Imperial College London

Lecture 7

Microarchitecture

of a simplified RISC-V

Peter Cheung
Imperial College London

URL: www.ee.imperial.ac.uk/pcheung/teaching/EE2_CAS/
E-mail: p.cheung@imperial.ac.uk

In previous lectures, we focused on how to design digital hardware in SystemVerilog. In this lecture, we examine the internal hardware of a simplified version of RISC-V processor using the execution of one instruction as an example. From this, you should be able to gradually add to the microarchitecture implementing more and more instructions.

What is microarchitecture?

- **Microarchitecture:** how to implement an architecture in hardware
- Processor:
  - **Datapath:** functional blocks
  - **Control:** control signals

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

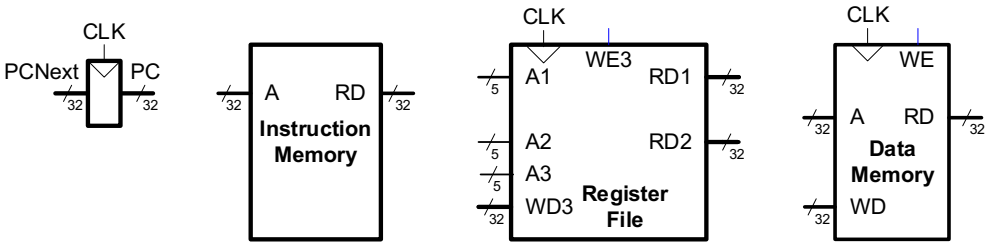
Microarchitecture refers to the detail digital circuits inside a processor that implements the ISA. There are two components to a microarchitecture:

The Datapath – this refers to the hardware components through which the instruction data or the information to be processed flow. This forms the bulk of the hardware ina processor. There are many strategy that one could use to implement the datapath. For example, we will start with a simple implementation where each instruction takes precisely ONE clock cycle. We then progress to an approach called pipelining, where several instructions may be executed at the same time in parallel but at different stages of execution.

The Control Unit – in designing the datapath, there are many signals that govern or control how data flows, and which part of the datapath circuit is enabled, and which is not. The control unit provides these control signals. In single cycle processor design, the control unit is mostly performing instruction decoding. In a multi-cycle and pipelined design, the control unit also implements a FSM to keep track of what state the CPU is progressing at in different stages of pipelining.

RISC-V State Elements

- **State elements**: determines everything about a processor:
  - **Architectural state:**
    - 32 registers
    - Program Counter (PC)
    - Memory



Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

In a processor, there are four components that determine the state of the CPU. They are:

The Program Counter – this is a counter that provides the address of the current instruction being executed.

The Instruction Memory – this stores the program code to the processor.

The Register File – this implements the registers of the processor and is always implemented as a multiport memory.

The Data Memory – this stores the data or information for processing.

Example Program

- Design datapath
- View example program executing

Address	Instruction	Type	Fields						Machine Language
			$imm_{11:0}$		rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100		01001	010	00110	0000011	FFC4A303
			$imm_{11:5}$	rs2	rs1	f3	$imm_{4:0}$	op	
0x1004	sw x6, 8(x9)	S	0000000	00110	01001	010	01000	0100011	0064A423
			funct7	rs2	rs1	f3	rd	op	
0x1008	or x4, x5, x6	R	0000000	00110	00101	110	00100	0110011	0062E233
			$imm_{12,10:5}$	rs2	rs1	f3	$imm_{4:1,11}$	op	
0x100C	beq x4, x4, L7	B	1111111	00100	00100	000	10101	1100011	FE420AE3

`lw x6, -4(x9)`
`lw rd, imm(rs1)`

I-Type

31:20	19:15	14:12	11:7	6:0
$imm_{11:0}$	rs1	funct3	rd	op
12 bits	5 bits	3 bits	5 bits	7 bits

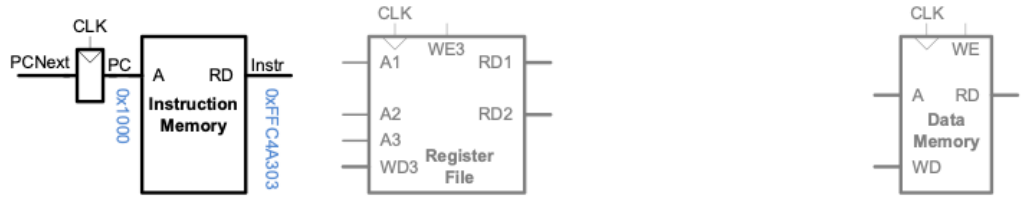
Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

As seen from last lecture, the RISC-V RV32I processor has four many types of instructions. The code snippet shown here covers all four type of instructions. We will focus in the first instruction : `lw x6, -4(x9).`

This instruction does the following: load Register 6 with the contents of from data memory at address specified by Register 9 with an offset of -4.

This is an I-type instruction because the offset -4 is specified in the instruction as a 12-bit immediate value. The load word (lw) instruction is specified by the opcode (instr[6:0]) and funct3 (instr[14:12]) fields.

Step 1: Instruction Fetch



Address	Instruction	Type	Fields					Machine Language
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

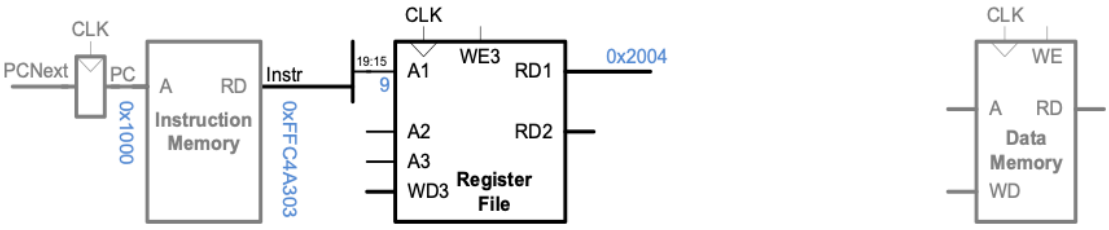
Consider what happens when this instruction is executed.

Step 1 is to **fetch the instruction** from instruction memory. The instruction is stored at address 0x1000. The machine code of the instruction is presented to the instruction memory which asynchronously (i.e. immediately) produces the 32-bit instruction 0xFFC4A303.

We use a block asynchrous memory here because the instruction must be complete in clock cycle. Therefore the instruction information is required immediately on the active edge of the clock signal. The delays incurred by this step are:

1. The clock to PC delay of the counter.
2. The address to data access time of the instruction memory block.

Step 2: Read Source Operand (rs1)

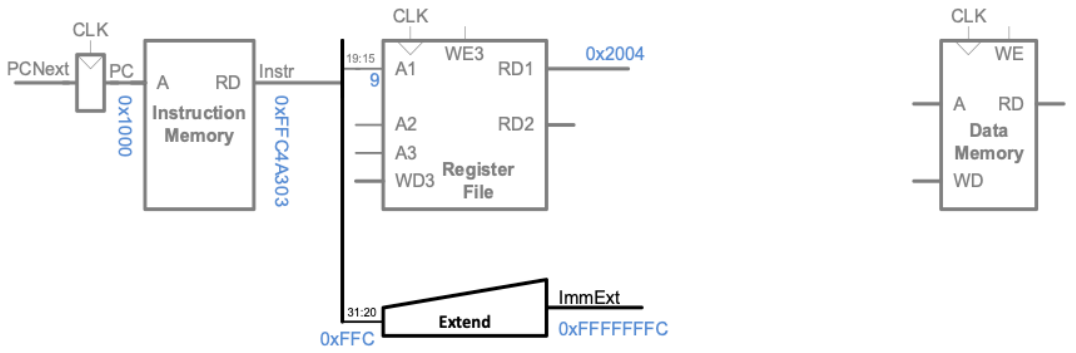


Address	Instruction	Type	Fields					Machine Language
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

In Step 2 is to retrieve the address pointer to data memory using the contents of x9. The rs1 field of the instruction (instr[19:15]) is always connected to A1 address of the Register File. The contents of x9 is provided on RD1 port. It is 0x2004, which will be used to calculate the address of data memory to read from.

Step 3: Extend the immediate constant



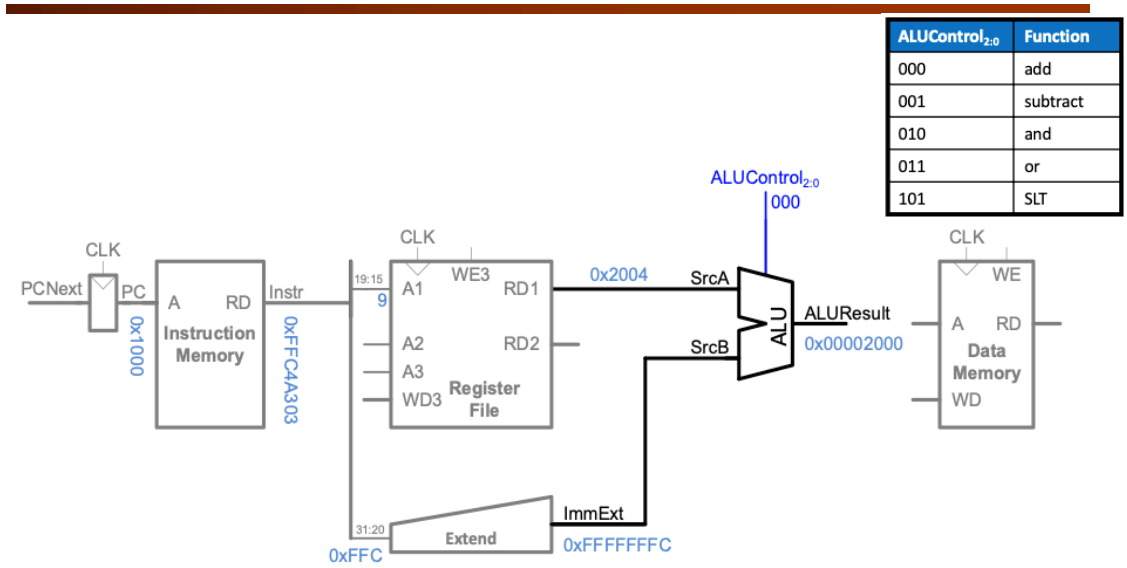
Address	Instruction	Type	Fields					Machine Language
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

Step 3 is to compute the immediate value from this instruction, which will be used as an offset to the address from x9.

The 12-bit immediate constant field (instr[31:20]) is used as the bottom 12 bits of a 32-bit 2's complement offset. This offset is -4 which has a 12-bit value of 0xFFC. This number is sign extended to provide ImmExt value of 0xFFFFFFFC as the offset address.

Step 4: Calculate memory address

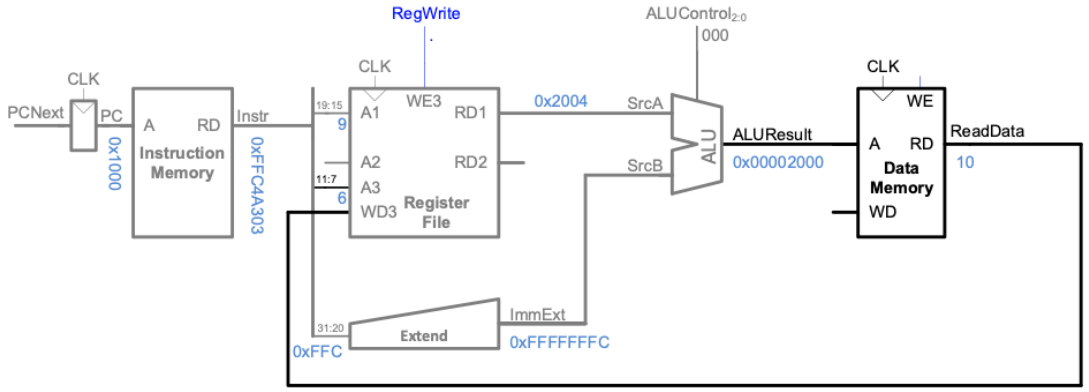


ALUControl$_{2:0}$	Function
000	add
001	subtract
010	and
011	or
101	SLT

Address	Instruction	Type	Fields				Machine Language
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

The offset address is passed to the ALU which performs a 32-bit addition. This ALU is also used for other arithmetic and logic operations. The little table above shows the funct3 instruction field that determines which operation is to be performed. 0x000 is for addition. ALU adds contents of x9, which is 0x2004, and the immediate offset value of -4 (0xFFFFFFFC) together to produce the effective address of 0x2000. This is address to data memory where the load word instruction is going to read from.

Step 5: Read data from memory & write to Reg



Address	Instruction	Type	Fields					Machine Language
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

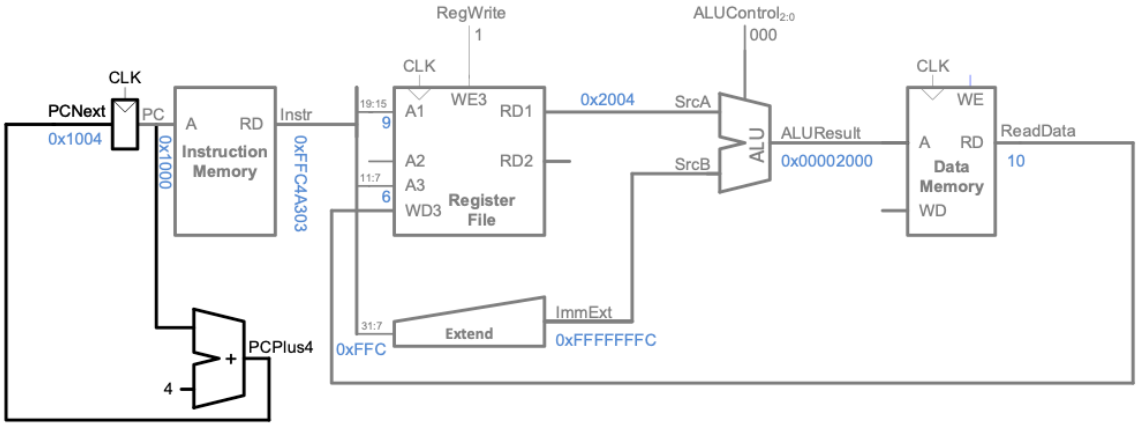
Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

The final step is to read the data from the memory address 0x2000, and store it in x6.

All read ports in both instruction, register file and data memory are asynchronous meaning that the read data is available as soon as the address is presented. The clock signal is ony used to control three things: 1) PC counter update; 2) Register file write; 3) Data memory write.

All these changes happens on the rising edge of the next instruction. That is, the Program Counter changes on the next rising edge of the clock. This is also the time that the register is updated with the new write value, and data memory is written to. If the register write operation is not synchronous to the next clock edge, there will be the potential of a race condition where there is a feedback loop that changes in the register file value continuously within a clock cycle. This is obviously not correct. For example, for the instruction: addi x3, zero, 1, which is equivalent to increment x3 value by 1, without the synchronous writing operation, x3 will be incremented continuously through the duration of the current clock cycle.

Step 6: Determine address of next instruction



Address	Instruction	Type	Fields				Machine Language
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

Step 6 is the update of the Program Counter. Since this instruction does not change the flow of the program, the next PC value must be the current PC + 4.

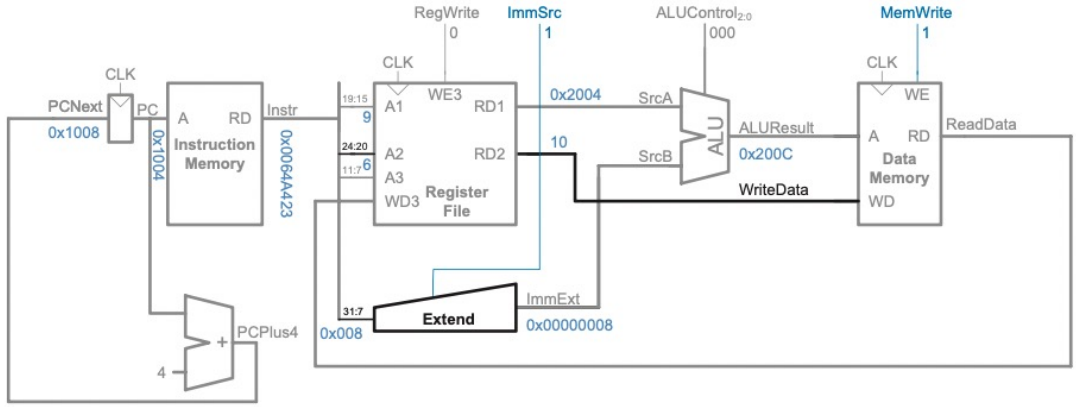
Now the instruction is complete.

Note also that although we divide the execution of this instruction into six steps, in reality, they occur "simultaneously" because this is hardware. For example RD1 value and ImmExt value are derived in parallel, but ALUResult cannot be computed until the two input source values to the ALU are stable.

Implementation of the ”sw” instruction

- **Immediate:** now in {instr[31:25], instr[11:7]}
- **Add control signals:** ImmSrc, MemWrite

Address	Instruction	Type	Fields						Machine Language
			$imm_{11:5}$	rs2	rs1	f3	$imm_{4:0}$	op	
0x1004	sw x6, 8(x9)	S	0000000	00110	01001	010	01000	0100011	0064A423



Based on: “*Digital Design and Computer Architecture (RISC-V Edition)*” by Sarah Harris and David Harris (H&H),

The second instruction is: `sw x6, 8(x9)`, which is an S-type instruction. This is similar to ”load word” instruction involving two registers and an immediate offset. However, the the destination is not a register but a memory location. Therefore the immediate constant (of the offset) is split into two parts as shown below.

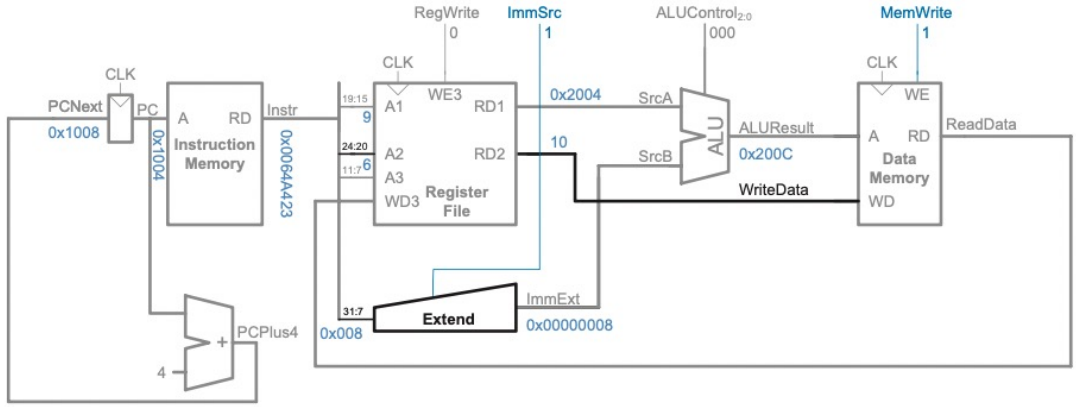
Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						

The actual hardware of the microarchitecture does not change except that we need a different control signal for the sign-extension unit (to generate the correct immediate offset), and for writing to data memory.

Implementation of the ”sw” instruction

- **Immediate:** now in {instr[31:25], instr[11:7]}
- **Add control signals:** ImmSrc, MemWrite

Address	Instruction	Type	Fields						Machine Language
			$imm_{11:5}$	rs2	rs1	f3	$imm_{4:0}$	op	
0x1004	sw x6, 8(x9)	S	0000000	00110	01001	010	01000	0100011	0064A423



Based on: “*Digital Design and Computer Architecture (RISC-V Edition)*” by Sarah Harris and David Harris (H&H),

The second instruction is: `sw x6, 8(x9)`, which is an S-type instruction. This is similar to ”load word” instruction involving two registers and an immediate offset. However, the the destination is not a register but a memory location. Therefore the immediate constant (of the offset) is split into two parts as shown below.

Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						

The actual hardware of the microarchitecture does not change except that we need a different control signal for the sign-extension unit (to generate the correct immediate offset), and for writing to data memory.

Immediate offset for I-type and S-type are different

Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						

ImmSrc	ImmExt	Instruction Type
0	{{20{instr[31]}}, **instr[31:20]**}	I-Type
1	{{20{instr[31]}}, **instr[31:25], instr[11:7]**}	S-Type

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

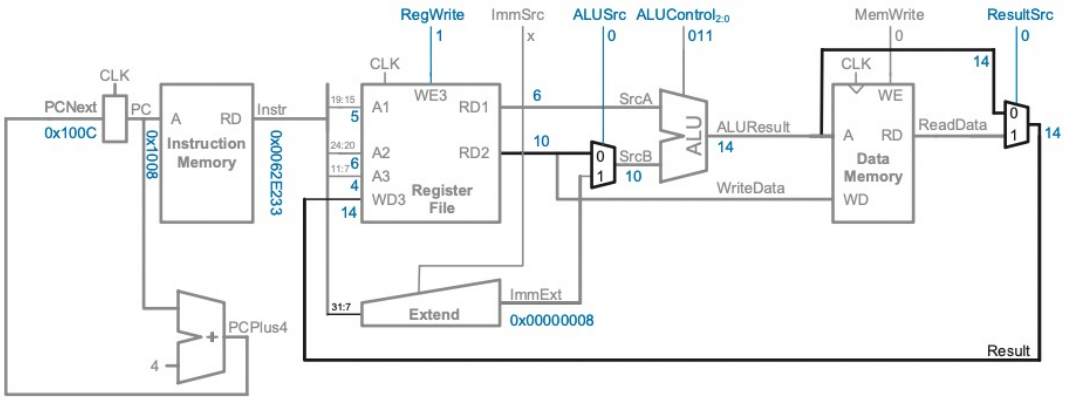
To reiterate, I and S-type instructions are similar in encoding except that they make up the 12-bit immediate offset constant using different bits in the instruction as shown here.

Using the concatenation operator in SystemVerilog {…}, it is easy construct the sign extension unit to cater for either type of instruction as shown here.

Implementation of the ”or” instruction

- Read from **rs1** and **rs2** (instead of **imm**)
- Write *ALUResult* to **rd**

Address	Instruction	Type	Fields					Machine Language
			funct7	rs2	rs1	f3	rd	op	
0x1008	or x4, x5, x6	R	0000000	00110	00101	110	00100	0110011	0062E233



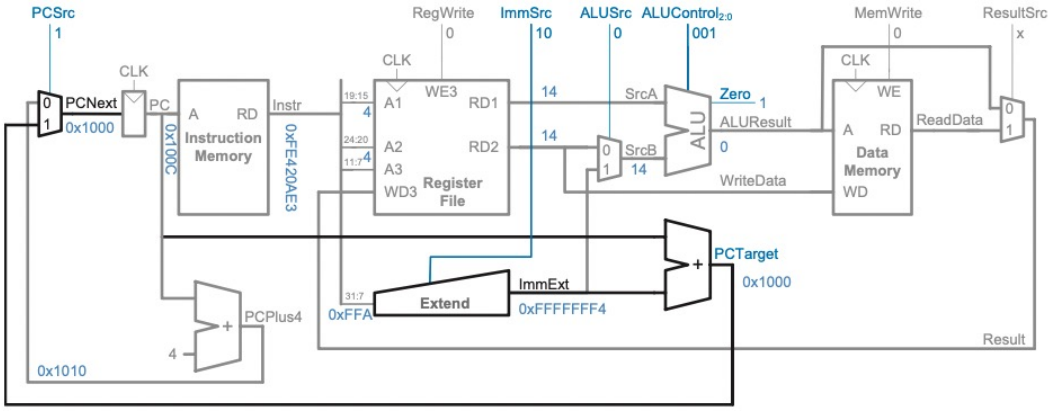
Based on: “*Digital Design and Computer Architecture (RISC-V Edition)*” by Sarah Harris and David Harris (H&H),

The third instruction is an R-type instruction: or x4, x5, x6. This instruction does not involve the data memory. This require the introduction of the two highlighted MUX component. First to select the Register data 2 instead of the immediate value. Second MUX select the ALU results to write back instead of the data memory (as in the as “lw” instruction). The rest of the hardware remains the same.

Implementation of the ”beq” instruction

Calculate **target address:** PCTarget = PC + imm

Address	Instruction	Type	Fields						Machine Language
			$imm_{12,10:5}$	rs2	rs1	f3	$imm_{4:1,11}$	op	
0x100C	beq x4, x4, L7	B	1111111	00100	00100	000	10101	1100011	FE420AE3



Based on: “*Digital Design and Computer Architecture (RISC-V Edition)*” by Sarah Harris and David Harris (H&H),

The final instruction in this simple program is the “branch if equal” instruction. Here the addition circuit is an adder which computes the target PC address as PC +Imm.

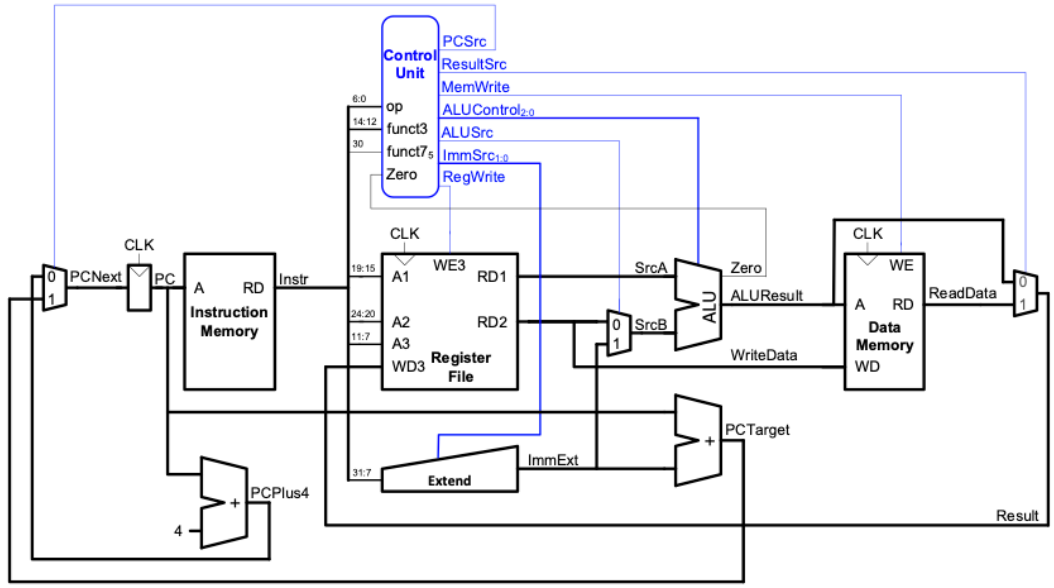
As discussed in Lecture 6, the immediate value for the PC-relative branch instruction is made up from various part of the instruction in a rather weird way. (See Lecture 6 slide 20 for detail explanation).

In summary , there are three different ways to compose the immediate value for I, S and B type instructions as summarized here:

Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Immediate	imm[11:0]												rs1					funct3			rd					opcode							
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						

ImmSrc	ImmExt	Type	Description
00	{{20{*Instr*[31]}}, *Instr*[31:20]}	I	12-bit signed immediate
01	{{20{*Instr*[31]}}, *Instr*[31:25], *Instr*[11:7]}	S	12-bit signed immediate
10	{{20{*Instr*[31]}}, *Instr*[7], *Instr*[30:25], *Instr*[11:8], 1’b0}	B	13-bit signed immediate

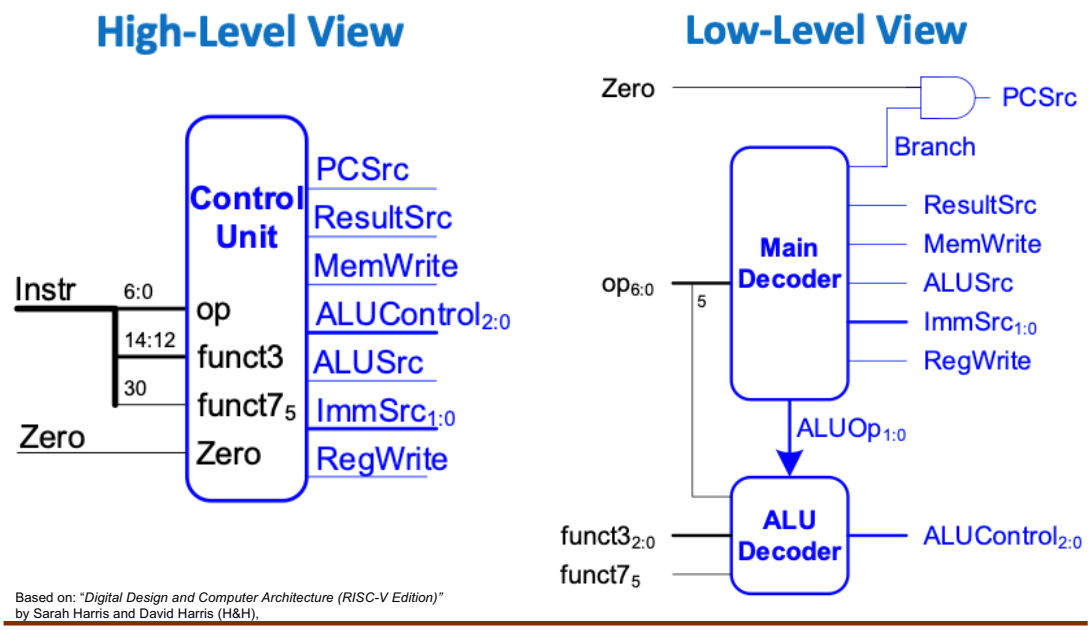
Adding the Control Unit



Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

Next, consider the Control Unit that generates all the control signals to the Datapath circuit.

Two different views of the Control Unit

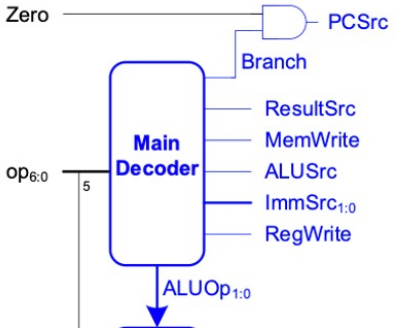


Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

The control unit can be divided into two separate, relatively independent parts:

1. The Main Decoder, which generates most of the control signals depending on the opcode field.
2. The ALU decoder which controls the ALU operation using opcode, funct3 and funct7 field.

Main decoder

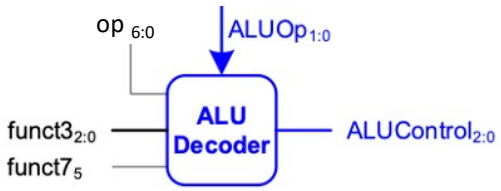


Instruction	Op	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
lw	0000011	1	00	1	0	1	0	00
sw	0100011	0	01	1	1	x	0	00
R-type	0110011	1	xx	0	0	0	0	10
beq	1100011	0	10	0	0	x	1	01

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

The main decoder used to determine the instruction type (i.e. I, S, R or B etc.), and for each type of instruction the datapath for the operands are different according to the true table here.

ALU Decoder



ALUOp	funct3	{op5, funct75}	ALUControl	Instruction
00	x	x	000 (add)	lw, sw
01	x	x	001 (subtract)	beq
10	000	00, 01, 10	000 (add)	add
	000	11	001 (subtract)	sub
	010	x	101 (set less than)	slt
	110	x	011 (or)	or
	111	x	010 (and)	and

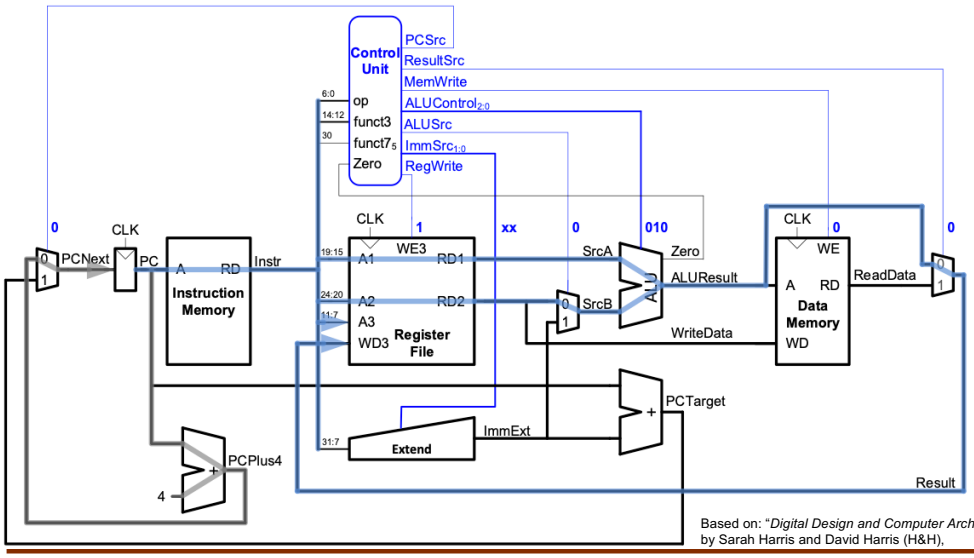
Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

The ALU Decoder unit controls the ALU and determines the type of ALU operations that it should perform. There are three ALU types of operations determined by funct3 (and in two cases, also funct7 bit 5):

1. lw, sw, where the ALU is used to computer the memory address (with an address pointer from Register and an immediate offset).
2. The branch equal instruction that performs a subtraction (or comparison).
3. The other ALU operations.

Example – Control for `and x5, x6, x7`

op	Instruct	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
51	**R-type**	1	XX	0	0	0	0	010



Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

Before we leave the single cycle RISC-V microarchitecture, let us examine the control signals required to implement the R-type AND instructions.

The control signal values are labelled in the diagram here with the datapath highlighted in bold BLUE lines.

Lab 4 – A Very Basic RISC-V CPU

- Start working as a Team – 2 pairs allocated by me
- **Lab objectives**:
  1. To get to know your teammates.
  2. To establish a Github Repo for your team where everyone's contribute towards.
  3. To learn about TWO RISC-V instructions in great details.
  4. To design a simple CPU that executes these two instructions.
  5. To use execute a short program using only these two instructions. The program implements the binary counter in Lab 1, but in software.
  6. Stretched goal – to implement a third instruction accessing data memory. With this, implement the sinewave generator in software.

Lab 4 is design with a number of goals in mind. This also form the basis for your Team Project, which is the main coursework assignment to be done by all as Teams of four students.

I would recommend you to complete everything including the stretched goal. It will force you to learn how to implement three of the four main type of instructions: I-type, B-type and S-type. R-type instructs are not include in this Lab, but is rather easy to implement.

Lab 4 – Program to execute

```
1  main:
2      addi    t1, zero, 0xff      # load t1 with 255
3      addi    a0, zero, 0x0       # a0 is used for output
4  mloop:
5      addi    a1, zero, 0x0       # a1 is the counter, init to 0
6  iloop:
7      addi    a0, a1, 0           # load a0 with a1
8      addi    a1, a1, 1           # increment a1
9      bne     a1, t1, iloop       # if a1 = 255, branch to iloop
10     bne     t1, zero, mloop     # else always branch to mloop
```

Online RISC-V Assembler:
https://riscvasm.lucasteske.dev

Hex Dump
0ff00313
00000513
00000593
00058513
00158593
fe659ce3
fe0318e3

This is the program that your Reduced RISC process need to execute. It performs exactly the same function as that of the simple binary counter, but in RISC-V instructions. The purpose of each instruction os described in the comments.

It is particularly interesting that we implement a binary up counter with only two instructions: addi and bne. This illustrates how “reduced” this architecture is!

The machine code for this program is shown as “Hex Dump”. This is produced by the online RISC-V assembler program – link given above.

Lab 4 – Pseudoinstruction is easier to read

```
1  main:
2      addi    t1, zero, 0xff
3      addi    a0, zero, 0x0
4  mloop:
5      addi    a1, zero, 0x0
6  iloop:
7      addi    a0, a1, 0
8      addi    a1, a1, 1
9      bne     a1, t1, iloop
10     bne     t1, zero, mloop
```

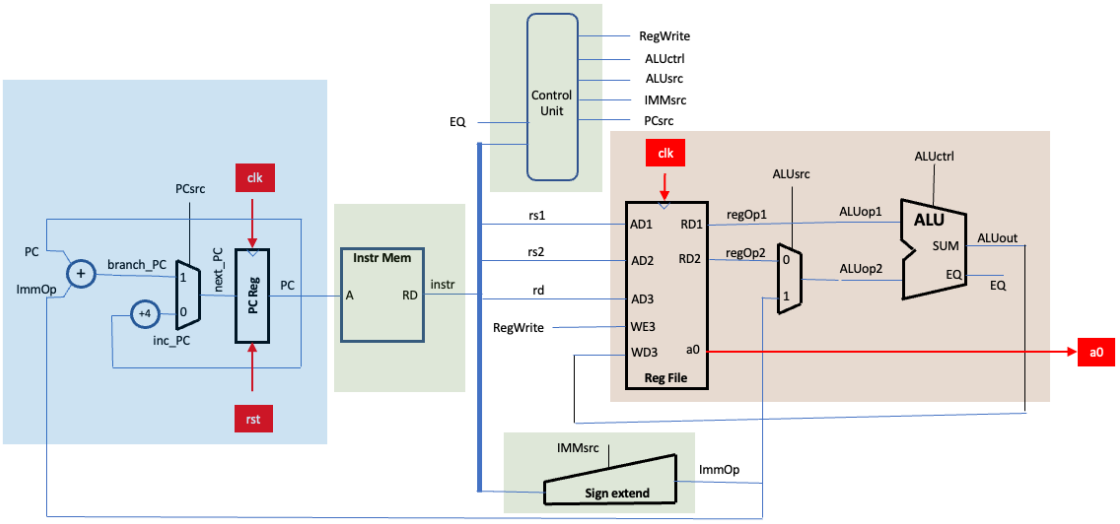
```
0000000000000000
:
   0:   0ff00313                li      t1,255
   4:   00000513                li      a0,0

0000000000000008 :
   8:   00000593                li      a1,0

000000000000000c :
   c:   00058513                mv      a0,a1
  10:   00158593                addi    a1,a1,1
  14:   fe659ce3                bne     a1,t1,c
  18:   fe0318e3                bnez    t1,8
```


This little program with the reduced RISC-V instruction is not easy to read. The dissassembled version using pseudoinstructions is shown alone side the original isntructios. “li” is load immediate. “mv” is moving values between registers. “bnez” is branch not equal zero.

Lab 4 – Overall block diagram



To help the Team making rapid progress in the right direction, you are given the overall block diagram of the Reduced RISC process along the ideas presented in this lecture.

The entire design is divided into three parts. One student should take charge of one of the three parts, with a fourth member of the Team looking after the testbench, the compilation script and the testing of the design.

This block diagram does not include the data memory, which will be required for the "Stretched Goal" of Lab 4.