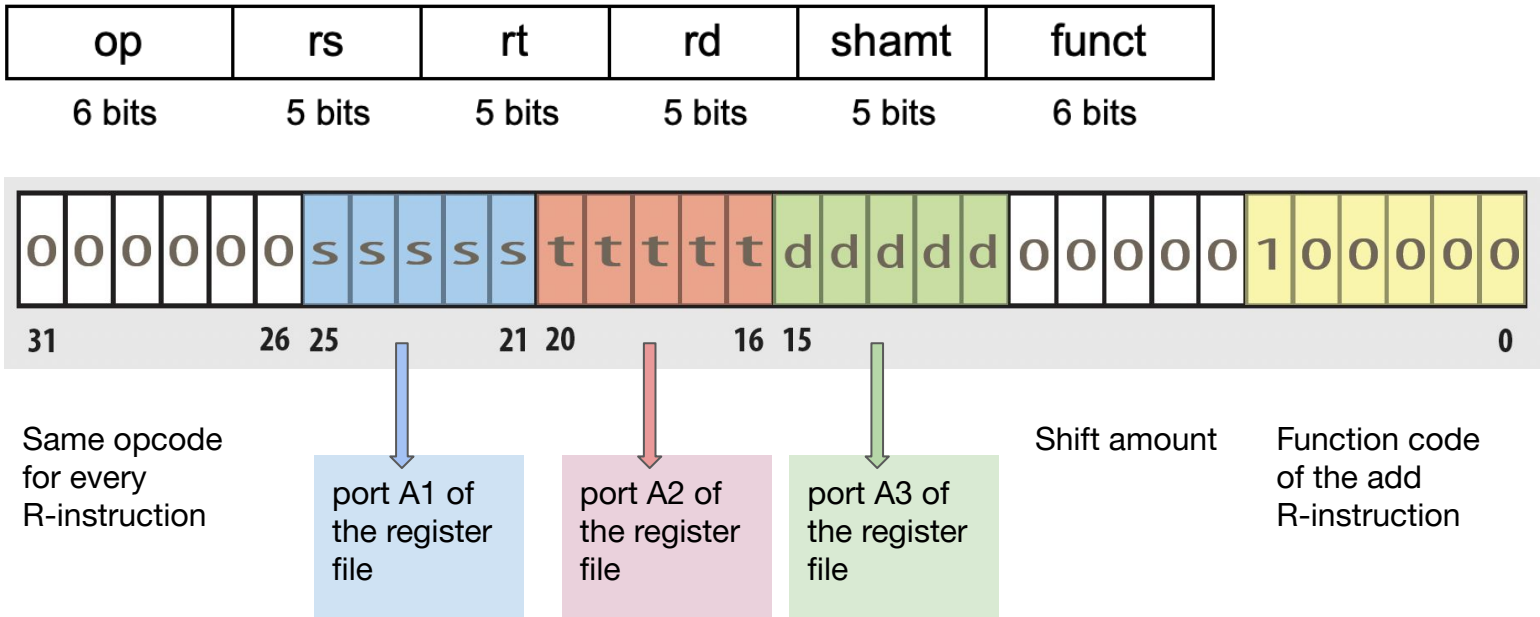
Address of the destination register

Shift amount

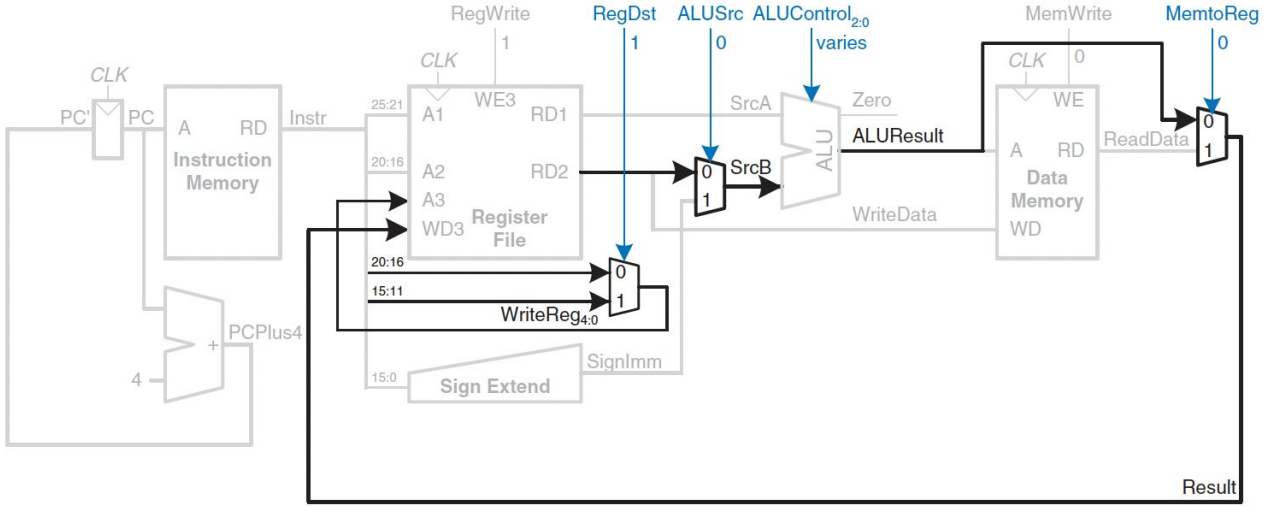
Function code of the add R-instruction

The add instruction

`add $d, $s, $t` #adds the values of the `$s` and `$t` and stores the result in `$d`

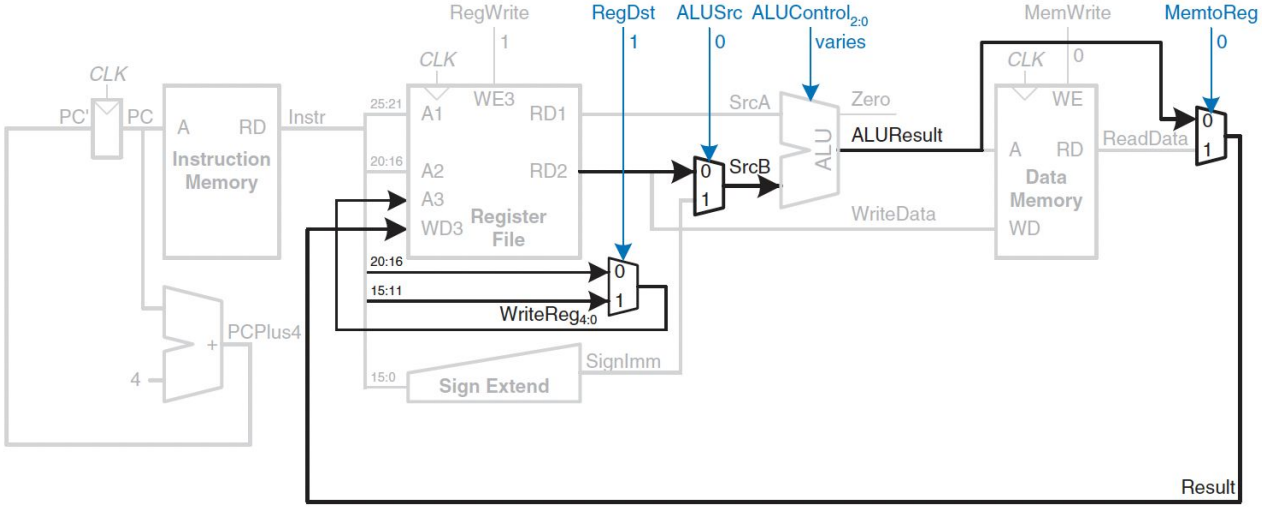


Data path enhancements for R-instructions



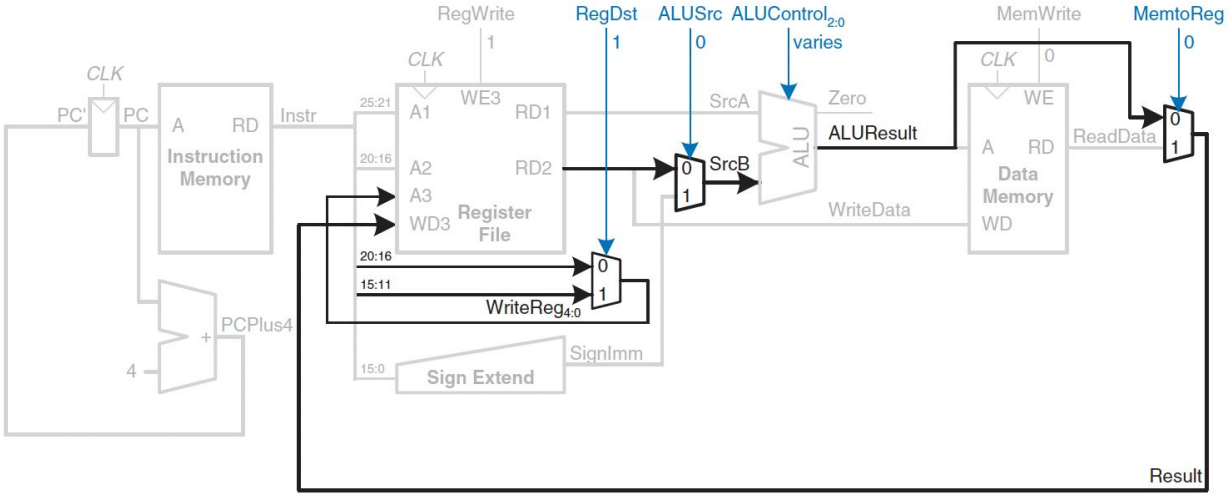
- The `add`, `sub`, `or`, and `slt` instructions can be handled with the same hardware, using different ***ALUControl*** signals.
- Now, we add a multiplexer in order to choose ***SrcB*** depending on the instruction either from the ***RD2*** port of the register file or from ***SignImm***.

Datapath enhancements for R-instructions



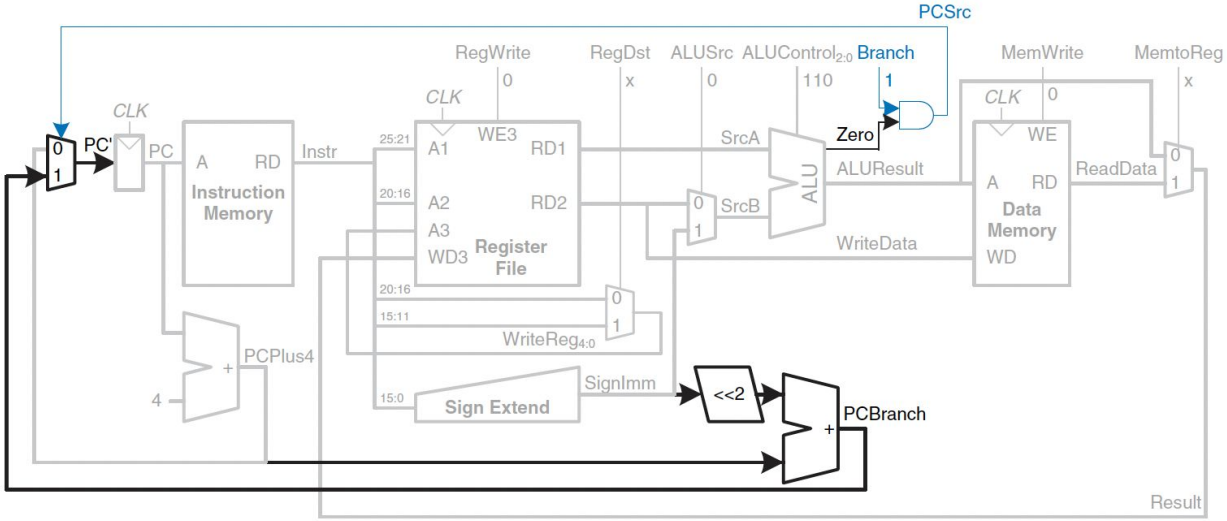
- We add a multiplexer to choose ***WriteReg*** depending on the instruction.
- The reason is that the address value ***WriteReg*** is stored:
  - in the **rt** field [20:16] in the case of an I-type instruction
  - in the **rd** field [15:11] in the case of an R-type instruction

Datapath enhancements for R-instructions



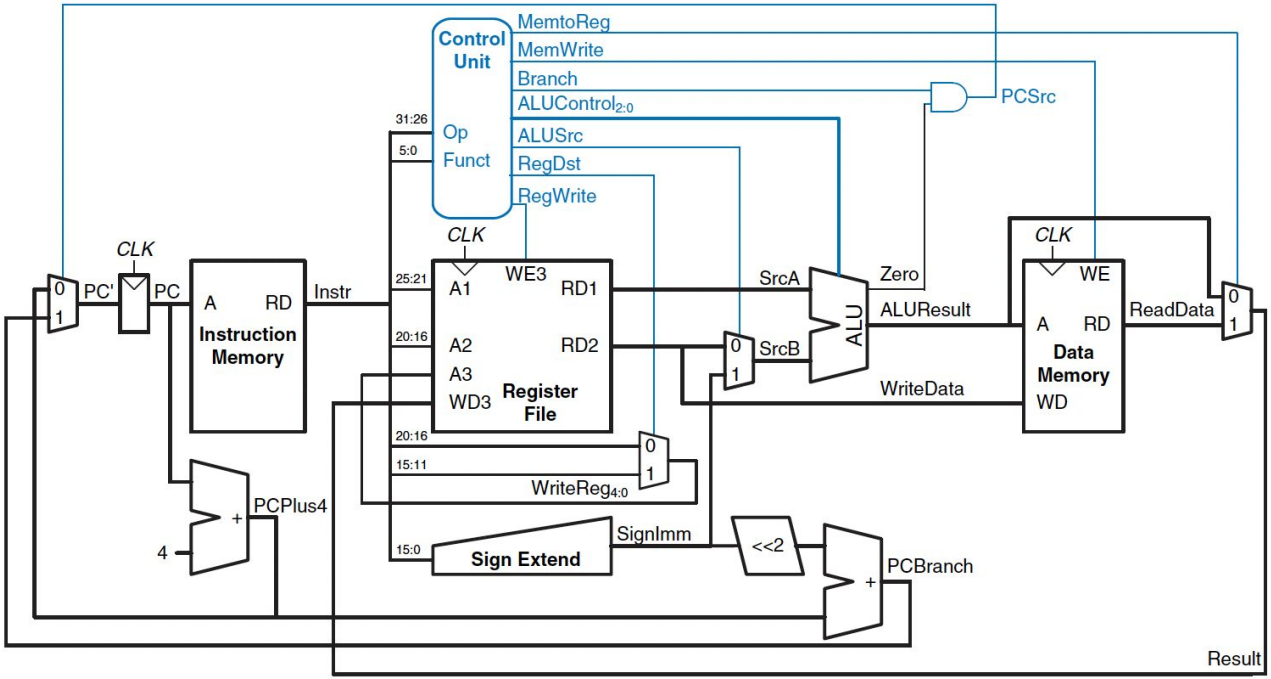
- Finally, we add a control signal ***MemtoReg*** and a multiplexer to select depending on the instruction whether the 32-bit data value ***Result*** is connected:
  - to the 32-bit data value ***ALUResult*** produced by the ALU
  - to the 32-bit data value ***ReadData*** produced by the data memory

Datapath enhancements for beq instruction



- The `beq` instruction compares two registers and branches if they are equal. We add the control signal ***Branch*** which is set to 1 during the `beq` instruction. Hence:
- If the signal ***Zero*** is equal to 1 then ***PC' = PC + 4 + SignImm ✕ 4***
- Otherwise: ***PC' = PC + 4*** and the next instruction is executed.

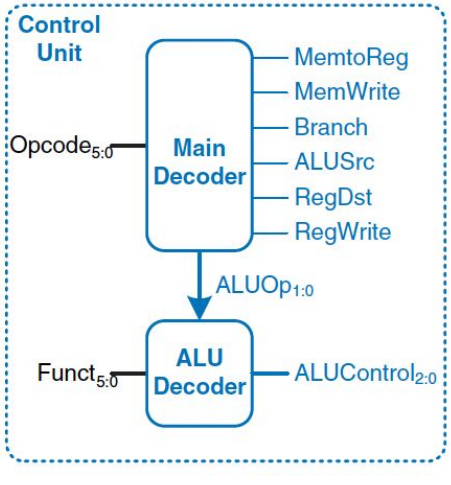
Complete single-cycle MIPS processor



The control unit computes the control signals based on the `opcode` and `funct` fields: ***Instr[31:26]*** and ***Instr[5:0]***.

The control signal controls the multiplexers and ALUs.

Control unit internal structure



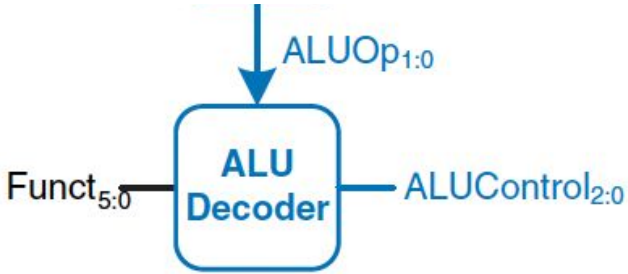
ALUOp	Meaning
00	add
01	subtract
10	look at `funct` field
11	n/a

Most of the control information comes from the `opcode`, and, in R-type instructions, it also uses the `funct` field.

To simplify the design, we can factor the control unit into two blocks of combinational logic.

- main **decoder**: it computes most of the outputs from the `opcode`, also, it determines a 2-bit ***ALUOp*** signal.
- ALU **decoder**: It uses the ***ALUOp*** signal in conjunction with the `funct` field to compute ***ALUControl.***

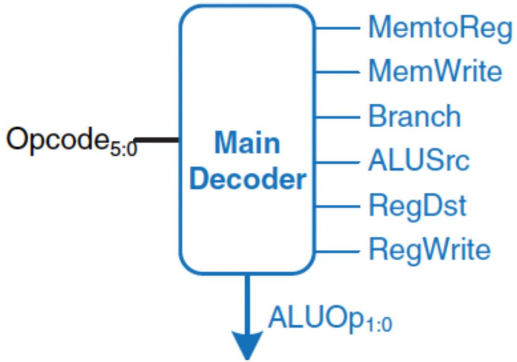
ALU decoder truth table



ALUOp	Funct	ALUControl
00	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 (set less than)

- When the ***ALUOp*** is 00 or 01, the **ALU** should add or subtract, respectively.
- When ***ALUOp*** is 10, the decoder examines the `funct` field to determine the ***ALUControl***.
  - For R-type instructions, the first two bits of the funct fields are always 10, so we ignore them to simplify the decoder.
- The ***ALUOp*** is never 11, so, the truth table can use don't care's ***X1*** and ***1X*** instead of 01 and 10 to simplify the logic. (i.e., we can decide the action when knowing one of the bits in the ALUOP is 1.)

Main decoder truth table



- All R-type instructions use the same main decoder values; they differ only in the ALU decoder output.
- Instructions that do not write to the register file, the ***RegDst*** and ***MemtoReg*** are set to don't cares (*X*).
- The decoder can be designed using different combinational logic designs.

Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01

Exercise 1

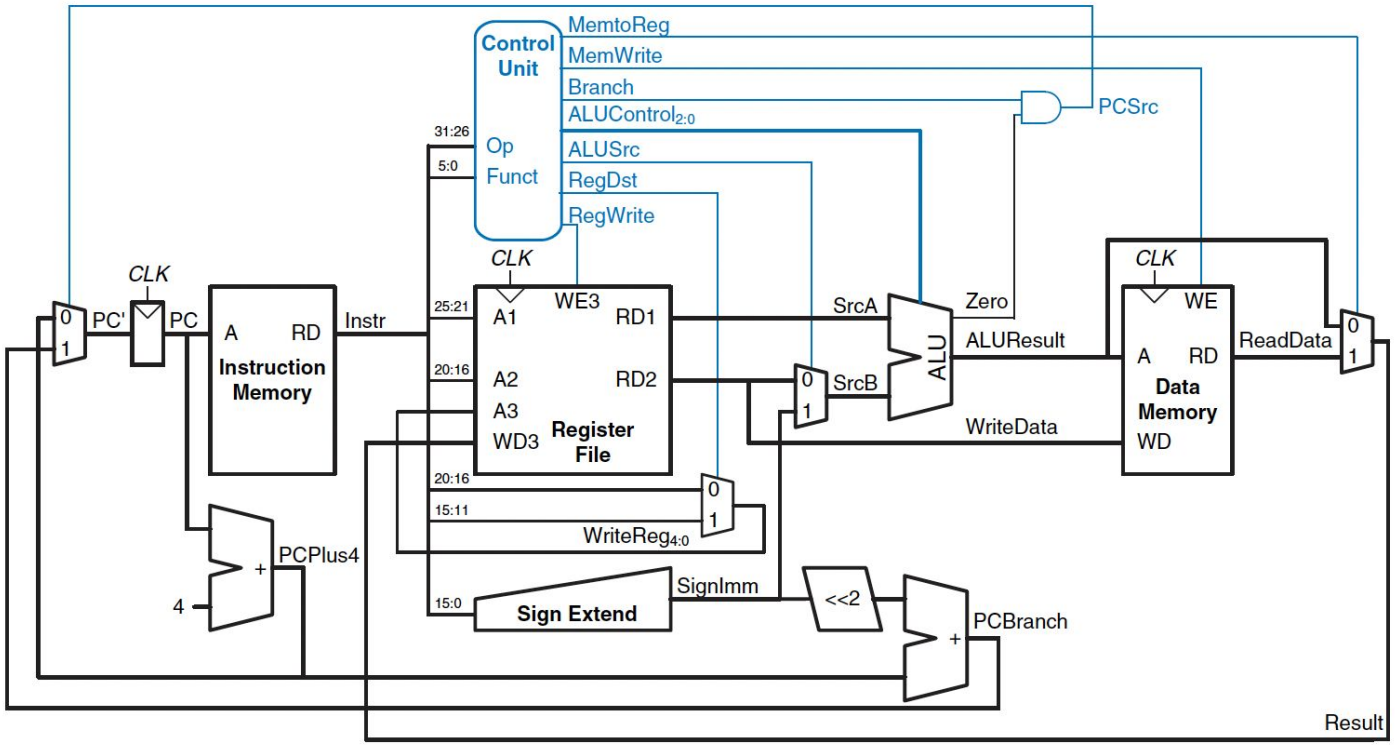
Describe the ***control***:

- Multiplexer select
- Register enable
- Memory write signals

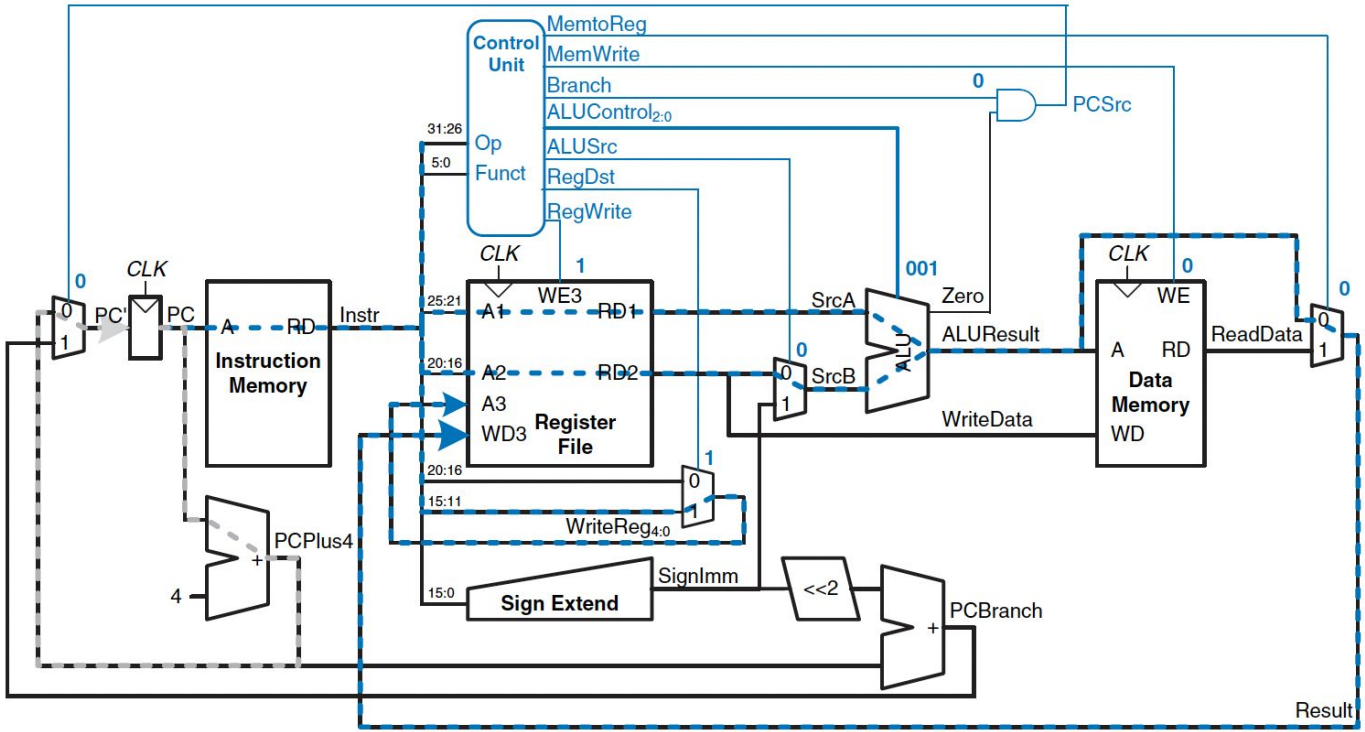
as well as the ***datapath*** which perform the `or` instruction.

- `or $t0, $t1, $t2`

Control and data flow while executing or instruction



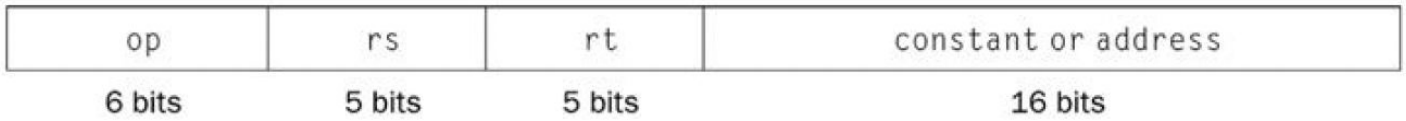
Control and data flow while executing or instruction



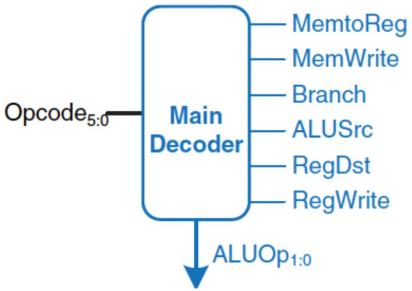
Exercise 2

How does one enhance the current microarchitecture in order to support `addi` instruction?

- `addi $rt $rs constant` #add the constant to `$rs` and store the result in `$rt`.



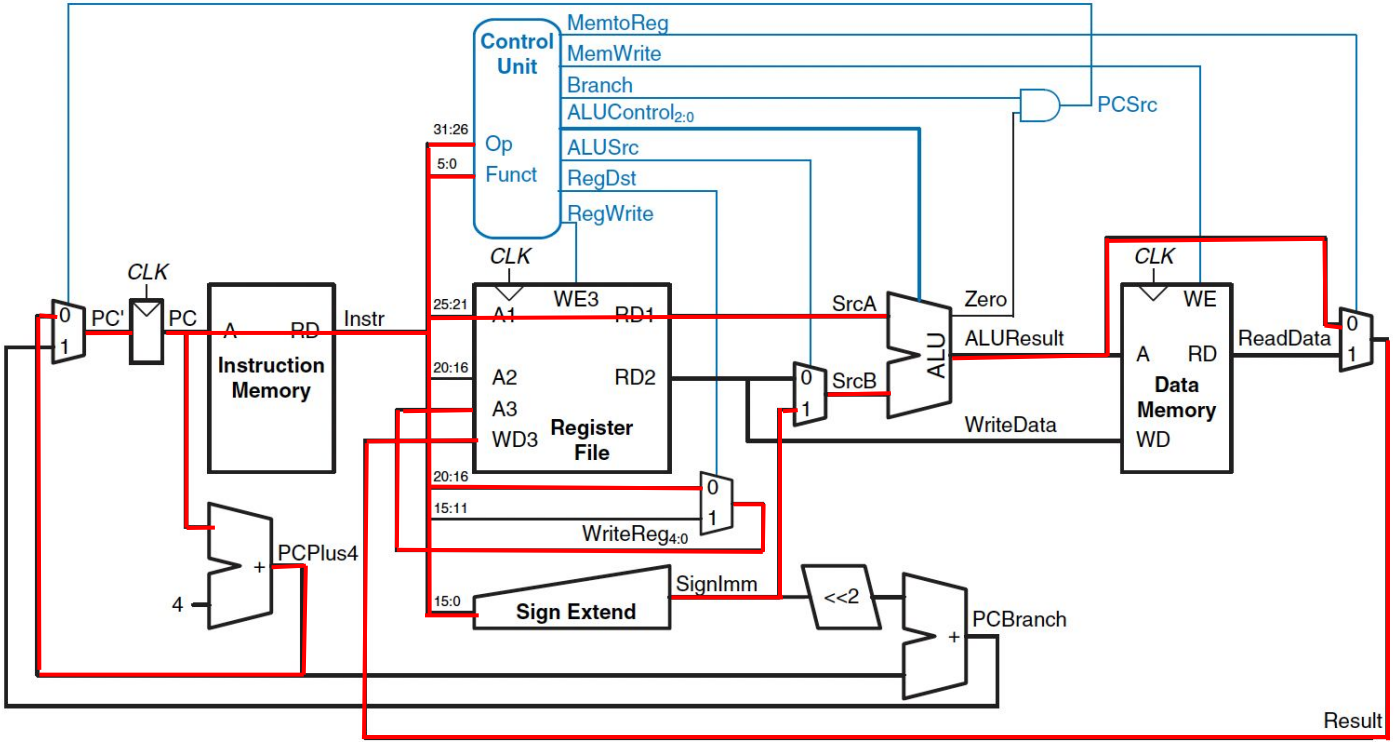
Control and data flow for the addi instruction



- `addi` is an **I-type** instruction;
- The result should be written to the register file, so ***RegWrite=1.***
- The destination register is specified in the **rt** field, so ***RegDst=0.***
- ***SrcB*** comes from the immediate, so ***ALUSrc=1.***
- Neither branch, nor does it write memory, so ***Branch=MemWrite=0***.
- The result comes from the **ALU**, not memory, so ***MemtoReg=0.***
- The **ALU** should add, so ***ALUOP=00***

Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	0	10
`lw`	100011	1	0	1	0	0	1	00
`sw`	101011	0	X	1	0	1	X	00
`beq`	000100	0	X	0	1	0	X	01
`addi`	001000	1	0	1	0	0	0	00

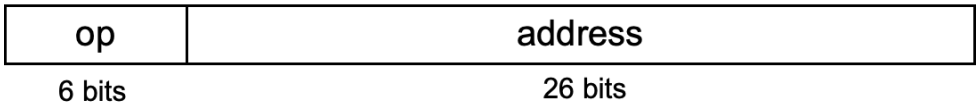
Control and data flow for the addi instruction



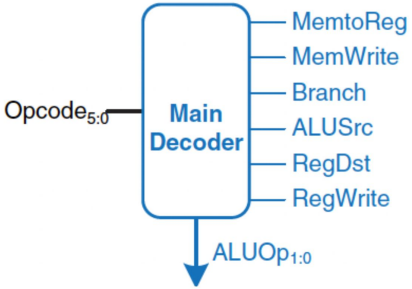
Exercise 3

How does one enhance the current microarchitecture in order to support the `j` instruction?

- `j Label` #jump to the address marked as Label



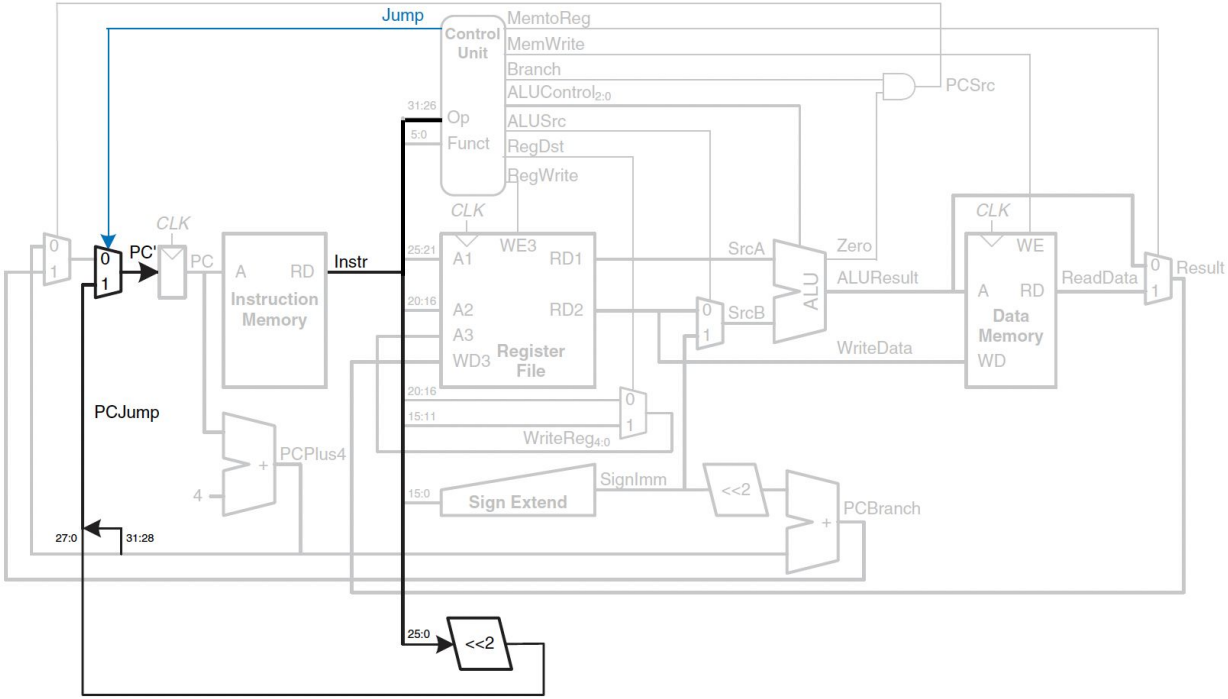
Control and data flow for the j instruction

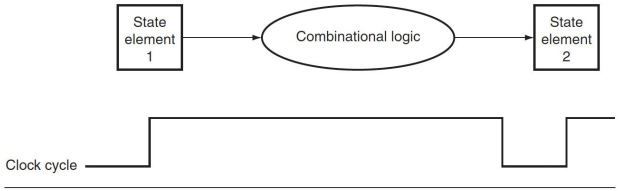


- `j` is an **J-type** instruction. It writes a new value into the ***PC***. The two least significant bits of the ***PC*** are always 0, because ***PC*** is word aligned (i.e., always a multiple of 4). The next 26 bits are taken from the jump address field in ***Instr[25:0]***. The upper four bits are taken from the old value of the ***PC***.
- So, we must add hardware to compute the next ***PC*** value in the case of a `j` instruction and a multiplexer to compute the next PC.

Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp	Jump
R-type	000000	1	1	0	0	0	0	10	0
`lw`	100011	1	0	1	0	0	1	00	0
`sw`	101011	0	X	1	0	1	X	00	0
`beq`	000100	0	X	0	1	0	X	01	0
`addi`	001000	1	0	1	0	0	0	00	0
`j`	000010	0	X	X	X	0	X	XX	1

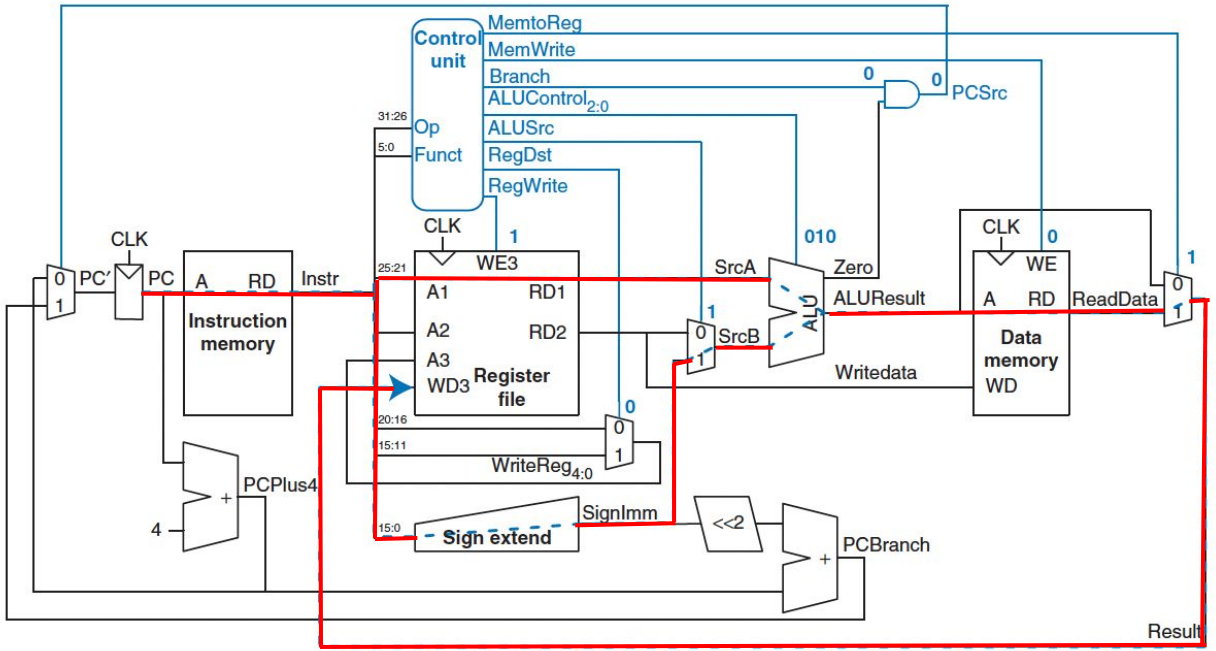
Control and data flow for the j instruction





Performance analysis

- Each instruction in the single-cycle processor takes one clock cycle, so the CPI is 1.



The figure shows the critical path for the `lw` instruction.

To complete the `lw` instruction, the cycle time should be

$$T_c = t_{pcq\_PC} + t_{mem} + max\{t_{RFread}, t_{sext} + t_{mux}\} + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$$

Other instructions have shorter critical path as they may not need to access the data memory. Since it is a synchronous sequential circuit, the clock cycle must be long enough to **accommodate the slowest** instruction, i.e., the `lw`.

Performance analysis

In most implementation technologies, the ALU, memory, and register file accesses are substantially slower than other operations. Therefore, the cycle time simplifies to

- $T_c = t_{pcq\_PC} + 2t_{mem} + 2t_{RFread} + t_{ALU} + t_{mux} + t_{RFsetup}$
- Given the delay of elements, we can calculate the cycle time of the single-cycle processor is $T_{c1} = 30+2(250)+150+200+25+20 = 925$.

If you want to run a program with 100 billion instructions, it will take,
$T_1 = (100 \times 10^9 \text{ instructions})\ (1 \text{ cycle/instruction})\ (925 \times 10^{-12} \text{s/cycle}) = 92.5 \text{ seconds}$

Table 7.6 Delays of circuit elements

Element	Parameter	Delay (ps)
register clk-to-Q	t_{pcq}	30
register setup	t_{setup}	20
multiplexer	t_{mux}	25
ALU	t_{ALU}	200
memory read	t_{mem}	250
register file read	t_{RFread}	150
register file setup	$t_{RFsetup}$	20