Computer organization

Lab2 Assembly language-MIPS(1)

Data Details & Implement

TOPIC

Data Processing Details and Implements

- storage
- transfer
- address
- value of Data(relate to the defination and usage)

Practice

- p1-1,p1-2,p1-3; p2-1,p2-2,p2-3; p3.

Assembly Language based on MIPS

- **Data declaration**
  - Data declaration section starts with "**. data**".
  - The declaration means **a piece of memory is required to be allocated**. The declaration usually includes **lable** (name of address on this meomory unit), **size**(optional), and **initial value**(optional).
- **Code definition**
  - Code definition starts with "**.text**", includes **basic instructions, extended instructions, labels of the code**(optional). At the end of the code, "**exit**" system service should be called.
- **Comments:**
  - Comments start from "#" till the end of current line

```
.data
        s1:     .ascii   "Welcome "
        sid:    .space  9
        e1:     .asciiz  "to MIPS World"
.text
main:
        li $v0,8        #to get a string
        la $a0,sid
        li $a1,9
        syscall

        #....

        li $v0,4        #to print a string
        la $a0,s1
        syscall

        li $v0,10       #to exit
        syscall
```

Data storage （1）

- Data Storage: instruction(fastest,smallest), register, memory(slowest, biggest)

```
//in Java， C， Phython
a = b + 1
# in MIPS
lw $t0, b               #get data from memory to register
addi $t1, $t0, 1
sw $t1, a               #get data from register to memory
```

- **Instruction :** data is part of instruction, the data in the instruction can be obtained while analyzing the instruction, the data is also called immediate data.

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Implement in verilog：
suppose the instruction is got and stored in “ins”, “ins” is 32 bits (the index range is 0 to 31).

“ins[15:0]” is the “constant or address” of I-type instruction
“ins[10:6]” is the “shamt” of R-type instruction.

Data storage(2)

- **Reg**

  - Registers are small storage areas used to store data in the CPU, which are used to temporarily store the data and results involved in the operation.
  - All MIPS **arithmetic** instructions MUST operate on registers.The **size of registers** in MIPS32 is **32 bits.**

```
//in Java， C， Phython
a = b + 1

# in MIPS
lw $t0, b      #from memory to register
addi $t1, $t0, 1
sw $t1, a      #from register to memory
```

*Implement **Registers** in verilog：*
Suppose there are x registers, each of them is y bits.

reg [y-1:0] regs **[x-1:0]**; //defination

“regs[a]” is one register of regs (“a” is an integer between 0 and x-1).

Data storage(3)

- **Memory**
  - Both instruction and data are stored in the memory.
    - In MIPS32, the bit width of instrucion code is **32**.
  - Address based on **bytes**.
  - **Continuous addressing** of memory units in memory.

Text Segment

pt	Address	Code	Basic
	0x00400000	0x24020008	addiu $2, $0, 0x00000008
	0x00400004	0x3c011001	lui $1, 0x00001001
	0x00400008	0x34240008	ori $4, $1, 0x00000008
	0x0040000c	0x24050009	addiu $5, $0, 0x00000009
	0x00400010	0x0000000c	syscall
	0x00400014	0x3c011001	lui $1, 0x00001001
	0x00400018	0x80280011	lb $8, 0x00000011($1)

Data Segment

Address	Value (+0)	Value (+4)
0x10010000	c l e W	e m o
0x10010020	\0 \0 \0 \0	\0 \0 \0 \0

```
//in Java， C， Phython
a = b + 1

# in MIPS
lw $t0, b      #from memory to register
addi $t1, $t0, 1
sw $t1, a      #from register to memory
```

*Implement **Memory** in verilog*：

- IP core "Block Memory" are used as instruction memory and data memory.
  (more details could be found in tips3)
  - sequential logic circuit
  - input: "clk", "address", "write_enable", "write_data"
  - output: "read_data"
  - How to determine the width of "address" , "write_data" and "read_data" ?
  - Can read and write memory occur at the same time?

MIPS Instruction： Load & Store

- In MIPS
  - Access the data in **memory** could ONLY be invoked by two types of instruction: **load** and **store**.
  - All the **calculation** are based on the data in **Registers**.
  - Unit Conversion(in MIPS32)
    - **1 word = 32bit = 2*half word(2*16bit) = 4* byte(4*8bit)**
    - **1 double word = 2 word = 64bit**

Name	Example	Comments
32 registers	\$s0-\$s7, \$t0-\$t9, \$zero, \$a0-\$a3, \$v0-\$v1, \$gp, \$fp, \$sp, \$ra, \$at	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register \$zero always equals 0, and register\$at is reserved by the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

Load (Load to Register)

```
lw          register_destination, RAM_source
                                     # copy word (4 bytes) at
                                     # source_RAM location
                                     # to destination register.
                                     # load word -> lw

lb          register_destination, RAM_source
                                     # copy byte at source RAM
                                     # location to low-order byte of
                                       # destination register,
                                       # and sign -e.g. tend to
                                       # higher-order bytes
                                       # load byte -> lb

 li          register_destination, value
                                       #load immediate value into
                                       #destination register
                                       #load immediate --> li
```

"la" (load address) is a extended (presudo) instruction, which is implemented by two basic instructions: **lui**(load upper immediate), **ori**(bitwise OR immediate).

Labels

Label	Address ▲
mips1.asm	
s1	0x10010000
sid	0x10010008
e1	0x10010010

Basic		
addiu $2, $0, 0x00000008	6:	li $v0, 8
lui $1, 0x00001001	7:	la $a0, sid
ori $4, $1, 0x00000008		

Store (Store to Memory)

```
sw          register_source, RAM_destination
                                #store word in source register
                                # into RAM destination

sb          register_source, RAM_destination
                                #store byte (low-order) in
                                #source register into RAM
                                #destination
```

Q: Is there any need to implement the “sa” instruction(store address), why ? If need to implement “sa”, how to do it ?

The Address of the Target Unit in the Memory(1)

- The **"label"**
  - The value of "label" is determined by the Assembler according to the assembly source code.

```
.data
	s1:	.ascii "Welcome "
	sid:	.space 8
	e1:	.asciiz " to MIPS World"
```

Labels

Label	Address ▲
mips1.asm	
s1	0x10010000
sid	0x10010008
e1	0x10010010

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	c l e W	e m o	\0 \0 \0 \0	\0 \0 \0 \0	o t	S P I M	r o W	\0 \0 d l

e.g: la $a0, sid

Basic		
addiu $2, $0, 0x00000008	6:	li $v0, 8
lui $1, 0x00001001	7:	la $a0, sid
ori $4, $1, 0x00000008		

The Address of the Target Unit in the Memory(2)

- The address need to be got from the **Register**(Using the content in register as address).
  - **Load** the word **from the memory unit** whose address is in the register "t0" **to the register** "t2". `lw $t2, ($t0)`
  - **Store** the word **from the register** "t2" **to the memory unit** whose address is in the register "t0". `sw $t2,($t0)`
- **The address need to be caculated by Baseline + offset**(Using the sum of the baseline address and offset as address).
  - Load the word from the memory unit whose **address is the sum of 4 and the value in register "t0"** to the register "t2". `lw $t2, 4($t0)`
  - Store the word in register "t2" to the memory unit whose **address is the sum of -12 and the value in the register "t0"**. `sw $t2,-12($t0)`

Practice 1

Use MIPS to program and realize the following functions on Mars: Using 2 syscall to get the sid which has 8 numbers from input, print out the string:
Welcome XXXXXXXX to MIPS World
(XXXXXXXX is an 8-digit number)

1-1. complete the code on the right hand, move the string “ to MIPS World” from the memory unit addressed by “e1” to the memory unit addressed by the sum of 8 and “sid”.
1-2. Is there any other way to implement the function
1-3. Which one would get better performance:
1-1 or 1-2 ?

Tips:
1. While get and put string by syscall, the end of string is “\0” which means getting a string would add a “\0” at the end of string, print a string would end with “\0”
2. The difference between “ascii” and “asciiz” is that “asciiz” would add “\0” at the end of the string while “ascii” would not.

```
.data
    s1:     .ascii   "Welcome "
    sid:    .space  9
    e1:     .asciiz  "to MIPS World"
.text
main:
    li $v0,8       #to get a string
    la $a0,sid
    li $a1,9
    syscall

#complete code here

    li $v0,4       #to print a string
    la $a0,s1
    syscall


    li $v0,10     #to exit
    syscall
```

The value of Data（1）relate to the defination

```
name:       storage_type    value(s)
```

```
example

var1:       .word   3       # create a single integer:
                            #variable with initial value 3

array1:     .byte   'a','b' # create a 2-element character
                            # array with elements initialized:
                            # to  a  and  b

array2:     .space  40      # allocate 40 consecutive bytes,
                            # with storage uninitialized
                            # could be used as a 40-element
                            # character array, or a
                            # 10-element integer array;
                            # a comment should indicate it.

string1:    .asciiz "Print this.\n"        #declare a string
```

```
.data
        var1:      .word  3
        array1:   .byte 'a', 'b'
```

Labels

Label	Address ▲
mips1.asm	
var1	0x10010000
array1	0x10010004

Data Segment

Address	Value (+0)	Value (+4)
0x10010000	0x00000003	0x00006261

The value of Data （2） relate to the usage(1)

- while calculate the data, if the instruction ends with “**u**” means the data are treated as **unsigned** integer, else the data are treated as **signed by defalut**.

```
.include "macro_print_str.asm"
.data
.text
main:
	print_string("\n -1 less than 1 using slt:")
	li $t0,-1
	li $t1,1
	slt $a0,$t0,$t1
	li $v0,1
	syscall

	print_string("\n -1 less than 1 using sltu:")
	sltu $a0,$t0,$t1
	li $v0,1
	syscall
	end
```

TIPS:

1) slt $t1,$t2,$t3

set less than: if $t2 is less than $t3, then set $t1 to 1 else set $t1 to 0

2) sltu $t1,$t2,$t3

set less than unsigned: if $t2 is less than $t3 using unsigned comparision, then set $t1 to 1 else set $t1 to 0

The value of Data （2） relate to the usage(2)

```
.data
	tdata: .byte 0x0F00F0FF
	sx: .asciiz "\n"
.text
main:
	lb $a0,tdata
	li $v0,1
	syscall

	li $v0,36
	syscall

	li $v0,10
	syscall
```

Q1. What’s the data stored in the byte of address “tdata”?
Q2. What’s the data stored in the $a0 after execute “lb $a0,tdata”?

Q3. What are their values when they are treated as unsigned and signed integers respectively ?

Tips: syscall
1) code in $v0 : **1**
Display data in **$a0** as **signed** decimal value

2) code in $v0 : **36**
Display data in **$a0** as **unsigned** decimal value

The value of Data （2） relate to the usage(3)

```
.include "macro_print_str.asm"
.data
	tdata: .byte 0x80
.text
main:
	lb $a0,tdata
	li $v0,1
	syscall

	print_string("\n")
	lb $a0,tdata
	li $v0,36
	syscall

	end
```

```
.include "macro_print_str.asm"
.data
	tdata: .byte 0x80
.text
main:
	lbu $a0,tdata
	li $v0,1
	syscall

	print_string("\n")
	lbu $a0,tdata
	li $v0,36
	syscall

	end
```

Tips: syscall
1) code in $v0 : **1**
Display data in **$a0** as **signed** decimal value

2)code in $v0 : **36**
Display data in **$a0** as **unsigned** decimal value

Q1: Run the two demos, what's the value stored in the register $a0 after the operation of ‘lb’ and ‘lbu’

Q2: using “-1” as initial value of tdata instead of “0x80”, answer Q1 again.

Practice 2(1)

2-1. The data in a word is 0x12345678, print it in hexdecimal, then exchange the bytes of this word to get the new value 0x78563412 and print the updated data in hexdecimal.

Tips: more information could be get from the help page of Mars.

2-2. Implement in verilog: the orignal data is stored in register “x” which is 8bits, get the data in “x”, extend the data to 32bit with sign bit of it, store the updated data to the register “y” which is 32bits.

For example:

x: 8’b**0**100_0000; y:32’b0000_0000_0000_0000_0000_0000_0100_0000

y: 8’b**1**100_0000; y:32’b1111_1111_1111_1111_1111_1111_1100_0000

Practice 2(2)

2-3. Run the code on the right hand

Answer the questions

1) what's the value of lable alice?

2) what's the value of lable tony?

3) what's the output after execute the syscall on line 23 ?

```
1  .data
2      name:    .space  16        #malloc 16 byte , not initialize   #####  name value : 0x10010000
3      mick:    .ascii "mick\n"    # malloc 4+1 = 5byte = 5 * asciic(byte)
4      alice:   .asciiz "alice\n"  #####  what's the value of alice ?
5      tony:    .asciiz "tony\n"   #####  what's the value of tony ?
6      chen:    .asciiz "chen\n"
7
8  .text
9  main:
10     la $t0,name          #using  name value which is an address, load this address to $t0
11
12     la $t1,mick
13     sw $t1,($t0)         #1,get value of $to, use it as the address of a piece of memory
14     la $t1,alice
15     sw $t1,4($t0)        #baseline : the content of  $t0 , offset :4
16     la $t1,tony
17     sw $t1,8($t0)
18     la $t1,chen
19     sw $t1,12($t0)
20
21     li $v0,4
22     lw $a0,0($t0)
23     syscall              #what's the output while this syscall is done
24
25     li $v0,10
26     syscall
```

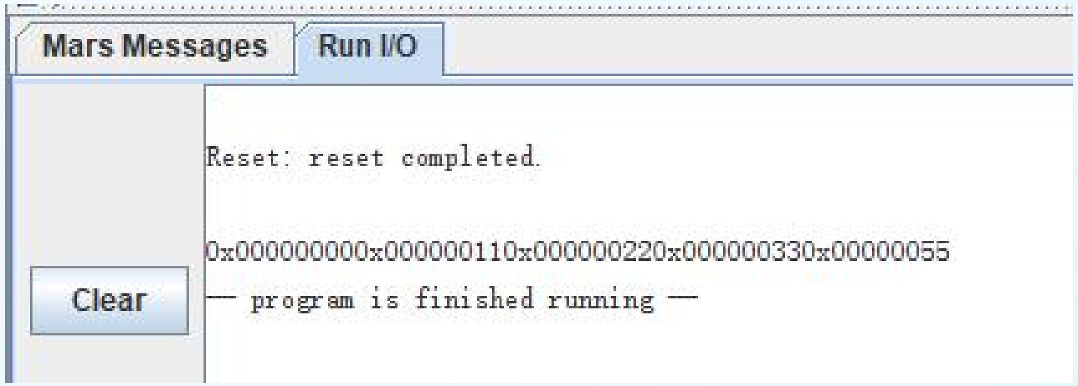
Tips1 : macro_print_str.asm

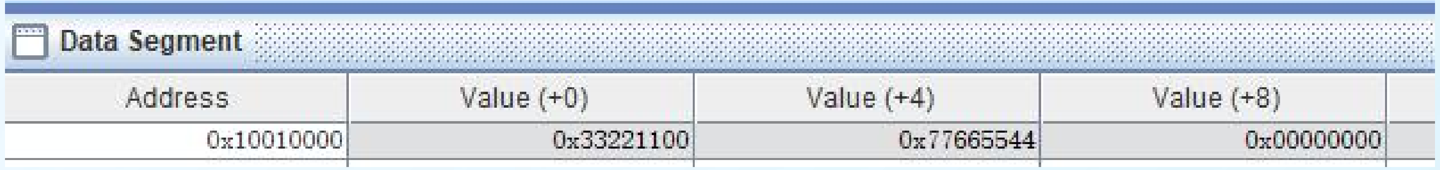
```
.macro print_string(%str)
	.data
		pstr:   .asciiz   %str
	.text
		la $a0,pstr
		li $v0,4
		syscall
.end_macro

.macro end
	li $v0,10
	syscall
.end_macro
```

Get help of defination and usage about macro from Mars‘ help page.

While using the macro, put this file to the same directory as the file which use the macro.

Tips2: the data address in Mars




Data Segment

Address	Value (+0)	Value (+4)	Value (+8)
0x10010000	0x33221100	0x77665544	0x00000000

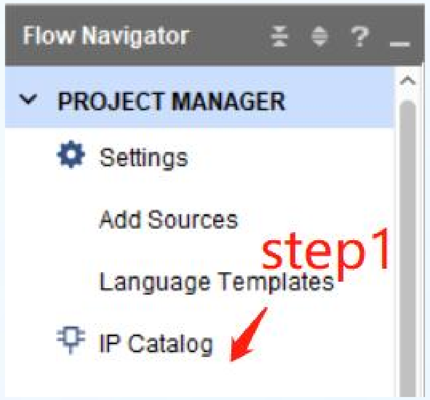
```
.include "macro_print_str.asm"
.data
	tdata0: .byte
0x00,0x11,0x22,0x33,0x44,0x55,0x66,0x77
.text
main:
	la $t0,tdata0
	lb $a0, ($t0)
	li $v0,34
	syscall

	la $t0,tdata0
	lb $a0, 1($t0)
	syscall

	lb $a0, 2($t0)
	syscall

	lb $a0, 3($t0)
	syscall

	lb $a0, 5($t0)
	syscall

	end
```

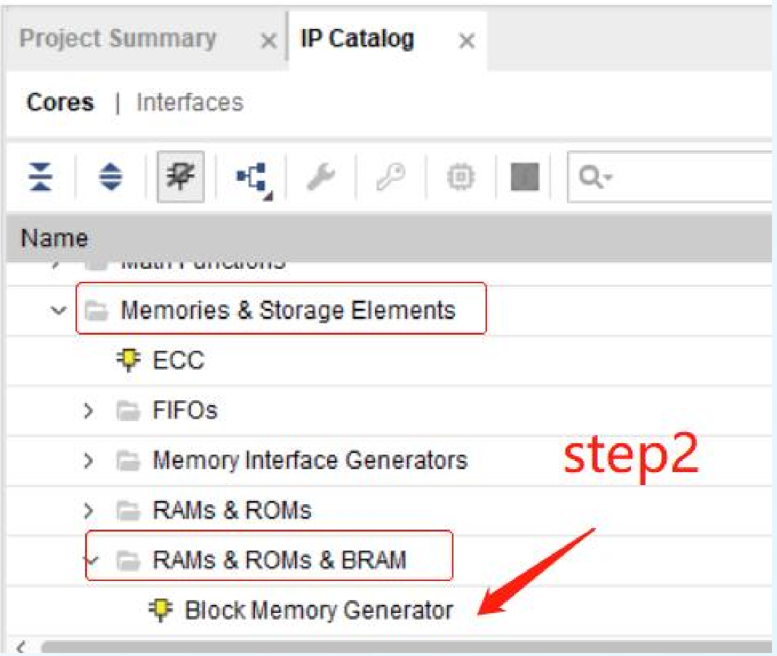
Practice 3

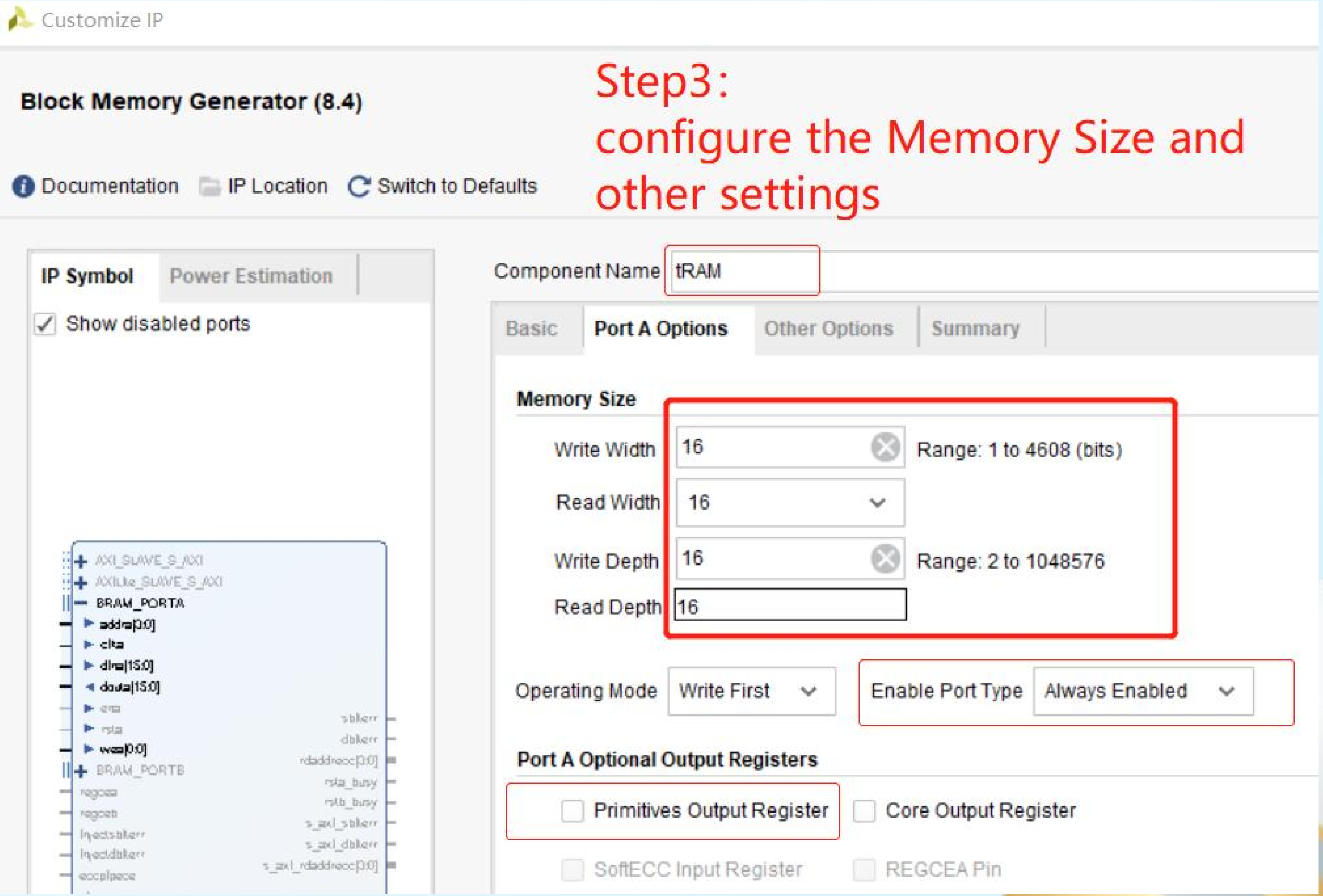
Custome a block memory and generate it by using IPcore of vivado, it should be a RAM, its size is 64KB, the bit width of data bus is 8. Build a testbench to verify its function, to write a byte to a memory unit and read it out.

1. When does reading and writing occur? Is it on the rising(posedge) or falling edge(negedge) of the clock?
2. If read and write occure at the same time, read first or write first?
3. Is there any delay about write memory? while write_enable is valid, would the write memory process at the same time, or a cycle delay ?

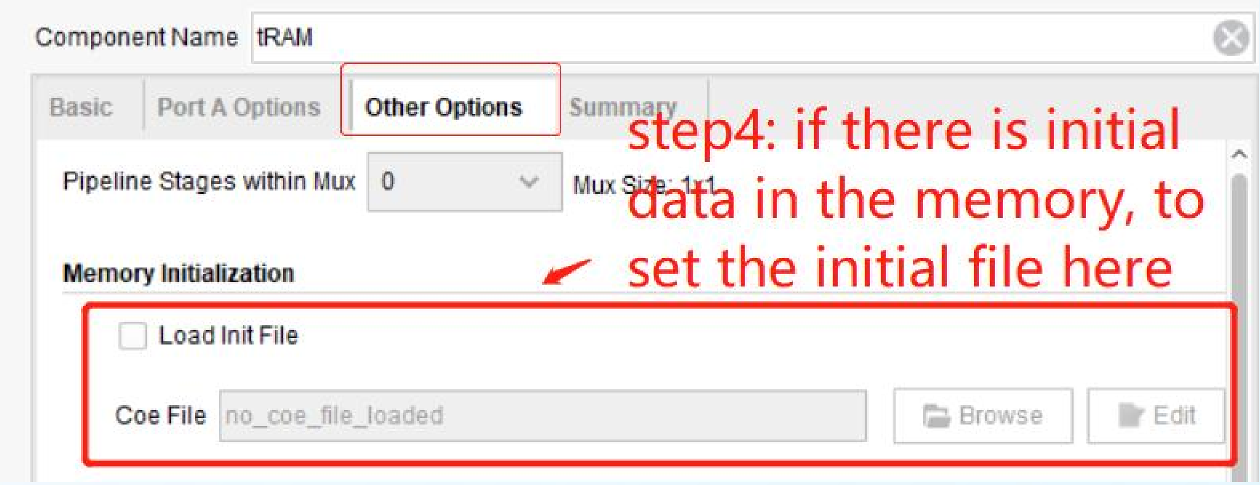
Tips3: Using IP cores(block memory) in Vivado(1)



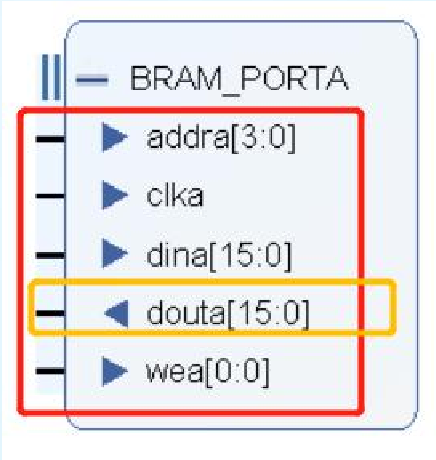


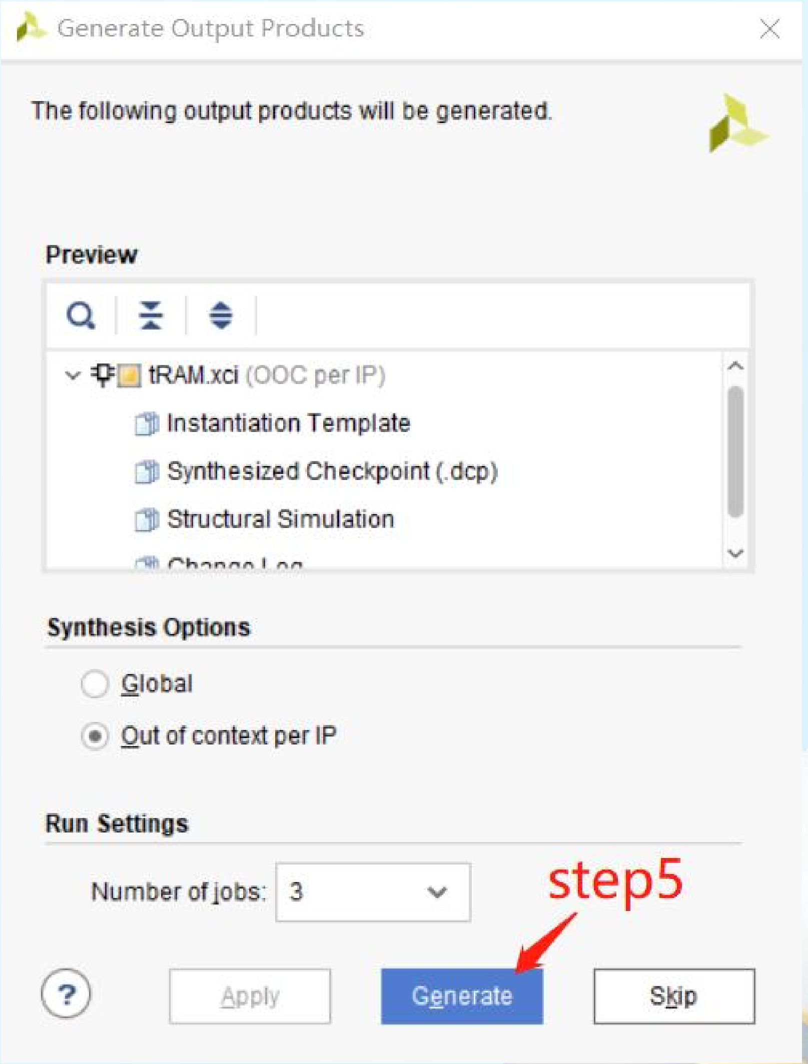


Tips3: Using IP cores(block memory) in Vivado(2)



step4: if there is initial data in the memory, to set the initial file here

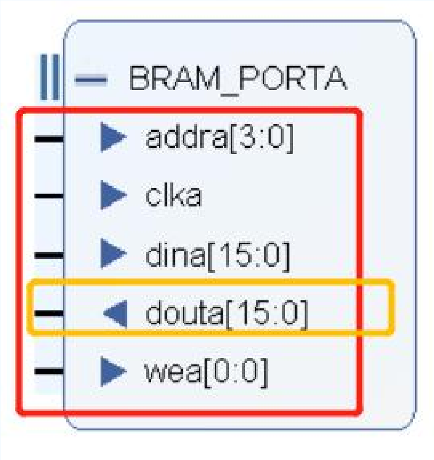




step5

Tips3: Using IP cores(block memory) in Vivado(3)

While the IP core has been generated, it could be found in the “Design Sources” and could be instanced.




```
// a demo about instance the IP core tRAM
module tRam(  /*to be complete*/);
   /*to be complete*/
    reg clk,we;
    reg [15:0] addr,wdata;
    wire [15:0] rdata;

    tRAM utram1(.addra(addr),.clka(clk),.dina(wdata),.douta(rdata),.wea(we));
endmodule
```