SUSTech Southern University of Science and Technology

SME309—Microprocessor Design

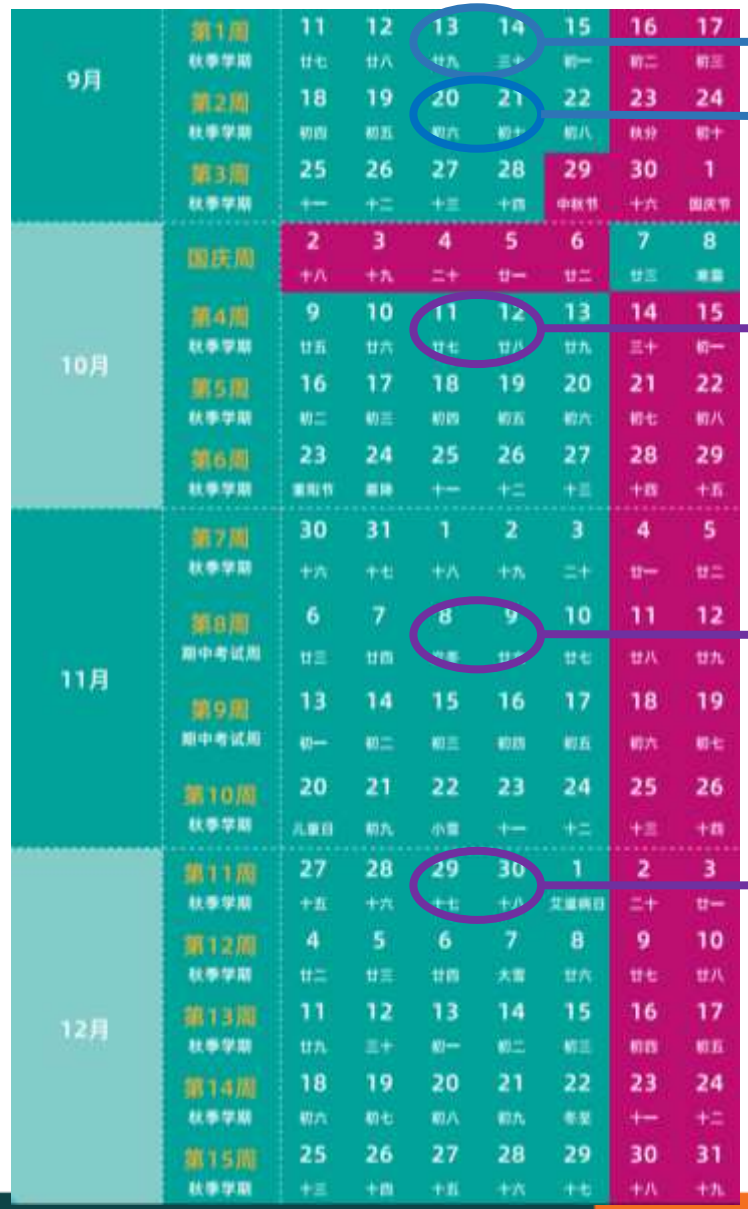
Lab2

Implementation of an ARMv3 Processor

2023.10

Lab Schedule

SUSTech Southern University of Science and Technology



Vivado installation and quick start. Verilog basic and practice.

Lab1 release

Lab1 checkpoint
Lab2 release

Lab2 checkpoint
Lab3 release

Lab3 checkpoint
Project release

Q & A

Q & A

Q & A

Q & A

Here Now

Content

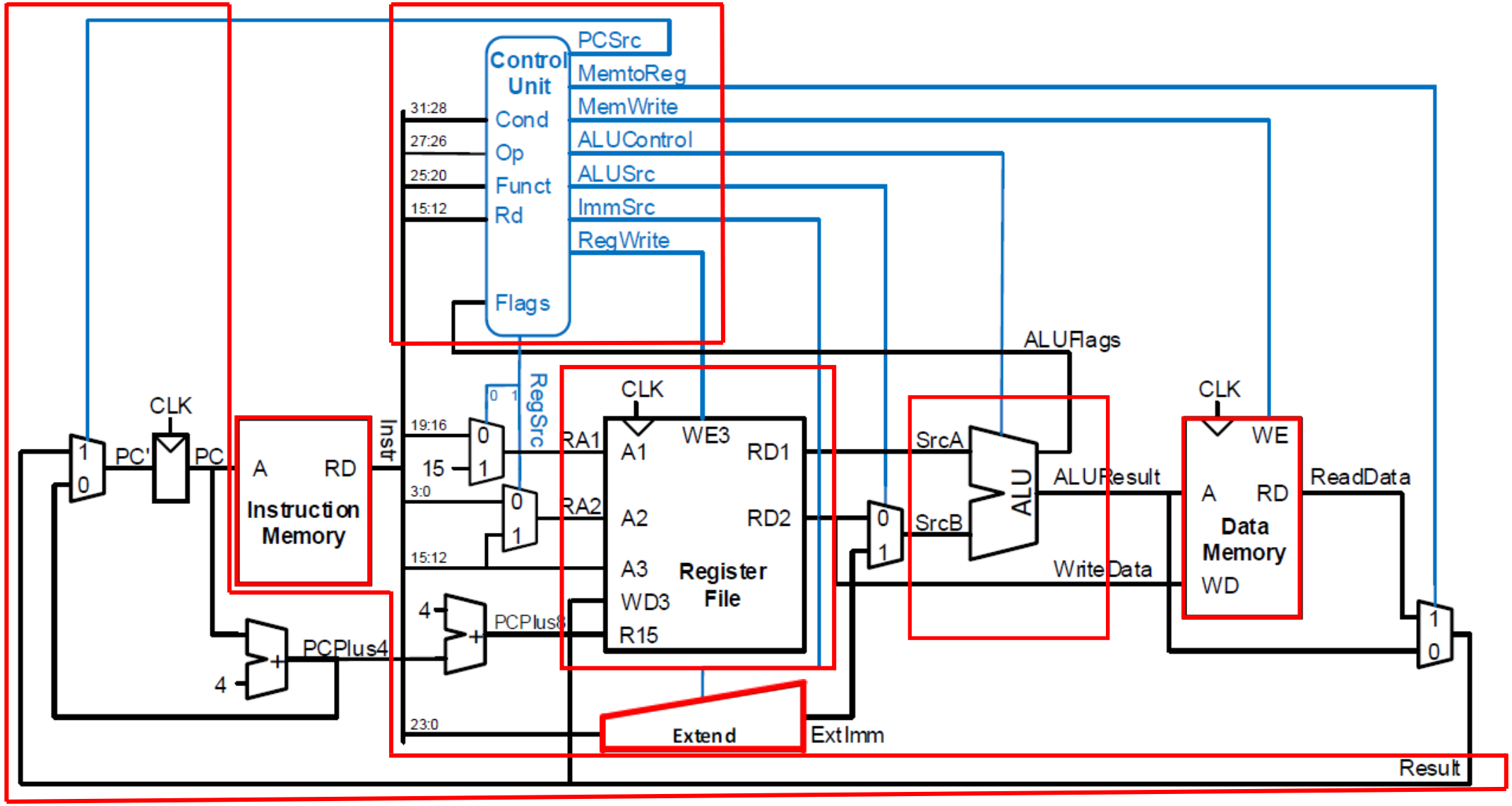
SUSTech Southern University of Science and Technology

1. **Objective & Overview**
2. **Submodule Details**
3. **How To Test**

Objective

- In this lab, you will be implementing an ARMcore you are learning in lectures.
- Essentially, it will support the following instructions
  - Memory instructions:
    - **LDR, STR**
    - With positive immediate offset(e.g. `LDR R6, [R4, #4]`)
  - Data-processing instructions:
    - **AND, OR, ADD, SUB**
    - with register or immediate `Scr2`, but no shifts(e.g. `ADD R9, R8, R5` or `SUB R3, R11, #1`)
  - Branch instructions:
    - **B**
- Further, improve the processor by adding the following features
  - Memory instructions:
    - **LDR, STR**
    - with negative immediate offset(e.g. `STR R6, [R4, #-4]`)
  - Data-processing instructions:
    - **CMP, CMN**
  - Data-processing instructions:
    - **AND, OR, ADD, SUB**
    - `Src2` support register with immediate shift(**LSL, LSR, ASR, ROR**) (e.g. `ADD R7, R2, R12, LSR #5`)

ARMcore Overview



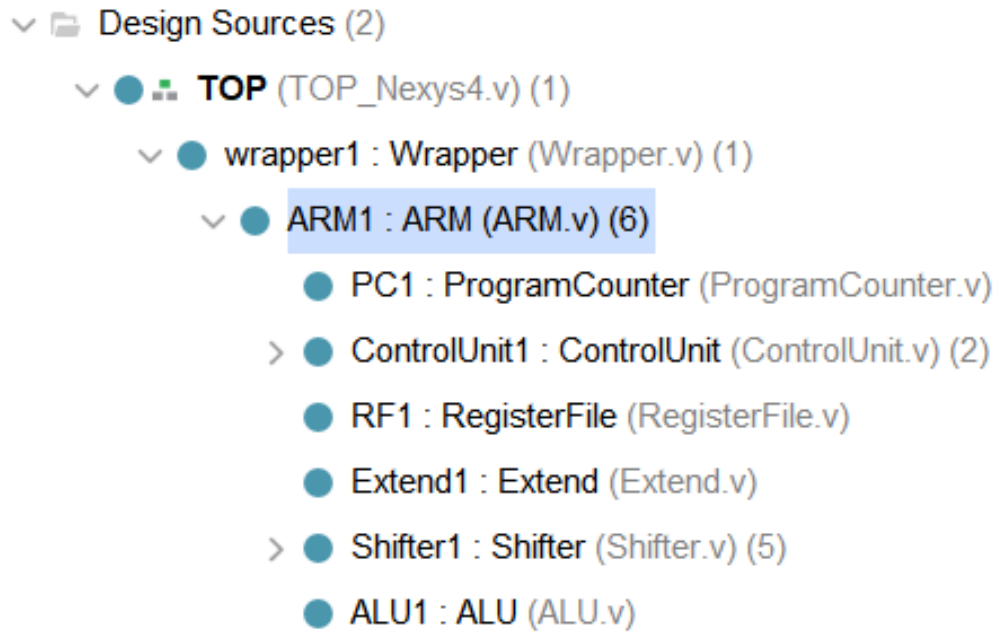
Project Structure

TOP_Nexys4.v, Wrapper.v and TOP.xdc are all completely provided so that you can directly use them to setup your project.

- *TOP_Nexys4.v: FPGA Onboard implementation for wrapper.v*
- *Wrapper.v: wrap up Memory and ARM Core together.*

For convenience to code check, recommend to complete by following:

- *Core Module: ARM.v*
- *Submodules: ProgramCounter.v, ControlUnit.v, RegisterFile.v, Extend.v, Shifter.v, ALU.v*




```
1  module ARM(
2      input CLK,
3      input Reset,
4      input [31:0] Instr,
5      input [31:0] ReadData,
6
7      output MemWrite,
8      output [31:0] PC,
9      output [31:0] ALUResult,
10     output [31:0] WriteData
11 );
```

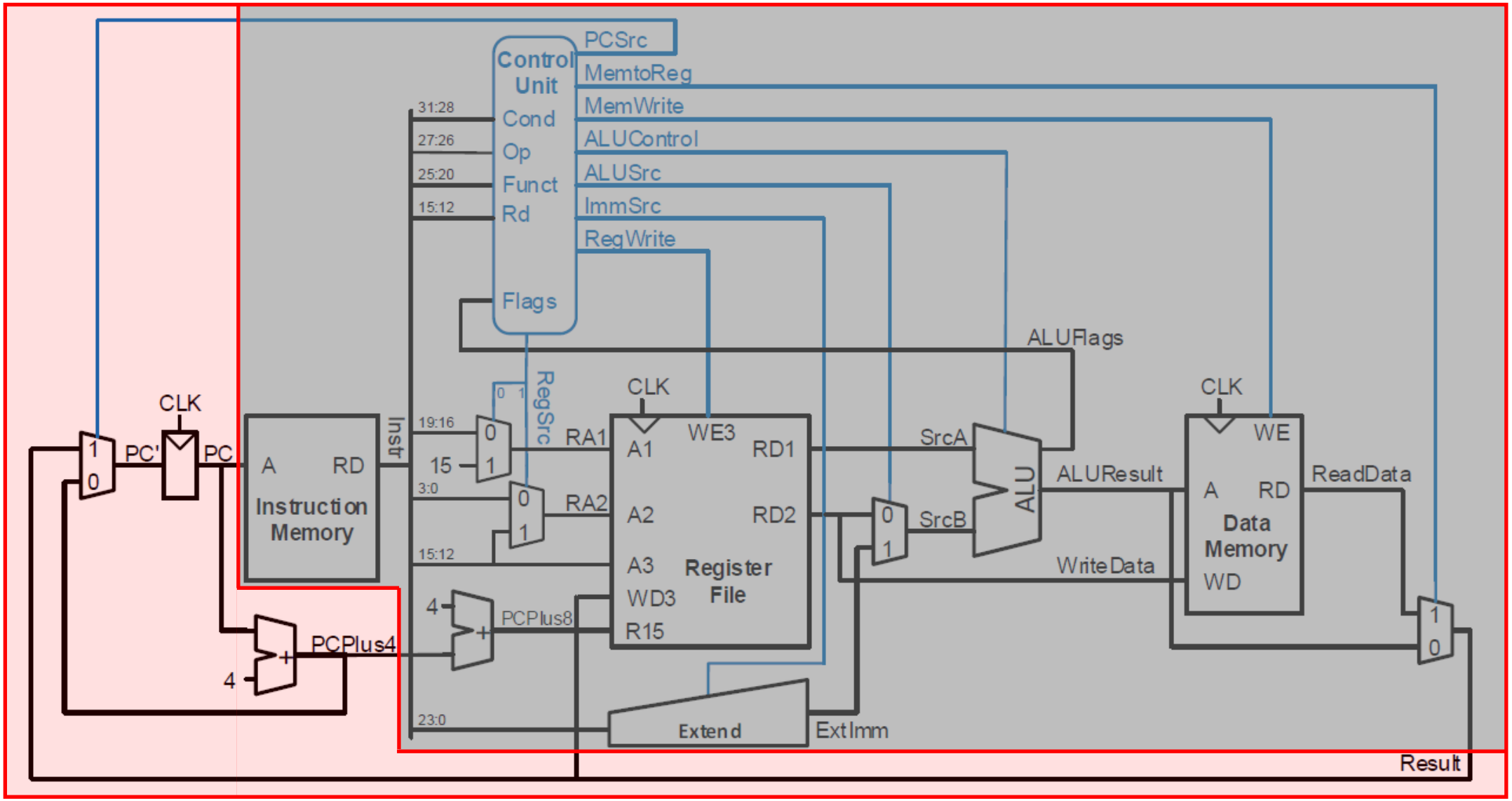
FPGA Onboard Demo

The role of FPGA peripherals:

- Switches: Represents the address of data memory (variable region).
- LEDs: Indicates the real-time PC value.
- SevenSegs: Indicates the data content of the data memory with the address corresponding to the switches.
- BTNU: Pause; BTND: Reset;

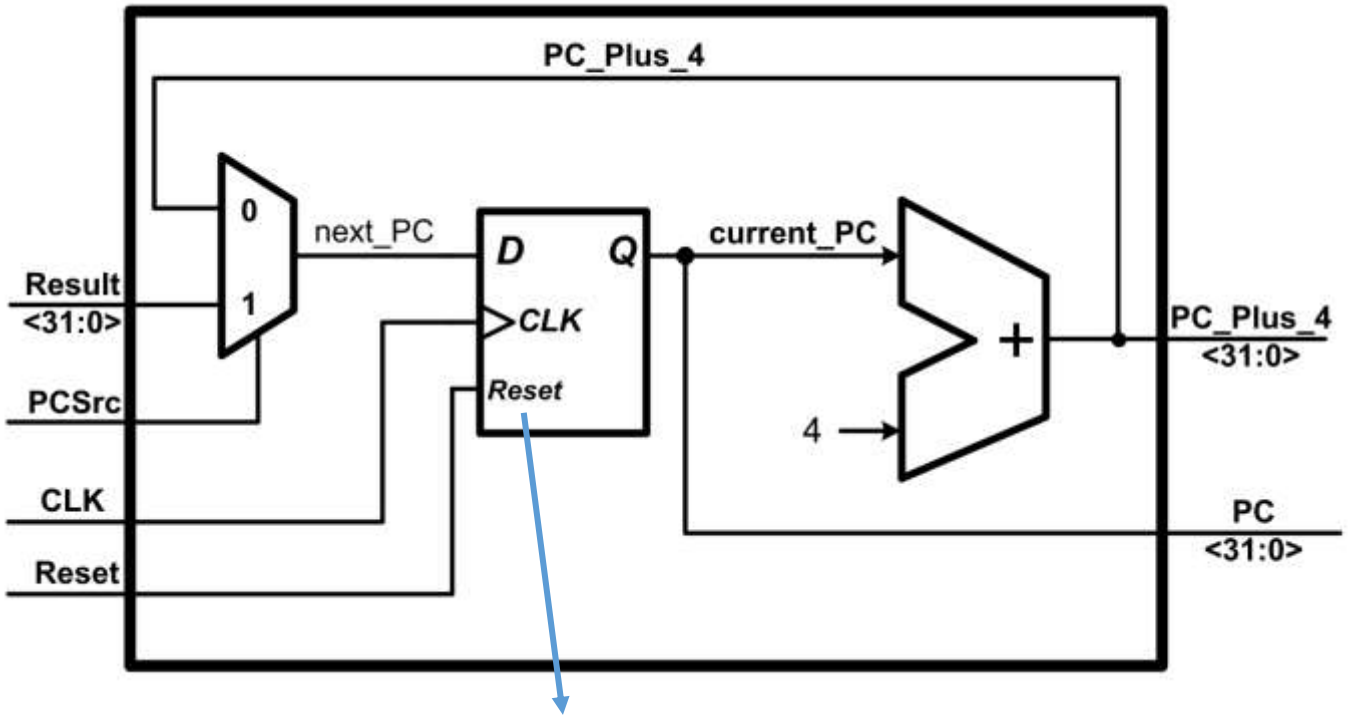
Content

Program Counter



Program Counter

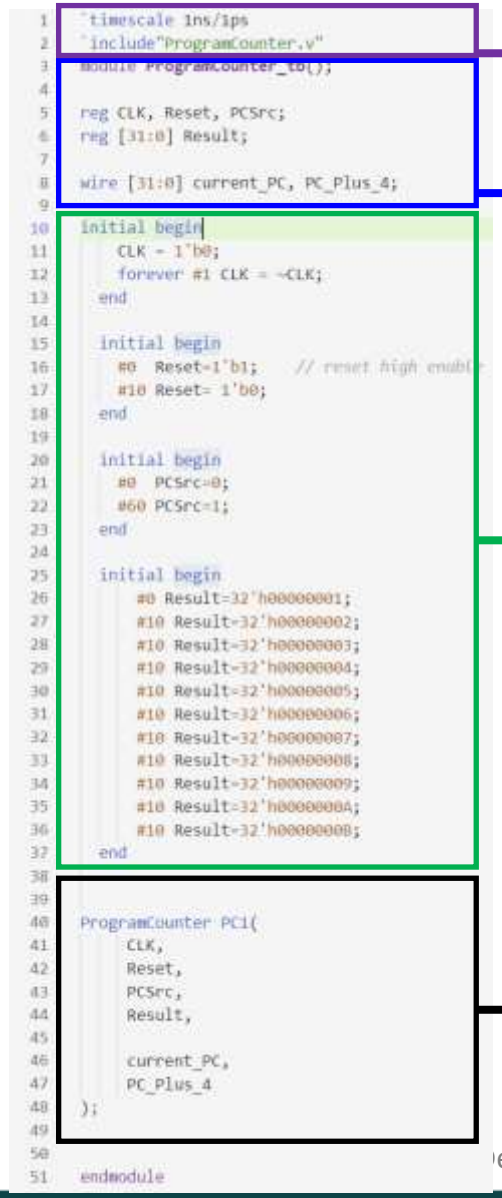
Module ProgramCounter



Asynchronous Reset

```
//ProgramCounter.v

module ProgramCounter(
    input CLK,
    input Reset,//reset is high-active
    input PCSrc, //PC source
    input [31:0] Result, //From ALU

    output reg [31:0] PC,
    output [31:0] PC_Plus_4
);

//fill your Verilog code here.

endmodule
```

Program Counter TestBench



Compile Option:
`timescale <time unit>/<resolution>

Signal Declaration:
Input of DUT should be declared as "reg", while output should be "wire"

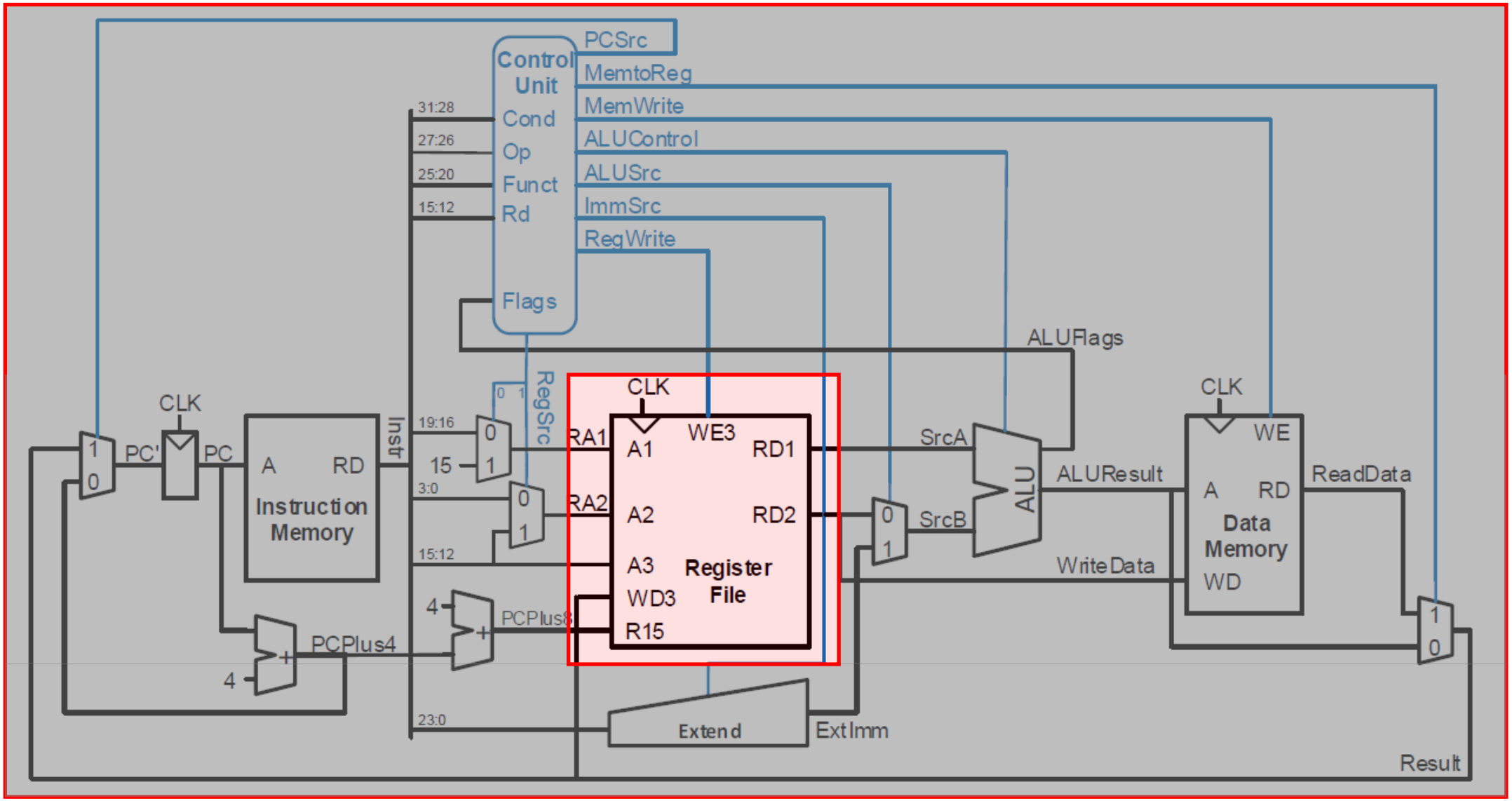
Stimuli Generation:
Generate proper input signal sequence (using initial/always statement)

Module Instantiation

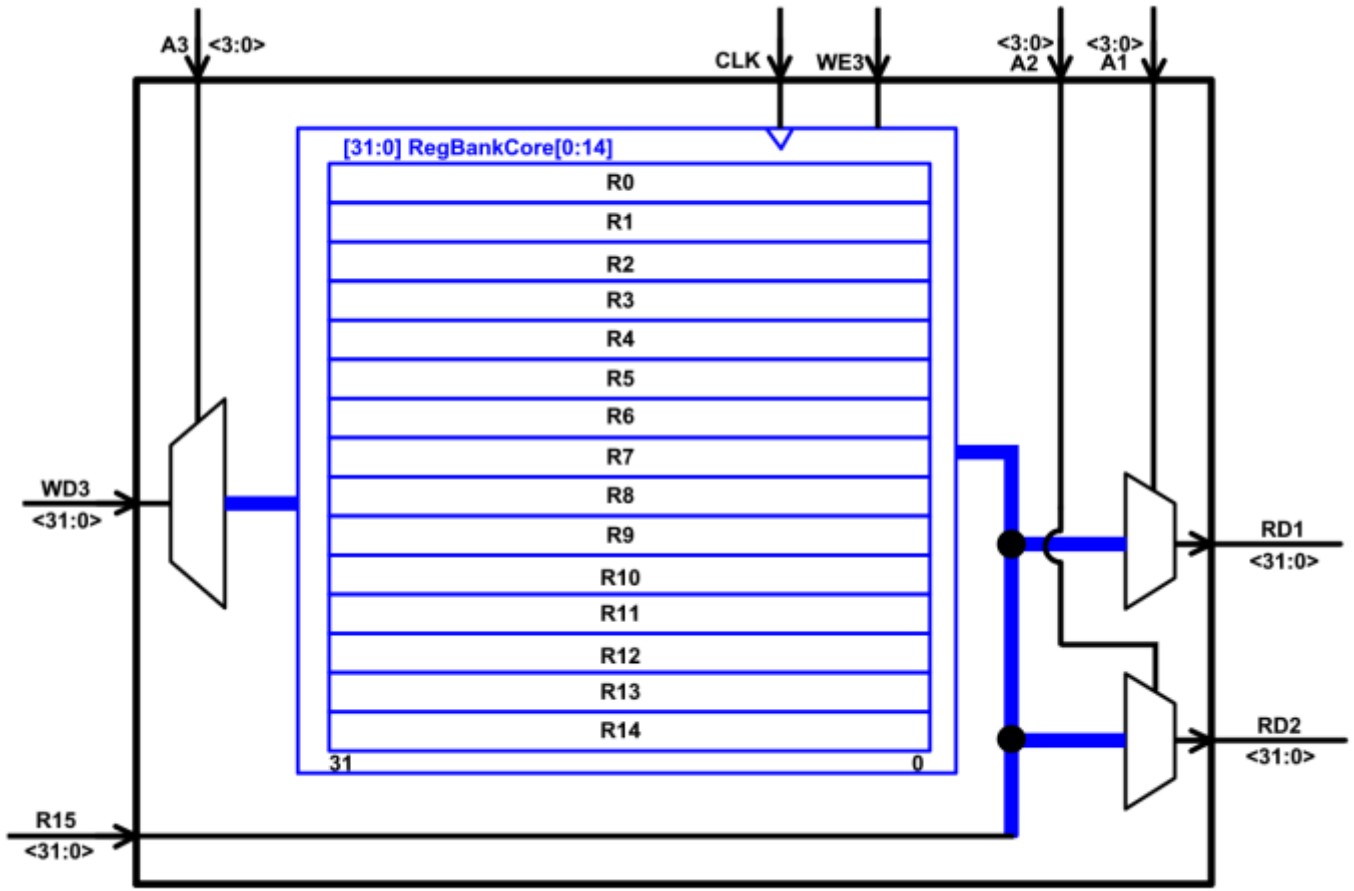
Compile Option:
`include"ProgramCounter.v "
Need to include the full file path if ProgramCounter.v and testbench are not in the same folder

```
ProgramCounter PC1(
      .CLK(CLK),
      .Reset(Reset),
      .PCSrc(PCSrc),
      .Result(Result),

      .current_PC(current_PC),
      .PC_Plus_4(PC_Plus_4)
);
```

Register File

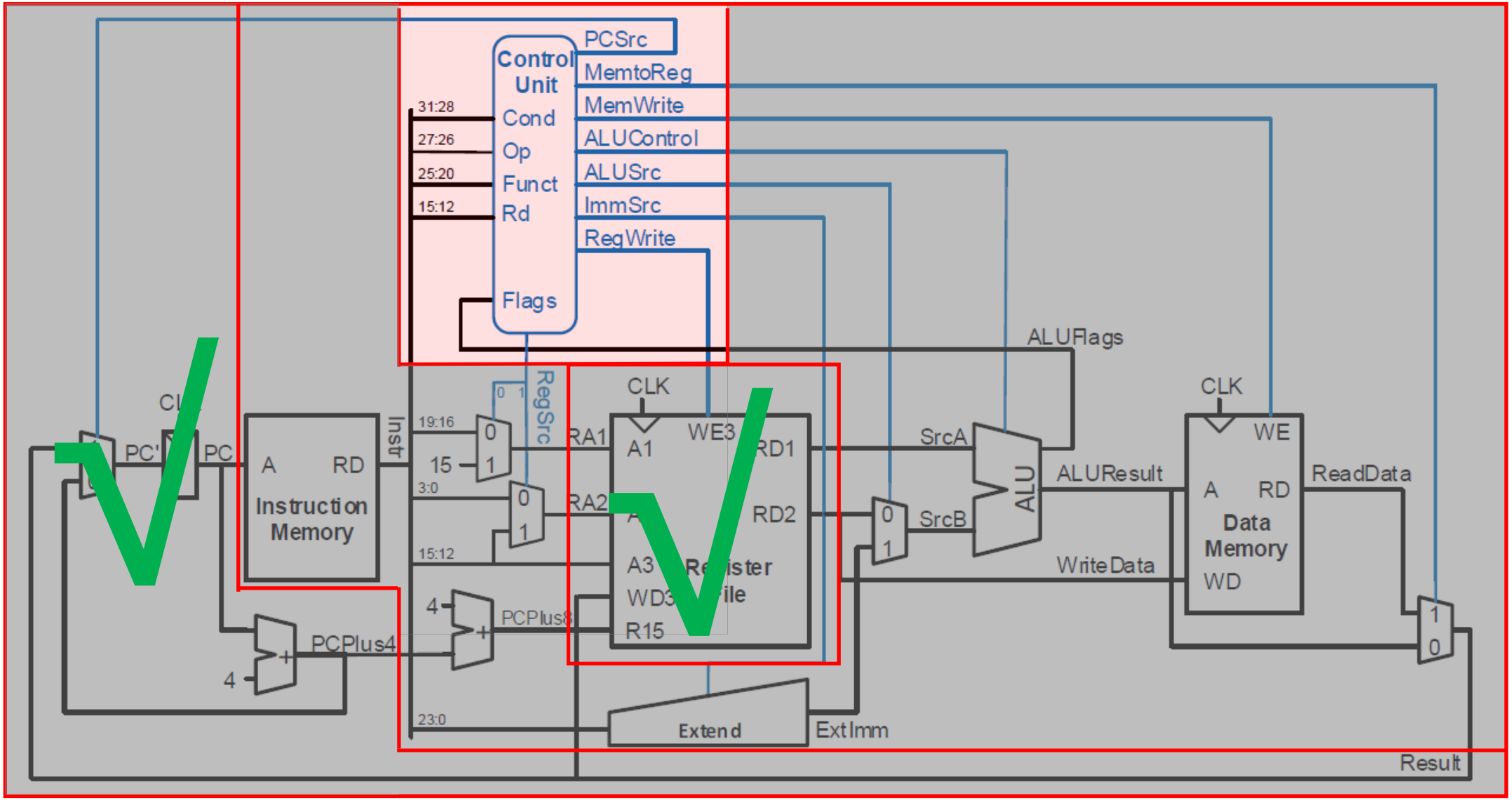


Register File

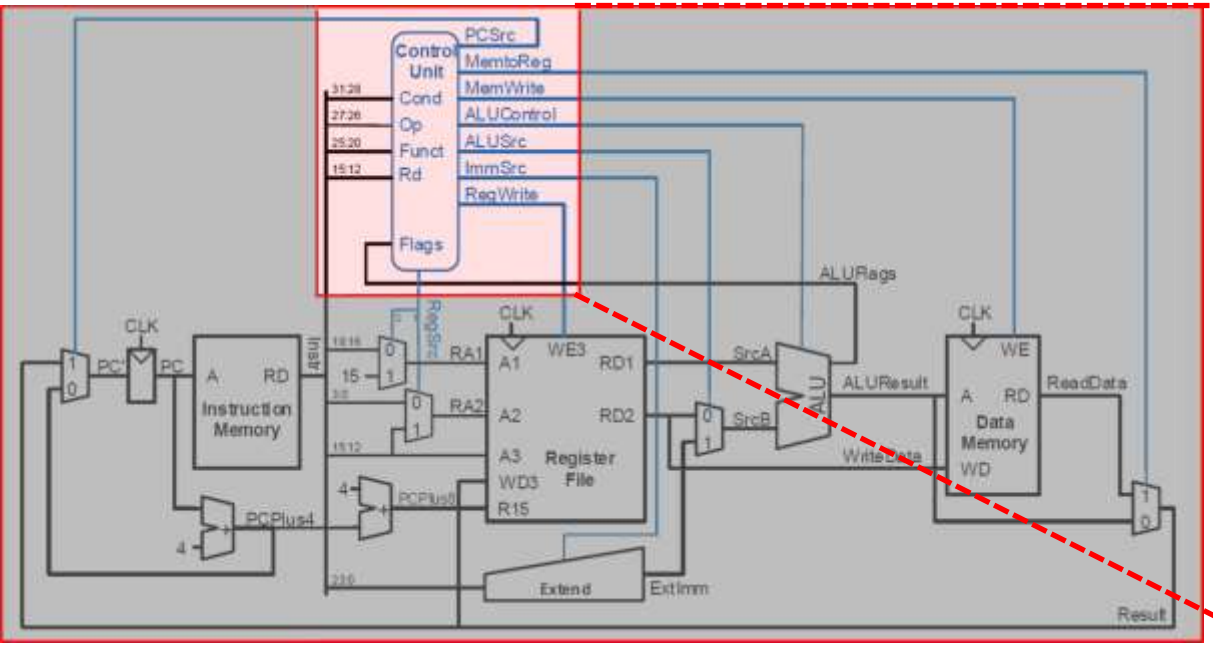


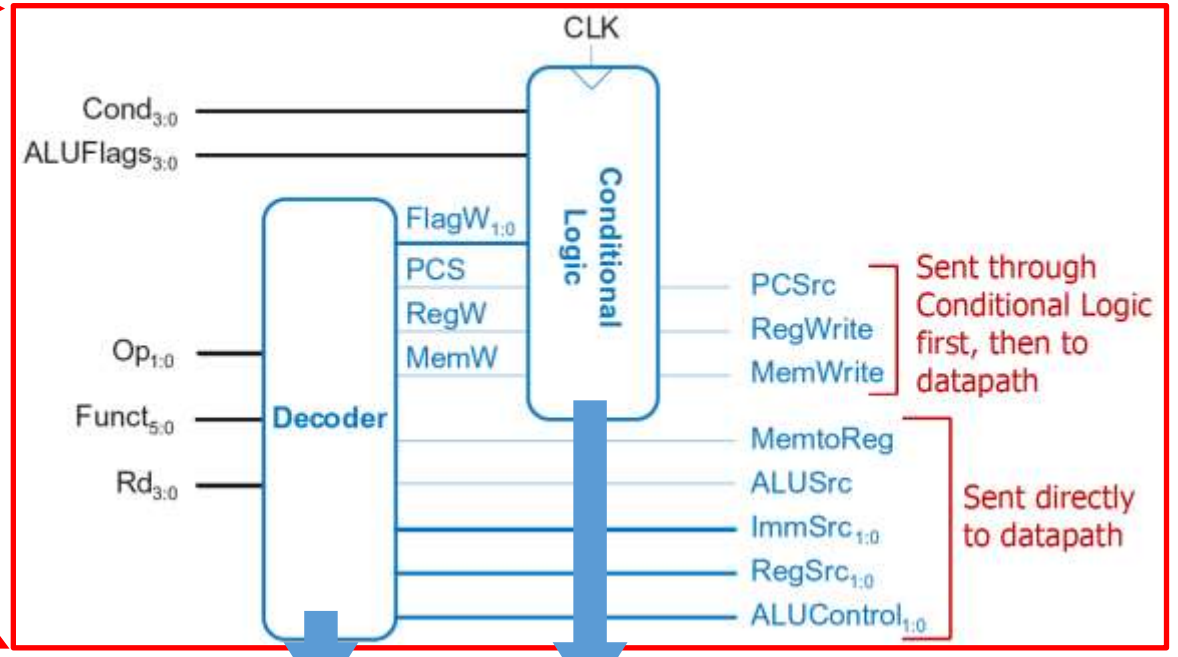

```
//RegisterFile.v

module RegisterFile(
    input CLK,
    input WE3,  //high active
    input [3:0] A1, //Read index1
    input [3:0] A2, //Read index2
    input [3:0] A3, //Write index
    input [31:0] WD3,//Write data
    input [31:0] R15, //R15 Data in
    output [31:0] RD1, // Read data1
    output [31:0] RD2  //Read data2
    );

reg [31:0] RegBankCore[0:14] ;

//fill your Verilog code here.

endmodule
```

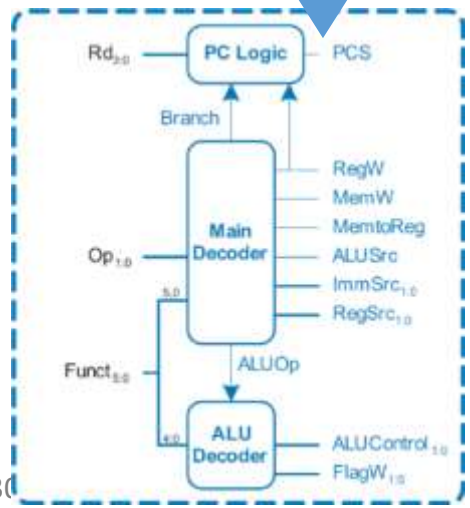
Control Unit

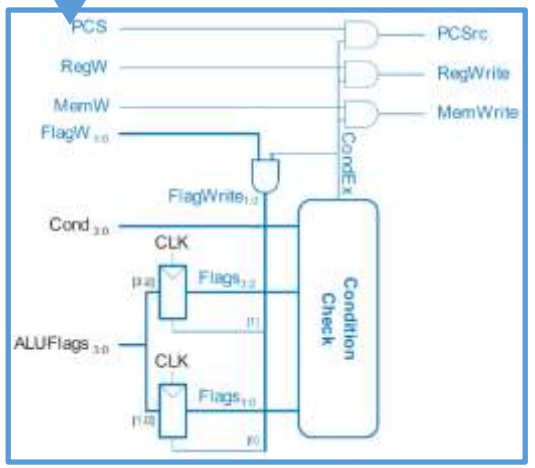


Control Unit



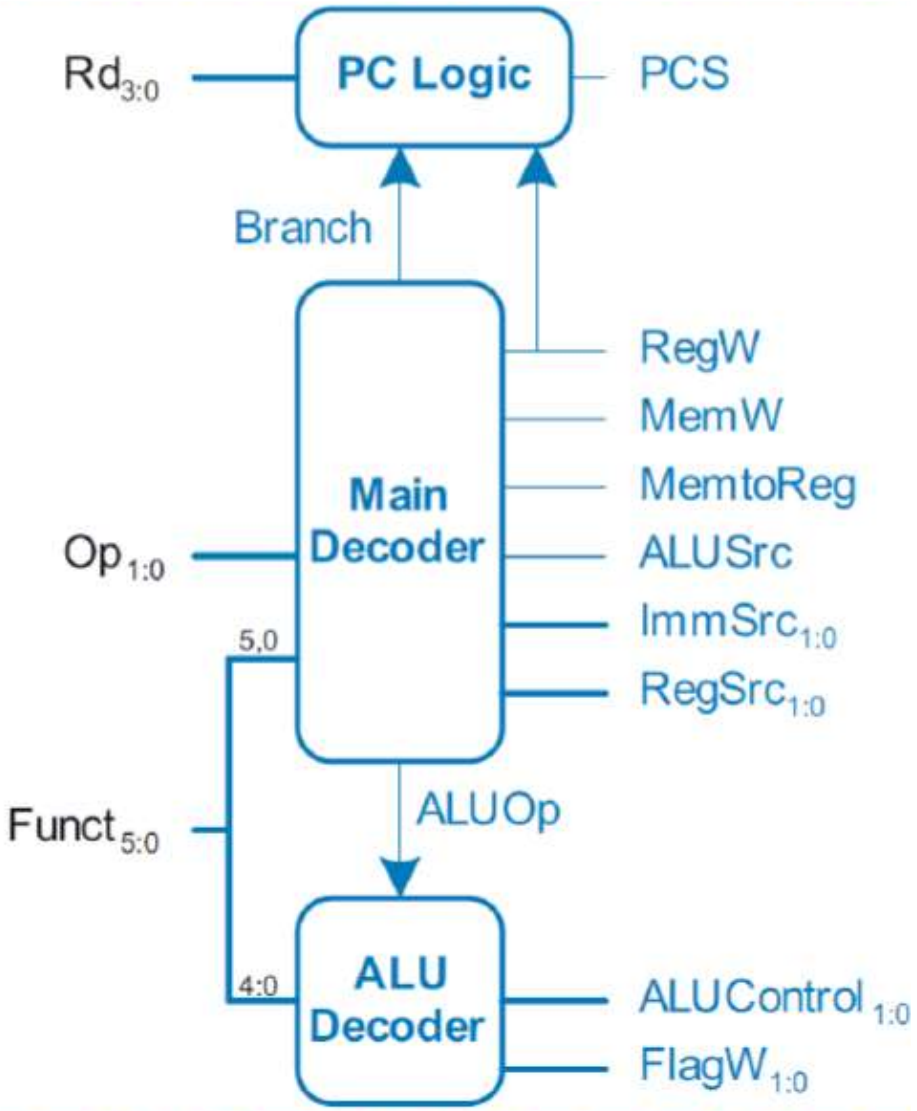


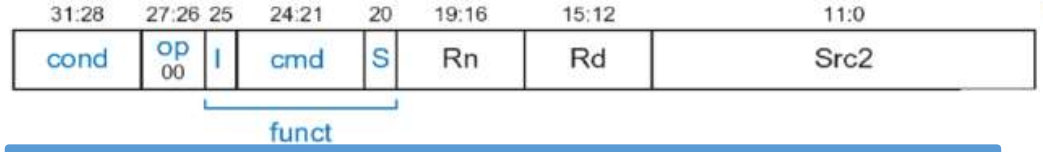




- Control Unit:
  - Decoder
  - Condition Logic

Decoder





- PCS = 1 if PC is written by an instruction or branch (B):
  PCS = ((Rd == 15) & RegW) | Branch
- If instruction is executed: PCSrc = PCS
- Else: PCSrc = 0 (i.e., PC=PC+4)

PC Logic

Op	$Funct_5$	$Funct_0$	Type	Branch	MemtoReg	MemW	ALUSrc	ImmSrc	RegW	RegSrc	ALUOp
00	0	X	DP reg	0	0	0	0	XX	1	00	1
00	1	X	DP imm	0	0	0	1	00	1	X0	1
01	X	0	STR	0	X	1	1	01	0	10	0
01	X	1	LDR	0	1	0	1	01	1	X0	0
10	X	X	B	1	0	0	1	10	0	X1	0

Main Decoder

ALUOp	$Funct_{4:1}$ (cmd)	$Funct_0$ (S)	Type	$ALUControl_{1:0}$	$FlagW_{1:0}$
0	X	X	Not DP	00	00
1	0100	0	ADD	00	00
		1			11
	0010	0	SUB	01	00
		1			11
	0000	0	AND	10	00
		1			10
	1100	0	ORR	11	00
		1			10

assume to be ADD

For positive immediate offset

ALU Decoder

Decoder

```
module Decoder(
    input [31:0] Instr,
    output PCS,
    output RegW,
    output MemW,
    output MemtoReg,
    output ALUSrc,
    output [1:0] ImmSrc,
    output [1:0] RegSrc,
    output reg [1:0] ALUControl,
    output reg [1:0] FlagW
    );

    reg ALUOp ;
    reg Branch ;

//Main Decoder

//ALU Decoder

//PC Logic


endmodule
```

Hint for Coding: Use case statement

Verilog HDL: A Guide to Digital Design and Synthesis, Chapter 7.5

7.5.1 case Statement

The keywords **case**, **endcase**, and **default** are used in the *case* statement. .

```
case (expression)
   alternative1: statement1;
   alternative2: statement2;
   alternative3: statement3;
      …
      …
   default: default_statement;
endcase
```

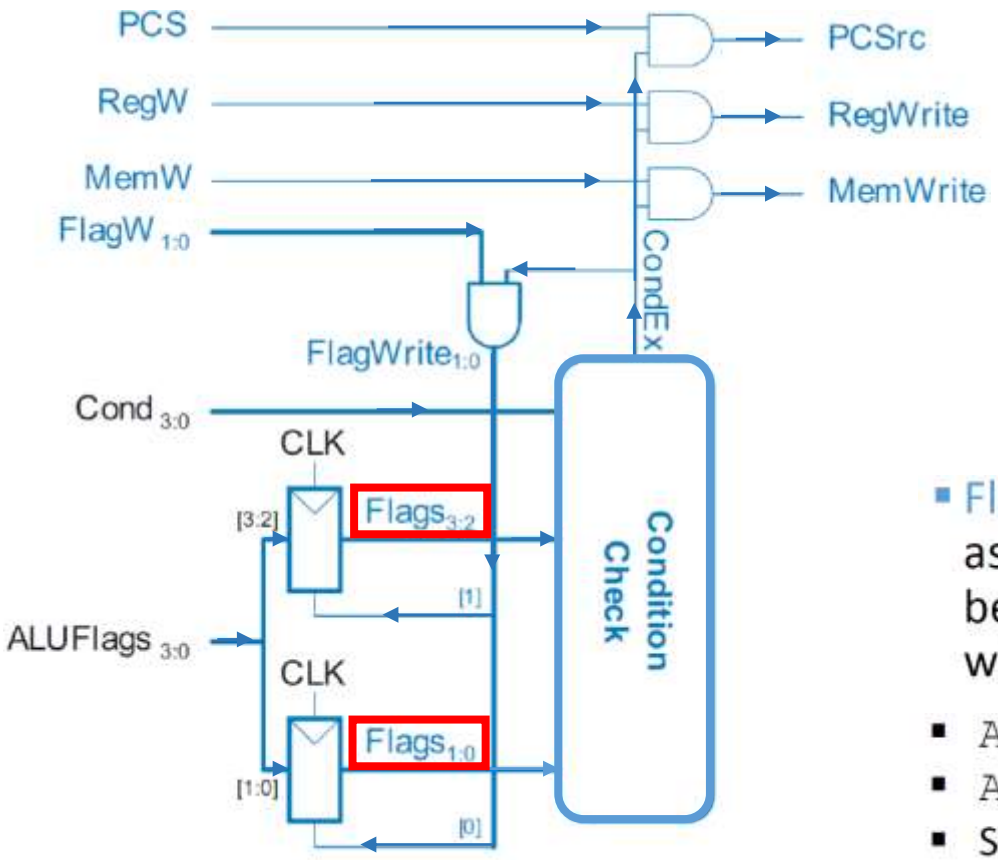
7.5.2 casex, casez Keywords

There are two variations of the **case** statement. They are denoted by keywords, **casex** and **casez**.

- **casez** treats all **z** values in the case alternatives or the case expression as don't cares. All bit positions with **z** can also represented by **?** in that position.
- **casex** treats all **x** and **z** values in the case item or the case expression as don't cares.

The use of **casex** and **casez** allows comparison of only non-**x** or -**z** positions in the case expression and the case alternatives. Example 7-16 illustrates the

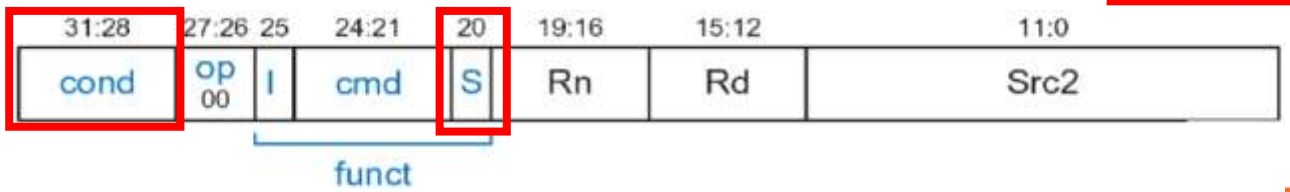
Condition Logic



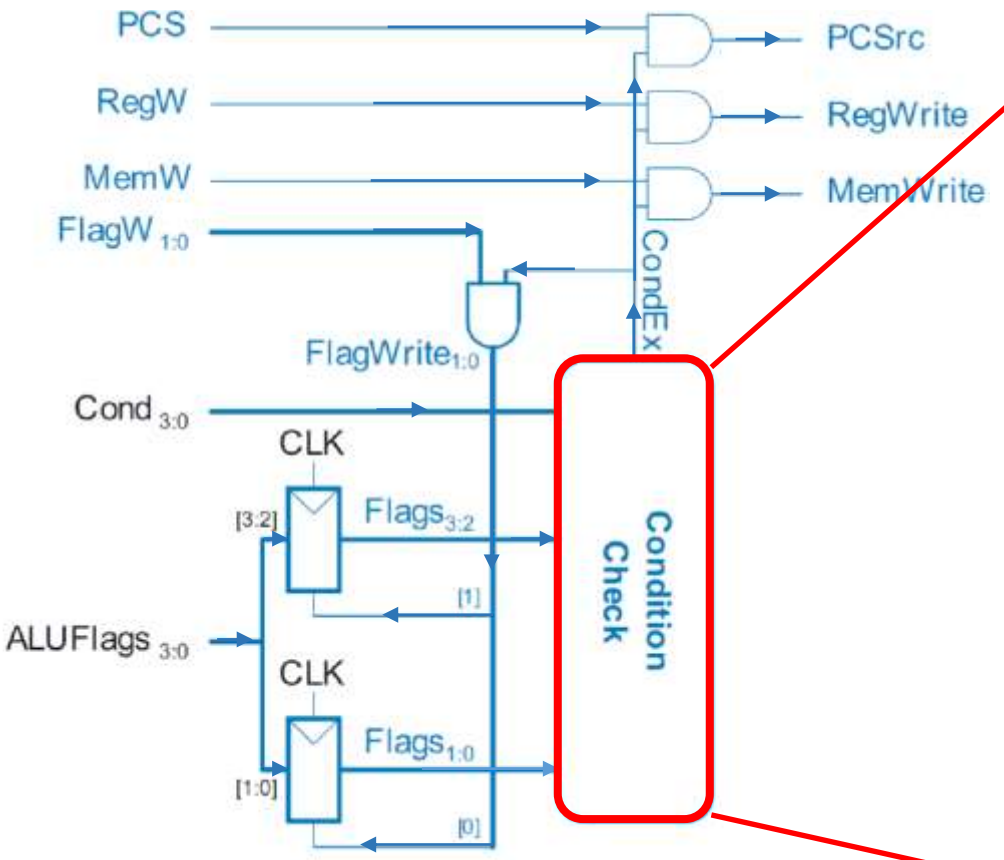
ALUOp	Funct$_{4:1}$ (*cmd*)	Funct$_0$ (*S*)	Type	ALUControl$_{1:0}$	FlagW$_{1:0}$
0	X	X	Not DP	00	00
1	0100	0	ADD	00	00
		1			11
	0010	0	SUB	01	00
		1			11
	0000	0	AND	10	00
		1			10
	1100	0	ORR	11	00
		1			10

- $FlagW_{1:0}$: Flag Write signal, asserted when ALUFlags should be saved (i.e., on instruction with S=1)
- ADD, SUB update all flags
- AND, OR update NZ only
- So Flags status register has two write enables. **$Flags_{3:0}$** = {N,Z,C,V}

- ADD, SUB update all flags (NZCV)
- AND, ORR only update NZ flags
- So, two bits needed:
  - $FlagW_1$ = 1: NZ saved ($ALUFlags_{3:2}$ saved)
  - $FlagW_0$ = 1: CV saved ($ALUFlags_{1:0}$ saved)



Condition Logic

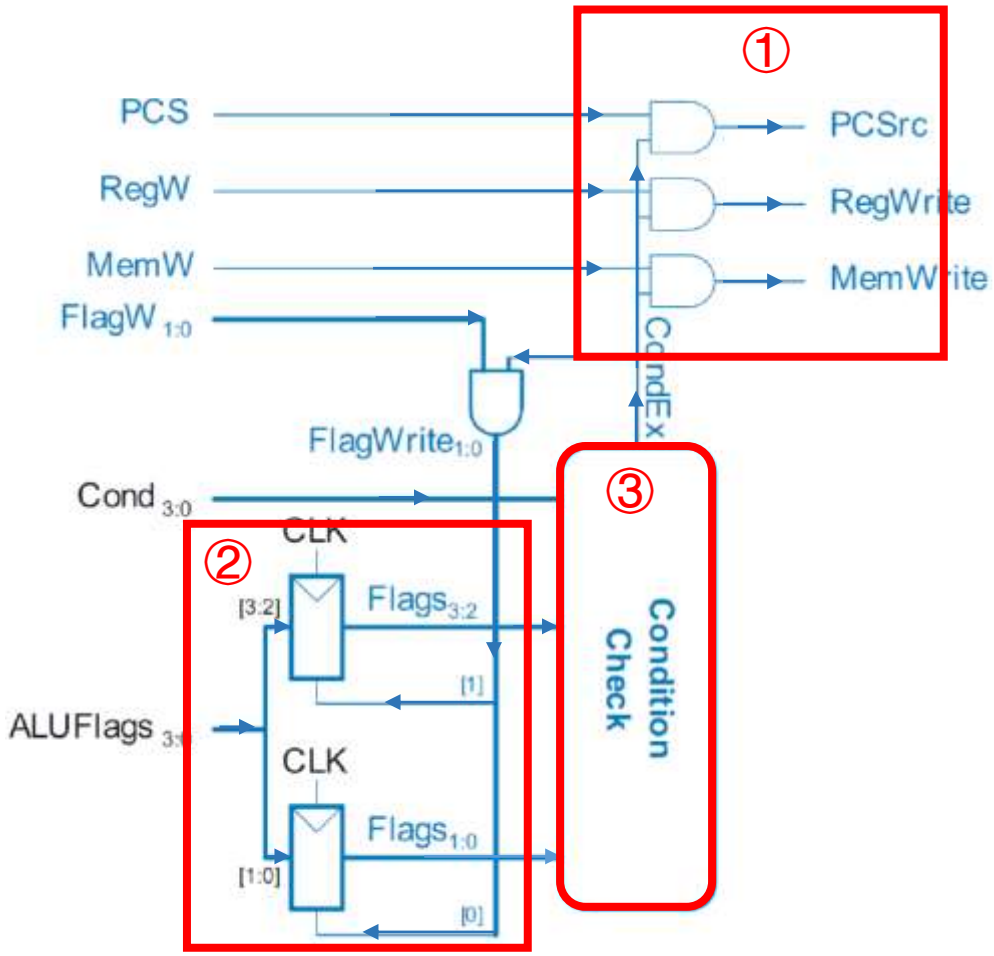


cond	Mnemonic	Name	CondEx
0000	EQ	Equal	Z
0001	NE	Not equal	$\bar{Z}$
0010	CS / HS	Carry set / Unsigned higher or same	C
0011	CC / LO	Carry clear / Unsigned lower	$\bar{C}$
0100	MI	Minus / Negative	N
0101	PL	Plus / Positive of zero	$\bar{N}$
0110	VS	Overflow / Overflow set	V
0111	VC	No overflow / Overflow clear	$\bar{V}$
1000	HI	Unsigned higher	$\bar{Z}C$
1001	LS	Unsigned lower or same	$Z\ OR\ \bar{C}$
1010	GE	Signed greater than or equal	$\overline{N \oplus V}$
1011	LT	Signed less than	$N \oplus V$
1100	GT	Signed greater than	$\bar{Z}(\overline{N \oplus V})$
1101	LE	Signed less than or equal	$Z\ OR\ (N \oplus V)$
1110	AL (or none)	Always / unconditional	ignored

Condition Logic

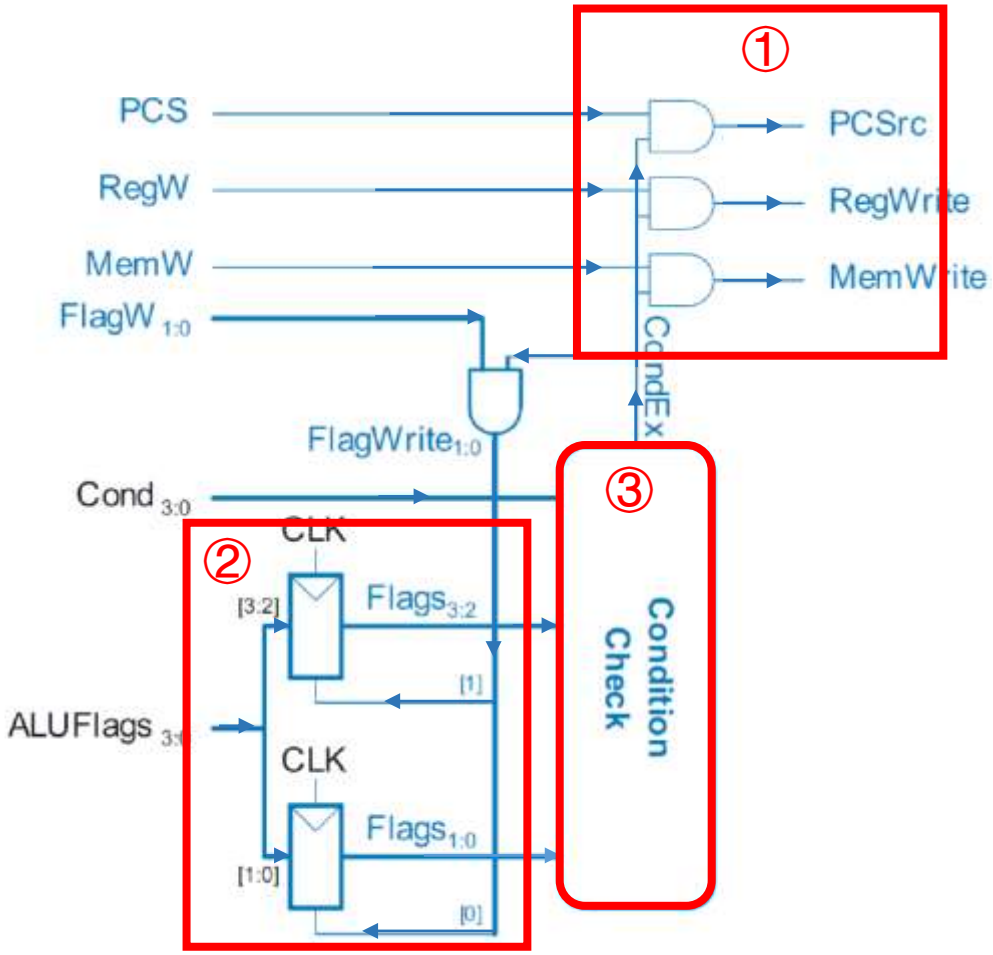
```
module CondLogic(
    input CLK,
    input PCS,
    input RegW,
    input MemW,
    input [1:0] FlagW,
    input [3:0] Cond,
    input [3:0] ALUFlags,

    output PCSrc,
    output RegWrite,
    output MemWrite,
    );

    reg CondEx ;
    reg N = 0, Z = 0, C = 0, V = 0 ;

endmodule
```

Code can be divided into 3 parts:



Condition Logic

Code can be divided into 3 parts:



① output stage: assign using '&' logic

② Flags Register update: $Flags_{3:0} = \{N,Z,C,V\}$

- $FlagW_1$ = 1: NZ saved ($ALUFlags_{3:2}$ saved)
- $FlagW_0$ = 1: CV saved ($ALUFlags_{1:0}$ saved)

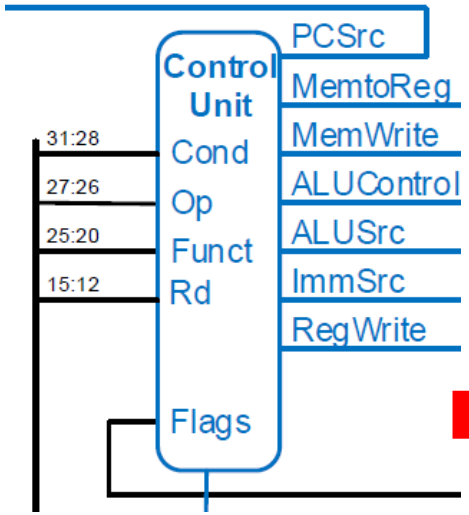
⇒ $\{N, Z\} <= ALUFlags_{3:2}$

⇒ $\{C, V\} <= ALUFlags_{1:0}$

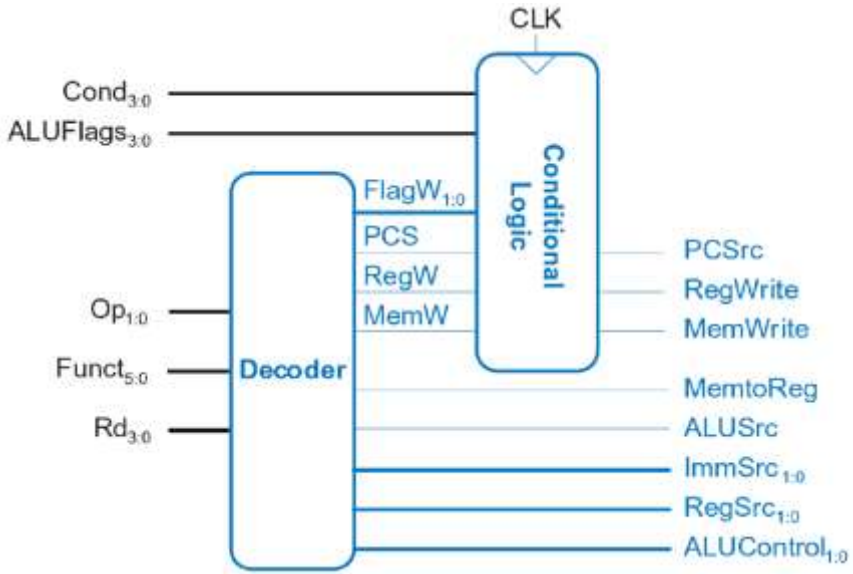
③ 'CondEX' generate: use case statement

cond	Mnemonic	Name	CondEx
0000	EQ	Equal	Z
0001	NE	Not equal	$\bar{Z}$
0010	CS / HS	Carry set / Unsigned higher or same	C
0011	CC / LO	Carry clear / Unsigned lower	$\bar{C}$
0100	MI	Minus / Negative	N
0101	PL	Plus / Positive of zero	$\bar{N}$
0110	VS	Overflow / Overflow set	V
0111	VC	No overflow / Overflow clear	$\bar{V}$
1000	HI	Unsigned higher	$\bar{Z}C$
1001	LS	Unsigned lower or same	$Z\ OR\ \bar{C}$
1010	GE	Signed greater than or equal	$\overline{N \oplus V}$
1011	LT	Signed less than	$N \oplus V$
1100	GT	Signed greater than	$\bar{Z}(\overline{N \oplus V})$
1101	LE	Signed less than or equal	$Z\ OR\ (N \oplus V)$
1110	AL (or none)	Always / unconditional	ignored

ControlUnit



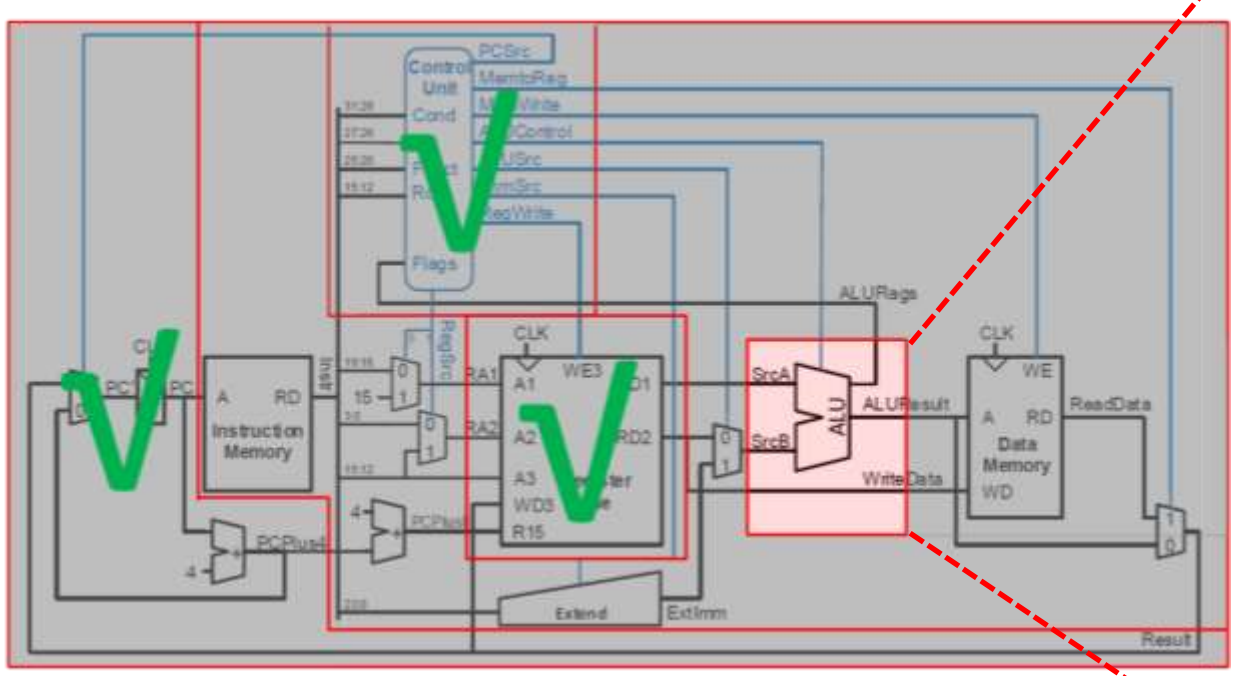
Combine the Decoder module and the Conditional Logic module to form the Control Unit module by connecting some internal wires.

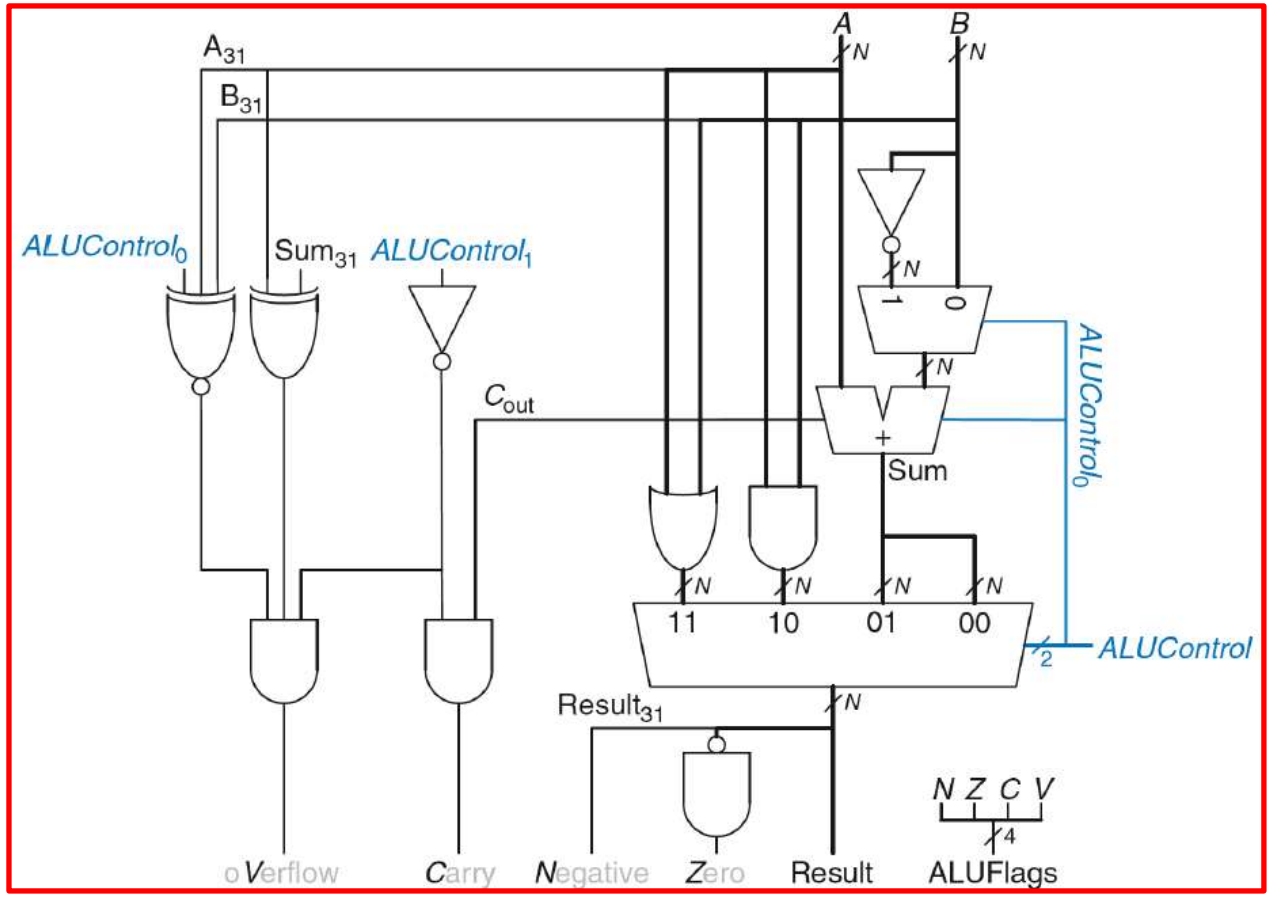



```
//ControlUnit.v

module ControlUnit(
    input [31:0] Instr,
    input [3:0] ALUFlags,
    input CLK,

    output MemtoReg,
    output MemWrite,
    output ALUSrc,
    output [1:0] ImmSrc,
    output RegWrite,
    output [1:0] RegSrc,
    output [1:0] ALUControl,
    output PCSrc
    );

    wire [3:0] Cond;
    wire PSC, RegW, MemW;
    wire [1:0] FlagW;

    assign Cond=Instr[31:28];
```

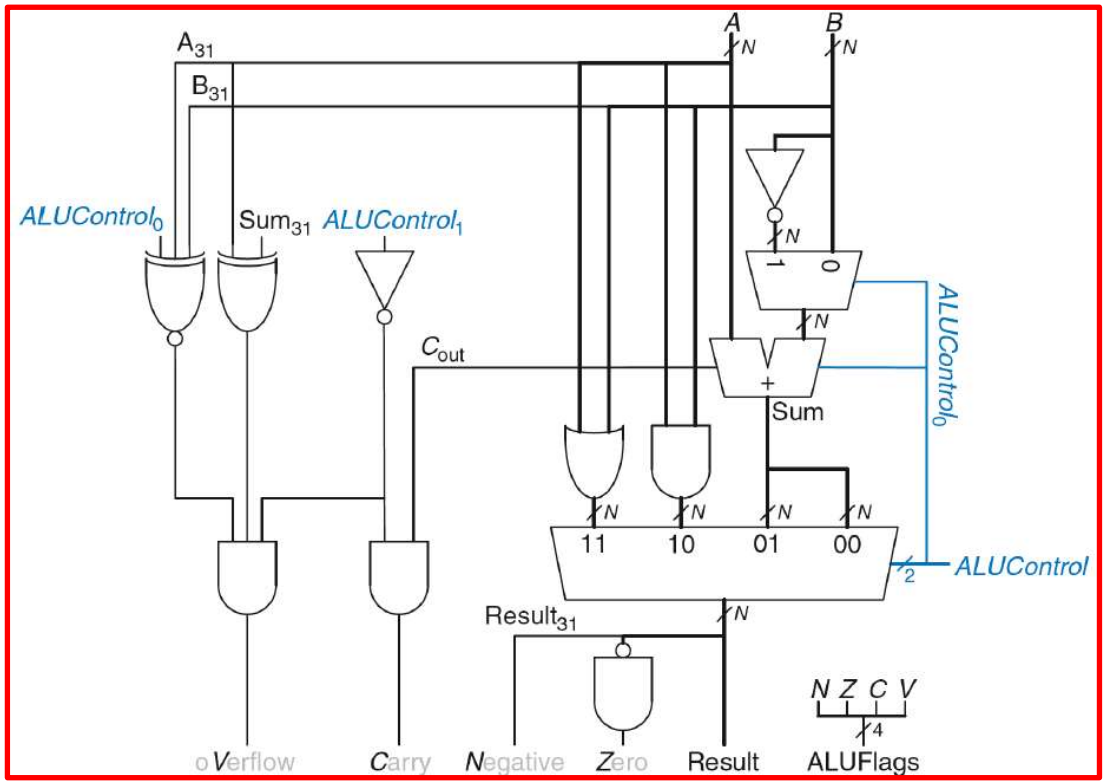
```
CondLogic CondLogic1(
      CLK,
      PCS,
      RegW,
      MemW,
      FlagW,
      Cond,
      ALUFlags,

      PCSrc,
      RegWrite,
      MemWrite,
    );
Decoder Decoder1(
      Instr,
      PCS,
      RegW,
      MemW,
      MemtoReg,
      ALUSrc,
      ImmSrc,
      RegSrc,
      ALUControl,
      FlagW
    );
    endmodule
```

ALU


Control Unit
PCSrc
MemtoReg
MemWrite
ALUControl
ALUSrc
ImmSrc
RegWrite
Flags
ALUFlags
CLK
PC
Instruction Memory
RA1
RA2
A1
A2
A3
WD3
R15
WE3
RD1
RD2
SrcA
SrcB
ALU
ALUResult
WriteData
WE
Data Memory
WD
ReadData
Extend
ExtImm
Result


A31
B31
ALUControl0
Sum31
ALUControl1
Cout
A
B
N
1
0
Sum
+
11
10
01
00
2
ALUControl
Result31
NZCV
4
oVerflow
Carry
Negative
Zero
Result
ALUFlags

ALU



- N = 1 (Negative) if Result is negative
- Z = 1 (Zero) if all of the bits of Result are 0
- C = 1 (Carry) if
  - C_{out} of adder is 1 **AND**
  - ALU is performing `ADD`/`SUB`
- V = 1 (oVerflow) if
  - ALU is performing `ADD`/`SUB` **AND**
  - A and Sum have opposite signs **AND**
  - If `ADD`: A and B have same sign
  - If `SUB`: A and B have different signs

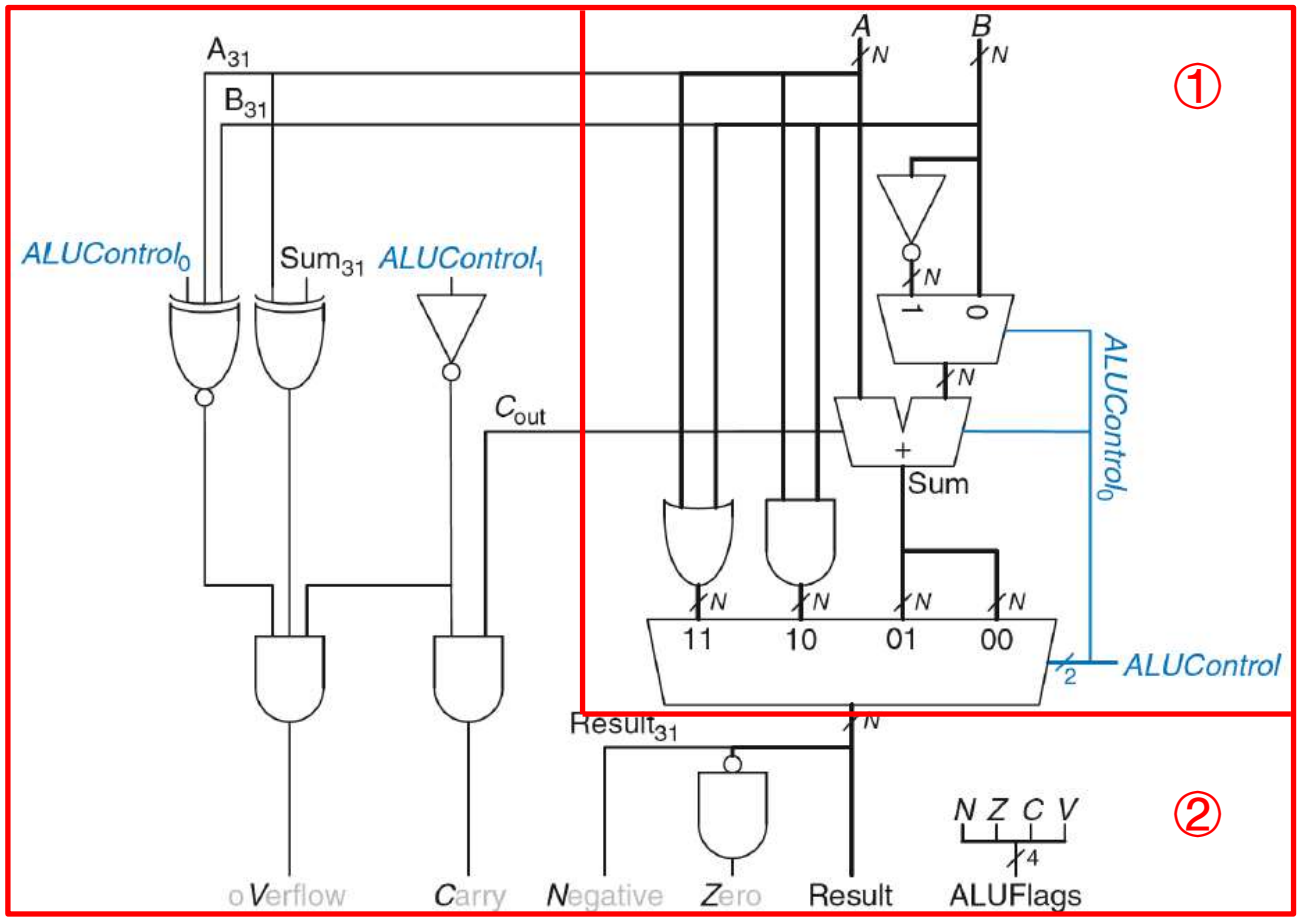
ALUControl$_{1:0}$	**00**	**01**	**10**	**11**
Type	**ADD**	**SUB**	**AND**	**ORR**

- **ADD**, **SUB** update all flags(NZCV)
- **AND**, **ORR** only update NZ flags*

ALU

```
module ALU(
        input [31:0] A,
        input [31:0] B,
        input [1:0]  ALUControl,

        output [31:0] ALUResult,
        output [3:0]  ALUFlags
);

endmodule
```

Code can be divided into 2 parts:

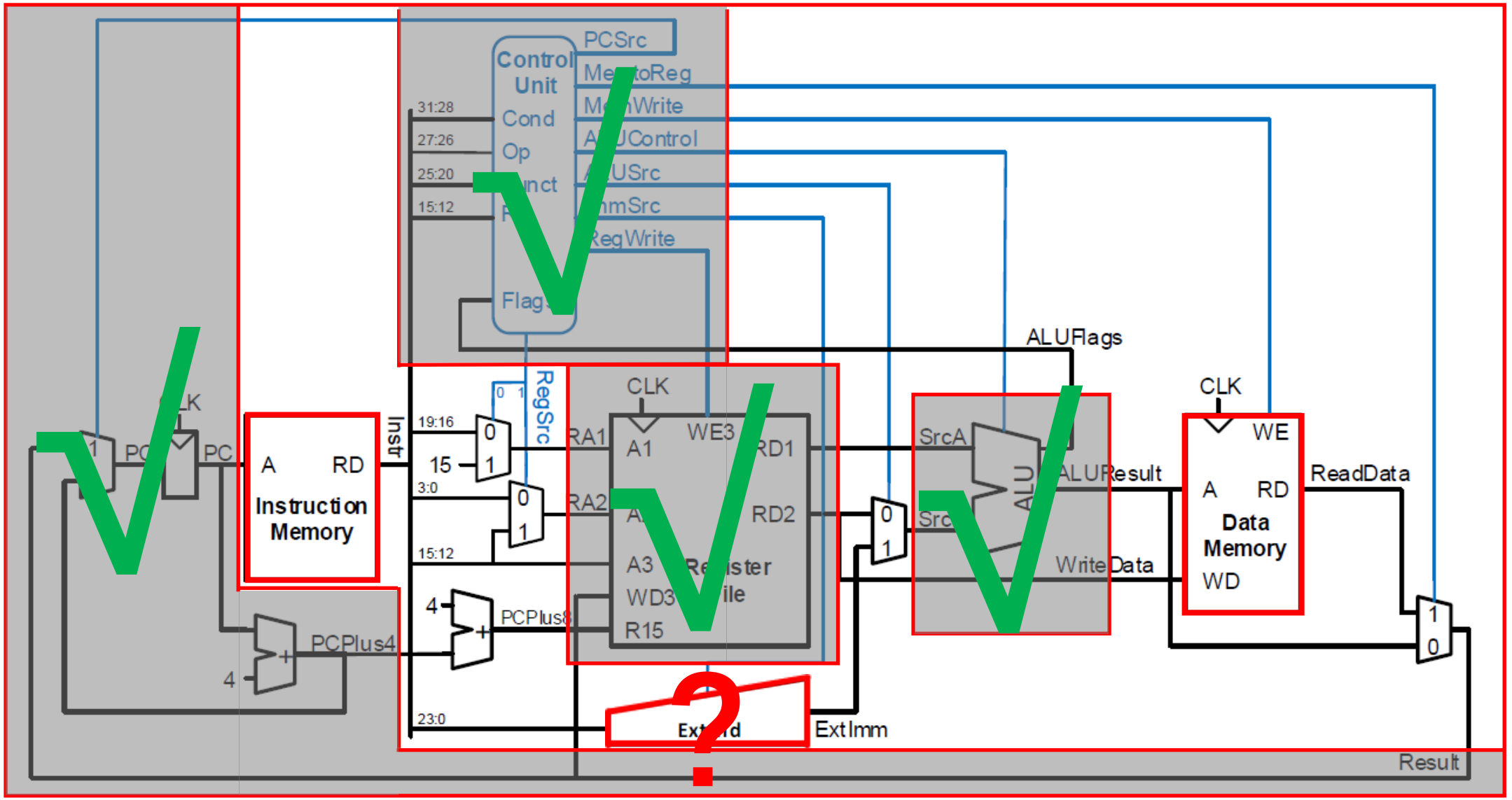


① **Arithmetic Operation**

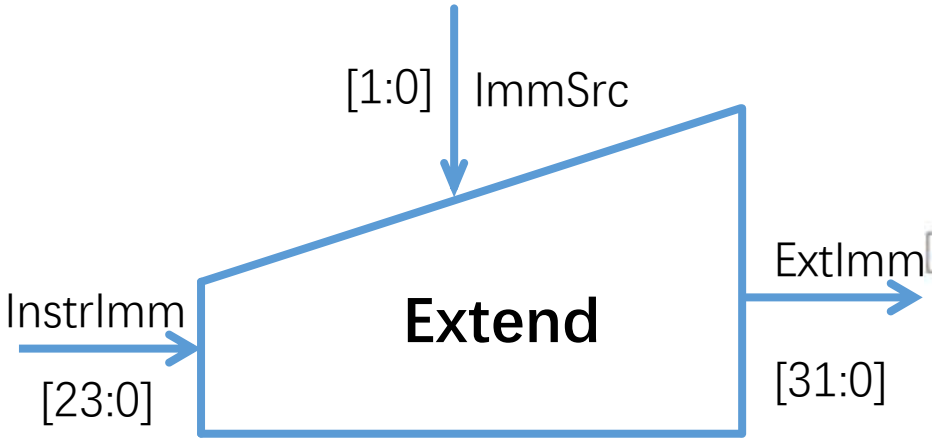
② **ALUFlags Generation**

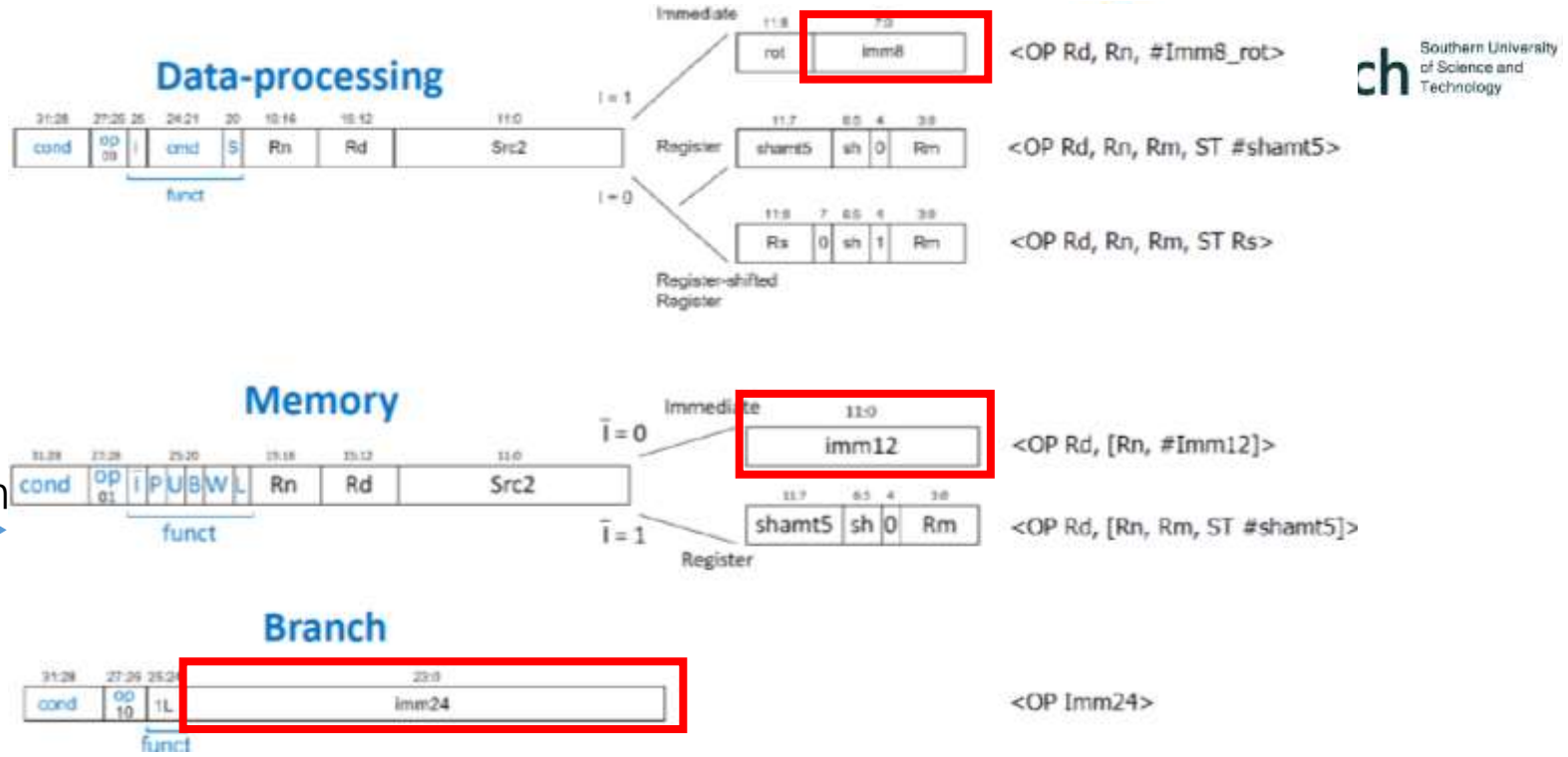
- According to the diagram, **only one** Adder is allowed!
- Subtraction shall NOT appear in your code!

Extend


Control Unit
PCSrc
MemtoReg
MemWrite
ALUControl
ALUSrc
ImmSrc
RegWrite
Flags
31:28
27:26
25:20
15:12
Cond
Op
Funct
Instruction Memory
A
RD
Instr
PC
PCPlus4
4
19:16
15
3:0
15:12
23:0
RegSrc
CLK
WE3
RA1
RA2
A1
A2
A3
WD3
R15
RD1
RD2
Register File
ALUFlags
SrcA
SrcB
ALU
ALUResult
WriteData
Data Memory
WE
WD
ReadData
ExtImm
Extend
Result

Extend



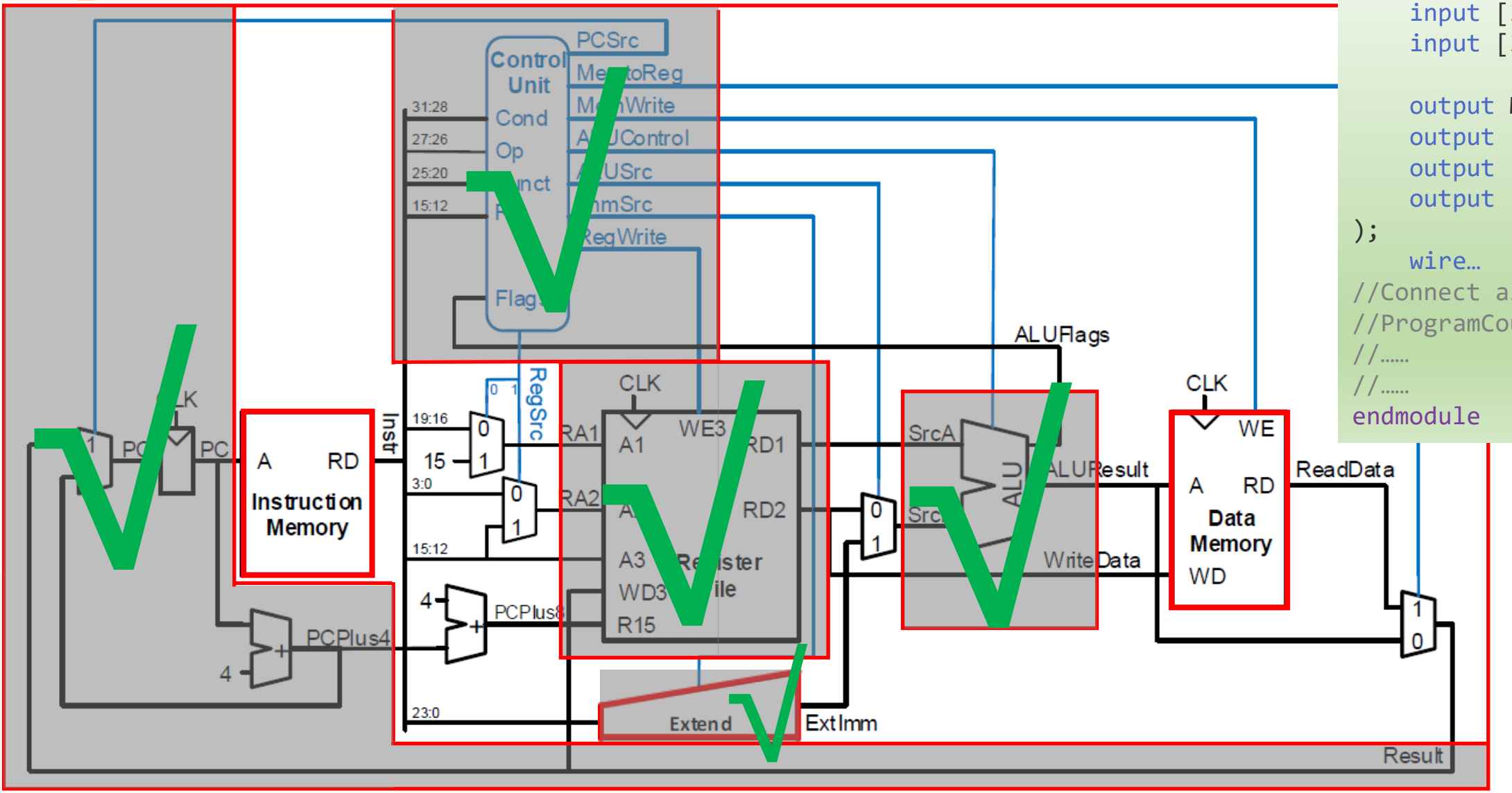


Different bit width of "Imm" for different types of instructions

	$ImmSrc_{1:0}$	ExtImm	Description
DP	00	(24{'0'}, $Instr_{7:0}$)	Zero-extended Imm8
LDR/STR	01	(20{'0'}, $Instr_{11:0}$)	Zero-extended Imm12
B	10	(6{$Instr_{23}$}, $Instr_{23:0}$, 2{'0'})	(Sign-extended Imm24) <<2

```
module Extend(
    input [1:0] ImmSrc,
    input [23:0] InstrImm,

    output reg [31:0] ExtImm
    );

endmodule
```

ARMcore (Essential Functionality)




```
module ARMcore(
    input CLK,
    input Reset,
    input [31:0] Instr,
    input [31:0] ReadData,

    output MemWrite,
    output [31:0] PC,
    output [31:0] ALUResult,
    output [31:0] WriteData
);
    wire…
//Connect all the components
//ProgramCounter
//……
//……
endmodule
```

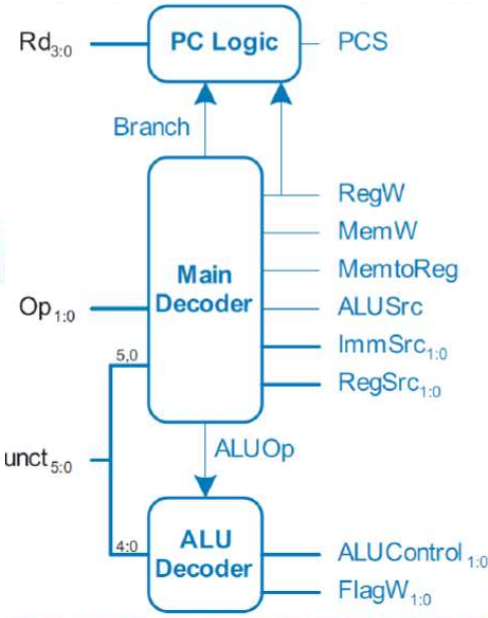
Extended Functionality

SUSTech Southern University of Science and Technology

STR, LDR with negative immediate offset

Op	$Funct_5$	$Funct_0$	Type	Branch	MemtoReg	MemW	ALUSrc	ImmSrc	RegW	Reg	ALU
00	0	X	DP reg	0	0	0	0	XX	1		
00	1	X	DP imm	0	0	0	1	00	1	X0	1
01	X	0	STR	0	X	1	1	01	0	10	0
01	X	1	LDR	0	1	0	1	01	1	X0	0
10	X	X	B	1	0	0	1	10	0	X1	0

what should be changed to support negative offset?



E.g. LDR R5,[R3,#-4];

Extended Functionality

?? Which Function to add

- funct (6-bit)
  - $\overline{I}$: Immediate bar
  - P : Preindex
  - U : Add
  - B : Byte
  - W : Writeback
  - L : Load

SUSTech Southern University of Science and Technology

STR, LDR with negative immediate offset

Op	$Funct_5$	$Funct_0$	Type	Branch	MemtoReg	MemW	ALUSrc	ImmSrc	RegW	RegSrc	ALUOp
00	0	X	DP reg	0	0	0	0	XX	1	00	1
00	1	X	DP imm	0	0	0	1	00	1	X0	1
01	X	0	STR	0	X	1	1	01	0	10	0
01	X	1	LDR	0	1	0	1	01	1	X0	0
10	X	X	B	1	0	0	1	10	0	X1	0

$ALUOp_2$

11

11

Pos Imm ??

Neg Imm ??

Pos Imm ??

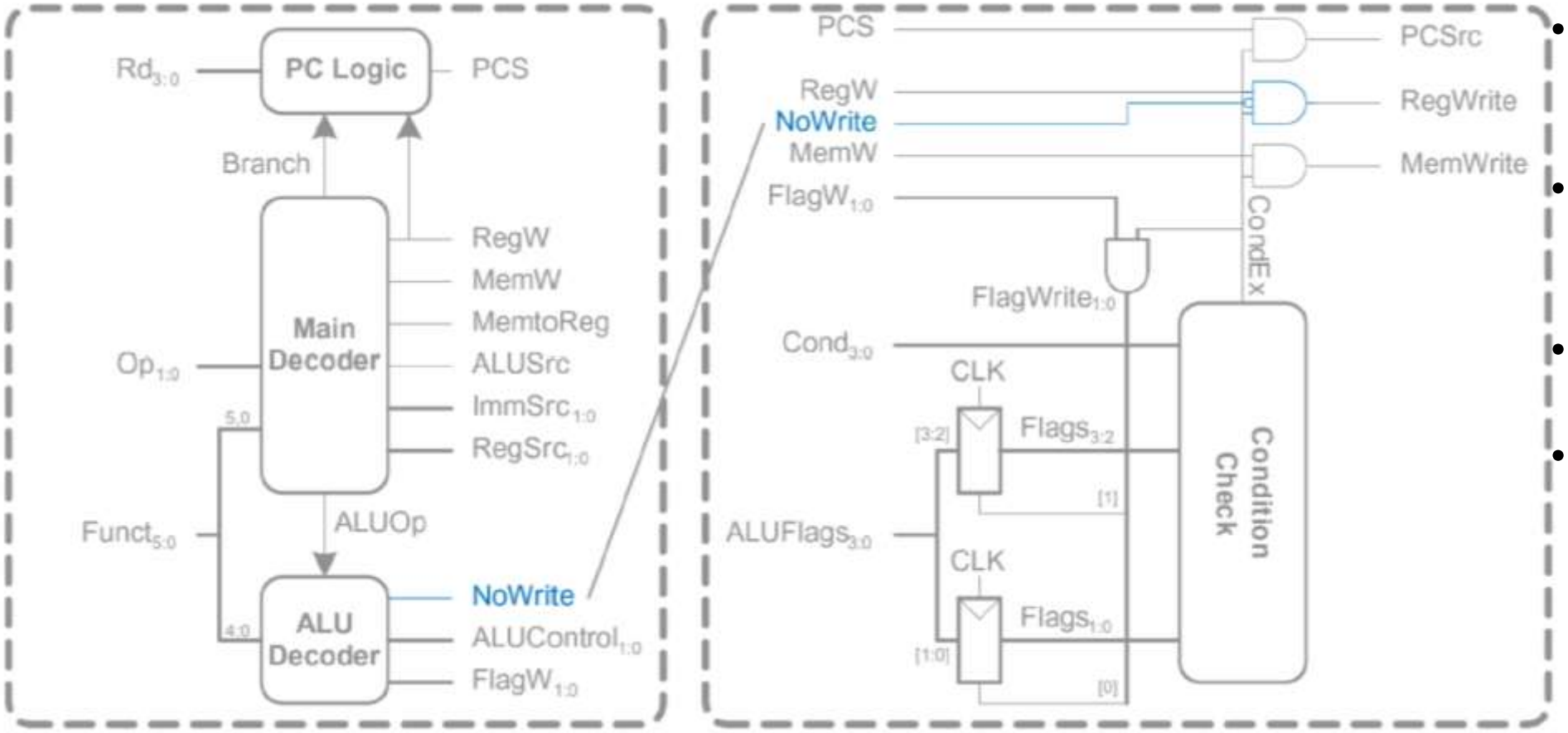
Neg Imm ??

??

Make your own choice!

Extended Functionality

CMP and CMN



- `CMP` does subtraction, sets the flag, and discards the result
- `CMP` does addition, sets the flag, and discards the result
- No change to datapath needed for implementation!
- Add “NoWrite”

Extended Functionality

SUSTech Southern University of Science and Technology

CMP and CMN

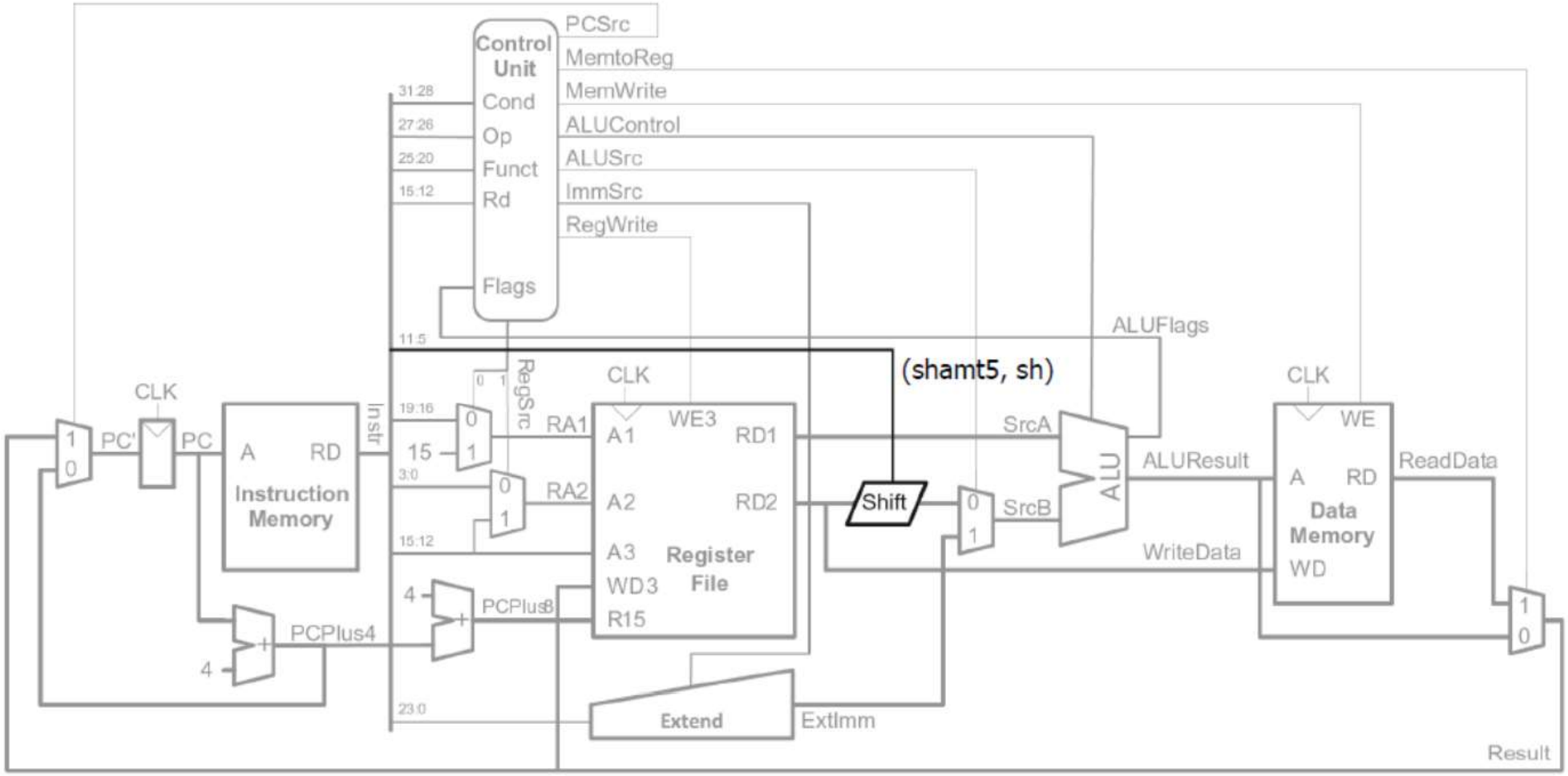
Funct[4:1]

1010	CMP	Compare	Update flags after Rn - shifter_operand
1011	CMN	Compare Negated	Update flags after Rn + shifter_operand

ALUOp	$Funct_{4:1}$ (cmd)	$Funct_0$ (S)	Type	$ALUControl_{1:0}$	$FlagW_{1:0}$	NoWrite
0??	X	X	Not DP	00	00	0
11	0100	0	ADD	00	00	0
		1			11	0
	0010	0	SUB	01	00	0
		1			11	0
	0000	0	AND	10	00	0
		1			10	0
	1100	0	ORR	11	00	0
		1			10	0
	1010	1	CMP	01	11	1
11	?	?	CMN	?	?	?

Extended Functionality

`Src2` support register with immediate shift(**LSL, LSR, ASR, ROR**)

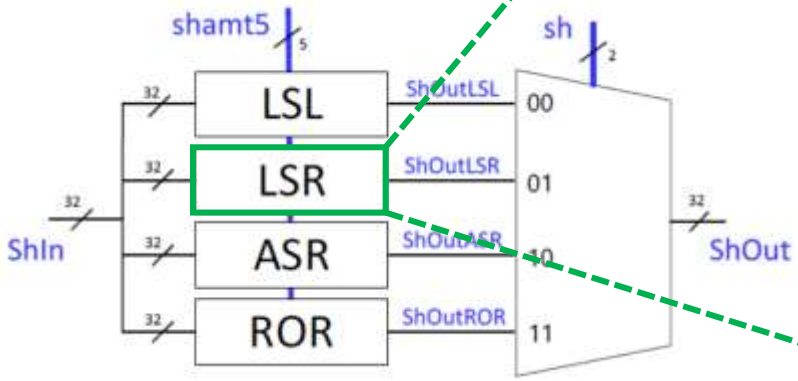


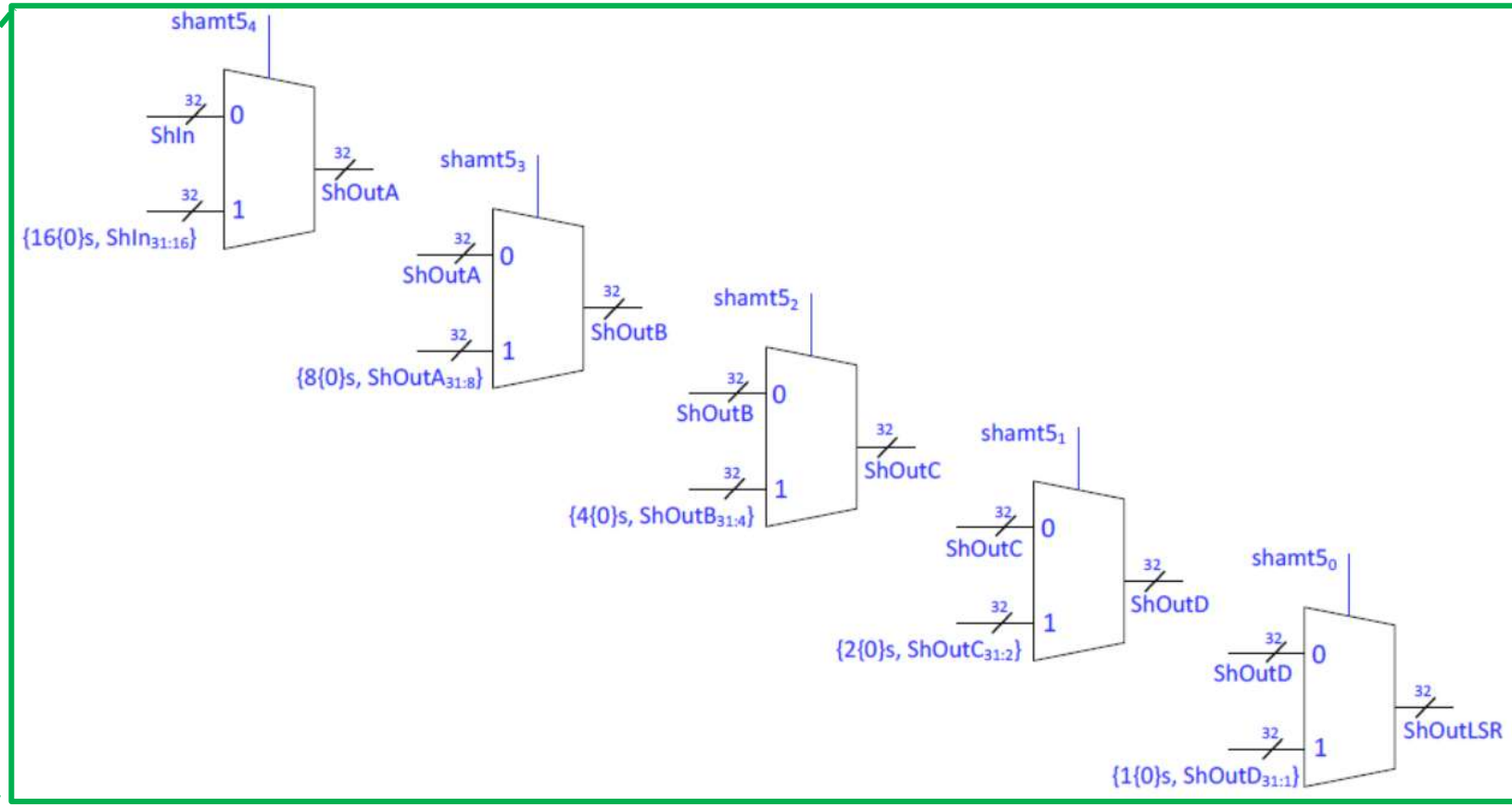
Extended Functionality

`Src2` support register with immediate shift(**LSL, LSR, ASR, ROR**)

Hardware-efficient Shifter

```
//Shifter.v

module Shifter(
    input [1:0] Sh,
    input [4:0] Shamt5,
    input [31:0] ShIn,
    output [31:0] ShOut
    );

endmodule
```





Content

1. Objective & Overview
2. Submodule Details
3. **How To Test**

How to Test

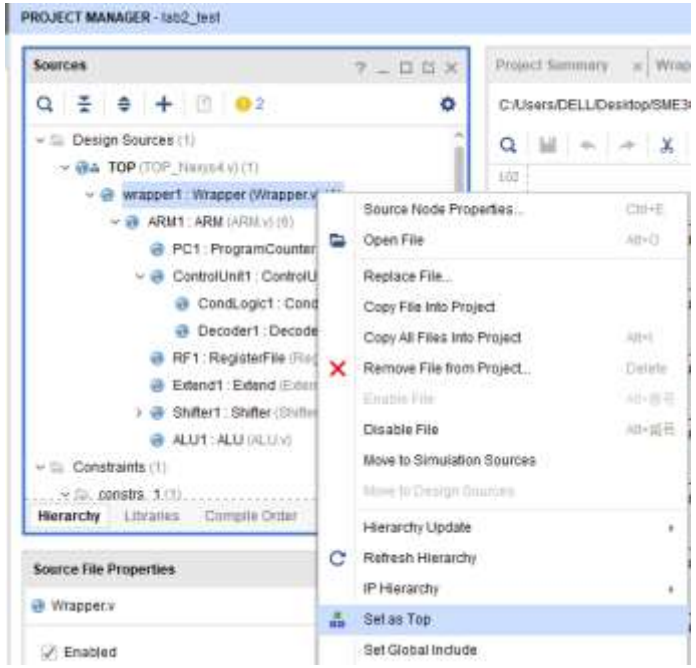
- We offer two up-level modules for ARM test: TOP and Wrapper.
- The design file hierarchy should be as follows:

  -- **TOP** (For on-board implementation, use the 100MHz on-board clock port.)

  ---- **Wrapper** (For simulation, use the post-division CLK.)

  ------ **ARM**

  -------- **Components of ARM** (ControlUnit, ALU, RegisterFile, ProgramCounter, etc.)



Q: How to convert the target module while converting synthesis to simulation?

*A: Right-click the target module, and choose **Set as Top**.*

How to Test

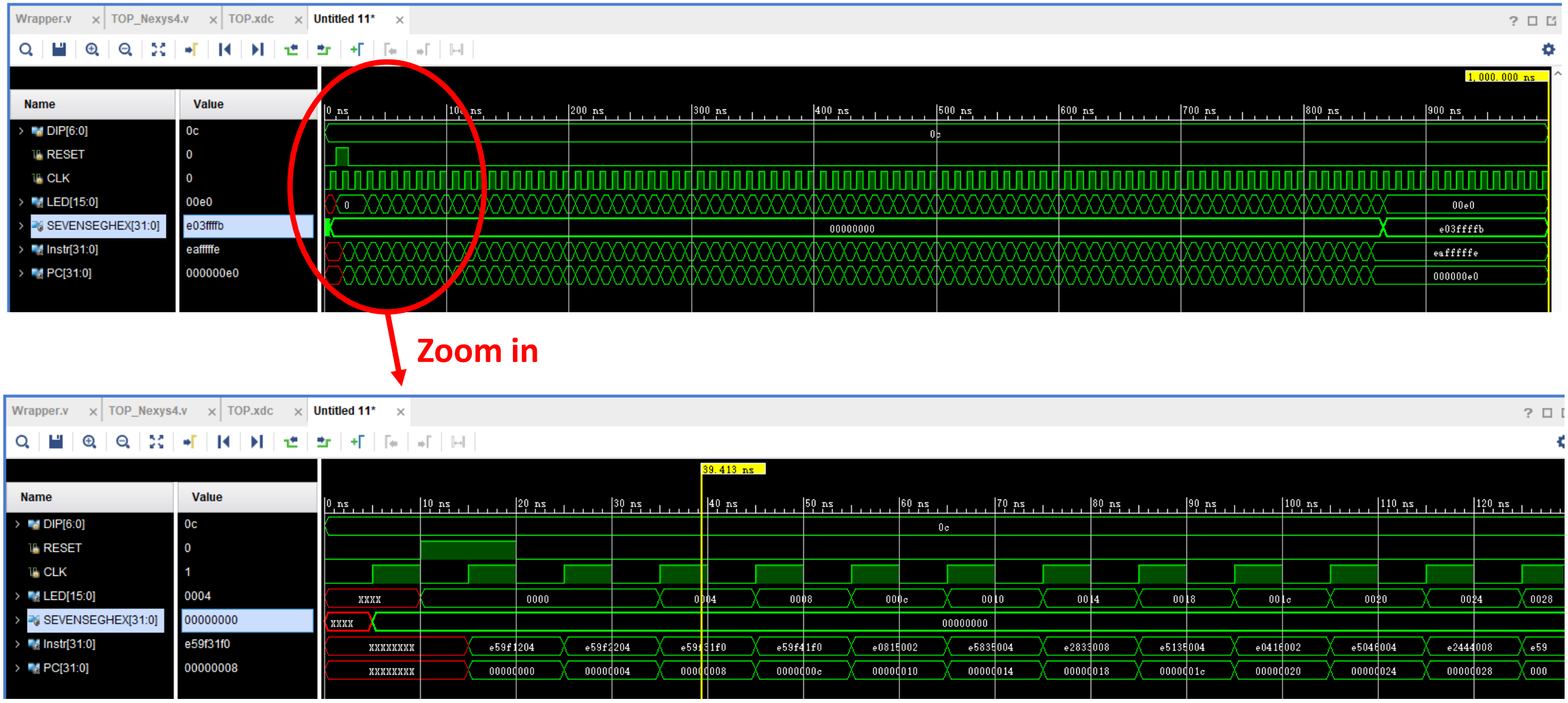
The role of FPGA peripherals:

- Switches: Represents the address of data memory (variable region).
- LEDs: Indicates the real-time PC value.
- SevenSegs: Indicates the data content of the data memory with the address corresponding to the switches.
- BTNU: Pause; BTND: Reset;
- 100MHz Oscillator: Clock provider.

*We offer the ARM instructions (machine code) and the initial data as a test sample, which are initialized in instruction memory and data memory in **Wrapper** module. Besides, TOP_Nexys4.v, Wrapper.v, Wrapper_tb.v and TOP.xdc are all completely provided so that you can directly use them to start your test. The assembly file corresponding to the test instructions can be also referred to have a better understanding.*

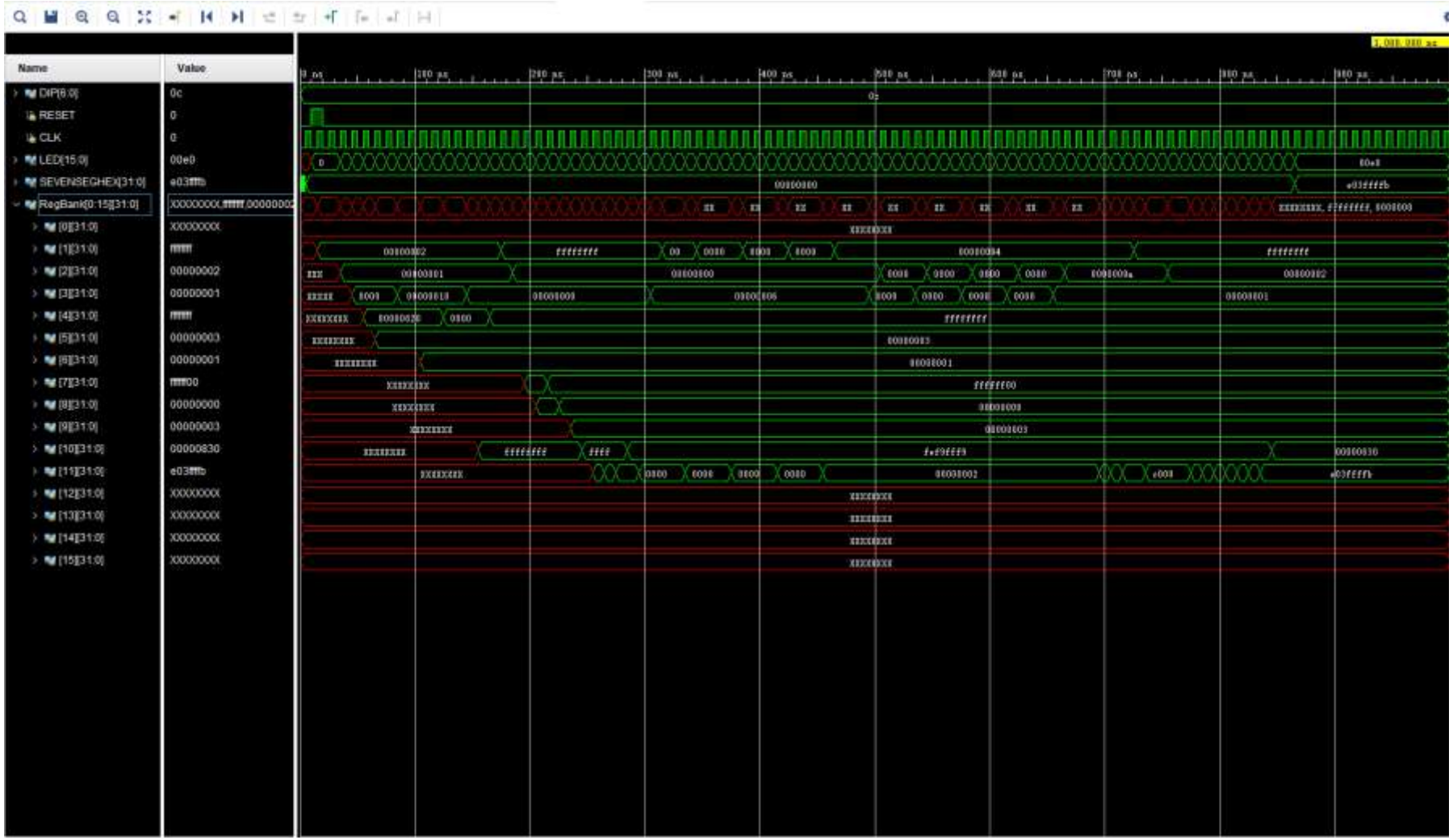
How to Test

The ideal simulation waveform should be like this:



How to Debug

Compare the sim_model， find the instruction and check.



How to Test

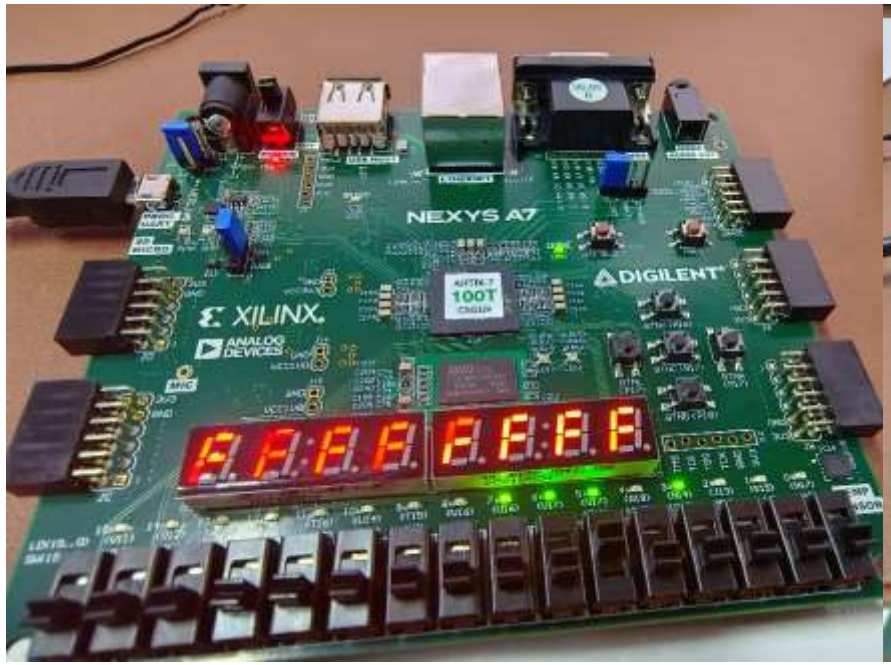
The ideal on-board implementation should be like this:

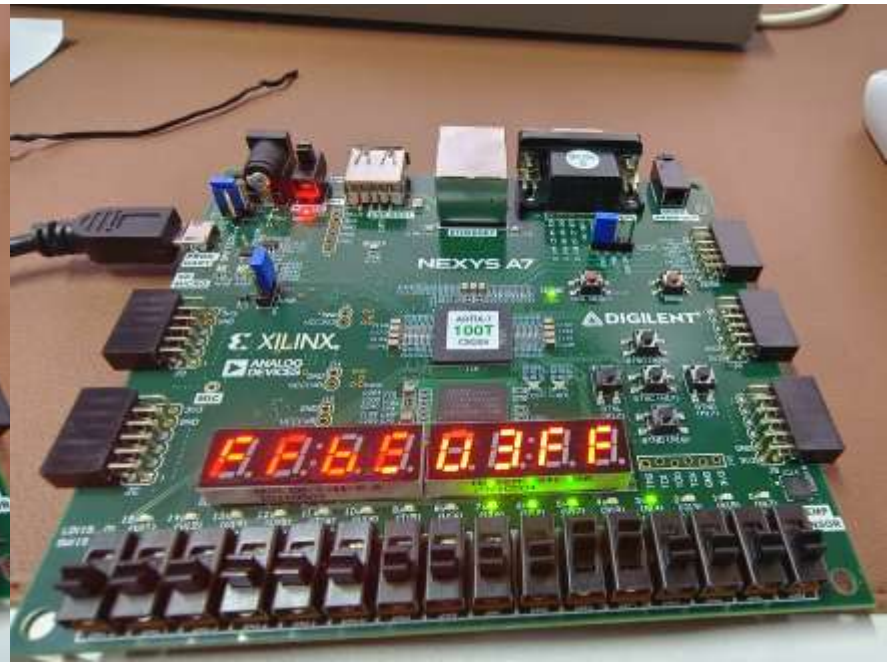
When SW5 and SW4 pulls up, the SevenSegs can show **FFBE03FF**.

When SW5 pulls up, the SevenSegs can show **FFFFFFFF**.

And the LEDs will finally stay at **0000000011101000.**

```
LDR R10,addr3;
STR R11,[R10];MEM[12]=R11
LDR R9,addr2;
STR R1,[R9];MEM[8]=R1
```





Cautious:
This test sample is given just for your own validation.
In Lab Check, we will replace it with one of the similar test samples to avoid some tricky.