



# Computer Organization

---

## Lab10 CPU Design(2)

Controller,  
ALU, Data Memory



# Topic

## ➤ CPU Design(2)

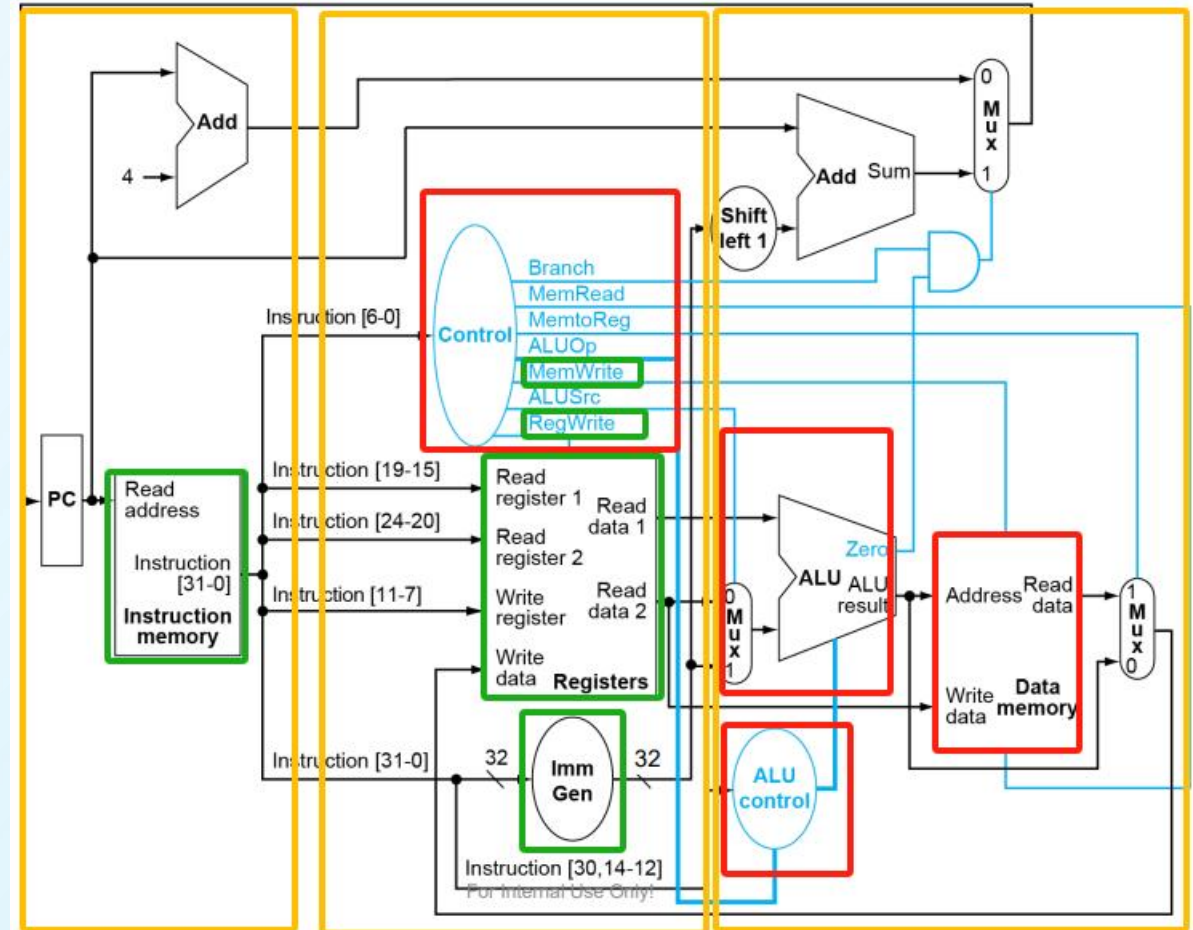
### ➤ Instruction Analysis(2)

#### ➤ Controller

### ➤ Instruction Execution(2)

#### ➤ ALU

#### ➤ Data Memory







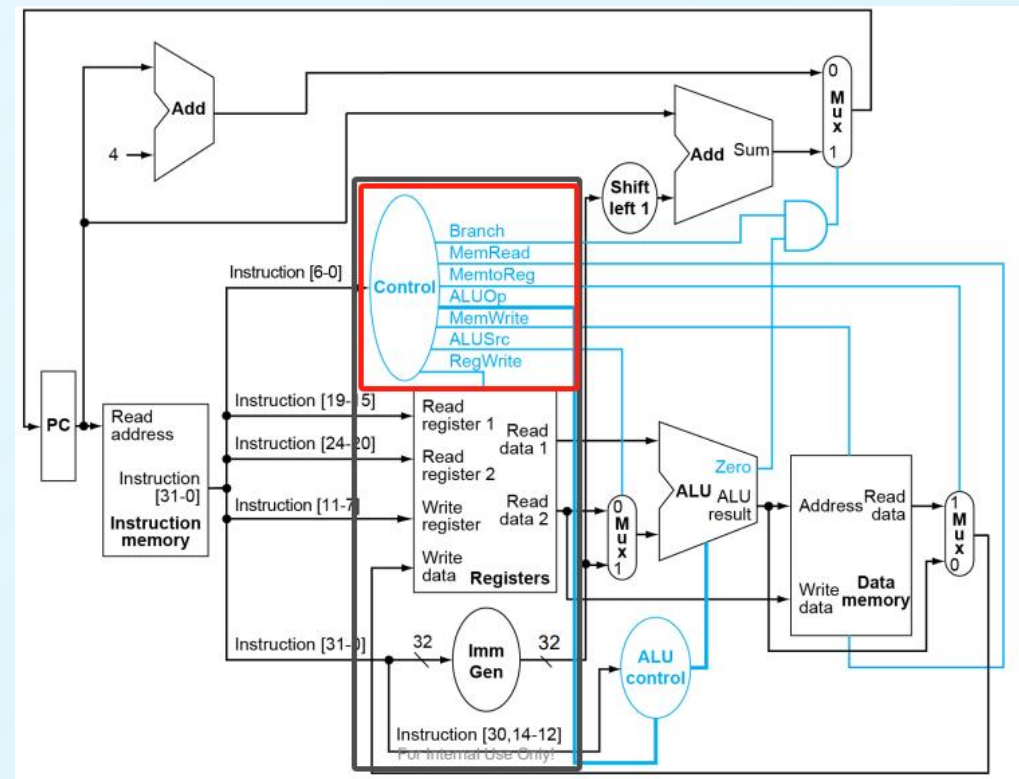
# Controller

## ➤ inputs

- opcode: instruction[6:0], Q: Are funct3 or funct7 needed here?

## ➤ outputs

- **Branch**: 1bit, 1'b1 while the instruction is beq, otherwise not.
- **MemRead**: 1bit, 1'b1 while need to read from Data Mem(load), otherwise not.
- **MemtoReg**: 1bit, 1'b1 while select the data read from memory to send to the Registers, otherwise not.
- **ALUOp**: 2bit, Need to be considered together with the ALU.
- **MemWrite**: 1bit, 1'b1 while need to write to Data Mem(store), otherwise not.
- **ALUSrc**: 1bit, 1'b1 while select Immediate as the operand of ALU, otherwise select the read data2 from Registers.
- **RegWrite**: 1bit, 1'b1 while need to write to Registers, otherwise not.



**NOTES:** Here is just a demo CPU which implements a subset of RISC-V : load, store, beq, add, sub, and, or. Larger instruction sets require deeper and more analysis and implementation.





# Controller continued

- **Circuit analysis:** After fetching the instruction, it is necessary to immediately align it for analysis, which does not involve storage. Therefore, a combinational logic circuit is used
- **outputs(demo1)**
  - MemRead: 1bit, 1'b1 while need to read from Data Mem(load), otherwise not.

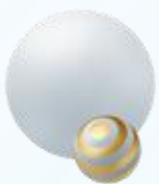
➤ option1:

```
output MemRead;
assign MemRead = (instruction[6:0]==7' h03)? 1' b1:1' b0;
```

➤ option2:

```
output reg MemRead;
always @ *
    if( instruction[6:0]==7' h03 )
        MemRead = 1' b1;
    else
        MemRead = 1' b0;
```

Inst	Name	FMT	Opcode	funct3	funct7
add	ADD	R	0110011	0x0	0x00
sub	SUB	R	0110011	0x0	0x20
xor	XOR	R	0110011	0x4	0x00
or	OR	R	0110011	0x6	0x00
and	AND	R	0110011	0x7	0x00
sll	Shift Left Logical	R	0110011	0x1	0x00
srl	Shift Right Logical	R	0110011	0x5	0x00
sra	Shift Right Arith*	R	0110011	0x5	0x20
slt	Set Less Than	R	0110011	0x2	0x00
sltu	Set Less Than (U)	R	0110011	0x3	0x00
addi	ADD Immediate	I	0010011	0x0	imm[11:5]=0x00 imm[11:5]=0x00 imm[11:5]=0x20
xori	XOR Immediate	I	0010011	0x4	
ori	OR Immediate	I	0010011	0x6	
andi	AND Immediate	I	0010011	0x7	
slli	Shift Left Logical Imm	I	0010011	0x1	
srli	Shift Right Logical Imm	I	0010011	0x5	
srai	Shift Right Arith Imm	I	0010011	0x5	
slti	Set Less Than Imm	I	0010011	0x2	
sltiu	Set Less Than Imm (U)	I	0010011	0x3	
lb	Load Byte	I	0000011	0x0	
lh	Load Half	I	0000011	0x1	
lw	Load Word	I	0000011	0x2	
lbu	Load Byte (U)	I	0000011	0x4	
lhu	Load Half (U)	I	0000011	0x5	
sb	Store Byte	S	0100011	0x0	
sh	Store Half	S	0100011	0x1	
sw	Store Word	S	0100011	0x2	
beq	Branch ==	B	1100011	0x0	
bne	Branch !=	B	1100011	0x1	
blt	Branch <	B	1100011	0x4	
bge	Branch ≥	B	1100011	0x5	
bltu	Branch < (U)	B	1100011	0x6	
bgeu	Branch ≥ (U)	B	1100011	0x7	



# Controller continued

## ➤ outputs(demo2)

### ➤ ALUOp: 2bit

➤ 2'b00 : load/store

➤ 2'b01: beq

➤ 2'b10: R-type

Q: What's the problem in the reference code below, fix it:

```
output [1:0] reg ALUOp;
```

```
always @ *
```

```
case( instruction[6:0])
```

```
7'h03,7'h23: ALUOp = 2'b00;
```

```
7'h33,7'h63: ALUOp = 2'b01;
```

```
default: ALUOp = 2'b10;
```

```
endcase
```

Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
ld	00	load doubleword	XXXXXXX	XXX	add	0010
sd	00	store doubleword	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

Inst	Name	FMT	Opcode	funct3	funct7
add	ADD	R	0110011	0x0	0x00
sub	SUB	R	0110011	0x0	0x20
xor	XOR	R	0110011	0x4	0x00
or	OR	R	0110011	0x6	0x00
and	AND	R	0110011	0x7	0x00
sll	Shift Left Logical	R	0110011	0x1	0x00
srl	Shift Right Logical	R	0110011	0x5	0x00
sra	Shift Right Arith*	R	0110011	0x5	0x20
slt	Set Less Than	R	0110011	0x2	0x00
sltu	Set Less Than (U)	R	0110011	0x3	0x00
addi	ADD Immediate	I	0010011	0x0	imm[11:5]=0x00 imm[11:5]=0x00 imm[11:5]=0x20
xori	XOR Immediate	I	0010011	0x4	
ori	OR Immediate	I	0010011	0x6	
andi	AND Immediate	I	0010011	0x7	
slli	Shift Left Logical Imm	I	0010011	0x1	
srli	Shift Right Logical Imm	I	0010011	0x5	
srai	Shift Right Arith Imm	I	0010011	0x5	
slti	Set Less Than Imm	I	0010011	0x2	
sltiu	Set Less Than Imm (U)	I	0010011	0x3	
lb	Load Byte	I	0000011	0x0	
lh	Load Half	I	0000011	0x1	
lw	Load Word	I	0000011	0x2	
lbu	Load Byte (U)	I	0000011	0x4	
lhu	Load Half (U)	I	0000011	0x5	
sb	Store Byte	S	0100011	0x0	
sh	Store Half	S	0100011	0x1	
sw	Store Word	S	0100011	0x2	
beq	Branch ==	B	1100011	0x0	
bne	Branch !=	B	1100011	0x1	
blt	Branch <	B	1100011	0x4	
bge	Branch ≥	B	1100011	0x5	
bltu	Branch < (U)	B	1100011	0x6	
bgeu	Branch ≥ (U)	B	1100011	0x7	





# Practice1

- Implement the sub-module of CPU: Controller
  - The controller is expected to support the specified set (add, sub, and, or, lw, sw, beq), demos on the courseware can serve as a reference.
  - Build the testbench to verify the function of Controller(here is a reference):

```
lw [0000a083]: Branch[0], ALUOp[0], ALUsrc[1], MemRead[1], MemWrite[0], MemtoReg[1], RegWrite[1]
add[001080b3]: Branch[0], ALUOp[2], ALUsrc[0], MemRead[0], MemWrite[0], MemtoReg[0], RegWrite[1]
sw [00102423]: Branch[0], ALUOp[0], ALUsrc[1], MemRead[0], MemWrite[1], MemtoReg[0], RegWrite[0]
beq[fe1008e3]: Branch[1], ALUOp[1], ALUsrc[0], MemRead[0], MemWrite[0], MemtoReg[0], RegWrite[0]
bne[00101663]: Branch[0], ALUOp[1], ALUsrc[0], MemRead[0], MemWrite[0], MemtoReg[0], RegWrite[0]
sub[403000b3]: Branch[0], ALUOp[2], ALUsrc[0], MemRead[0], MemWrite[0], MemtoReg[0], RegWrite[1]
and[0041f133]: Branch[0], ALUOp[2], ALUsrc[0], MemRead[0], MemWrite[0], MemtoReg[0], RegWrite[1]
or [005261b3]: Branch[0], ALUOp[2], ALUsrc[0], MemRead[0], MemWrite[0], MemtoReg[0], RegWrite[1]
```

- it's strongly recommended to think about that which input and output ports, as well as internal processing logic, the controller needs to add if it needs to support instructions other than the specified set (add, sub, and, or, lw, sw, beq) in RISC-V32I.



➤ inputs:

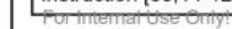
- Q: Are funct3 or funct7 needed here?**

➤ outputs:

- ✓ **ALUResult:** 32bit, the result of calculation in ALU
- ✓ **zero:** 1bit, 1' b1 while AULResult is zero, otherwise not

➤ functions:

- ✓ 1. generate **ALUControl (4bit)**
- ✓ 2. select **operand2** from ReadData2 and imm32
- ✓ 3. calculate on ReadData1 and operand2 according to the **ALUControl**, assign the calculation result to **ALUResult**
- ✓ 4. generate **zero** according to ALUResult







# ALU continued

## ALU:

- **Circuit analysis:** After get the control signals from controller and datas from Decoder, it is necessary to immediately calculate the result which does not involve storage. Therefore, a combinational logic circuit is used.
- Implements (Demo for load, store, beq, add, sub, and, or):
  - ✓ 1. generate **ALUControl**

```
reg [3:0] ALUControl;
```

```
always @ *  
    case( ALUOp)  
        2'b00,2'b01: ALUControl = { ALUOp, 2'b10};  
        2'b10:  
            //complete code here  
    endcase
```

Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
ld	00	load doubleword	XXXXXXX	XXX	add	0010
sd	00	store doubleword	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

## ALU:

- Implements(Demo):
  - ✓ 3. calculate on ReadData1 and operand2 according to the ALUControl, assign the calculation result to **ALUResult**

```
output reg [31:0] ALUResult;
```

```
//operand2 is the 2nd operand for the calculation  
always @ *  
    case( ALUControl)  
        4'b0010: ALUResult= ReadData1 + operand2;  
        4'b0110: ALUResult= ReadData1 - operand2;  
        //complete code here  
    endcase
```



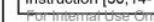
➤ Demos:

- ✓ 2. select **operand2** from ReadData2 and imm32

```
assign operand2 = (ALUSrc==1)? ReadData2 : imm32;
```

- ✓ 4. generate **zero** according to ALUResult

```
assign zero = (ALUResult==1)? 1'b1: 1'b0;
```





# Practice2

- Implement the sub-module of CPU: ALU
  - The ALU is expected to support the specified set (add, sub, and, or, lw, sw, beq), demos on the courseware can serve as a reference.
  - Build the testbench to verify the function of ALU
    - TIPS: It's suggested that integrating controller and alu into one module for testing after finish the function verification on Controller.
- it's strongly recommended to think about that which input and output ports, as well as internal processing logic, the ALU needs to add if it needs to support instructions other than the specified set (add, sub, and, or, lw, sw, beq) in RISC-V32I.



# Data Path(2) - Data Memory

- **Circuit analysis:** Due to the involvement of storage, this circuit is a sequential logic circuit.

- ✓ **Q1:** What's the bitwidth of address and data interface?
- ✓ **Q2:** Would memory units be read and written simultaneously in a single cycle RISC-V CPU?

- **inputs**

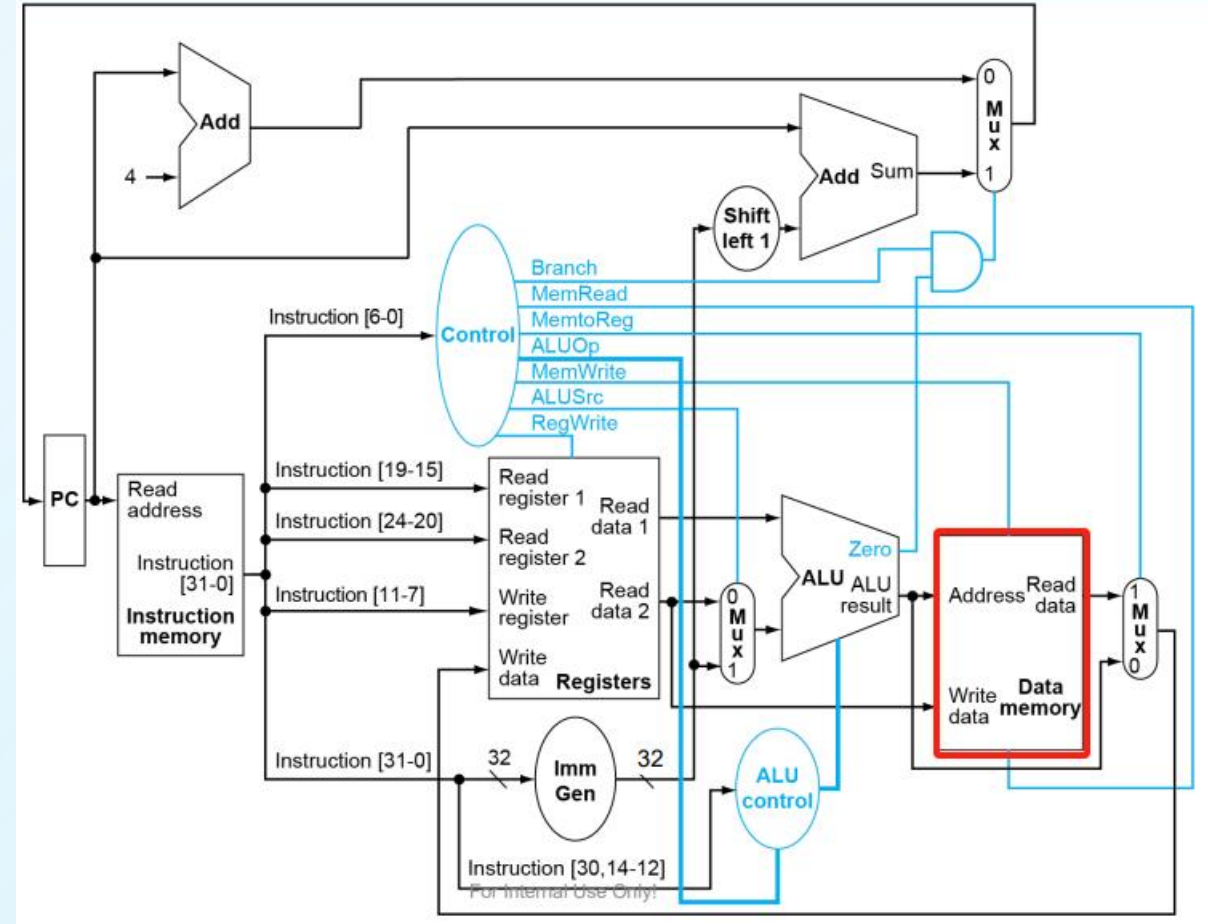
- ✓ **clk**
- ✓ **MemRead, MemWrite**
- ✓ **Address**
- ✓ **WriteData**

- **outputs**

- ✓ **ReadData**

- **Implements:**

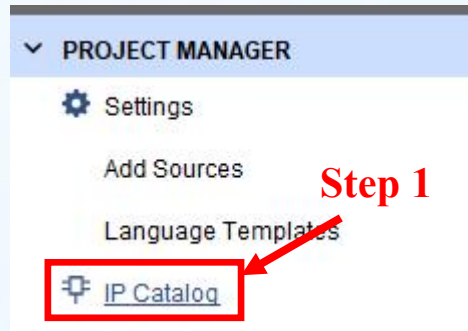
- ✓ **Using the IP core 'Block Memory' of Xilinx to implement the Data-memory.**





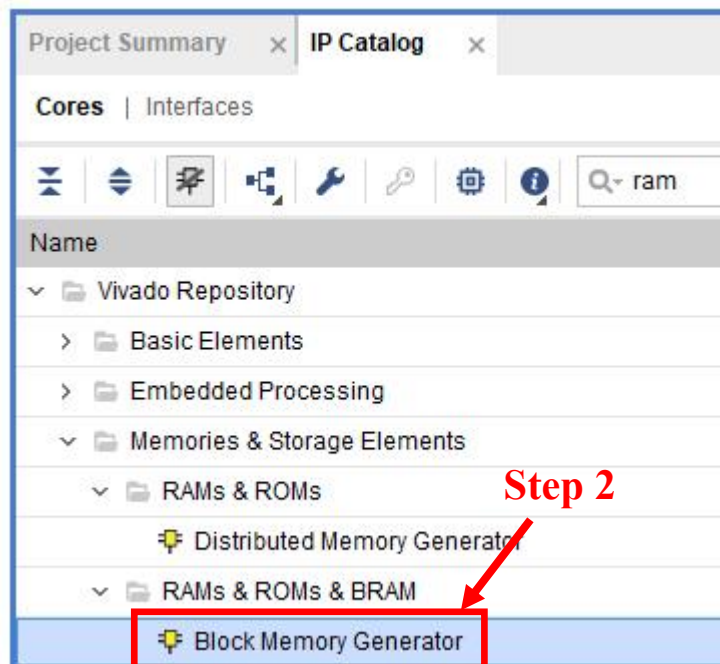
# Using IP core in Vivado: Block Memory

Using the IP core 'Block Memory' of Xilinx to implement the Data-memory.



Import the IP core in vivado project

1) in “**PROJECT MANAGER**” window  
click “**IP Catalog**”



2) in “**IP Catalog**” window

> Vivado Repository

> Memories & Storage Elements

> RAMs & ROMs & BRAM

> **Block Memory Generator**



# Customize Memory IP core

Component Name

**Basic** | Port A Options | Other Options | Summary

Interface Type:  ☐ Generate address interface with 32 bits

Memory Type:  ☐ Common Clock

**ECC Options**

ECC Type:  ☐ Error Injection Pins:

**Write Enable**

☐ Byte Write Enable

Byte Size (bits):

**Algorithm Options**

Defines the algorithm used to concatenate the block RAM primitives. Refer datasheet for more information.

Algorithm:  Primitive:

## Customize memory IP core

- 1) Component Name: **RAM**
- 2) Basic settings:
  - Interface Type: **Native**
  - Memory Type: **Single-port RAM**
  - ECC options: **no ECC check**
  - Algorithm options: **Minimum area**





# Customize Memory IP core continued

Component Name RAM

Basic Port A Options Other Options Summary

**Memory Size**

Write Width	32	Range: 1 to 4608 (bits)
Read Width	32	
Write Depth	16384	Range: 2 to 1048576
Read Depth	16384	

Operating Mode Write First Enable Port Type Always Enabled

**Port A Optional Output Registers**

<input type="checkbox"/> Primitives Output Register	<input type="checkbox"/> Core Output Register
<input type="checkbox"/> SoftECC Input Register	<input type="checkbox"/> REGCEA Pin

**Port A Output Reset Options**

<input type="checkbox"/> RSTA Pin (set/reset pin)	Output Reset Value (Hex) 0
<input type="checkbox"/> Reset Memory Latch	Reset Priority CE (Latch or Register Enable)

## 3) PortA Options settings:

➤ Data read and write **bit width**: 32 bits (4Byte)

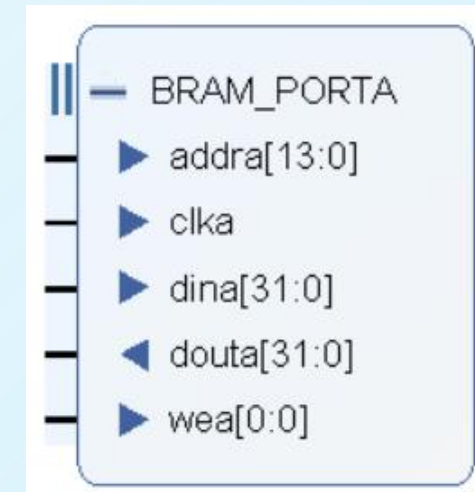
➤ Write/Read **Depth**: 16384

$$\text{size: } 2^{14} * 4\text{Byte} = 64\text{KB}$$

➤ Operating Mode: Write First

➤ Enable Port Type: Always Enabled

➤ PortA Optional Output Registers: **NOT SET**

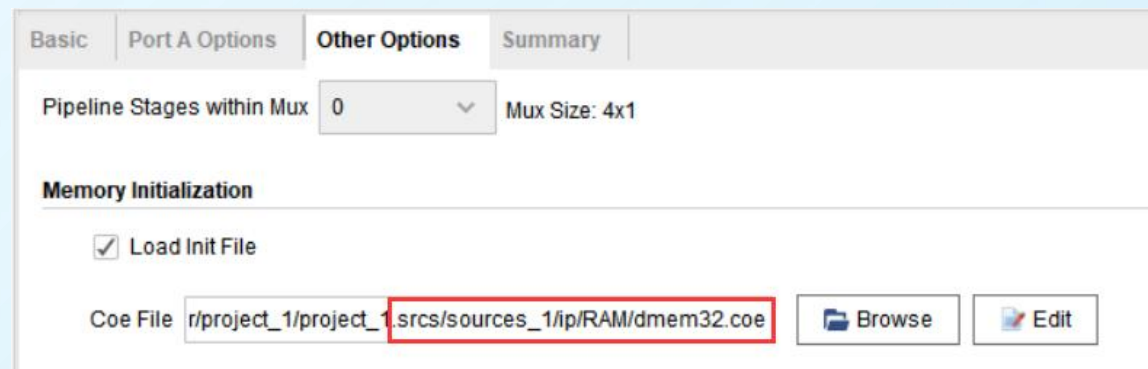
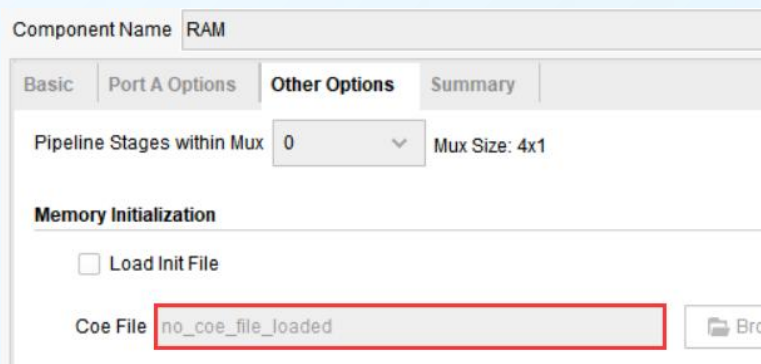




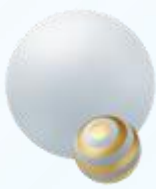
# Customize Memory IP core continued

## 4) Other Options settings:

- 1. When **specifying the initialization file** for customize the RAM on the 1st time, the IP core RAM just customized **WITHOUT initial file** and **corresponding path**, so set it to **no initial file** when creating RAM.
- 2. **After** the RAM IP core created
  - 2-1. **COPY** the initialization file **dmem32.coe** to **projectName.srscs/sources\_1/ip/ComponentName**. (“projectName.srscs” is under the project folder, “componentName” here is ‘RAM’)
  - 2-2. Double-click the newly created RAM IP core, **RESET** it with the **initialization file**, select the **dmem32.coe** file that has been in the directory of projectName.srscs/sources\_1/ip/RAM.



Tips: “dmem32.coe” file could be found in the directory “labs/lab10” of course blackboard site



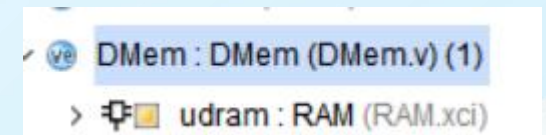
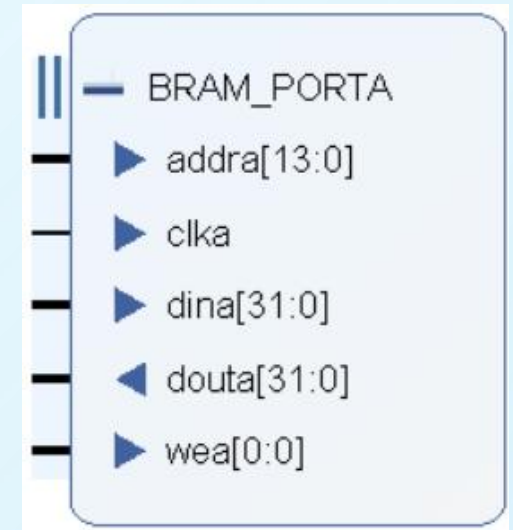
# Instance the Memory IP core

**Step1.** Find the name and the ports of the IP core:

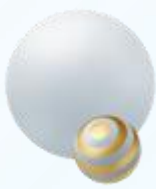
Component Name

**Step2.** Build a module DMem in verilog to instance the IP core and bind its ports:

```
module DMem(  
    input clk,  
    input MemRead, MemWrite,  
    input [31:0] addr,  
    input [31:0] din,  
    output [31:0] dout);  
  
    RAM udrum(.clka(clk), .wea(MemWrite), .addra(addr[13:0]), .dina(din), .douta(dout));  
  
endmodule
```







# Test the IP core(1)

**Step1.** Build the testbench to verify the function of the IP core.

**Step2.** do the simulation based on the testbench.

**Step3.** Check the waveform generated by the simulation and the coe file which used to initialize the IP core to check if the RAM IP core work as a RAM.

```
module tb_dmem();  
  
    reg clk, MemRead, MemWrite;  
    reg [31:0] addr, din;  
    wire [31:0] dout;  
    DMem udmem(.clk(clk),  
        .MemRead(MemRead), .MemWrite(MemWrite),  
        .addr(addr), .din(din),  
        .dout(dout));  
  
    initial begin...  
  
    initial begin...  
        1  
  
    initial begin...  
  
    initial begin...  
endmodule
```





## Test the IP core(2)

```
DMem udmem(.clk(clk),  
  .MemRead(MemRead), .MemWrite(MemWrite),  
  .addr(addr), .din(din),  
  .dout(dout));
```

```
initial begin  
  clk = 1'b0;  
  forever #5 clk=~clk;  
end
```

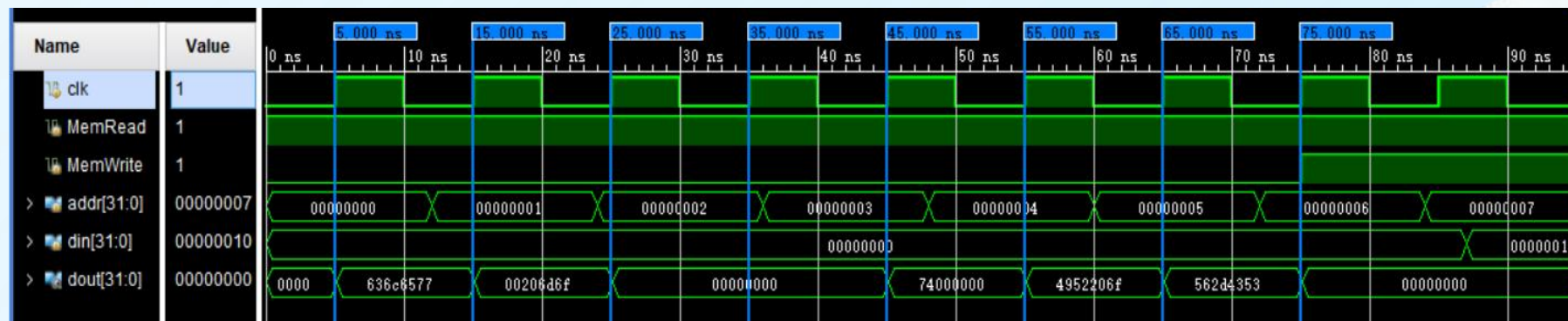
```
initial begin  
  MemWrite = 1'b0;  
  MemRead = 1'b1;  
  #75  
  MemWrite = 1'b1;  
end
```

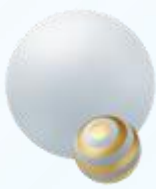
```
initial begin  
  din = 32'h0;  
  #75 repeat(10) #12 din = din+16;  
end
```

```
initial begin  
  addr = 32'h0;  
  repeat(20) #12 addr = addr + 1;  
  #20 $finish;  
<
```

Determine whether the module can accurately read the data stored in the corresponding storage unit in the RAM(which is **initialized with file dmem32.coe**) based on the address on the **rising edge** of the clock.(while wea is 1'b0)

```
1  memory_initialization_radix = 16;  
2  memory_initialization_vector =  
3  636c6577,  
4  00206d6f,  
5  00000000,  
6  74000000,      dmem32.coe  
7  4952206f,  
8  562d4353,  
9  00000000,  
10 00000000,  
11 00000000,  
12 00000000,
```





# Test the IP core(3)

```
DMem udmem(.clk(clk),  
  .MemRead(MemRead), .MemWrite(MemWrite),  
  .addr(addr), .din(din),  
  .dout(dout));
```

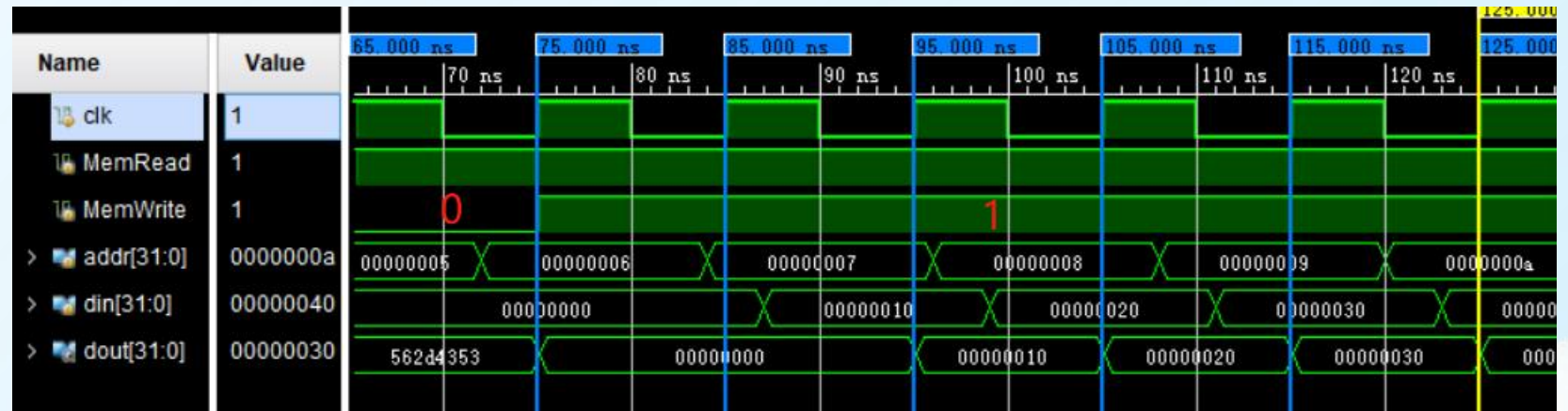
```
initial begin  
  clk = 1'b0;  
  forever #5 clk=~clk;  
end
```

```
initial begin  
  MemWrite = 1'b0;  
  MemRead = 1'b1;  
  #75  
  MemWrite = 1'b1;  
end
```

```
initial begin  
  din = 32'h0;  
  #75 repeat(10) #12 din = din+16;  
end
```

```
initial begin  
  addr = 32'h0;  
  repeat(20) #12 addr = addr + 1;  
  #20 $finish;  
<
```

Determine whether the module can accurately write and read the data stored in the corresponding storage unit in the RAM based on the address on the rising edge of the clock.





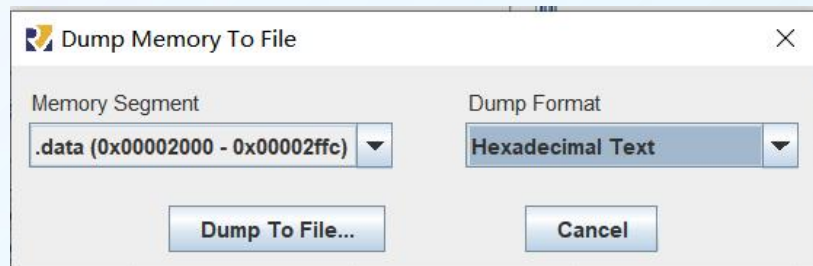


# Tips. How to generate coe file

- ✓ 1-1. build a RISC-V assembly source file.
- ✓ 1-2. Using Rars to assemble the source file.
- ✓ 1-3. Dump the data as **Hexadecimal Text**.
- ✓ 1-4. Using rars2coe.exe to generate the related coe file.

```
.data  
str1: .ascii "welcom "  
bs: .space 8  
str2: .asciz "to RISC-V"
```

```
>rars2coe.exe test_data.txt dmem32.coe
```



dmem32	COE 文件
rars2coe	应用程序
test_data	文本文档

```
1 memory_initialization_radix = 16;  
2 memory_initialization_vector =  
3 636c6577,  
4 00206d6f,  
5 00000000,  
6 74000000,  
7 4952206f,  
8 562d4353,  
9 00000000,  
10 00000000,
```

# Practice 3-1

```
module DMem(
input clk,
input MemRead,MemWrite,
input [31:0] addr,
input [31:0] din,
output[31:0] dout);

RAM udram(.clka(clk), .wea(MemWrite),
.addra(addr[13:0]), .dina(din),
.douta(dout));

endmodule
```

```
module tb_dmem(); //part1 of tb

reg clk, MemRead, MemWrite;
reg [31:0] addr, din;
wire [31:0] dout;
DMem udmem(.clk(clk),
.MemRead(MemRead), .MemWrite(MemWrite),
.addr(addr), .din(din),
.dout(dout));

initial begin
clk =1'b0;
forever #5 clk=~clk;
end
```

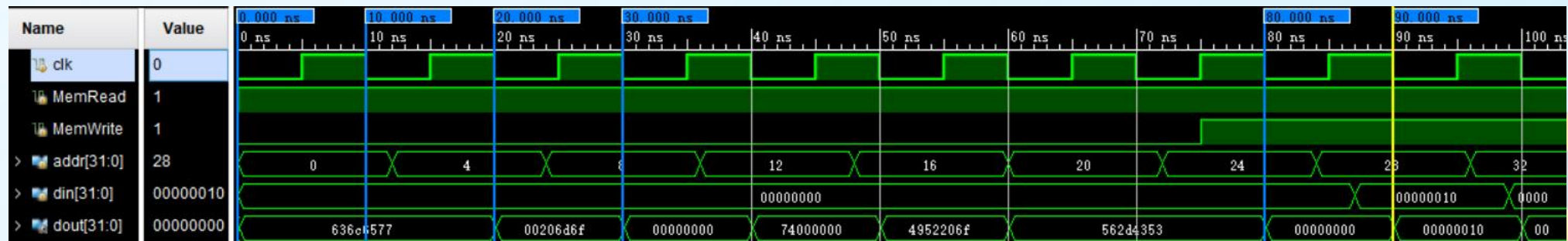
```
initial begin //part2 of tb
MemWrite = 1'b0;
MemRead = 1'b1;
#75
MemWrite = 1'b1;
end

initial begin
din = 32'h0;
#75 repeat(10) #12 din = din+16;
end

initial begin
addr =32'h0;
repeat(20) #12 addr = addr + 4;
#20 $finish;
end
endmodule
```

Modify the module "DMem" on the left top hand to achive:

- 1) the module read and write RAM on the **negedge** of the clk
- 2) the **"addr"** in DMem are based on the **byte** instead of word.
- Simulate the updated DRAM using the tb on this page, and the expected waveform is as follows





## Practice 3-2

3-2-1. Please consider whether the address (value of PC register) in IMEM is based on bytes or words. If it is based on bytes, how to modify the IFetch module?

3-2-2. Please review the relationship between each sequential circuit module in the CPU and the clock. If it is necessary to complete the process of fetching instructions, analyzing instructions, and executing instructions within one cycle, what type of clock edge would be arranged to the each sequential circuit module?