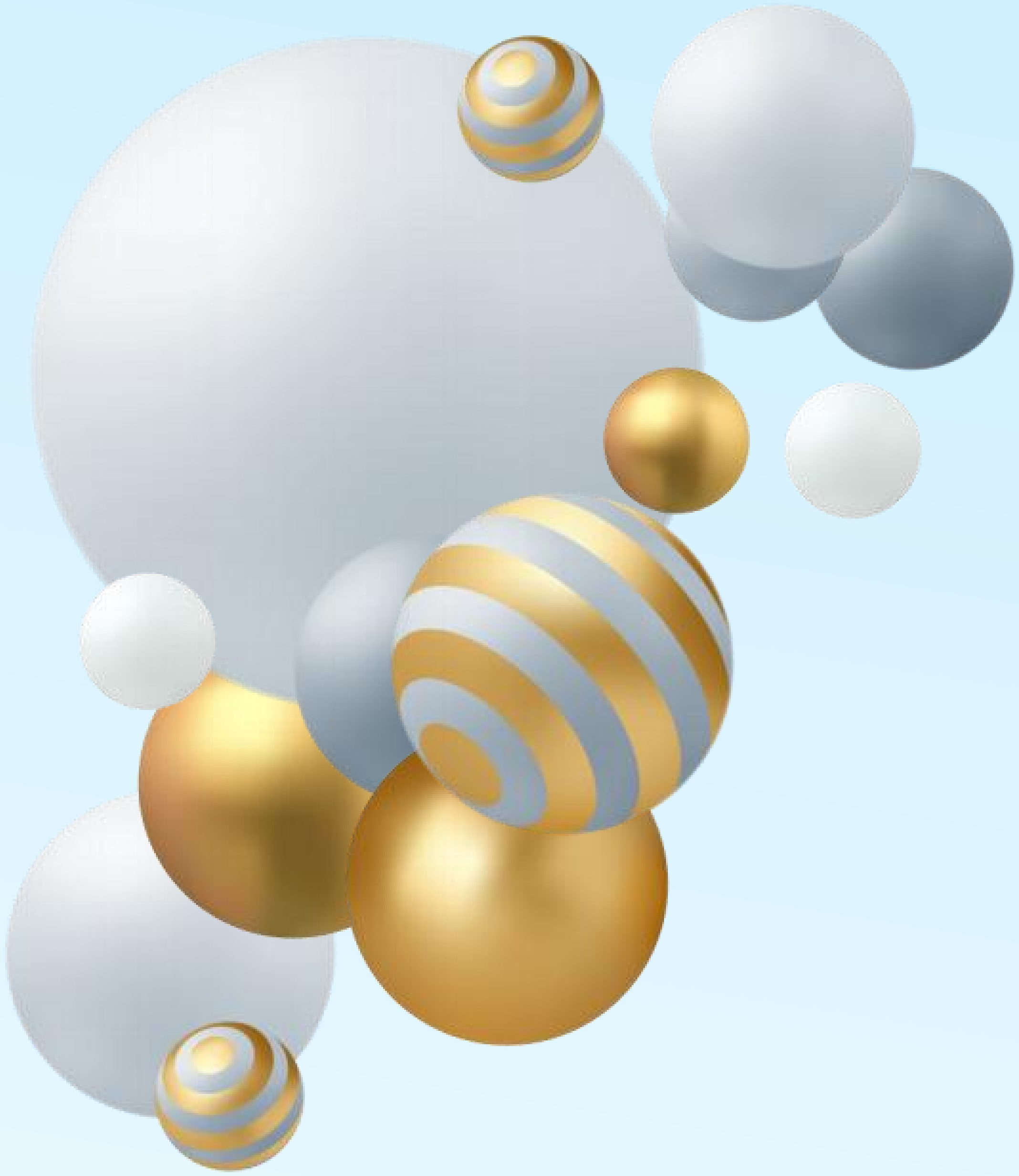


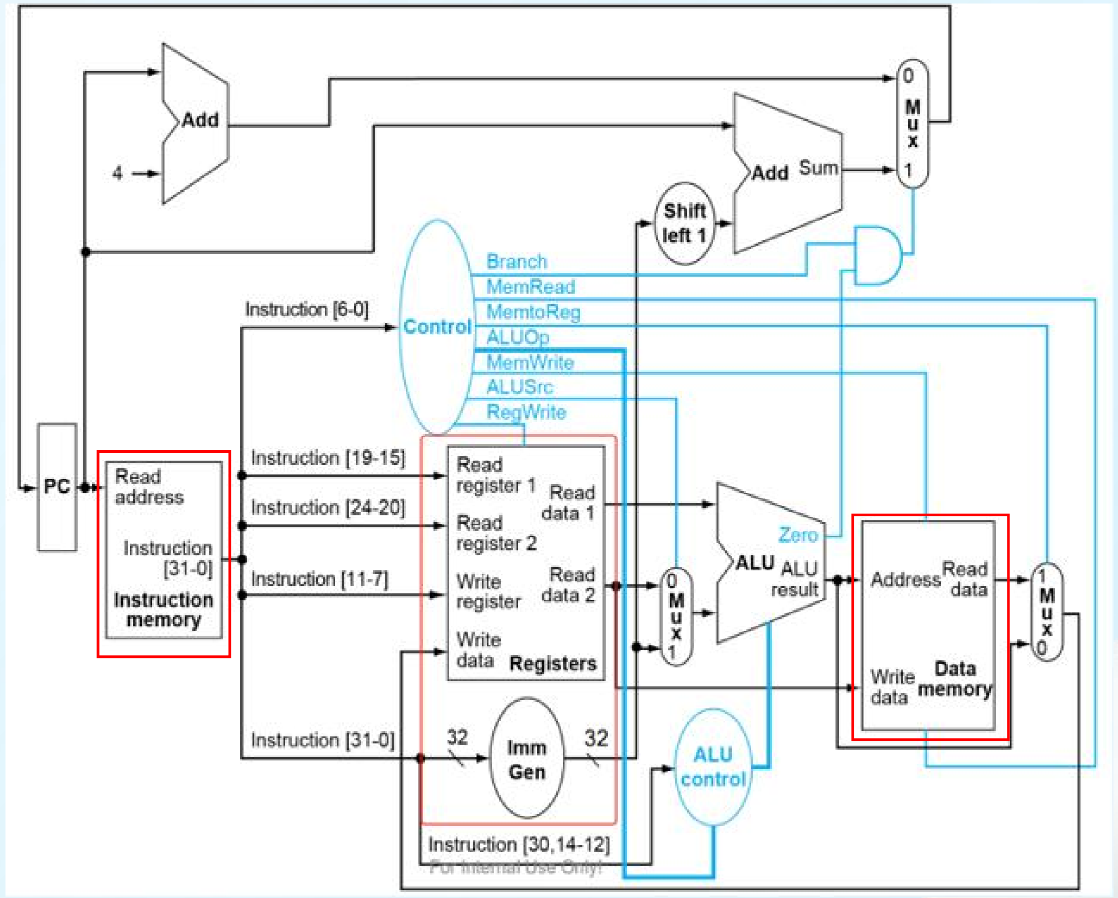
Computer Organization

Lab9 CPU Design(1)

ISA, Control + Data
Block Memory,Decoder

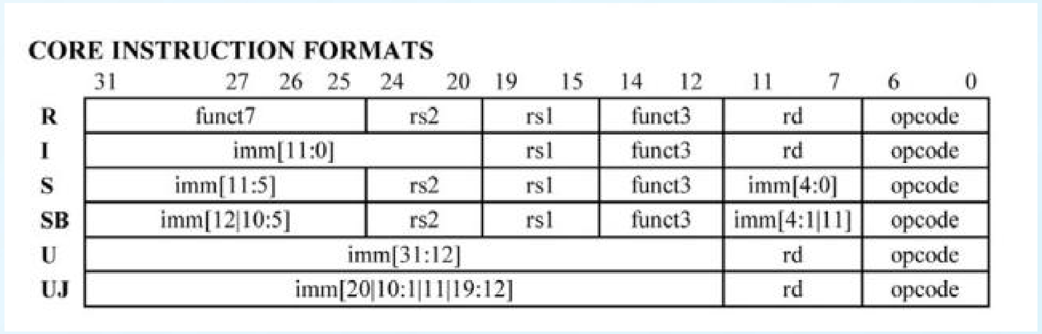
Topic

- CPU Design(1)
  - ISA
  - CPU = Control Path + Data Path
  - Data Path(1)
    - Instruction Fetch(1)
      - IP cores(Block Memory) ROM
    - Instruction analysis(1)
      - Decoder of Data Path
    - Instruction execution(1)
      - IP cores(Block Memory) RAM



RISC-V32I

RISC-V-Reference-Data.pdf



CORE INSTRUCTION FORMATS

	31 27 26 25	24 20	19 15	14 12	11 7	6 0
R	funct7	rs2	rs1	funct3	rd	opcode
I	imm[11:0]		rs1	funct3	rd	opcode
S	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]				rd	opcode
UJ	imm[20|10:1|11|19:12]				rd	opcode

I type {imm[11:0], rs1, funct3, rd, opcode}

inst	OPCODE	FUNCT3	FUNC7	hex
lb	**000_0011**	0		**03**/0/-
lh	**000_0011**	1		03/1/-
lw	**000_0011**	10		03/2/-
ld	**000_0011**	11		03/3/-
lbu	**000_0011**	100		03/4/-
lhu	**000_0011**	101		03/5/-
jalr	**110_0111**	0		**67/0/-**
addi	**001_0011**	0		**13/0/-**
slli	**001_0011**	1	0	13/1/0
slti	**001_0011**	10		13/2/-
sltiu	**001_0011**	11		13/3/-
xori	**001_0011**	100		13/4/-
srli	**001_0011**	101	0	13/5/0
srai	**001_0011**	101	010_0000	13/5/20
ori	**001_0011**	110		13/6/-
andi	**001_0011**	111		13/7/-

S type {imm[11:5], rs2, rs1, funct3, imm[4:0], opcode}

inst	OPCODE	FUNCT3	FUNC7	hex
sb	**010_0011**	0		**23/**
sh	**010_0011**	1		
sw	**010_0011**	10		
sd	**010_0011**	11		

R type {func7, rs2, rs1, func3, rd, opcode}

inst	OPCODE	FUNCT3	FUNC7	hex
add	**011_0011**	0	0	**33/**
sub	**011_0011**	0	010_0000	
sll	**011_0011**	1	0	
slt	**011_0011**	10		
sltu	**011_0011**	11		
xor	**011_0011**	100		
srl	**011_0011**	101	0	
sra	**011_0011**	101	010_0000	
or	**011_0011**	110		
and	**011_0011**	111		

SB type {imm[12|10:5], rs2, rs1, funct3, imm[4:1|11], opcode}

inst	OPCODE	FUNCT3	FUNC7	hex
beq	**110_0011**	0		**63/**
bne	**110_0011**	1		
blt	**110_0011**	100		
bge	**110_0011**	101		
bltu	**110_0011**	110		
bgeu	**110_0011**	111		

U type {imm[31:12], rd, opcode}

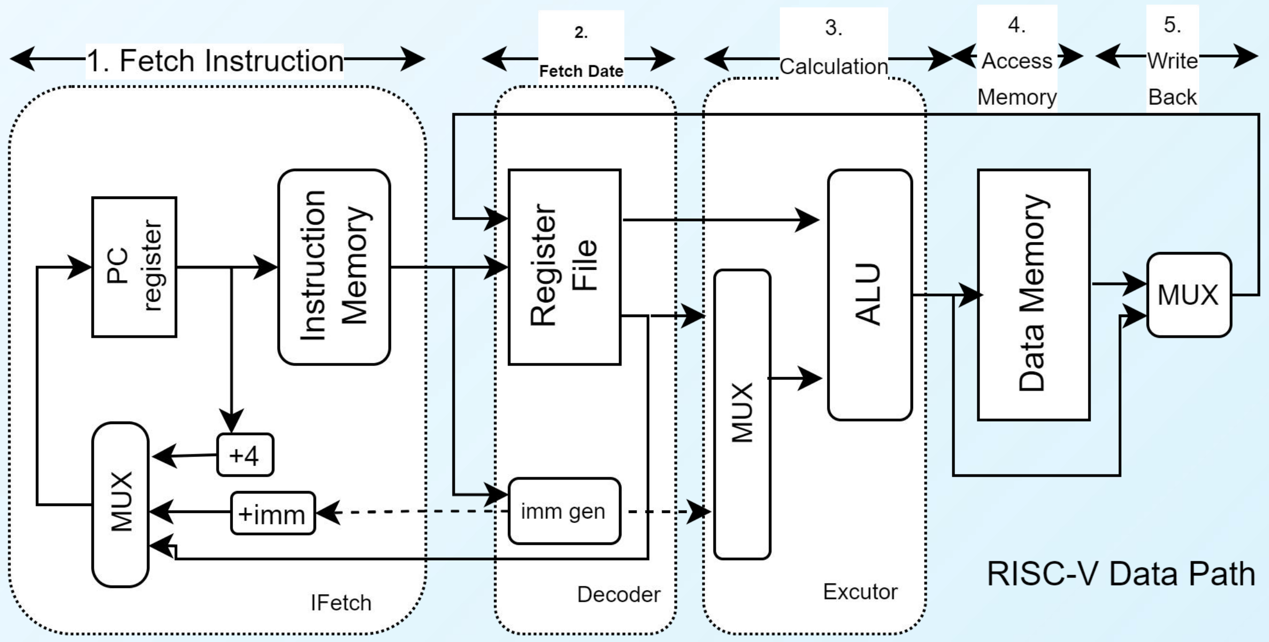
inst	OPCODE	FUNCT3	FUNC7	hex
auipc	**001_0111**			**17/**
lui	**011_0111**			**37/**

UJ type {imm[20|10:1|11|19:12], rd, opcode}

inst	OPCODE	FUNCT3	FUNC7	hex
jal	**110_1111**			**6f/**

Data Path(1)

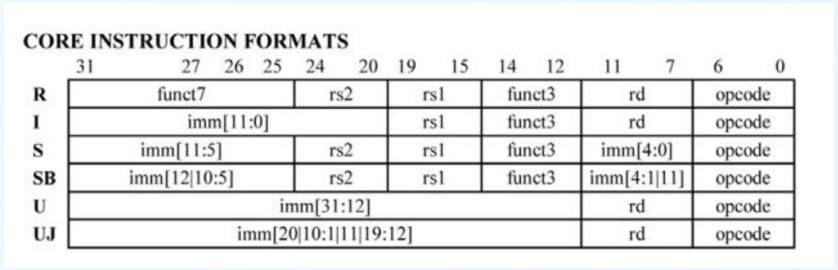
Data Path:
The parts in CPU with componets which are involved to execute instructions.



RISC-V Data Path

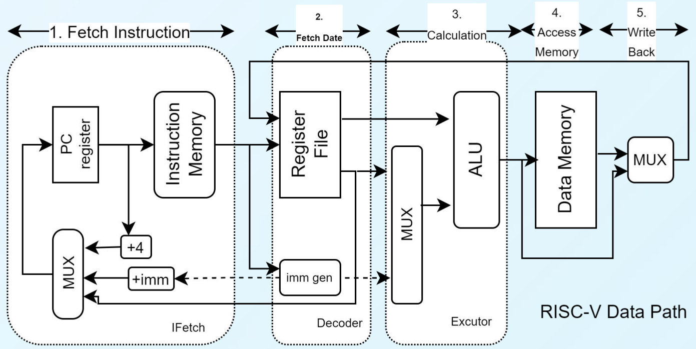
	1. Instruction fetch	2. Data Fetch	3. Instruction Execute	4. Memory Access	5. Register WriteBack
add[R]	Y	Y	Y		Y
addi[I]	Y	Y	Y		Y
sw[S]	Y	Y	Y	Y	
lw[I]	Y	Y	Y	Y	Y
branch[SB]	Y	Y	Y		
jalr[I]	Y	Y	Y		Y
jal [UJ]	Y	Y	Y		Y

Data Path(2)



CORE INSTRUCTION FORMATS

	31 … 25	24 … 20	19 … 15	14 … 12	11 … 7	6 … 0
R	funct7	rs2	rs1	funct3	rd	opcode
I	imm[11:0]		rs1	funct3	rd	opcode
S	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]				rd	opcode
UJ	imm[20|10:1|11|19:12]				rd	opcode



RISC-V Data Path

Module	How To Process	Instructions and Comments
IFetch	Determine how to update the value of PC register	1. pc+immediate(sign extended) (bne [**b**], jal[J]) 2. the value of the specified register (ra by default) (jalr [I]) 3. pc+4 (other instruction except branch, jal and jal)
Decoder	Determine whether to write register or not	lw [I], R type, U type vs beq([B]) vs sw([S])
	Get the source of data to be written Get the address of register to be written	1. Data memory (lw[I]) -> rd 2. ALU ([R]) -> rd 3. address of instruction (jal[J]) -> the specified register
	Determine whether to get immediate data from the instruction and expand it to 32bit	add([R]) vs addi([I]) vs lui([U]) vs sw([S]) vs beq([B]) vs jal([J])
Memory	Determine whether to write memory or not	(sw[S]) vs lw[I]
	Get the source of data to be written	rs1 vs rs2 of registers (sw[S])
	Get the address of memory unit to be written	the output of ALU (sw[S])
ALU	Determine how to calculate the datas	add, sub, or, sll, sltiu, sra, slt, branch ...
	determin the source of one operand from register or immediate extended	R(register), I(sign extended immediate)

Instruction Fetch(1) - Instruction Memory

Using the **IP core** ‘**Block Memory’** of Xilinx to implement the Instruction-memory.

Import the **IP core** in vivado project

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

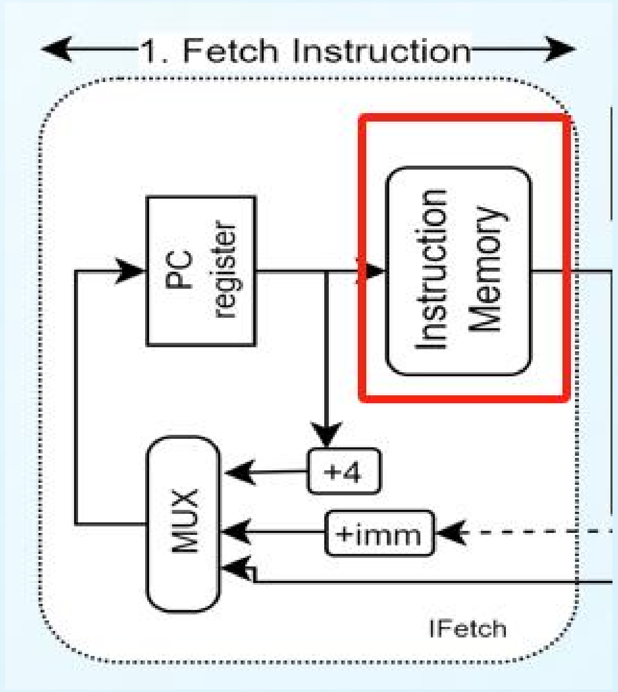
2) in “**IP Catalog**” window

> Vivado Repository

> Memories & Storage Elements

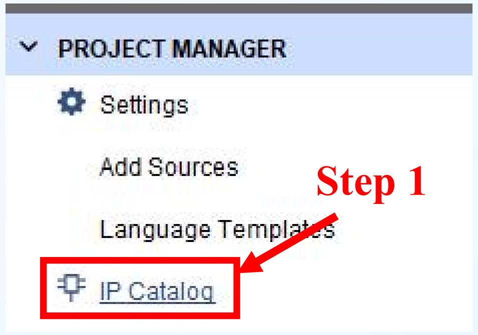
> RAMs & ROMs & BRAM

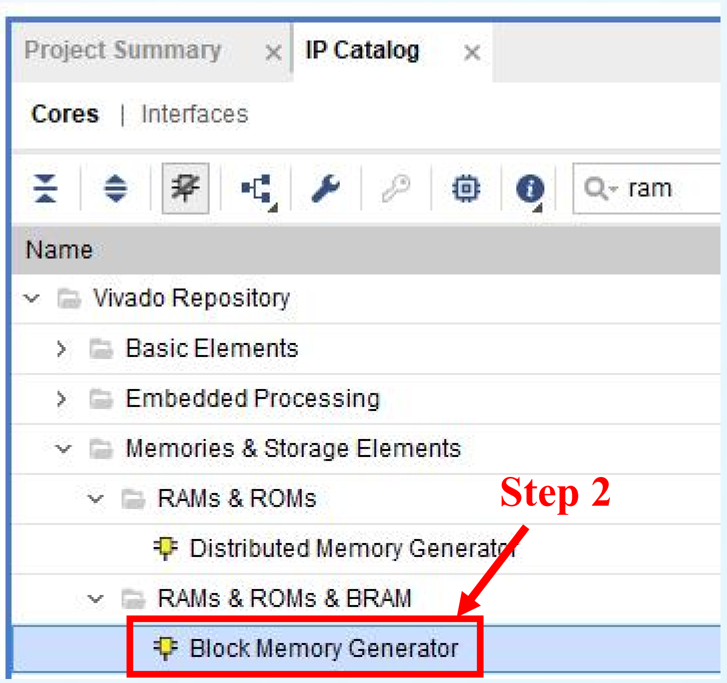
> **Block Memory Generator**



Using IP core in Vivado: Block Memory

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


PROJECT MANAGER
Settings
Add Sources
Language Templates
IP Catalog

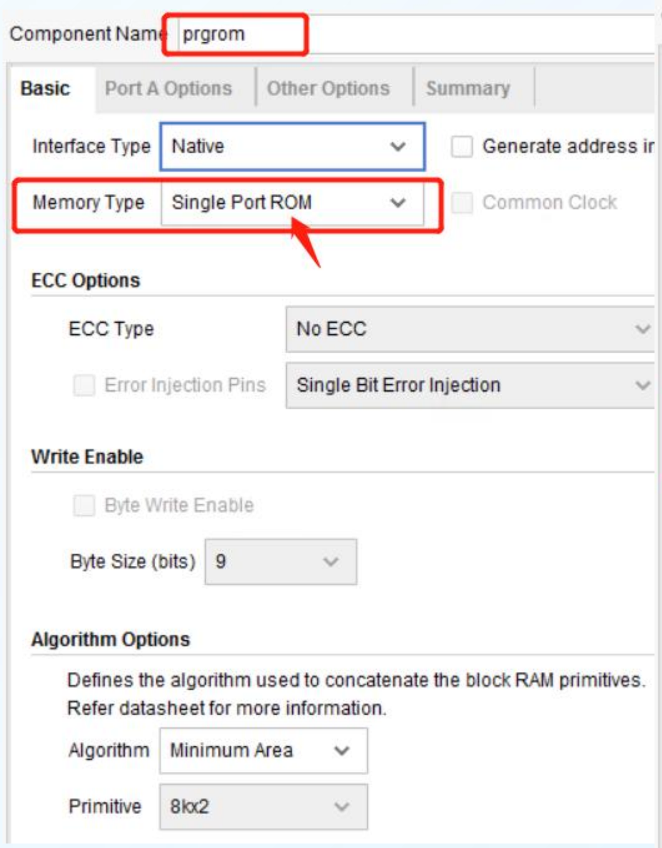

Project Summary | IP Catalog
Cores | Interfaces
Search: ram
Name
- Vivado Repository
  - Basic Elements
  - Embedded Processing
  - Memories & Storage Elements
    - RAMs & ROMs
      - Distributed Memory Generator
    - RAMs & ROMs & BRAM
      - Block Memory Generator

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: prgrom

Basic | Port A Options | Other Options | Summary

Interface Type: Native — ☐ Generate address ir

Memory Type: Single Port ROM — ☐ Common Clock

ECC Options

ECC Type: No ECC

☐ Error Injection Pins: Single Bit Error Injection

Write Enable

☐ Byte Write Enable

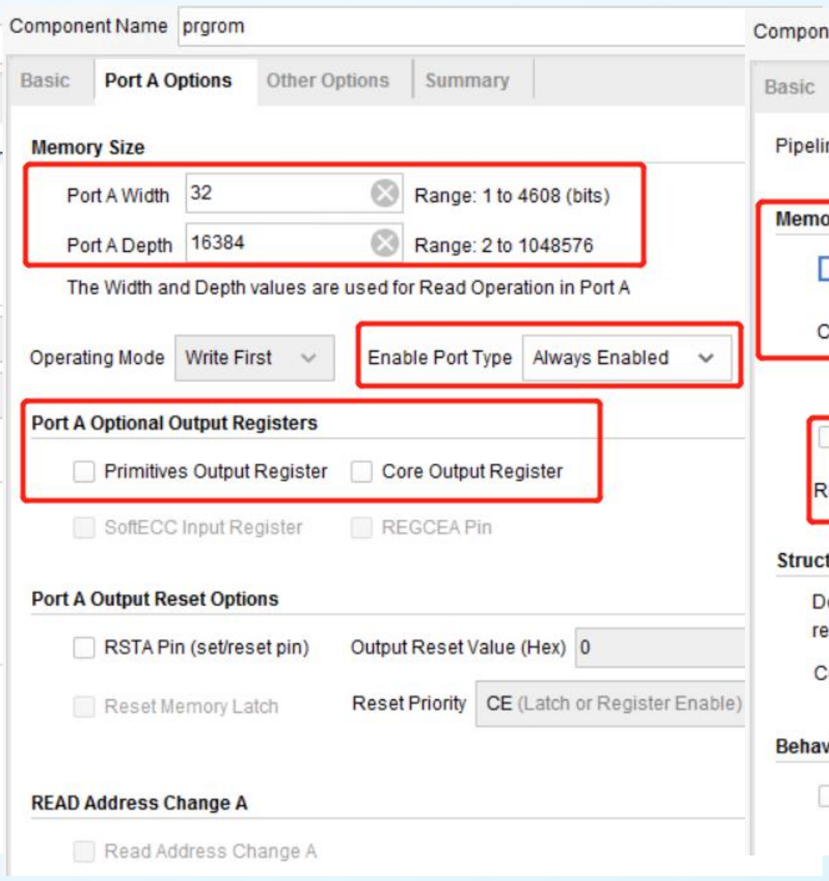
Byte Size (bits): 9

Algorithm Options

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

Algorithm: Minimum Area

Primitive: 8kx2



Component Name: prgrom

Basic | Port A Options | Other Options | Summary

Memory Size

Port A Width: 32 — Range: 1 to 4608 (bits)

Port A Depth: 16384 — Range: 2 to 1048576

The Width and Depth values are used for Read Operation in Port A

Operating Mode: Write First — Enable Port Type: Always Enabled

Port A Optional Output Registers

☐ Primitives Output Register ☐ Core Output Register

☐ SoftECC Input Register ☐ REGCEA Pin

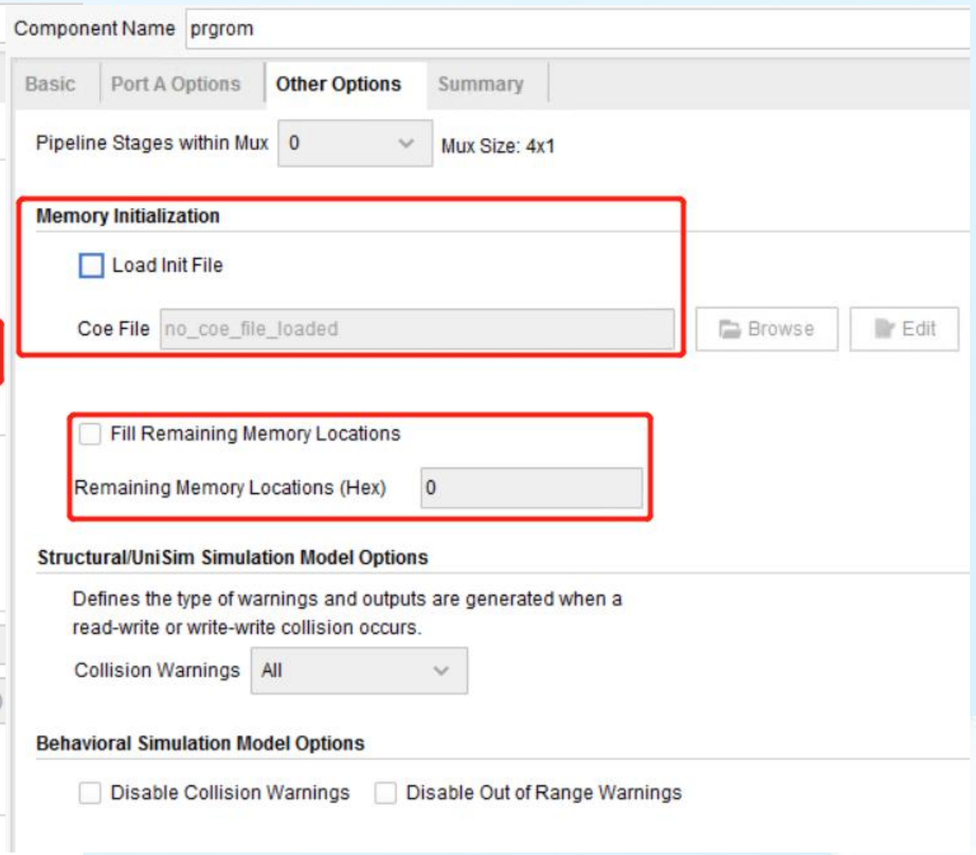
Port A Output Reset Options

☐ RSTA Pin (set/reset pin) — Output Reset Value (Hex): 0

☐ Reset Memory Latch — Reset Priority: CE (Latch or Register Enable)

READ Address Change A

☐ Read Address Change A



Component Name: prgrom

Basic | Port A Options | Other Options | Summary

Pipeline Stages within Mux: 0 — Mux Size: 4x1

Memory Initialization

☐ Load Init File

Coe File: no_coe_file_loaded — Browse — Edit

☐ Fill Remaining Memory Locations

Remaining Memory Locations (Hex): 0

Structural/UniSim Simulation Model Options

Defines the type of warnings and outputs are generated when a read-write or write-write collision occurs.

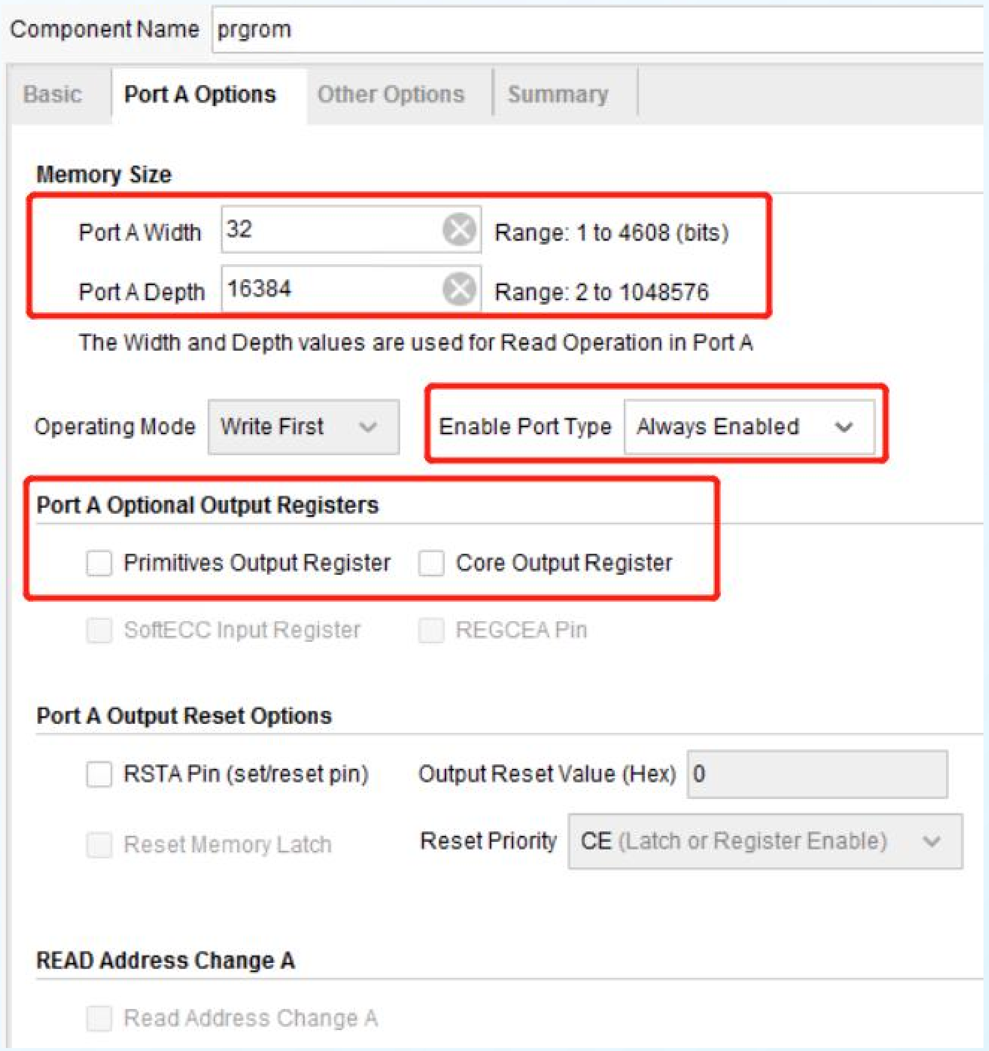
Collision Warnings: All

Behavioral Simulation Model Options

☐ Disable Collision Warnings ☐ Disable Out of Range Warnings

NOTE: set the init file of prgrom after this IP core has been added into vivado project. following the setting steps on page 10 of this slides.

Customize Memory IP core continued



Component Name prgrom

Basic | Port A Options | Other Options | Summary

Memory Size

Port A Width 32 Range: 1 to 4608 (bits)

Port A Depth 16384 Range: 2 to 1048576

The Width and Depth values are used for Read Operation in Port A

Operating Mode Write First | Enable Port Type Always Enabled

Port A Optional Output Registers

☐ Primitives Output Register ☐ Core Output Register

☐ SoftECC Input Register ☐ REGCEA Pin

Port A Output Reset Options

☐ RSTA Pin (set/reset pin) Output Reset Value (Hex) 0

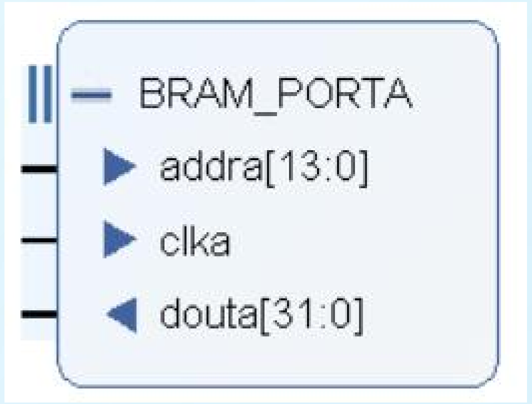
☐ Reset Memory Latch Reset Priority CE (Latch or Register Enable)

READ Address Change A

☐ Read Address Change A

3) **PortA Options** settings:

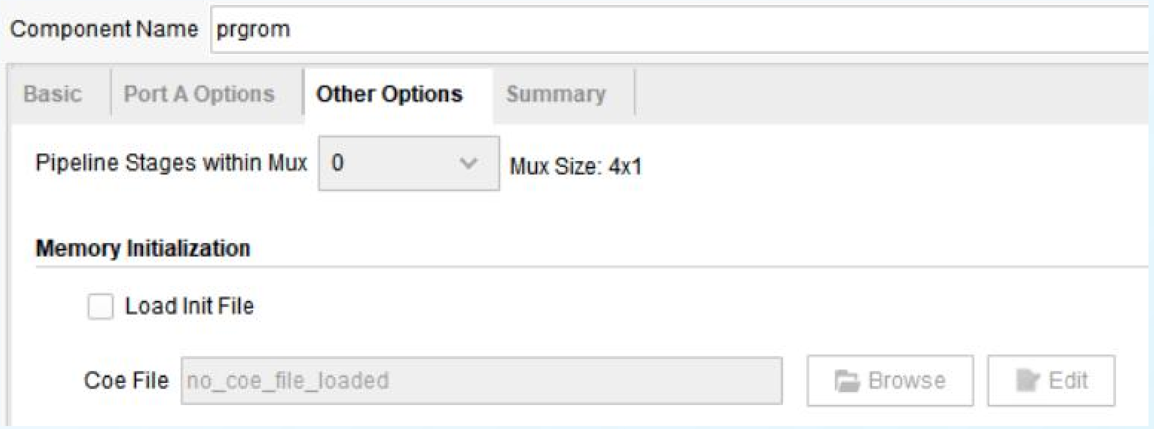
- Data read and write **bit width**: **32 bits (4Byte)**
- Read **Depth**: **16384, size: 2^{14} * 4Byte = 64KB**
- Operating Mode: **Write First**
- Enable Port Type: **Always Enabled**
- PortA Optional Output Registers: **NOT SET**



Customize Memory IP core continued

4) **Other Options** settings:

- **1.** When **specifying the initialization file** for customize the ROM on the 1st time, the IP core ROM just customized WITHOUT initial file and **corresponding path**, so set it to no initial file when creating ROM.
- **2. After** the ROM IP core created
  - **2-1. COPY** the initialization file prgrom32.coe **to** projectName.srcs/sources_1/ip/ComponentName. ("projectName.srcs" is under the project folder, "componentName" here is 'prgrom')
  - **2-2.** Double-click the newly created ROM IP core, **RESET** it with the **initialization file**, select the prgrom32.coe file that has been in the directory of projectName.srcs/sources_1/ip/prgrom.

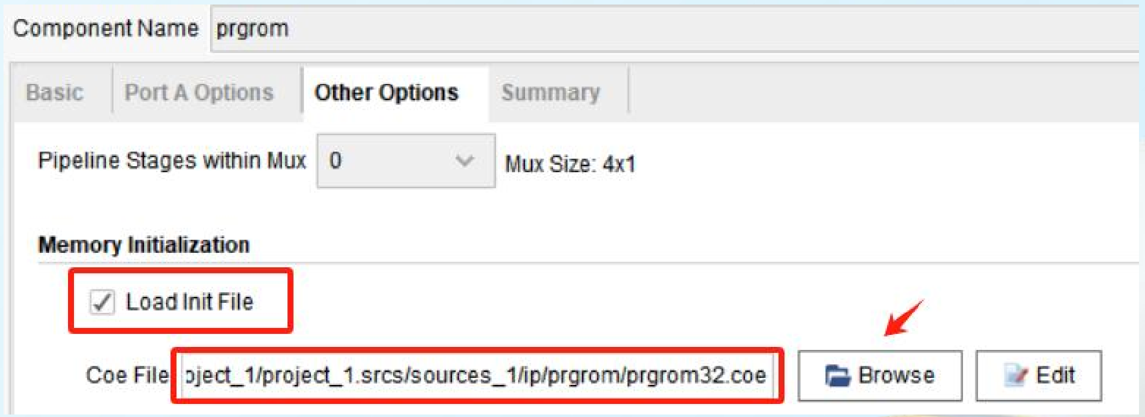

Component Name: prgrom

Pipeline Stages within Mux: 0 — Mux Size: 4x1

Memory Initialization

☐ Load Init File

Coe File: no_coe_file_loaded


Component Name: prgrom

Pipeline Stages within Mux: 0 — Mux Size: 4x1

Memory Initialization

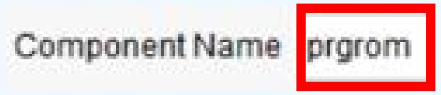
☑ Load Init File

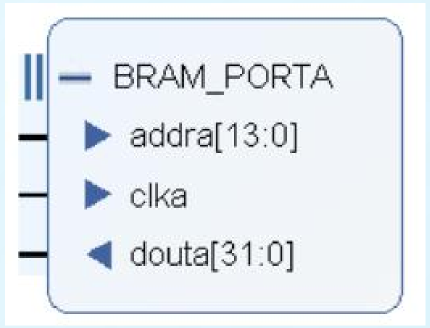
Coe File: oject_1/project_1.srcs/sources_1/ip/prgrom/prgrom32.coe

Tips: "**prgrom32.coe**" file could be found in the directory "**labs/lab9_CPU_1_IMem_Decoder_DMem**" of course blackboard site

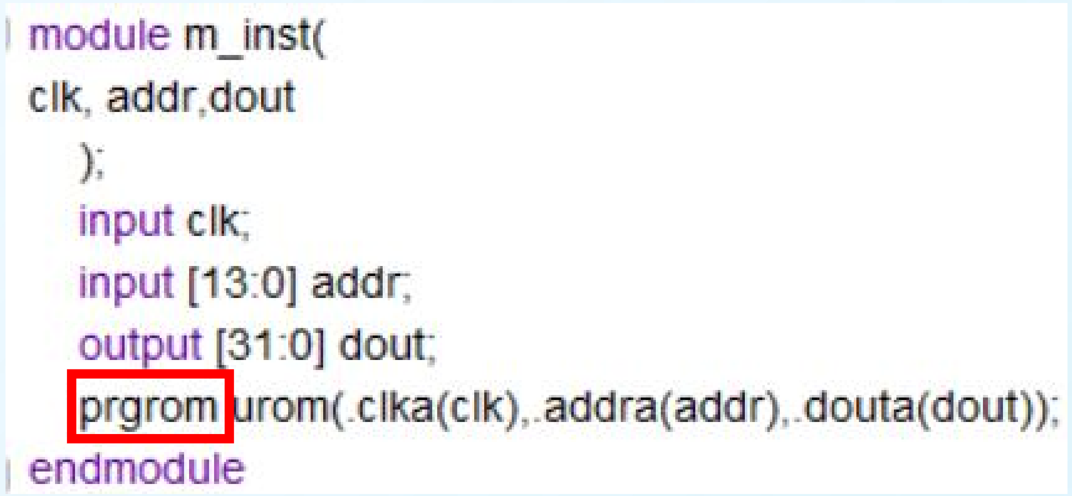
Instance the Memory IP core

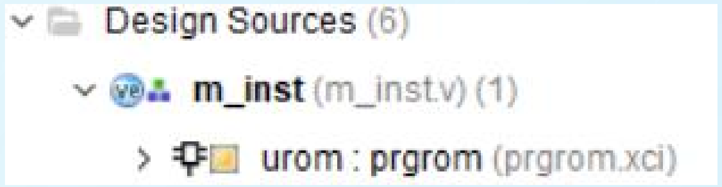
Step1. Find the name and the ports of the IP core:


Component Name prgrom



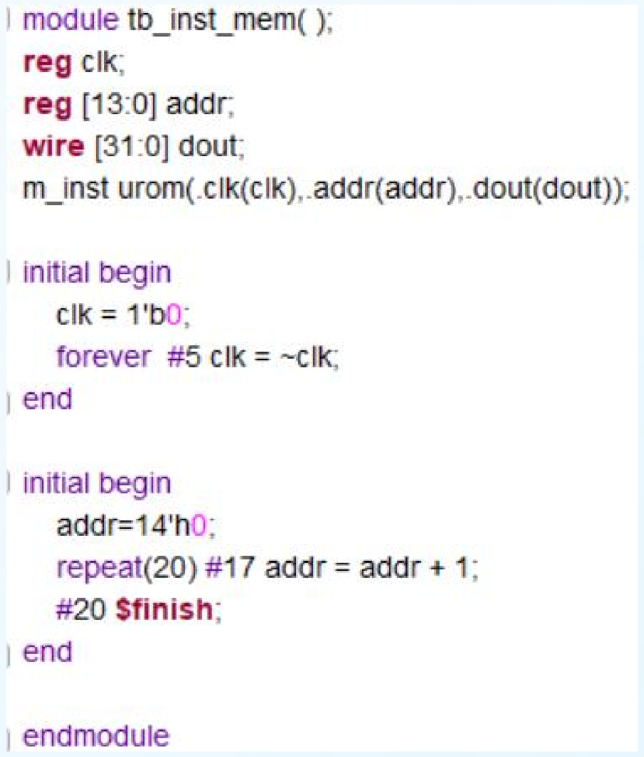
Step2. Build a module in verilog to instance the IP core and bind its ports:


```
module m_inst(
clk, addr,dout
    );
    input clk;
    input [13:0] addr;
    output [31:0] dout;
    prgrom urom(.clka(clk),.addra(addr),.douta(dout));
endmodule
```

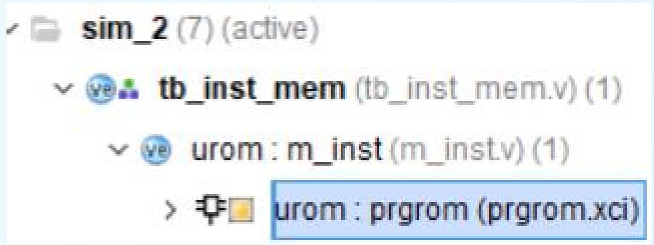

```
Design Sources (6)
    m_inst (m_inst.v) (1)
        urom : prgrom (prgrom.xci)
```

Test the IP core

Step1.Build the testbecn to verify the function of the IP core.



```
module tb_inst_mem( );
reg clk;
reg [13:0] addr;
wire [31:0] dout;
m_inst urom(.clk(clk),.addr(addr),.dout(dout));

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

initial begin
    addr=14'h0;
    repeat(20) #17 addr = addr + 1;
    #20 $finish;
end

endmodule
```



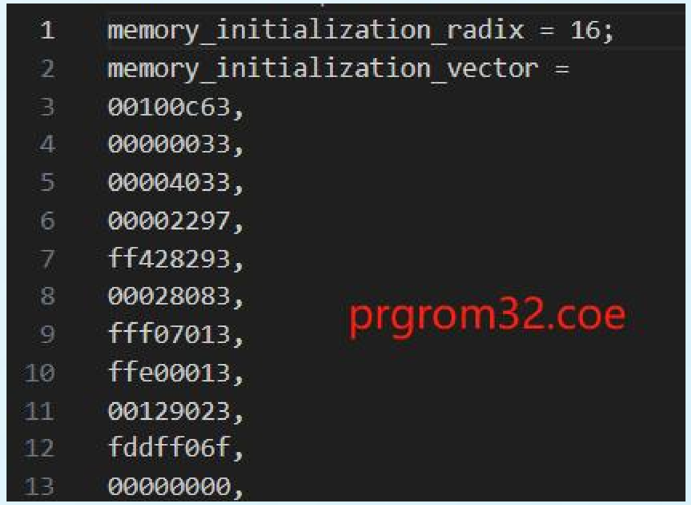
```
sim_2 (7) (active)
  tb_inst_mem (tb_inst_mem.v) (1)
    urom : m_inst (m_inst.v) (1)
      urom : prgrom (prgrom.xci)
```

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 prgrom IP core work as a ROM(Read Only Memory):

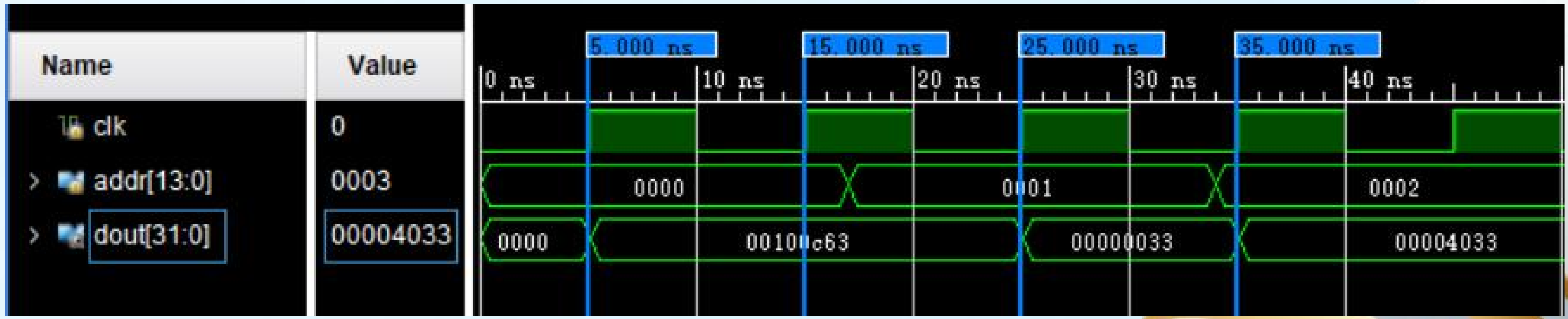
Determine whether the module can accurately read the data stored in the corresponding storage unit in the ROM based on the address on the rising edge of the clock.

The prgrom IP core is **initialized with file prgrom.coe**



```
memory_initialization_radix = 16;
memory_initialization_vector =
00100c63,
00000033,
00004033,
00002297,
ff428293,
00028083,
fff07013,
ffe00013,
00129023,
fddff06f,
00000000,
```

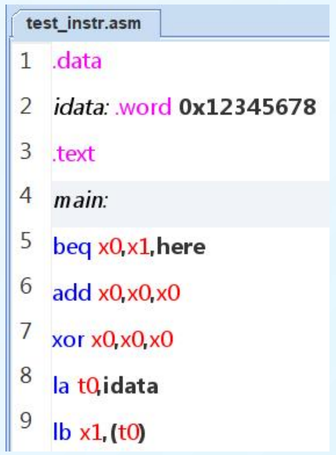
prgrom32.coe



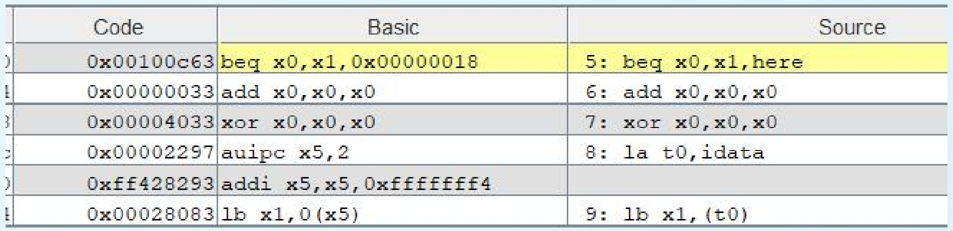
Name	Value
clk	0
addr[13:0]	0003
dout[31:0]	00004033

Tips. How to generate coe file

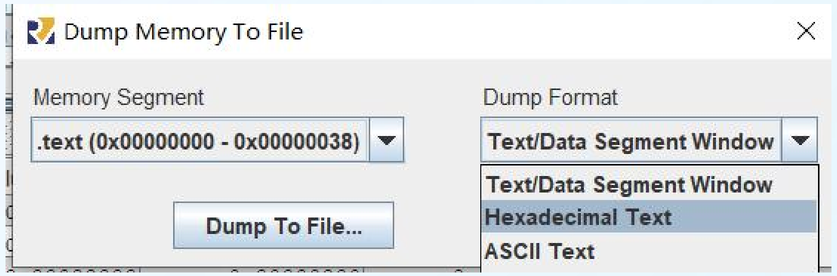
- ✓ 1-1. build a RISC-V assembly soure file.
- ✓ 1-2. Using Rars to asemble the source file, and generate the machine code.
- ✓ 1-3. Dump the machine code as Hexadecimal Text.
- ✓ 1-4. Using rars2coe.exe to generate the related coe file.


test_instr.asm

```
1 .data
2 idata: .word 0x12345678
3 .text
4 main:
5 beq x0,x1,here
6 add x0,x0,x0
7 xor x0,x0,x0
8 la t0,idata
9 lb x1,(t0)
```

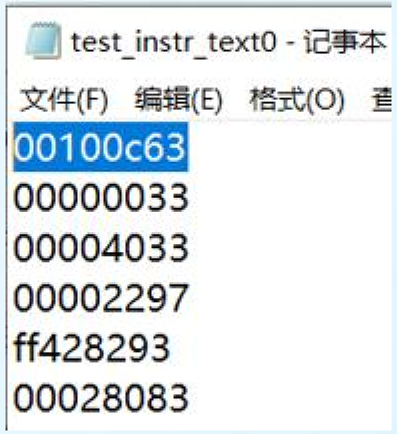


Code	Basic	Source
0x00100c63	beq x0,x1,0x00000018	5: beq x0,x1,here
0x00000033	add x0,x0,x0	6: add x0,x0,x0
0x00004033	xor x0,x0,x0	7: xor x0,x0,x0
0x00002297	auipc x5,2	8: la t0,idata
0xff428293	addi x5,x5,0xfffffff4	
0x00028083	lb x1,0(x5)	9: lb x1,(t0)

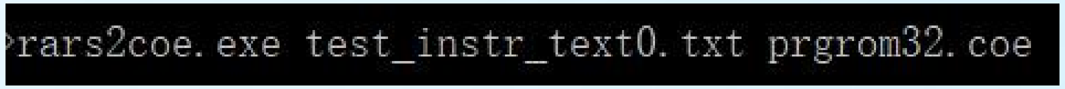

Dump Memory To File

Memory Segment: .text (0x00000000 - 0x00000038)

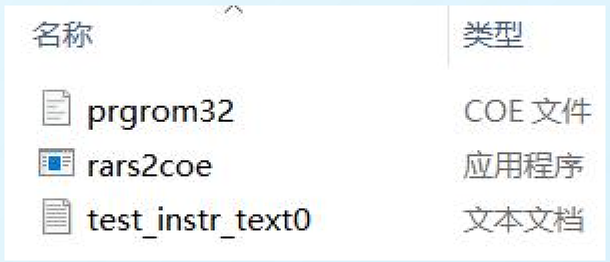
Dump Format: Text/Data Segment Window


test_instr_text0 - 记事本

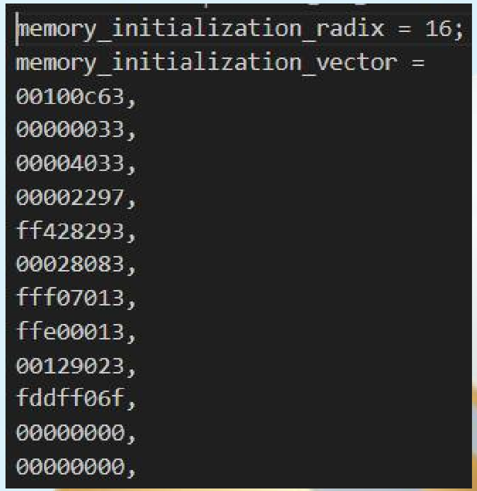
```
00100c63
00000033
00004033
00002297
ff428293
00028083
```



```
>rars2coe.exe test_instr_text0.txt prgrom32.coe
```



名称	类型
prgrom32	COE 文件
rars2coe	应用程序
test_instr_text0	文本文档



```
memory_initialization_radix = 16;
memory_initialization_vector =
00100c63,
00000033,
00004033,
00002297,
ff428293,
00028083,
fff07013,
ffe00013,
00129023,
fddff06f,
00000000,
00000000,
```

Practice 1

- Set the IP core of the Block memory type, observe and compare the circuit behavior of the IP cores.

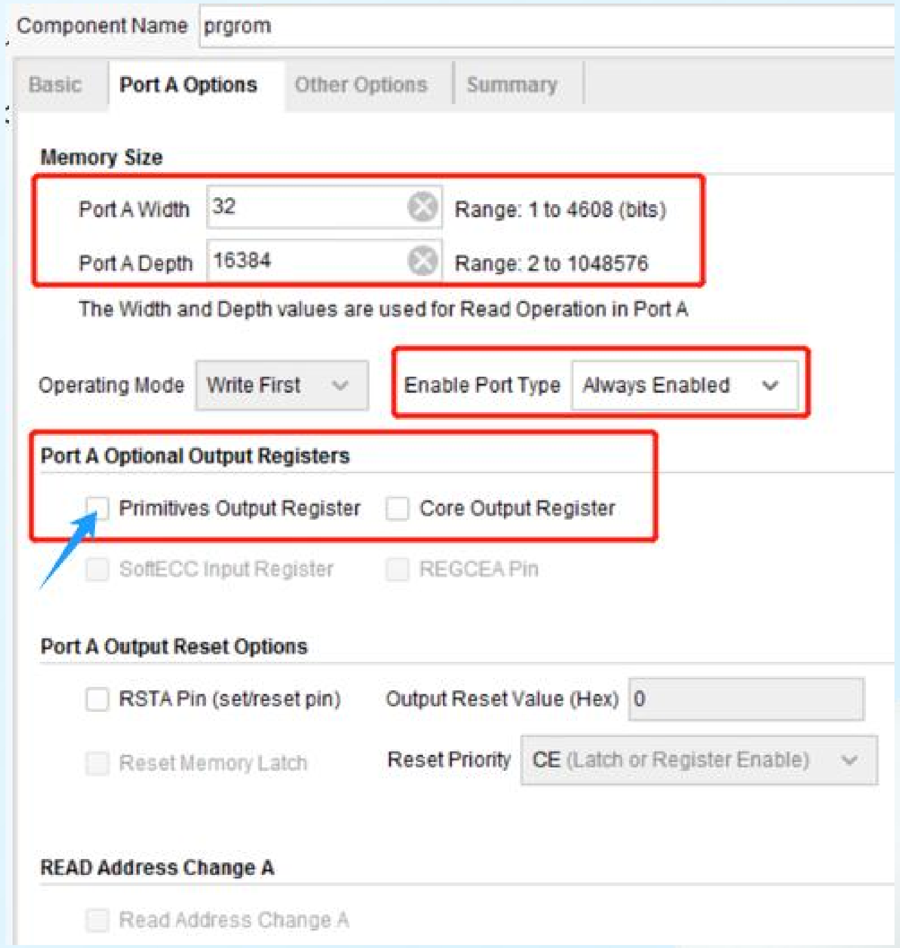
  - Step1.

  Refer to pages 6 to 12 of the courseware to create a block ROM IP core, except for modifying the option settings marked by the blue arrow in the right image, all other settings remain unchanged.

  - Step2.

  Instantiate and simulate the IP core under this configuration to observe the differences in their behavior under different settings

- Tip: For a ROM IP core, it' s suggested to pay attention to the data storage, data reading, and related timing.



Component Name: prgrom

Memory Size

Port A Width: 32 — Range: 1 to 4608 (bits)
Port A Depth: 16384 — Range: 2 to 1048576

The Width and Depth values are used for Read Operation in Port A

Operating Mode: Write First
Enable Port Type: Always Enabled

Port A Optional Output Registers

☐ Primitives Output Register ☐ Core Output Register
☐ SoftECC Input Register ☐ REGCEA Pin

Port A Output Reset Options

☐ RSTA Pin (set/reset pin) — Output Reset Value (Hex): 0
☐ Reset Memory Latch — Reset Priority: CE (Latch or Register Enable)

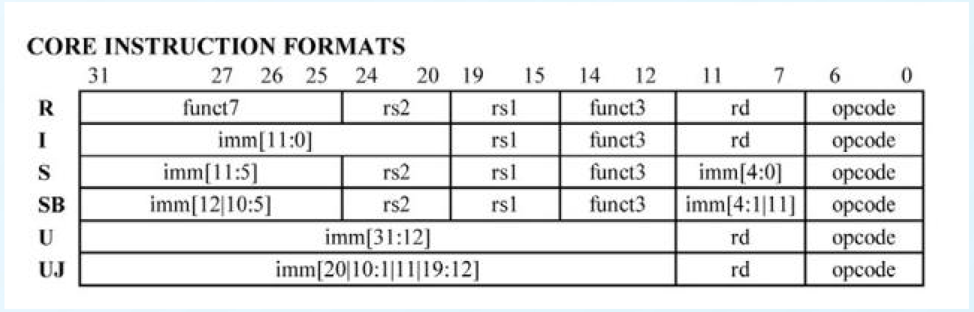
READ Address Change A

☐ Read Address Change A

Instruction Analysis

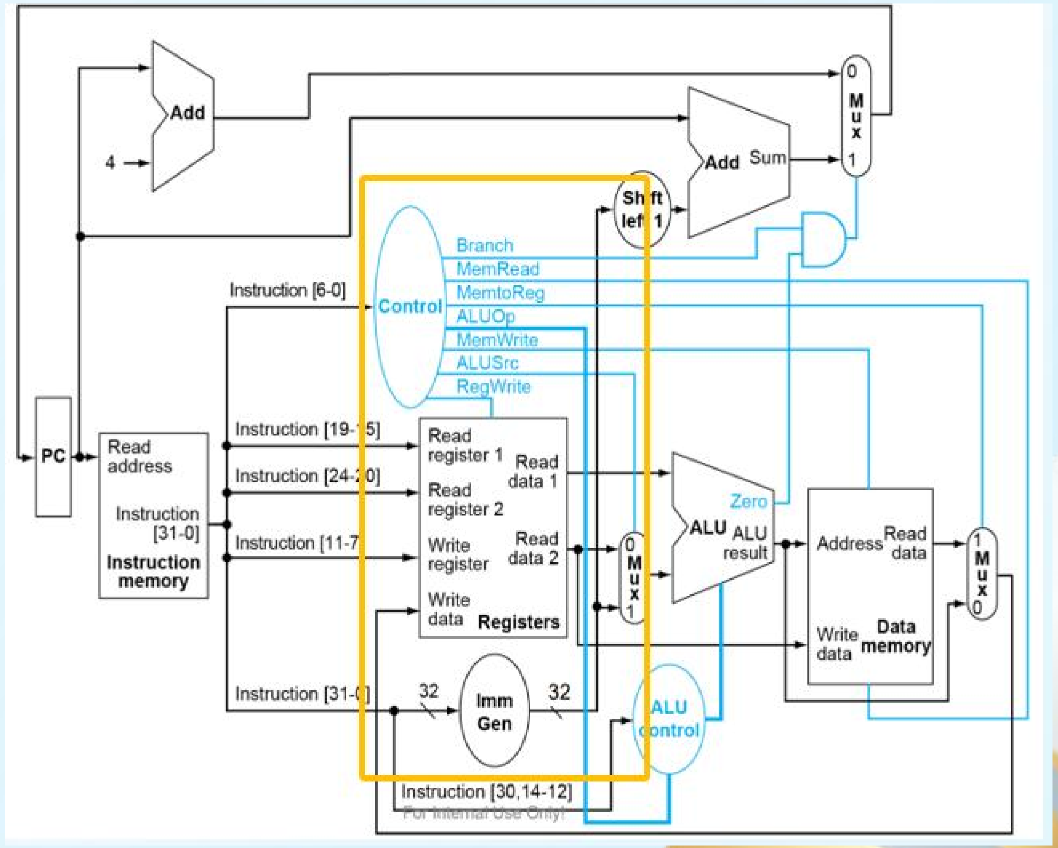
- **Task 1: get information about the data from the instruction (Decoder in Data Path)**
  - address of **registers**: **rs2**(Instruction[24:20]), **rs1**(Instruction[19:15]) and **rd**(Instruction[11:7])
  - **shift mount**(instruction[24:20]) (R type and I type)
  - **immediate(12bits for I/S/SB, 20bits for U/UJ)**

- **Task 2: generated control signals according to the instruction (Controller in Control Path)**
  - **get Operation code**(inst[**6:0**]) **and function code**(funct7,funct3) **in the instruction**
  - **generate control signals** to submodules of CPU

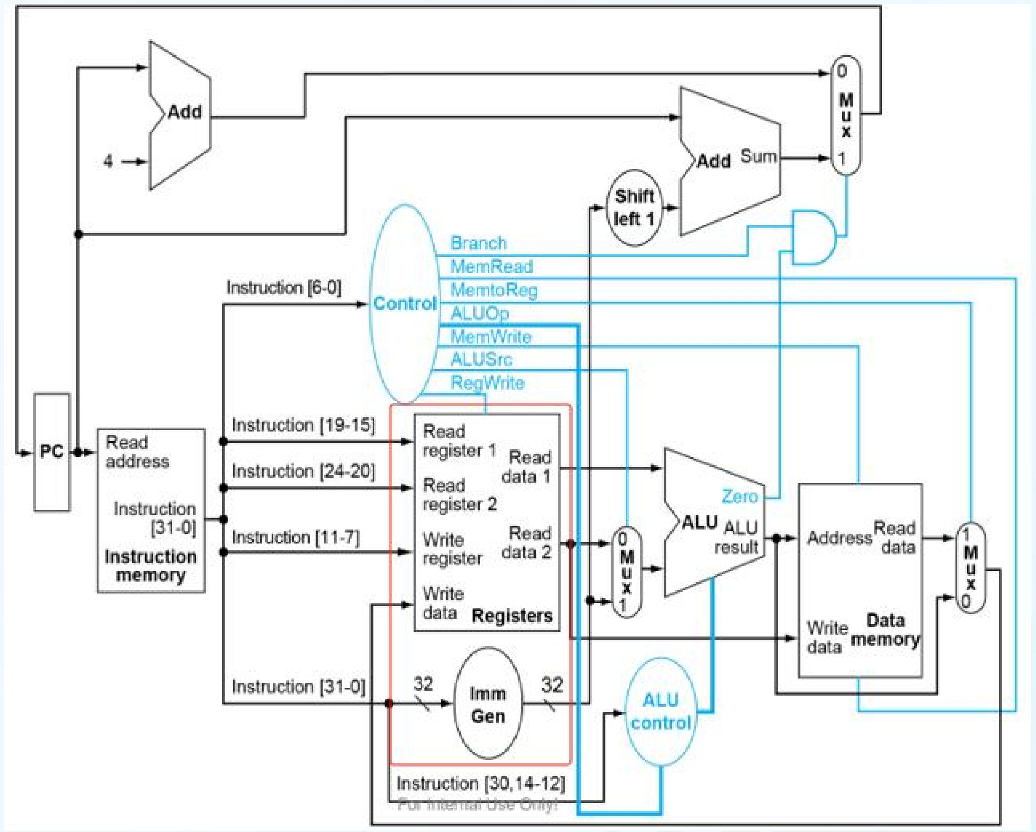


CORE INSTRUCTION FORMATS

	31 27 26 25	24 20	19 15	14 12	11 7	6 0
R	funct7	rs2	rs1	funct3	rd	opcode
I	imm[11:0]		rs1	funct3	rd	opcode
S	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]				rd	opcode
UJ	imm[20|10:1|11|19:12]				rd	opcode

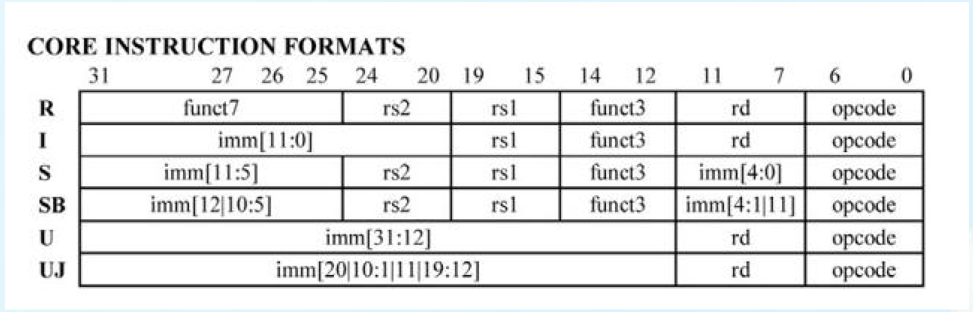


Data Path(1) Decoder



- **Get data** from the instruction directly or indirectly
  - opcode, function code : how to get data, where to get data
  - **immediate data** needs to be **signextended to 32bits for calculation.**
  - **data in the register**, the address of the register is coded in the instruction. e.g. rs2 = Instruction[24:20];
  - **data in the memory**, the **address** of the memory unit need to be calculated by ALU with **base address** stored in the **register** and **offset** as **immediate data in the instruction**

- **Read/Write data from/to Register File**

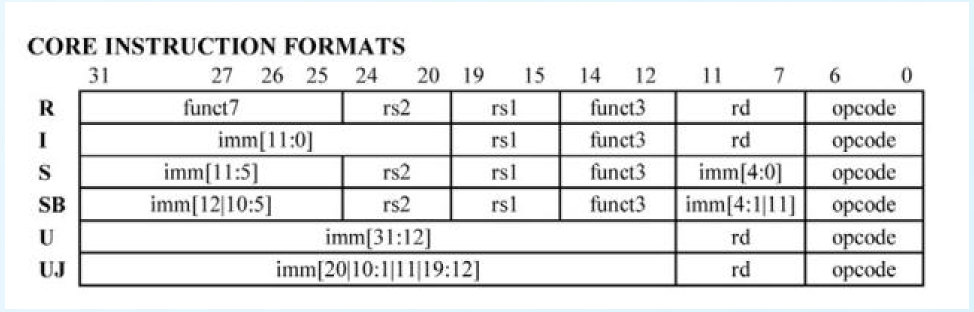


CORE INSTRUCTION FORMATS

	31 27 26 25	24 20	19 15	14 12	11 7	6 0
R	funct7	rs2	rs1	funct3	rd	opcode
I	imm[11:0]		rs1	funct3	rd	opcode
S	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]				rd	opcode
UJ	imm[20|10:1|11|19:12]				rd	opcode

Tips:
The submodule 'Imm Gen' has already been done in lab8. It is suggested to instance 'Imm Gen' in the Decoder to speed up the coding.

Data Path(1) Decoder continued



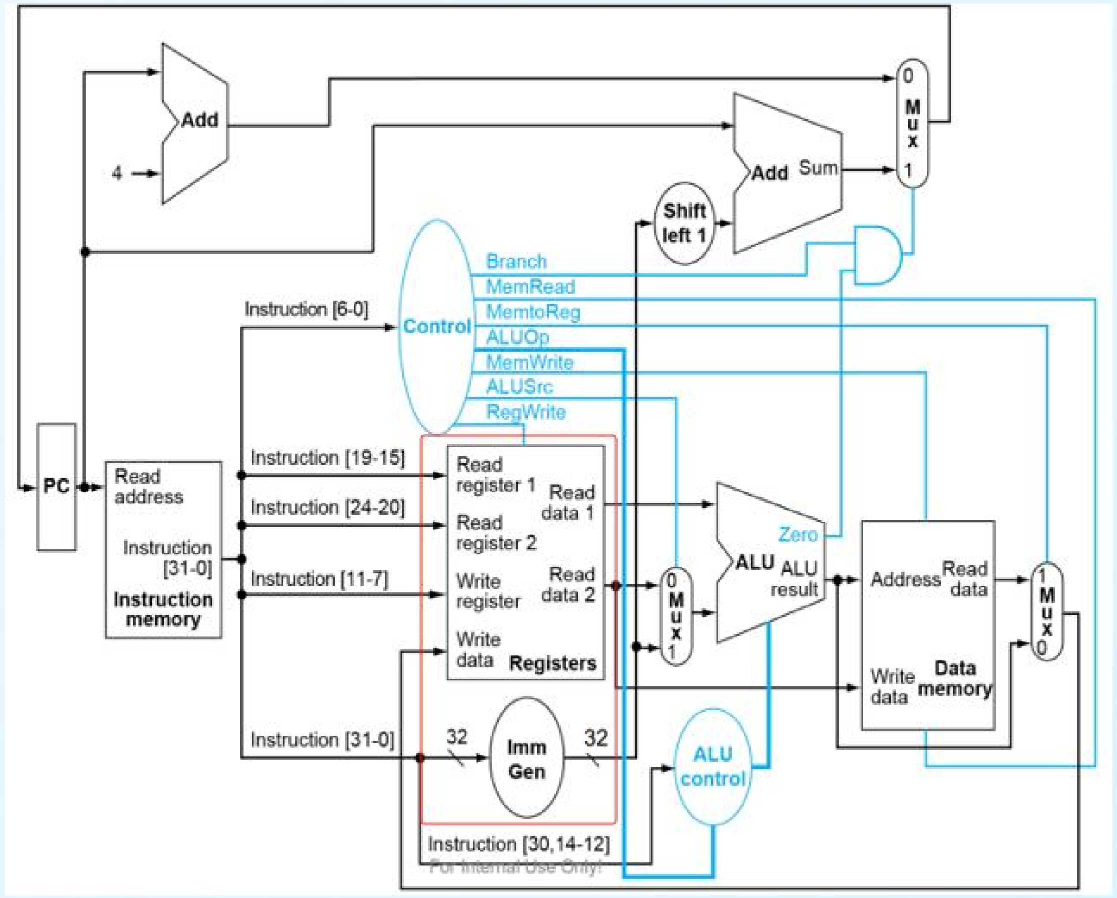
CORE INSTRUCTION FORMATS

	31 27 26 25	24 20	19 15	14 12	11 7	6 0
R	funct7	rs2	rs1	funct3	rd	opcode
I	imm[11:0]		rs1	funct3	rd	opcode
S	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]				rd	opcode
UJ	imm[20|10:1|11|19:12]				rd	opcode

- **Registers(Register File)**

  -Inputs

  - **read address**
    - [R] add: rs1,rs2
    - [B] beq : rs1, rs2
    - [I_calculate] addi: rs1
    - …
  - **write address**
    - [R] add: rd
    - [J] jal : rd/[31]
    - [I_calculate] addi: rd
    - …
  - **write data**
    - [R]/[I_calculate] add: data from alu_result
    - [I_load] lw: data from memory
    - …
  - **control signal**
    - [R]/[I]/[J] RegWrite
    - [J] jal
    - …

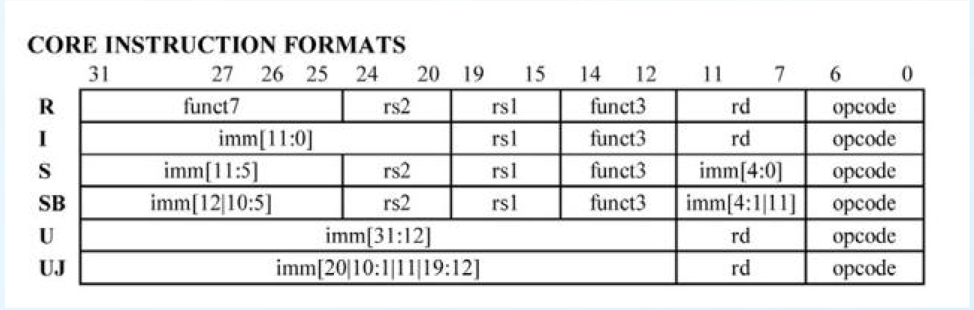


Registers(Register File)

-Outputs

- read data1
- read data2
- extended Immi

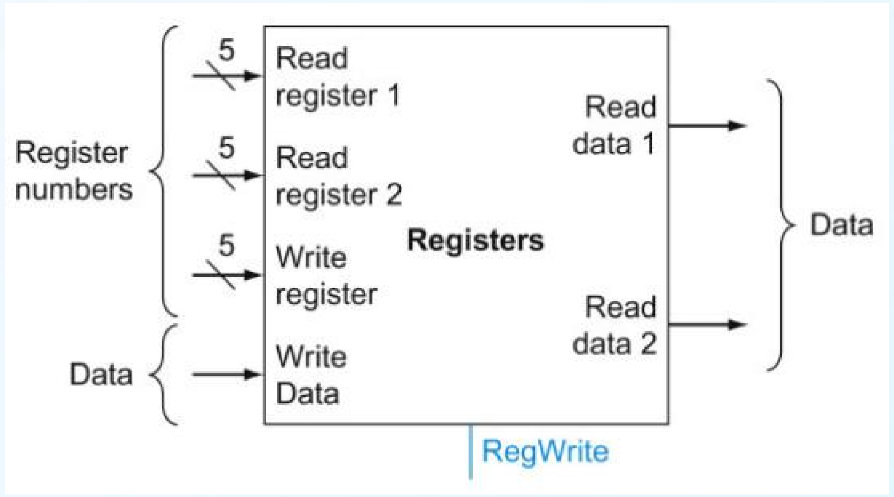
Data Path(1) Decoder continued



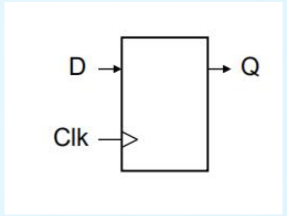
CORE INSTRUCTION FORMATS

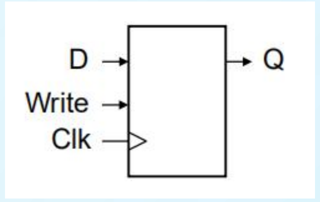
	31 27 26 25	24 20	19 15	14 12	11 7	6 0
R	funct7	rs2	rs1	funct3	rd	opcode
I	imm[11:0]		rs1	funct3	rd	opcode
S	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]				rd	opcode
UJ	imm[20|10:1|11|19:12]				rd	opcode

Register File(Registers):
Almost all the instructions need to read or write register file in CPU;
32 common registers with same bitwidth: 32



```
//verilog tips:
reg[31:0] register[0:31];
assign Rdata1 = register[Read register 1];
register[Write register] <= WriteData;
```



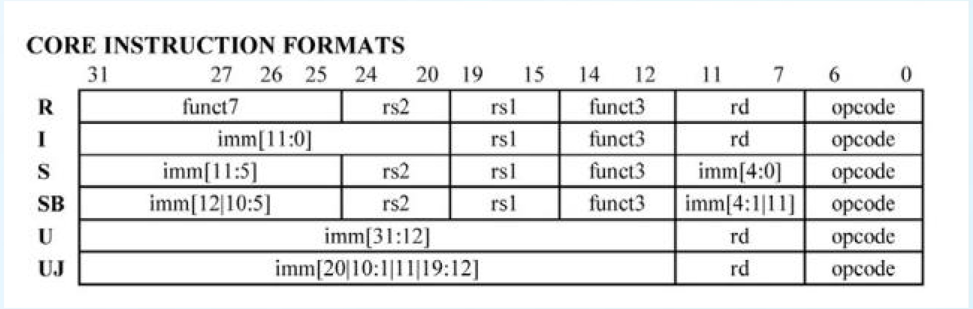


Q1:
How to avoid the confliction between register read and register write?

Q2: Which kind of circuit is this register file, combinatorial circuit or sequential circuit?

Q3: How to determine the size of address bus on register file?

Practice2



CORE INSTRUCTION FORMATS

	31 … 27 26 25	24 … 20	19 … 15	14 … 12	11 … 7	6 … 0
R	funct7	rs2	rs1	funct3	rd	opcode
I	imm[11:0]		rs1	funct3	rd	opcode
S	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]				rd	opcode
UJ	imm[20|10:1|11|19:12]				rd	opcode

- ➢ 1. Implement the sub-module of CPU: Decoder
  - ➢ There are **32** registers(each register is **32bits),** All the registers are **readable** and **writeable** except **x0**, **x0 is readonly**.
  - ➢ The **reading** should be done at any time while **writing** only happens on the **posedge** of the clock.
  - ➢ The register file should support **R/I/S/SB/U/UJ** type instructions(extend the immediate to be 32bits if needed).
    - ➢ such as: **add; addi; jalr; lw; sw; beq;jal;**
- ➢ 2. Verify its function by simulation. NOTES: The verification should be done on the full set of testcase.
- ➢ 3. List the signals which are used by the Decoder (NOTE: Signals' name are determined by designer)

tips: following table could be used as a reference

name	from	to	bits	function
regWrite	Controller	Decoder	1	1 means write the register identified by writeAdress
imm32	Decoder	IFetch	32	the signextended immediate
instruction	IFetch	Decoder	32	the instruction
rdata1	Decoder	ALU	32	the data read from the register which is specified by rs1
…				

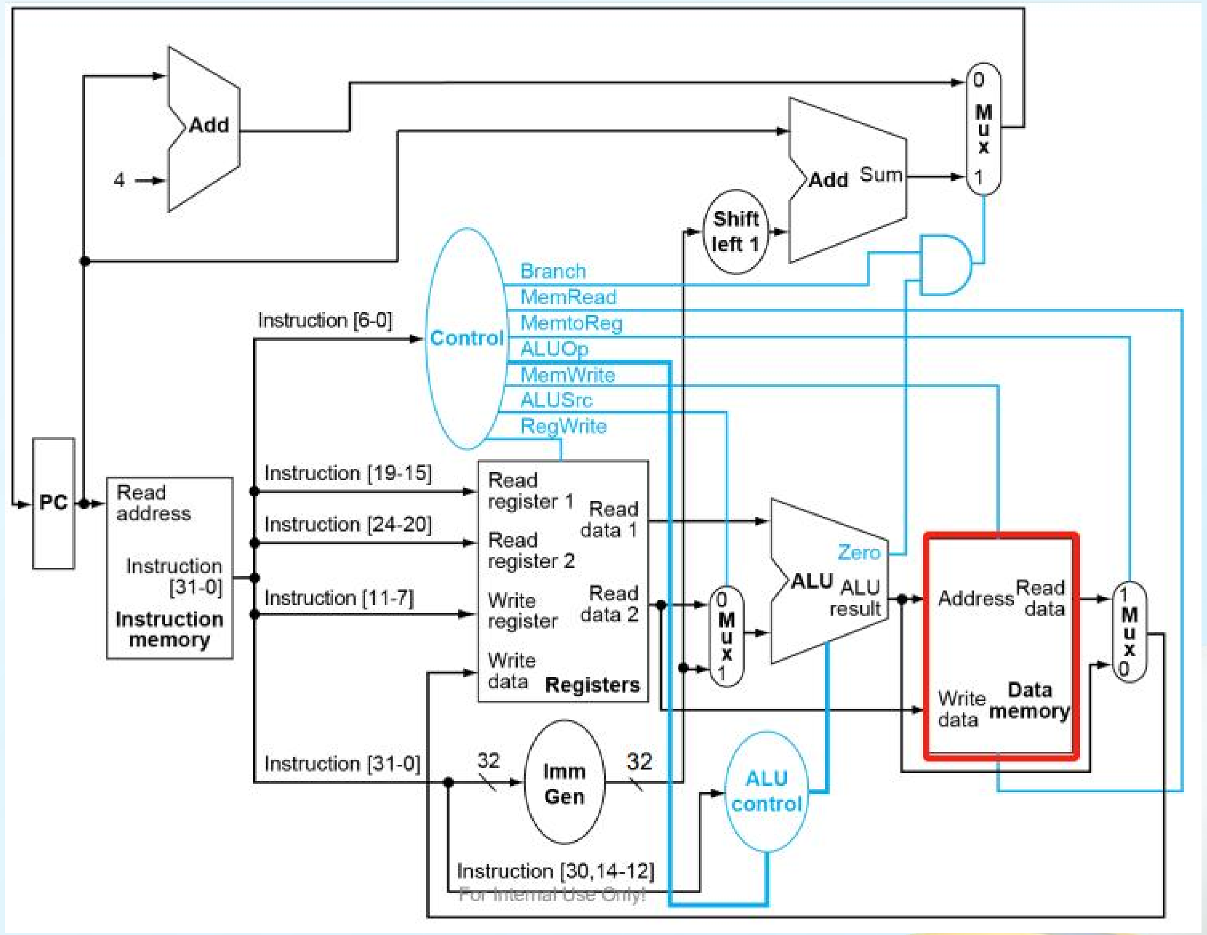
Data Path(2) - Data Memory

- **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**

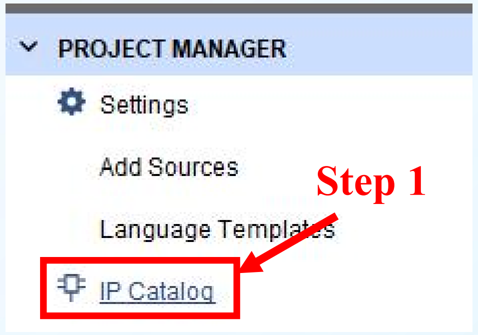
- **outputs**
  - ✓ **ReadData**

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



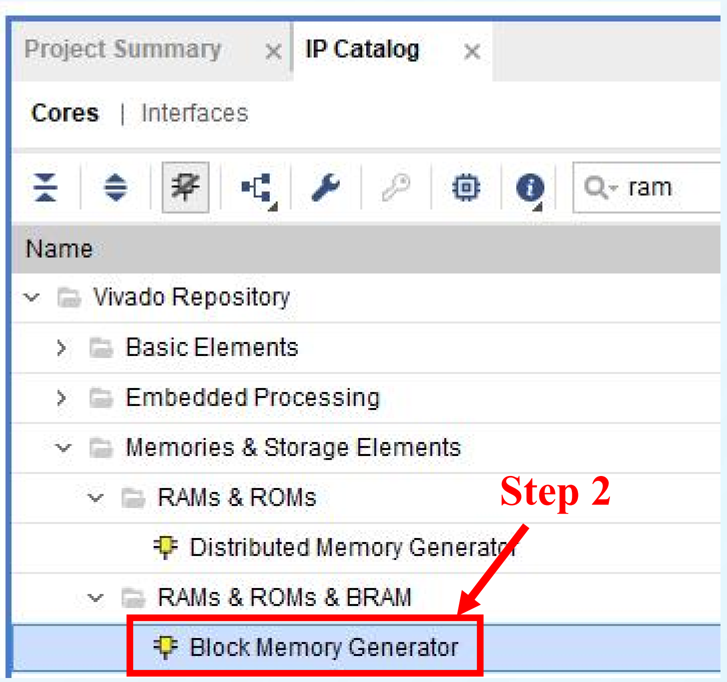
Using IP core in Vivado: Block Memory

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

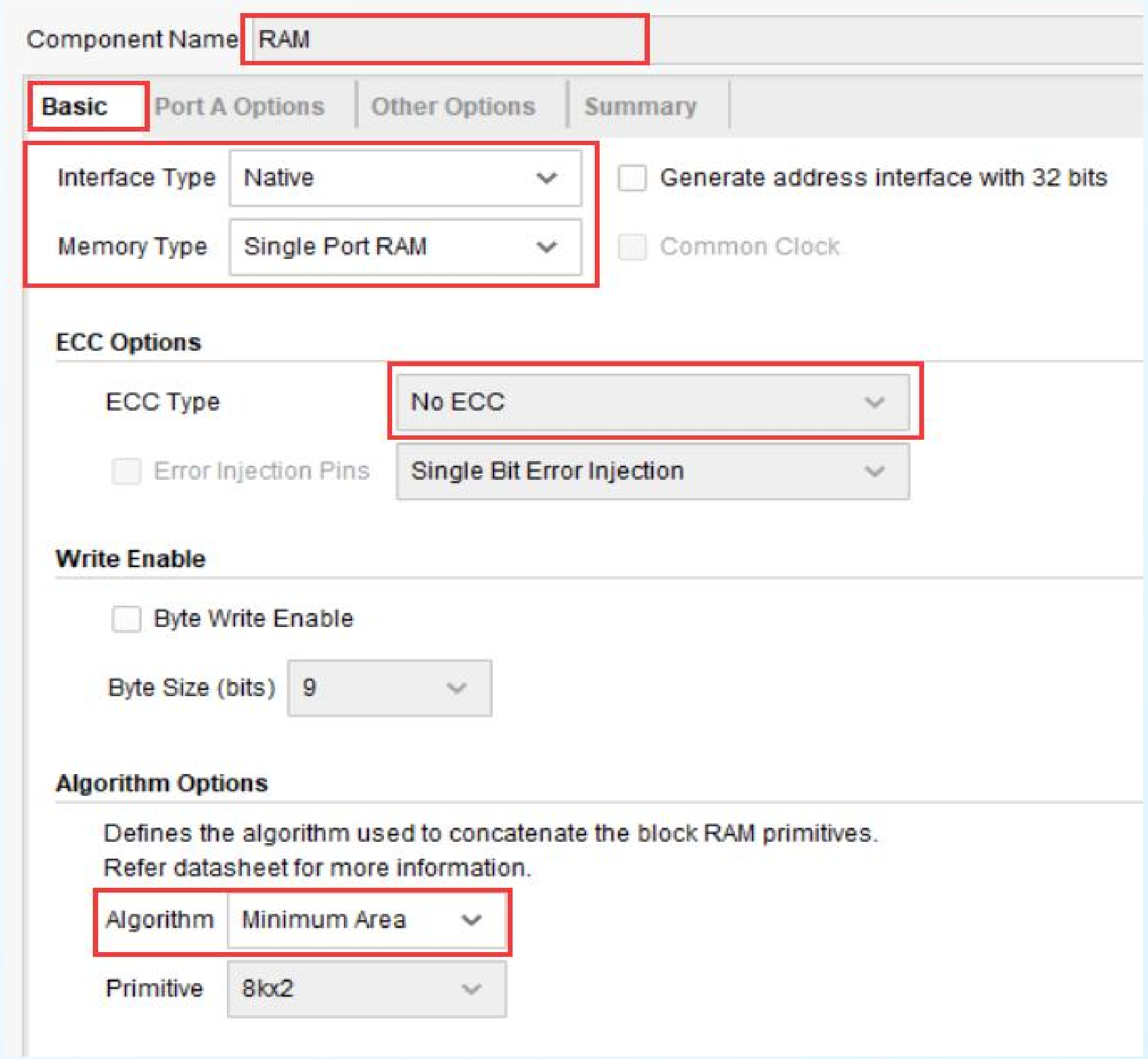
> Vivado Repository

> Memories & Storage Elements

> RAMs & ROMs & BRAM

> **Block Memory Generator**

Customize Memory IP core



Component Name: RAM

Basic | Port A Options | Other Options | Summary

Interface Type: Native
Memory Type: Single Port RAM
Generate address interface with 32 bits [ ]
Common Clock [ ]

ECC Options

ECC Type: No ECC
Error Injection Pins [ ]: Single Bit Error Injection

Write Enable

Byte Write Enable [ ]
Byte Size (bits): 9

Algorithm Options

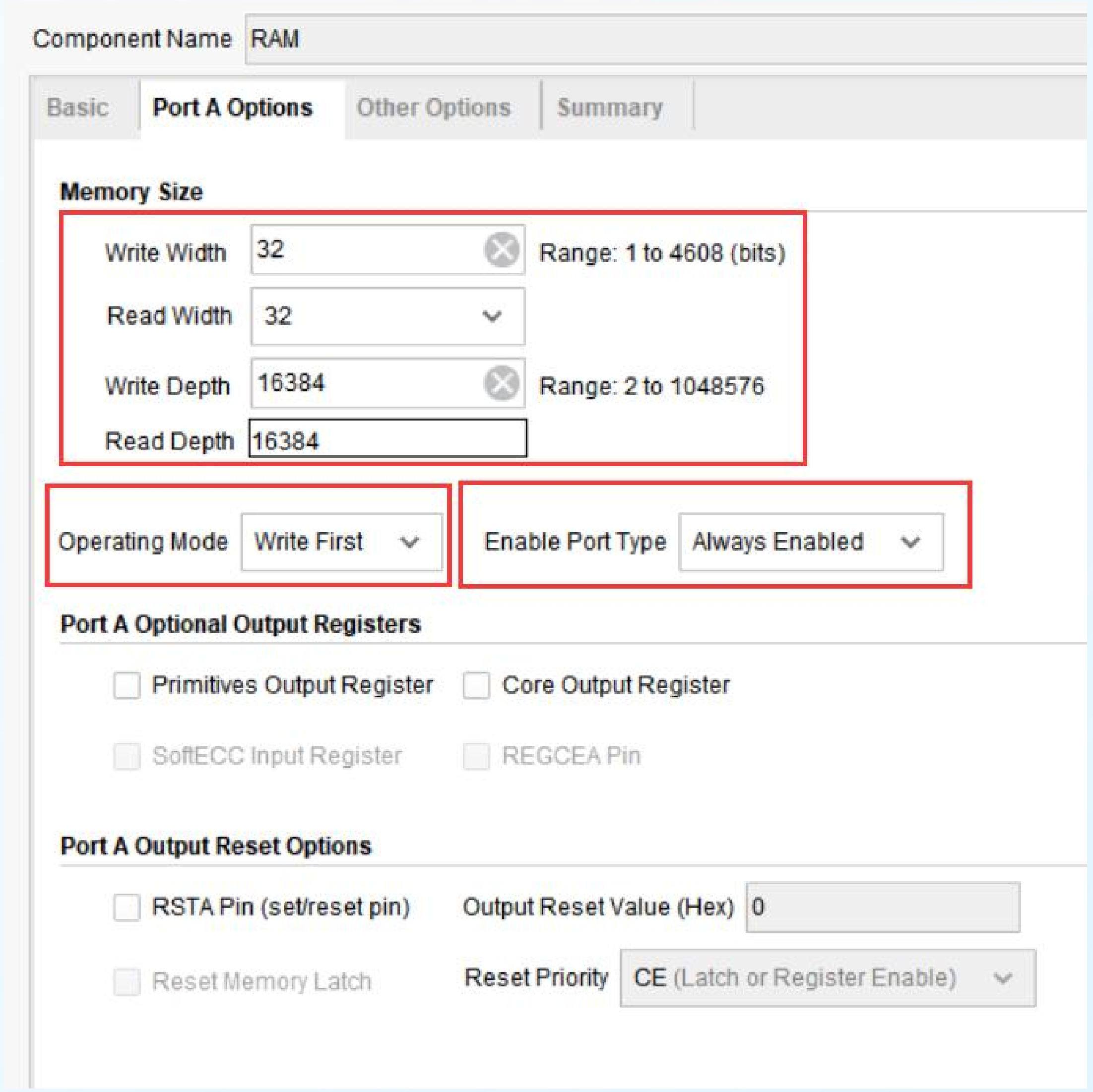
Defines the algorithm used to concatenate the block RAM primitives.
Refer datasheet for more information.

Algorithm: Minimum Area
Primitive: 8kx2

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

- [ ] Primitives Output Register
- [ ] Core Output Register
- [ ] SoftECC Input Register
- [ ] REGCEA Pin

Port A Output Reset Options

- [ ] RSTA Pin (set/reset pin)

Output Reset Value (Hex): 0

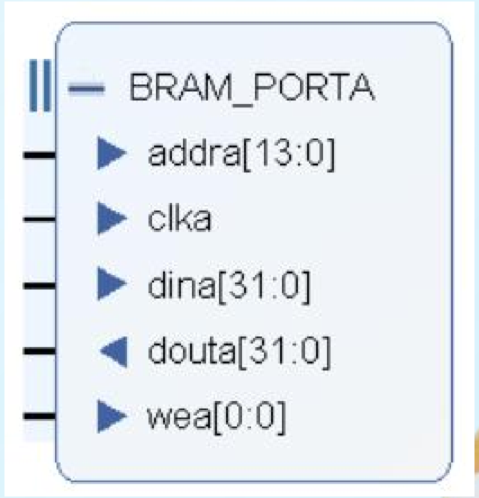
- [ ] Reset Memory Latch

Reset Priority: CE (Latch or Register Enable)

3) **PortA Options** settings:

- Data read and write **bit width**: **32 bits (4Byte)**
- Write/Read **Depth**: **16384**

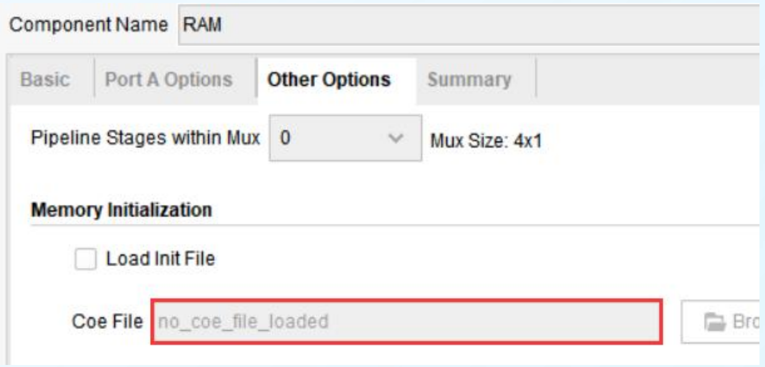
  size: 2^{14} * 4Byte = 64KB
- Operating Mode: **Write First**
- Enable Port Type: **Always Enabled**
- PortA Optional Output Registers: **NOT SET**



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.srcs/sources_1/ip/ComponentName. ("projectName.srcs" 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.srcs/sources_1/ip/RAM.

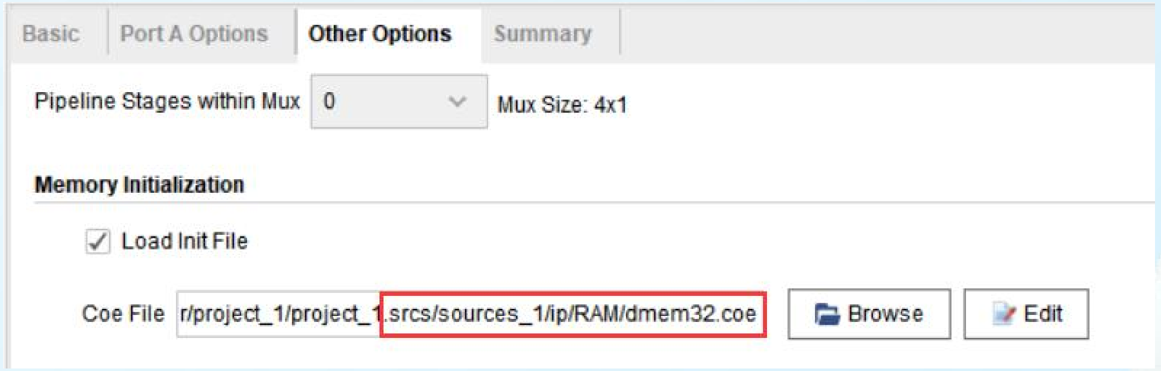

Component Name: RAM

Pipeline Stages within Mux: 0 — Mux Size: 4x1

Memory Initialization

☐ Load Init File

Coe File: no_coe_file_loaded


Pipeline Stages within Mux: 0 — Mux Size: 4x1

Memory Initialization

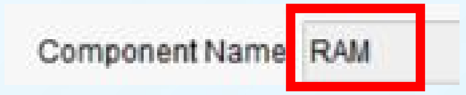
✓ Load Init File

Coe File: r/project_1/project_1.srcs/sources_1/ip/RAM/dmem32.coe

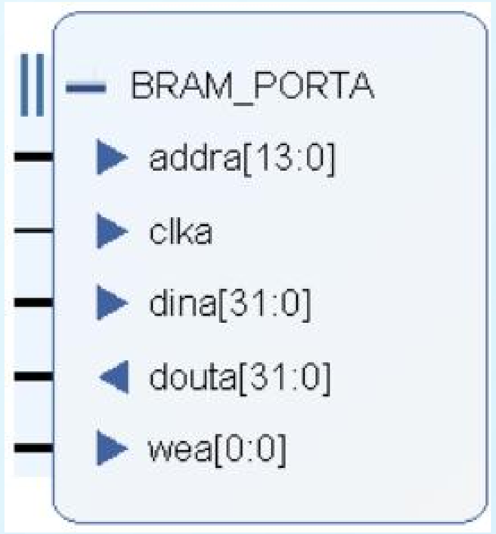
Tips: "**dmem32.coe**" file could be found in the directory "**labs/lab9_CPU_1_IMem_Decoder_DMem**" of course blackboard site

Instance the Memory IP core

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



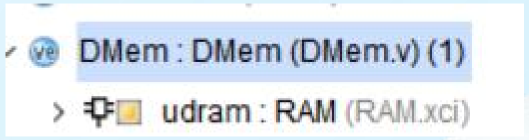
Component Name RAM



BRAM_PORTA
addra[13:0]
clka
dina[31:0]
douta[31:0]
wea[0:0]

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

endmodule
```



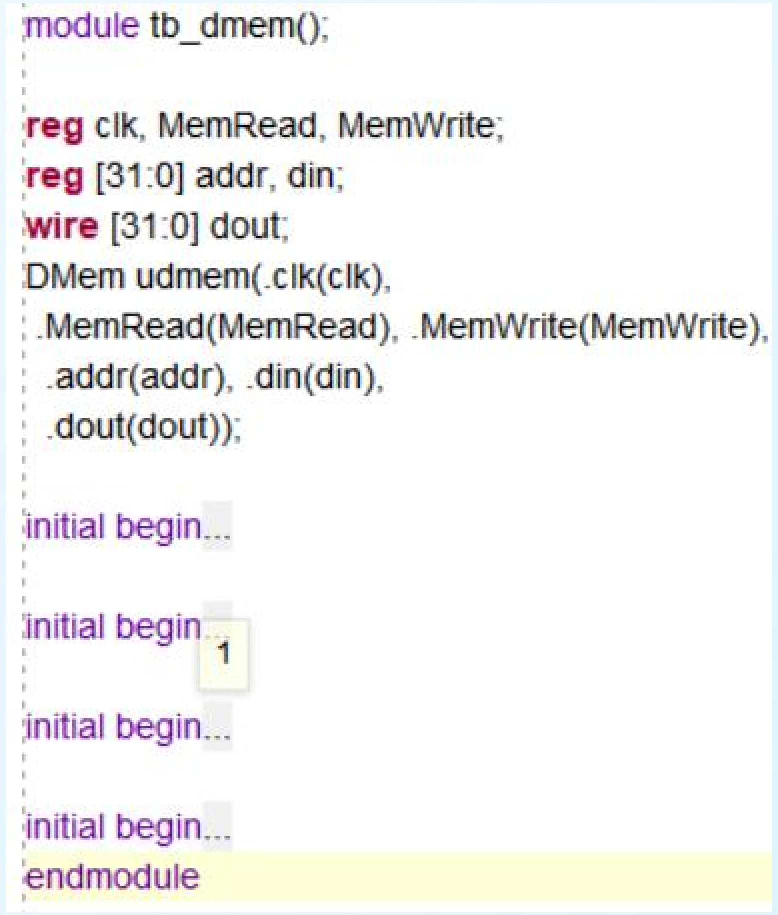
DMem : DMem (DMem.v) (1)
udram : RAM (RAM.xci)

Test the IP core(1)

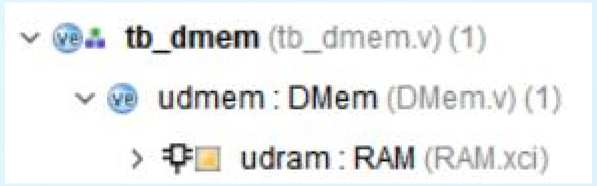
Step1.Build the testbecn to verify the function of the IP core.

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.

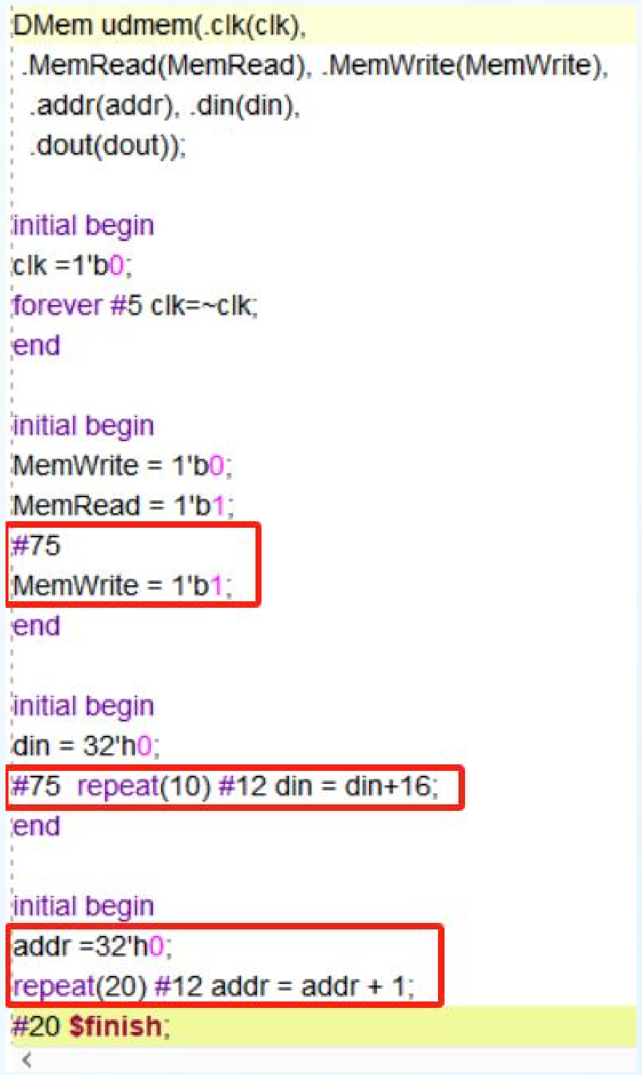


```
module tb_dmem();

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
```



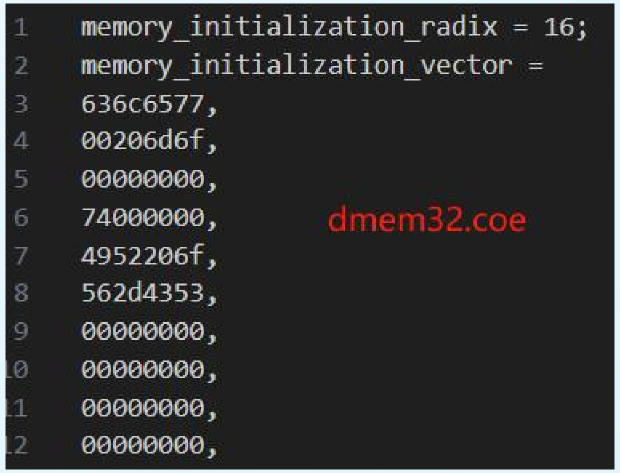
```
˅ tb_dmem (tb_dmem.v) (1)
  ˅ udmem : DMem (DMem.v) (1)
    › udram : RAM (RAM.xci)
```

Test the IP core(2)



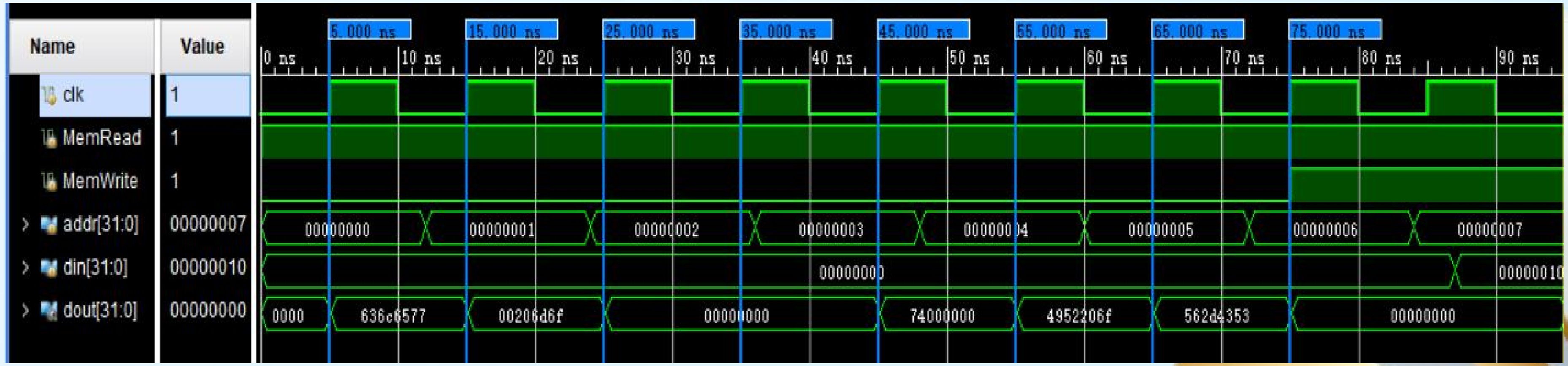
```
DMem udmem(.clk(clk),
 .MemRead(MemRead), .MemWrite(MemWrite),
  .addr(addr), .din(din),
  .dout(dout));

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)



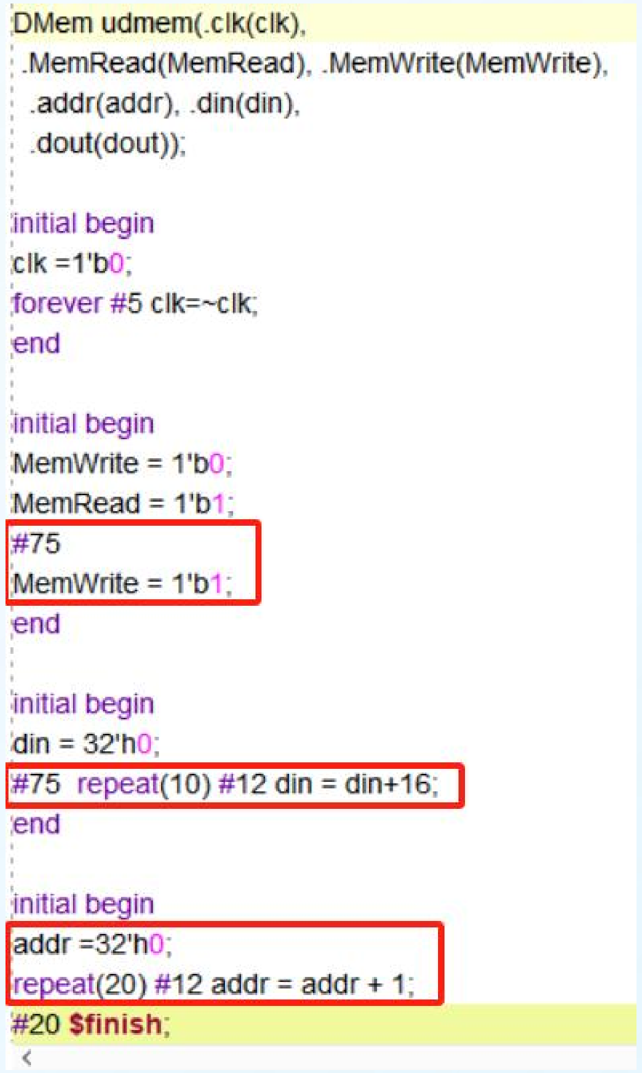
```
memory_initialization_radix = 16;
memory_initialization_vector =
636c6577,
00206d6f,
00000000,
74000000,
4952206f,
562d4353,
00000000,
00000000,
00000000,
00000000,
```

dmem32.coe



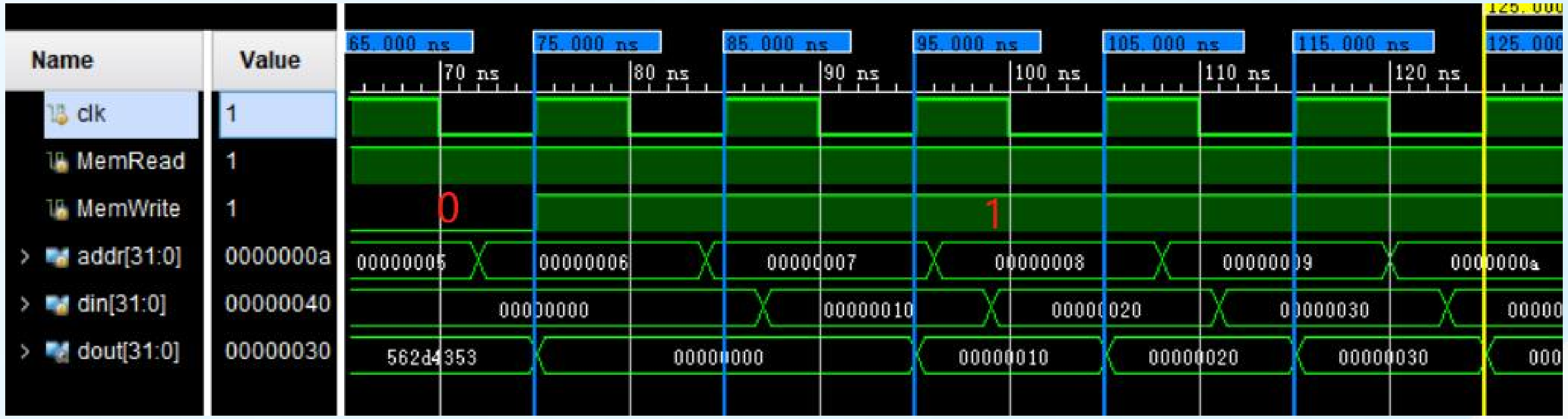
```
Name        Value
clk         1
MemRead     1
MemWrite    1
addr[31:0]  00000007
din[31:0]   00000010
dout[31:0]  00000000
```

Test the IP core(3)



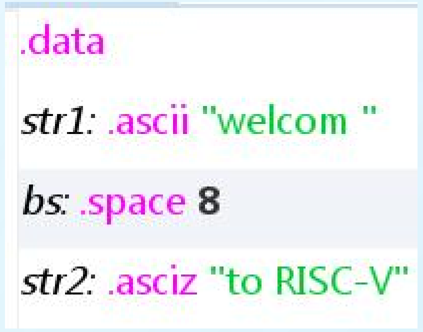
```
DMem udmem(.clk(clk),
 .MemRead(MemRead), .MemWrite(MemWrite),
  .addr(addr), .din(din),
  .dout(dout));

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.

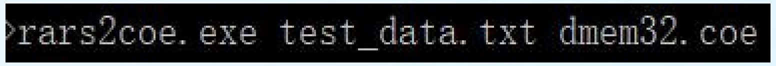


Tips. How to generate coe file

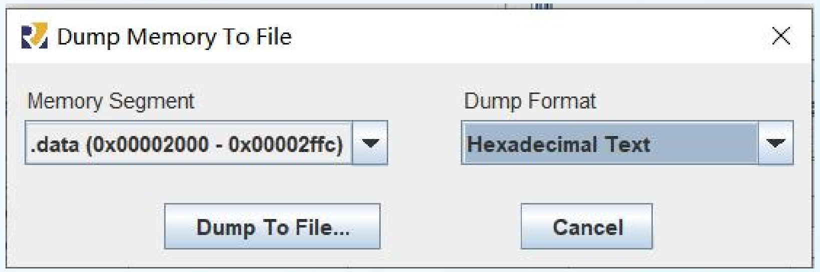
- ✓ 1-1. build a RISC-V assembly soure file.
- ✓ 1-2. Using Rars to asemble the source file.
- ✓ 1-3. Dump the data as Hexadecimal Text.
- ✓ 1-4. Using rars2coe.exe to generate the related coe file.



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



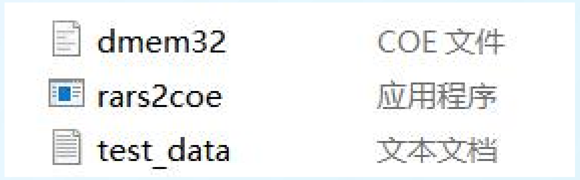
```
>rars2coe.exe test_data.txt dmem32.coe
```



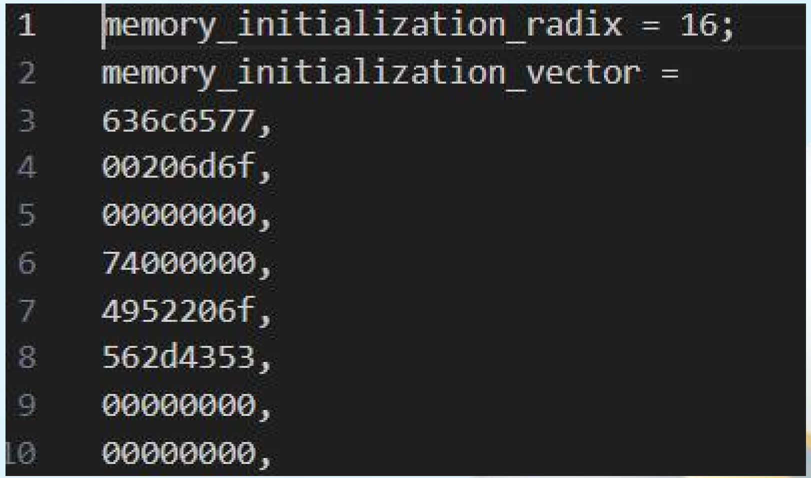
Dump Memory To File

Memory Segment: .data (0x00002000 - 0x00002ffc)

Dump Format: Hexadecimal Text



dmem32	COE 文件
rars2coe	应用程序
test_data	文本文档



```
memory_initialization_radix = 16;
memory_initialization_vector =
636c6577,
00206d6f,
00000000,
74000000,
4952206f,
562d4353,
00000000,
00000000,
```

Practice 3

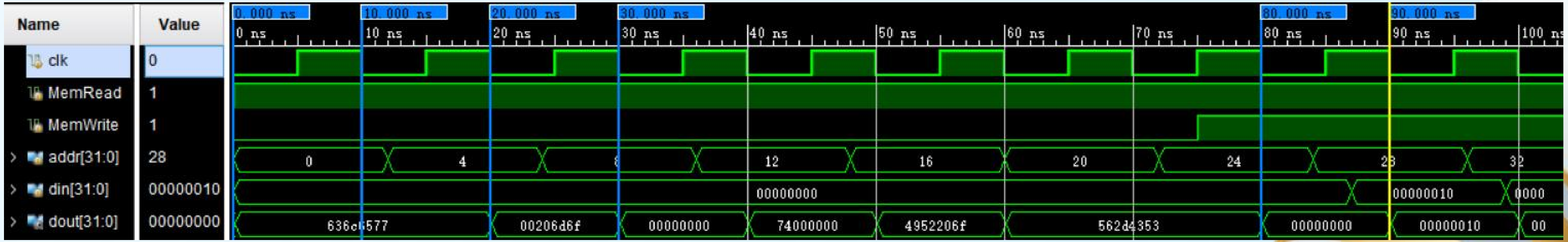
```
module DMem(
input clk,
input MemRead,MemWrite,
input [31:0] addr,
input [31:0] din,
output[31:0] dout);

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
```

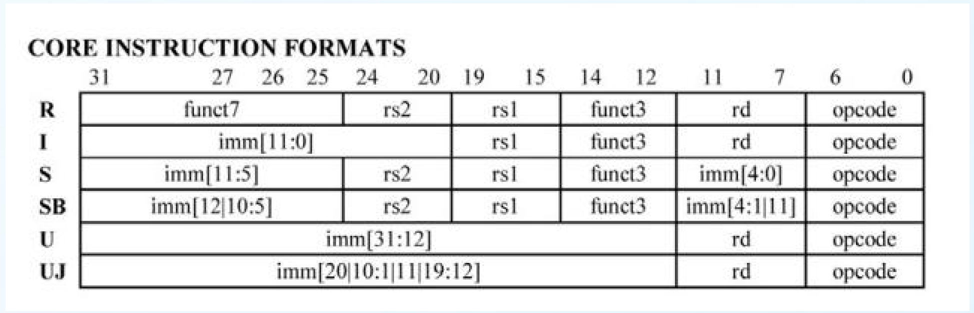
```
initial begin              //part2 of tb
MemWrite = 1'b0;
MemRead = 1'b1;
#75
MemWrite = 1'b1;
end

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

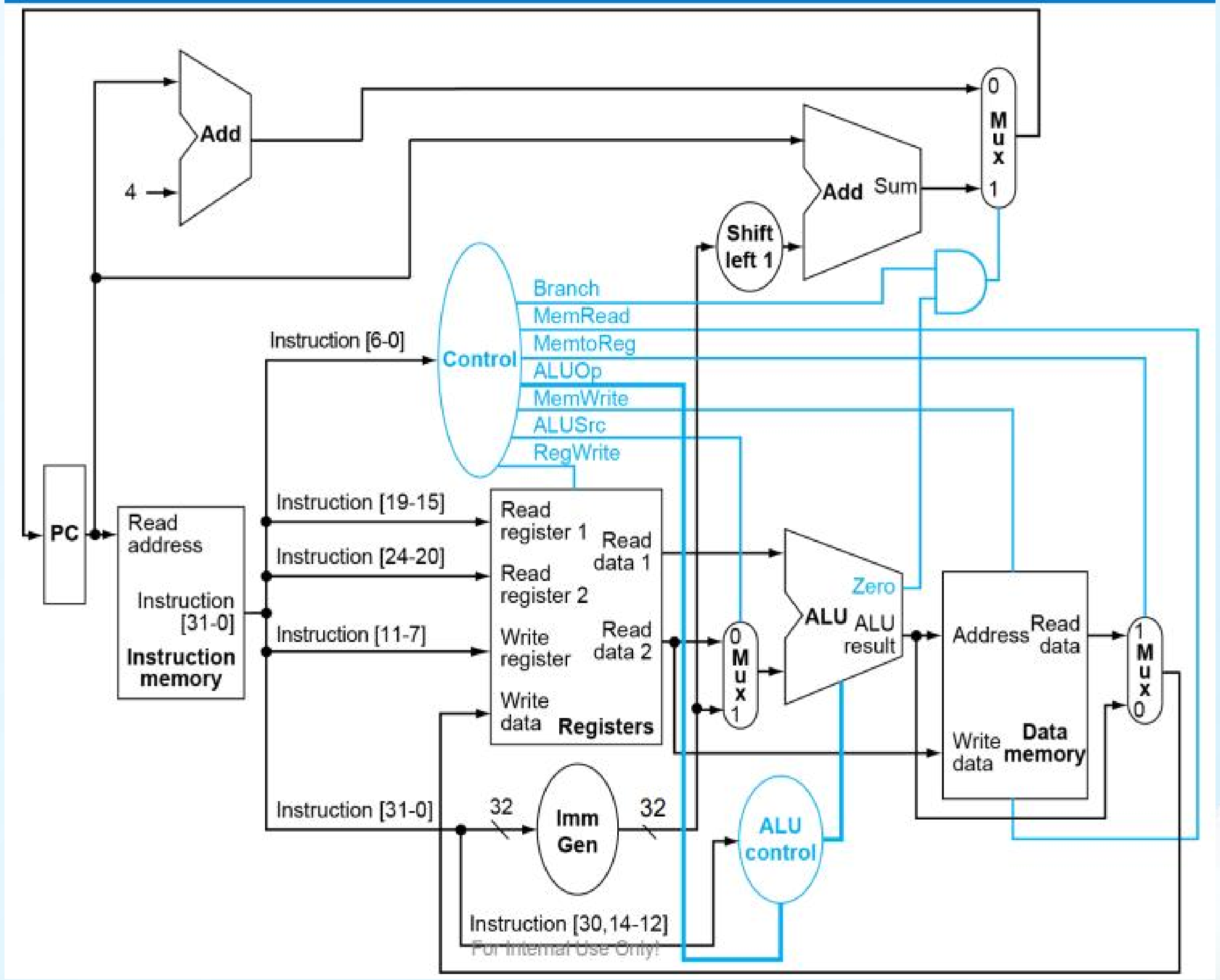


Tips: Control Path & Data Path of CPU



CORE INSTRUCTION FORMATS

	31 27 26 25	24 20	19 15	14 12	11 7	6 0
R	funct7	rs2	rs1	funct3	rd	opcode
I	imm[11:0]		rs1	funct3	rd	opcode
S	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]				rd	opcode
UJ	imm[20|10:1|11|19:12]				rd	opcode



Control Path: Interprete instructions and generate signals to control the data path to execute instructions

Data Path: The parts in CPU with componets which are involved to execute instructions