차세대 반도체 디지털 혁신공유대학 사업단

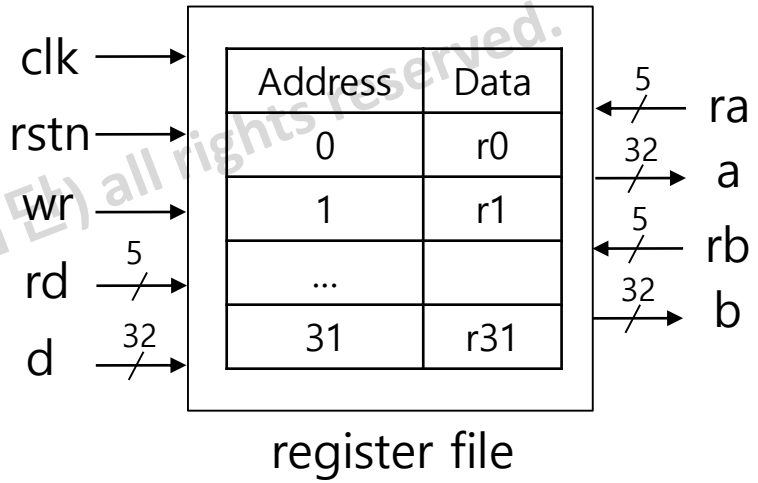
POLARIS

Lecture 3: Introduction to RISC-V Instruction/data memory, decoder

Xuan-Truong Nguyen

Recap: Register File (RF)

- Keep frequently-used data
- Architecture: RF consist of 32 registers, 32-bits each
  - Numbered from 0 to 31
  - Can be referred by number: r0, r1, …, r31
- Interface
  - **One write port** d indexed via an address rd
    - On falling edge when wr = 1
  - **Two read ports a, b** indexed via addresses ra, rb

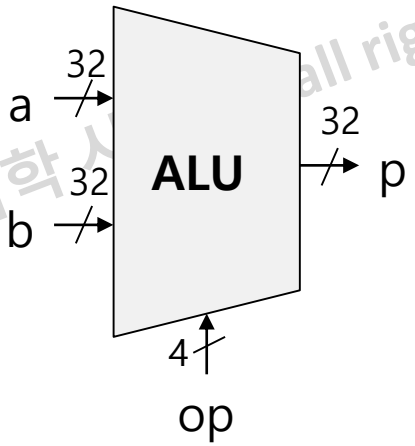


Recap: ALU

- Inputs: two 32-bit operands and one 4-bit operation code (opcode)
- Output: 32-bit result

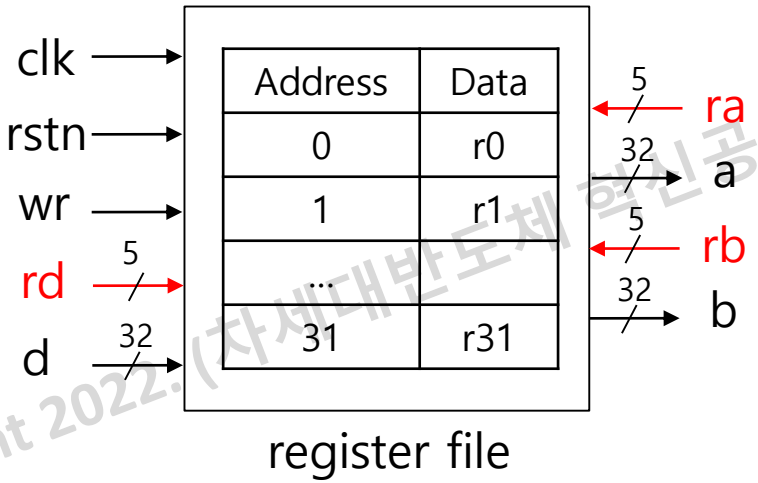
- Pseudocode
  - Case (opcode)
    - ADD
    - SUB
    - AND
    - OR
    - SLL
    - SRL

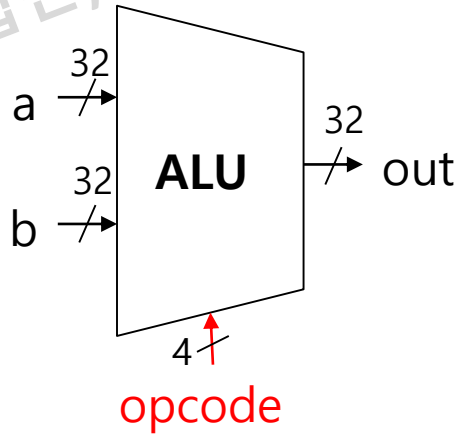
      .....
  - Endcase



Recap: Register File and ALU

- How to define those following signals?
  - Addresses ra, rb, rd of register file
  - Opcode of ALU





Lecture plan

- Today, we will
  - Work on a RISC-V simulator
  - Study a RISC-V memory model
  - Study the RISC-V instruction format
- Labs
  - Lab 1: Instruction memory
  - Lab 2: Decoder
  - Lab 3: RISC-V sim

Road map

RISC-V Simulator

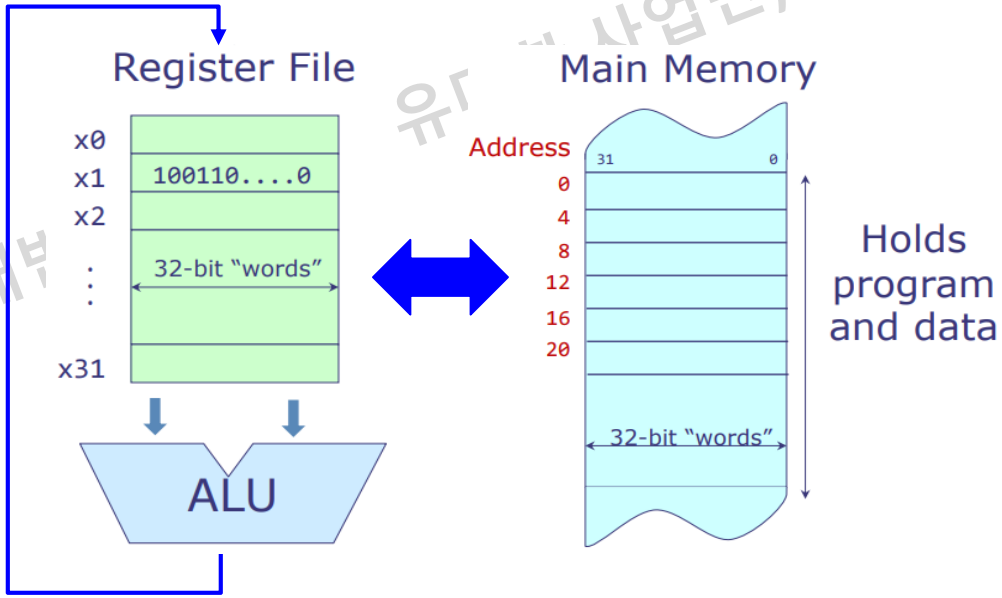
Memory
Instruction/data memory

RISC-V Instruction format
Decoder

Simple core

Components of MicroProcessor

- Machine language can directly represent this structure.
  - Register files: store frequently-used data
  - Arithmetic and Logic Unit (ALU): Do calculation
  - Main memory: Hold program/instruction codes and data



Register Operands

- Arithmetic instructions use register operands

- RISC-V 32I has a 32 × 32-bit register file
  - Use for frequently accessed data
  - 32-bit data is called a "word"
    - 32 x 32-bit general purpose registers x0 to x31

#	Name	Usage
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5~x7	t0~t2	Temporaries (Caller-save registers)
x8	s0/fp	Saved registers/ Frame pointer
x9	s1	Saved registers
x10~x17	a0~a8	Function arguments
x18~x26	s2~s11	Temporaries (Caller-save registers)
x28~x31	t3~t6	Temporaries (Caller-save registers)

RISC-V Registers

Register Operand Example

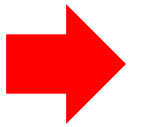
- C code:

  f = (g + h) - (i + j);

  - Variables f, …, j are in x19, x20, …, x23

- Compiled RISC-V code:

```
add t0, g, h   // temp t0 = g + h
add t1, i, j   // temp t1 = i + j
add f, t0, t1  // f = t0 - t1
```



```
add x5, x20, x21
add x6, x22, x23
sub x19, x5, x6
```

Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store results from register to memory
- Memory is byte-addressed
  - Each address identifies an 8-bit byte
- RISC-V is Little Endian
  - Least-significant byte at least address of a word
  - *c.f.* Big Endian: most-significant byte at least address
- RISC-V does NOT require words to be aligned in memory
  - Unlike some other ISAs

Question :→
RISC-V → Little Endian
But ARM follows big endian

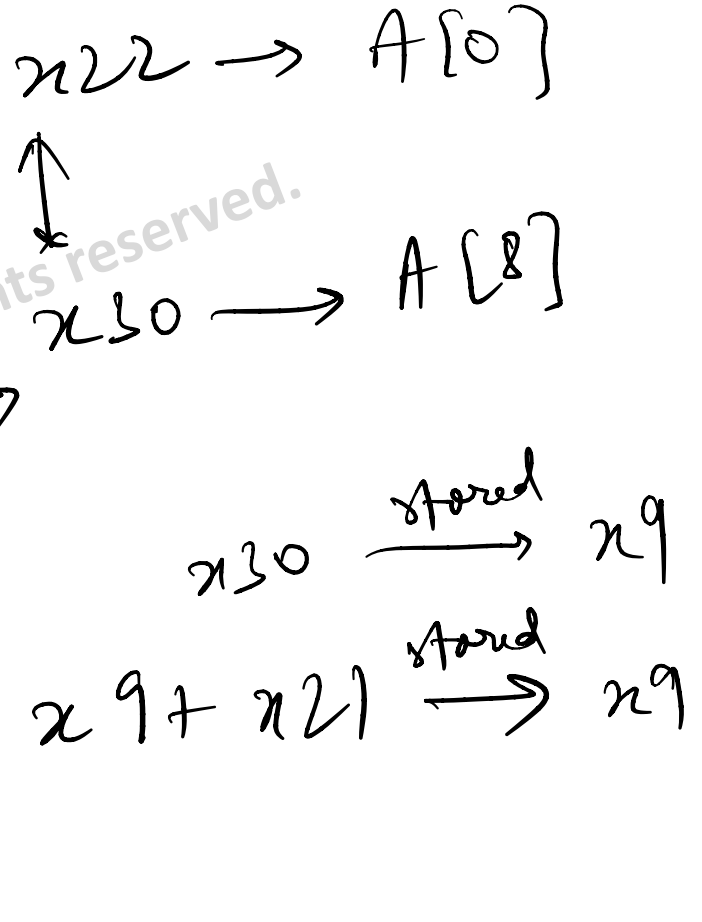
Memory Operand Example

- C code:

  `A[12] = h + A[8];`

  - h in x21, the base address of A in x22
- Compiled RISC-V code:
  - Index 4 requires an offset of 32
    - 4 bytes per word

```
ld      x9, 32(x22)
add     x9, x21, x9
sd      x9, 48(x22)
```



RISC-V Simulator: Venus

- Venus: A RISC-V Simulator
  - A web-based simulator for RISC-V Processor
- Example code

what this really does?

variables used by computer

C code

```
// f in x19
// g in x20
// h in x21
// I in x22
// j in x23

f = ( g + h ) – ( i + j )
```

Compiled RISC-V code

```
add x5, x20, x21
add x6, x22, x23
sub x19, x5, x6
```

RISC-V Simulator: Venus

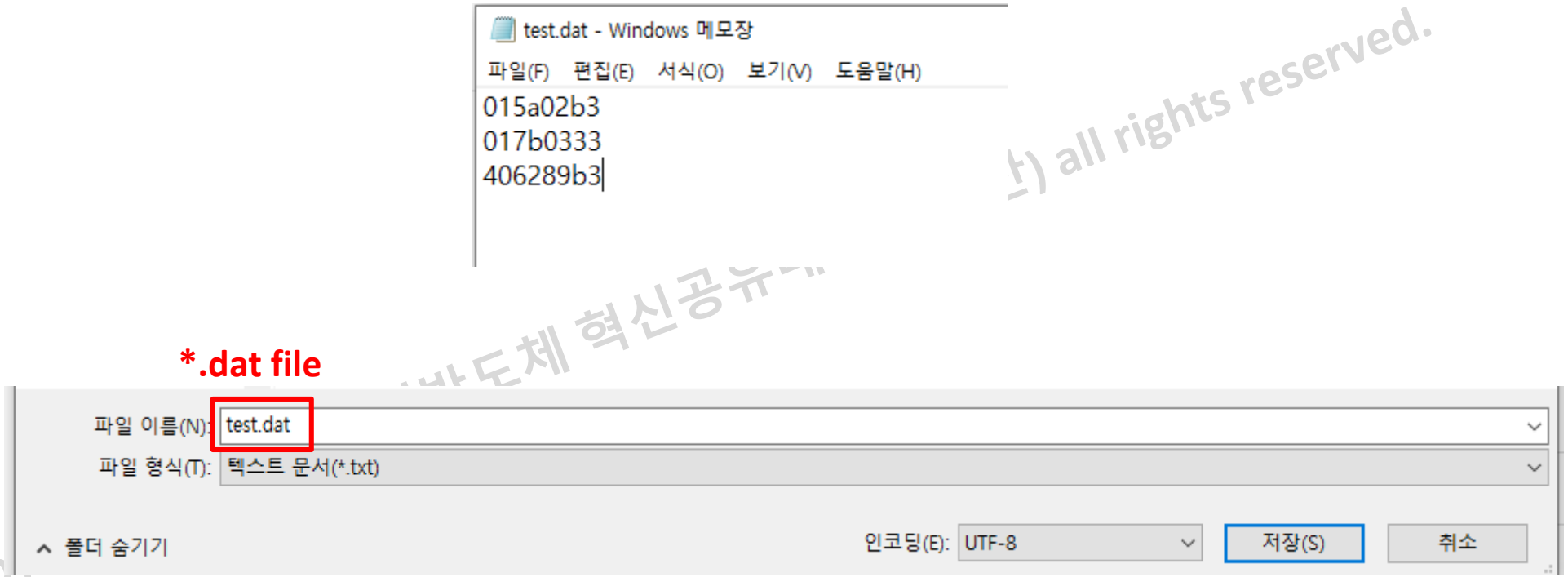
- Venus: A RISC-V Simulator
- **https://www.kvakil.me/venus/**
- Click to **editor** tab

RISC-V Simulator: Venus

- Venus: A RISC-V Simulator
- Click to **Simulator**

RISC-V Simulator: Venus

- Save generate machine codes in *.dat file



To do …

- Generate machine code of examples
- Save generate machine codes in mem.hex file

```
int A[64];
int sum = 0;
for (int i=0; i<64; i++)
    sum +=  A[i];
```

```
addi x9, x0, 64 # x9=&A[0]
add x10, x0, x0 # sum=0
add x11, x0, x0 # i=0
Loop:
  lw x12, 0(x9) # x12=A[i]
  add x10,x10,x12 # sum+=
  addi x9,x9,4  # &A[i++]
  addi x11,x11,1 # i++
  addi x13,x0,64 # x13=64
  blt x11,x13,Loop
```

Road map

RISC-V Simulator

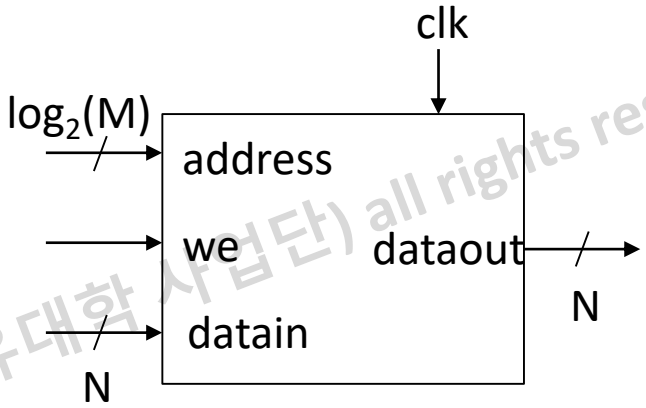
Memory
Instruction/data memory

RISC-V Instruction format
Decoder

Simple core

Memory-Block Basics

- Uses
  - Whenever a large collection of state elements is required
  - Data and program storage
  - General purpose registers
  - Data buffering
  - Table lookups
- Basic Types
  - RAM: Random access memory
  - ROM: Read-only memory
  - EPROM, FLASH: Electrically programmable read-only memory



M ×N memory
M: Depth
N: Width
M words of memory
Each word has N bits

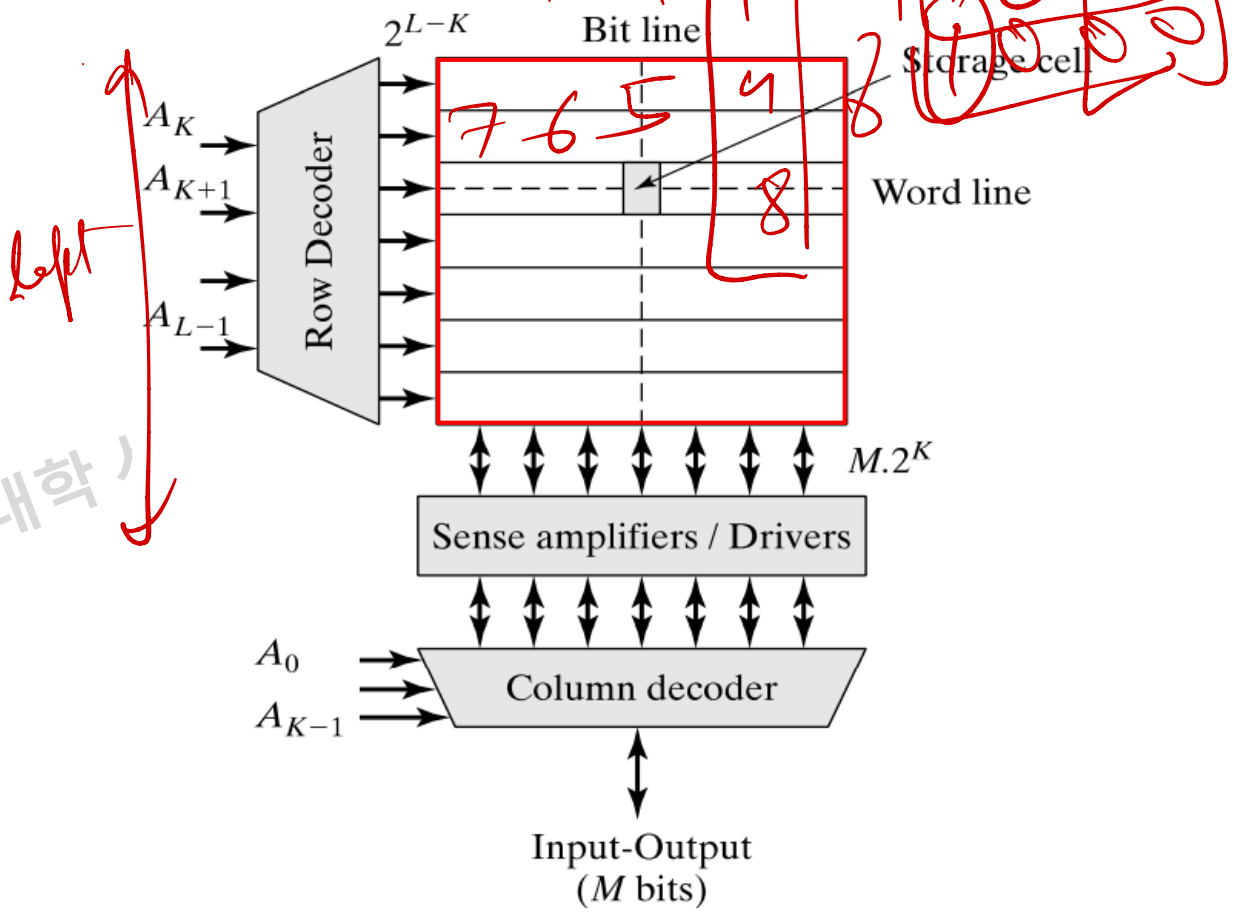
Memory Components

- Volatile: ***Values lost*** after memory devices are shut down.
  - Random Access Memory (RAM)
    - DRAM: "Dynamic" RAM
    - SRAM: "Static" RAM
- Non-volatile: ***Values remain*** after memory devices are shut down
  - Read Only Memory (ROM)
    - Mask ROM: "mask programmable"
    - EPROM: Electrically programmable
    - EEPROM erase electrically programmable
    - FLASH memory – similar to EEPROM with programmer integrated on chip

Memory Block Architecture

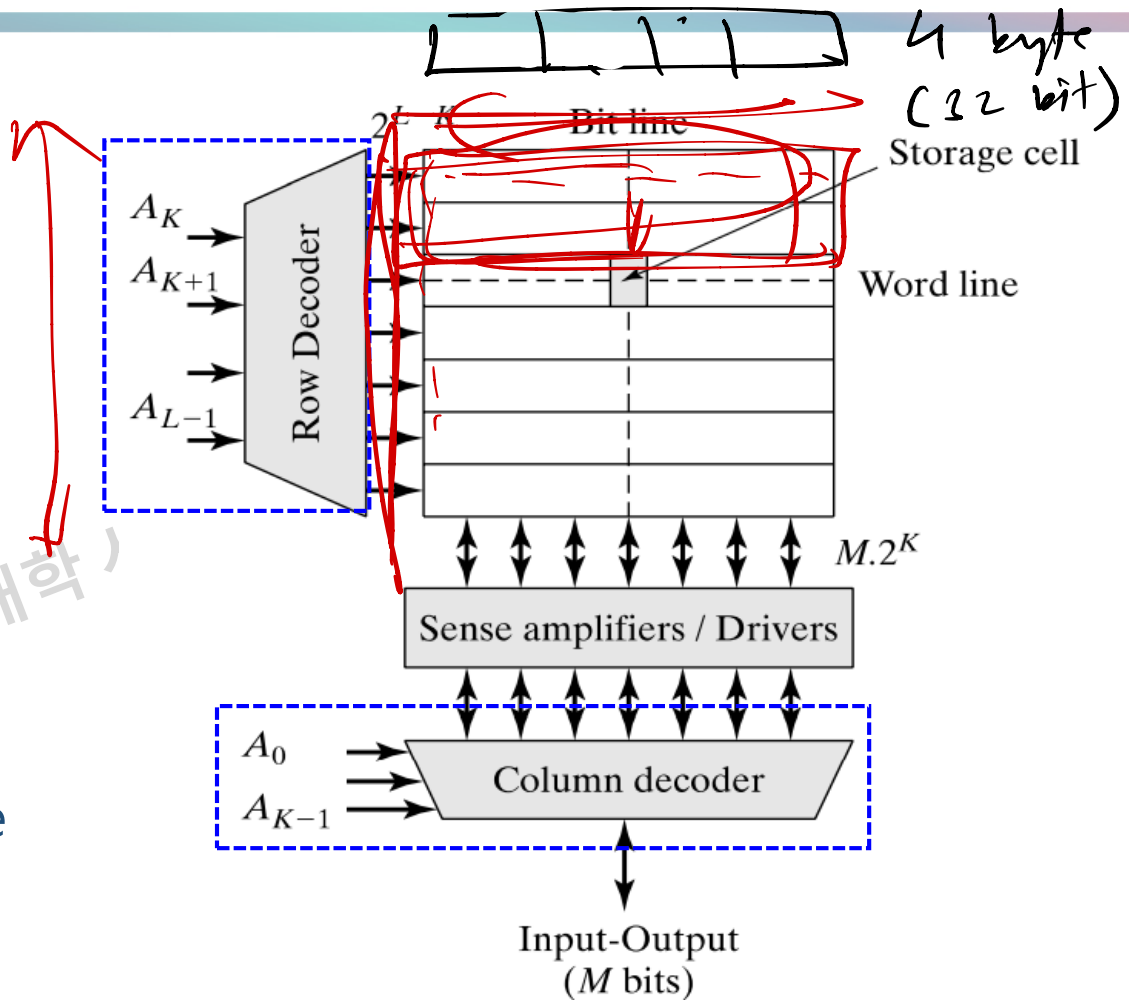
- Memory/Storage cells
  - Word lines are used to select a row for reading or writing
  - Bit lines carry data to/from periphery

- Core aspect ratio keep close to 1 to help balance delay on word line versus bit line



Memory Block Architecture

- Address bits are divided between the two decoders
  - A[0]~A[K-1]: Column decoder → Bit
  - A[K]~A[L-1]: Row decoder → Word
- Row decoder used to select word line
- Column decoder used to select one or more columns for input/output of data

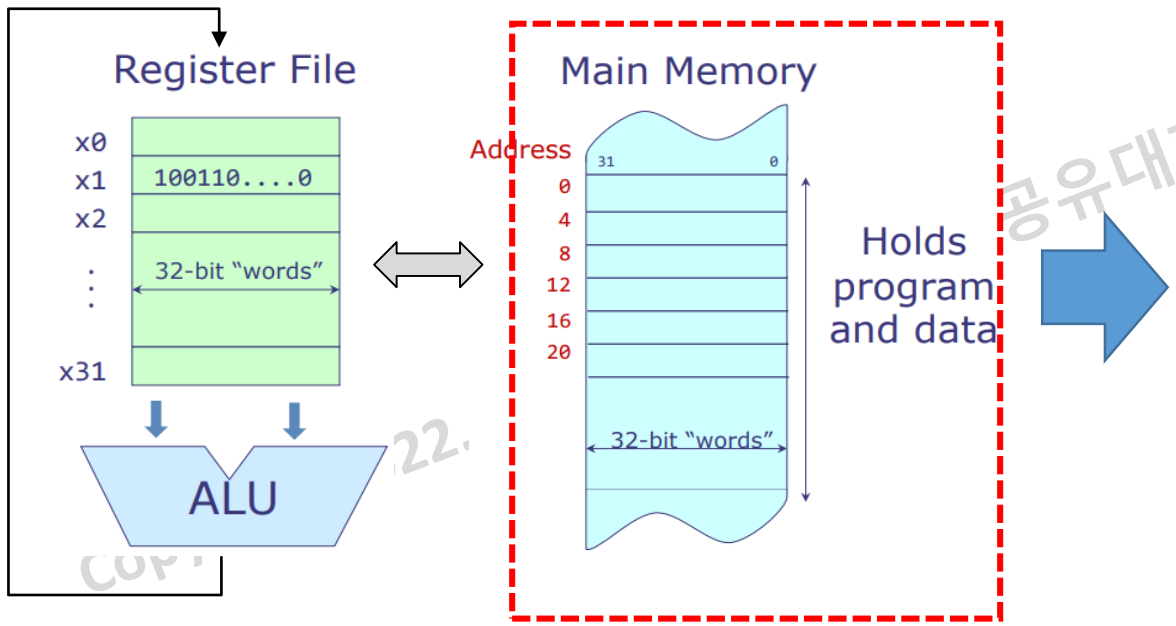


Lab 1: Memory model

- Lab 1: Memory model (~Instruction/Data cache)
  - Implement a memory model for RISC-V
  - Run a simulation
  - Show the output result

Memory model (riscv_memory.v)

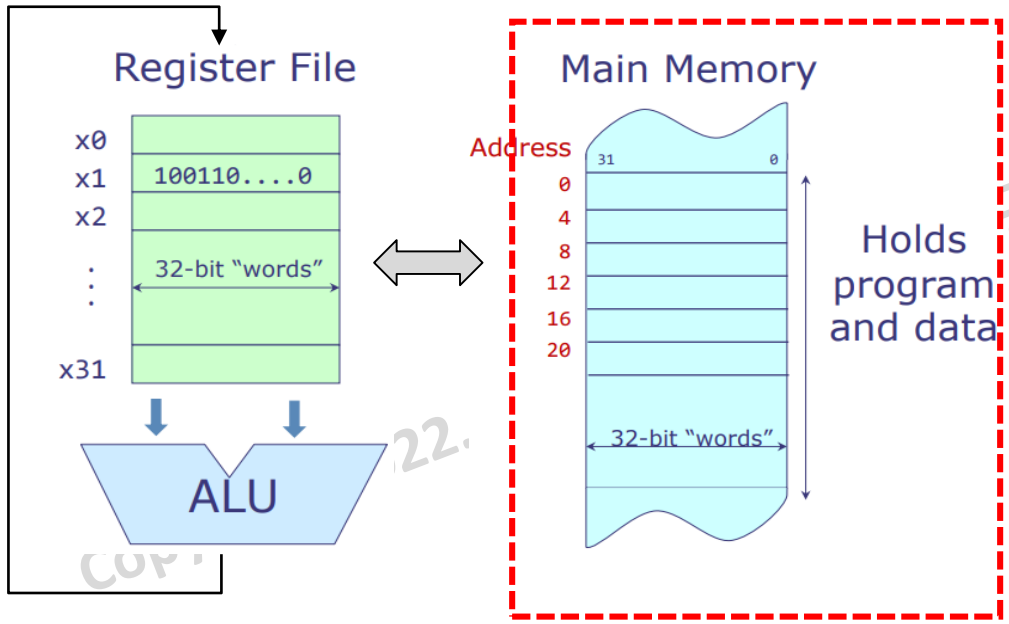
- Main memory: Hold program/instruction codes and data
- Parameters: SIZE=8192, initialized file (FIRMWARE = mem.hex).




```
module riscv_memory #(
    parameter       SIZE     = 8192,
    parameter       FIRMWARE = "mem.hex"
)(
    input           clk_i,
    input           reset_i,

    input   [31:0] iaddr_i,
    output  [31:0] irdata_o,
    input          ird_i,

    input   [31:0] daddr_i,
    input   [31:0] dwdata_i,
    output  [31:0] drdata_o,
    input    [1:0] dsize_i,
    input          drd_i,
    input          dwr_i
);
```

Memory cells

- Memory cells are defined as a 2D array. Its size is 8192 bytes
- Each word has 32 bits or 4 bytes
- $readmemh: Read a hexadecimal file
  - Initialize memory cells




```
//--------------------------------
// Memory Initialization
//--------------------------------
reg [31:0] mem_r [0:SIZE/4-1];

initial begin
  $readmemh(FIRMWARE, mem_r);
end
```

Memory model (riscv_memory.v)

- Ports for **instruction**: READ ONLY
  - Input
    - ird_i: enable signal
    - iaddr_i: request address
  - Output
    - irdata_o: return instruction

```
module riscv_memory #(
  parameter      SIZE     = 8192,
  parameter      FIRMWARE = "mem.hex"
)(
  input          clk_i,
  input          reset_i,

  input  [31:0] iaddr_i,
  output [31:0] irdata_o,
  input          ird_i,

  input  [31:0] daddr_i,
  input  [31:0] dwdata_i,
  output [31:0] drdata_o,
  input   [1:0] dsize_i,
  input          drd_i,
  input          dwr_i
);
```

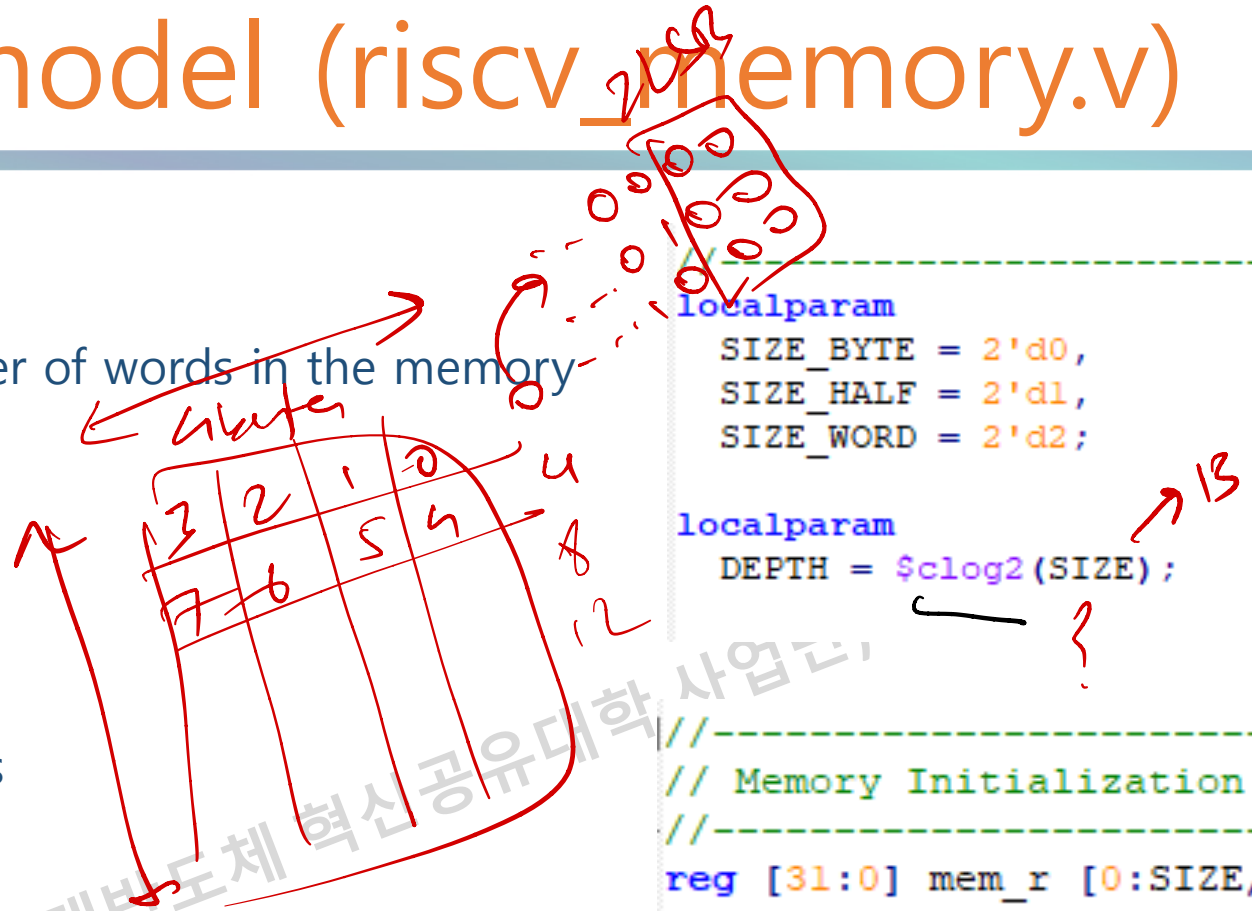
Memory model (riscv_memory.v)

- Ports for **data**: READ/WRITE
  - Input
    - drd_i: enable signal
    - dwr_i: write enable signal
    - daddr_i: request address
    - dsize_i: access a byte in a word
    - dwdata_i: write data
  - Output
    - drdata_o: return data

```
module riscv_memory #(
  parameter      SIZE     = 8192,
  parameter      FIRMWARE = "mem.hex"
)(
  input          clk_i,
  input          reset_i,

  input  [31:0] iaddr_i,
  output [31:0] irdata_o,
  input          ird_i,

  input  [31:0] daddr_i,
  input  [31:0] dwdata_i,
  output [31:0] drdata_o,
  input   [1:0] dsize_i,
  input          drd_i,
  input          dwr_i
);
```

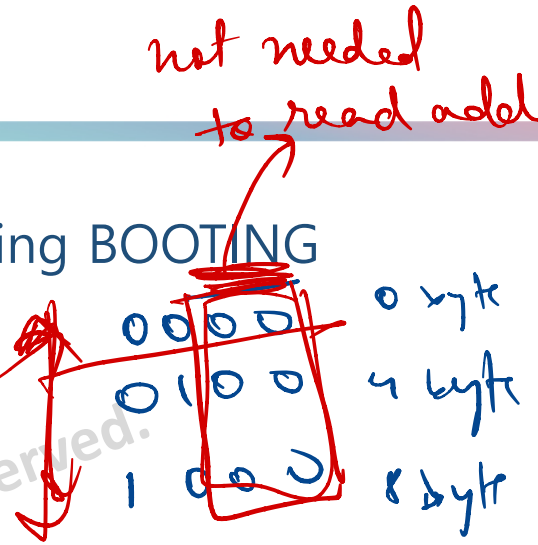
Memory model (riscv_memory.v)

- Local parameters
  - DEPTH: The number of words in the memory
  - Size mode:
    - BYTE: 8 bits
    - HALF: 16 bits
    - WORD: 32 bits
- Memory initialization
  - Memory cell: mem_r
  - During initialization, load the content from the FIRMWARE file into mem_r



```
//------------------------------
localparam
  SIZE_BYTE = 2'd0,
  SIZE_HALF = 2'd1,
  SIZE_WORD = 2'd2;

localparam
  DEPTH = $clog2(SIZE);

//--------------------------------
// Memory Initialization
//--------------------------------
reg [31:0] mem_r [0:SIZE/4-1];

initial begin
  $readmemh(FIRMWARE, mem_r);
end
```

Read operations for instructions



- READ ONLY for instruction because the instructions are initialized during BOOTING
  - **irdata_r ← mem_r[iaddr_i[DEPTH:2]]**

⇒ Why do we ignore the two least significant bits (LSBs)?

```
module riscv_memory #(
  parameter      SIZE     = 8192,
  parameter      FIRMWARE = "mem.hex"
)(
  input          clk_i,
  input          reset_i,

  input  [31:0] iaddr_i,
  output [31:0] irdata_o,
  input          ird_i,

  input  [31:0] daddr_i,
  input  [31:0] dwdata_i,
  output [31:0] drdata_o,
  input   [1:0] dsize_i,
  input          drd_i,
  input          dwr_i
);
```

```
//------------------------------------------
// Instruction memory: READ ONLY
//------------------------------------------
always @(posedge clk_i) begin
  if (~reset_i)
    irdata_r <= 32'h0;
  else begin
    // Insert your code
    //if (ird_i)
    //  irdata_r <= /*Insert your code */
  end
end
// Output ports for Instruction
assign irdata_o = irdata_r;
```

Read operations for data

- Read operation
  - **drdata_r ← mem_r[daddr_i[DEPTH:2]]**

```
module riscv_memory #(
  parameter      SIZE     = 8192,
  parameter      FIRMWARE = "mem.hex"
)(
  input          clk_i,
  input          reset_i,

  input  [31:0] iaddr_i,
  output [31:0] irdata_o,
  input          ird_i,

  input  [31:0] daddr_i,
  input  [31:0] dwdata_i,
  output [31:0] drdata_o,
  input   [1:0] dsize_i,
  input          drd_i,
  input          dwr_i
);
```

```
wire [7:0] rdata_byte_w =           //*** extract valid size from 32bit
    (2'b00 == daddr_r) ? drdata_r[7:0] :    //    If Byte, daddr_i LSB
    (2'b01 == daddr_r) ? drdata_r[15:8] :   //       [ #3 (1 Byte) | #2
    (2'b10 == daddr_r) ? drdata_r[23:16] :      //       |<------------
    drdata_r[31:24];                    //         2'b11       2'b10     2'



wire [15:0] rdata_half_w =                  //    If Half word,
    daddr_r[1] ? drdata_r[31:16] : drdata_r[15:0];      //    daddr_i[1
]                                   //    [      #1 (1 Half Word)      |
                                    //    |<----------------- 32bits-----------
                                    //                l'b1              l'b0

]//-----------------------------------------------------------------
// Data memory: Read out
// Read data size rescale to fit 32bits module output
//-----------------------------------------------------------------
assign drdata_o =
    (SIZE_BYTE == dsize_r) ? { 24'b0, rdata_byte_w } :
    (SIZE_HALF == dsize_r) ? { 16'b0, rdata_half_w } : drdata_r;
//---------------------------------------
// Read operation
//---------------------------------------

always @(posedge clk_i) begin
  if (~reset_i)
      drdata_r <= 32'h0;
  else begin
    // Insert your code
  end
end
```

Write operations for data

- Copy data from the input port (dwdata_i) to memory
  - **mem_r[daddr_i[DEPTH:2]] ← dwdata_i**
  - Support three modes: BYTE, HALF, and WORD

```
wire [31:0] dwdata_w =                          //*** dwdata_w[31:0] reflects write data with repetition.
    (SIZE_BYTE == dsize_i) ? {4{dwdata_i[7:0]}} :       //    If Byte write, x4 repeat
    (SIZE_HALF == dsize_i) ? {2{dwdata_i[15:0]}} : dwdata_i;    //    If Half Word, x2 repeat
```

```
module riscv_memory #(
  parameter      SIZE     = 8192,
  parameter      FIRMWARE = "mem.hex"
)(
  input          clk_i,
  input          reset_i,

  input  [31:0] iaddr_i,
  output [31:0] irdata_o,
  input          ird_i,

  input  [31:0] daddr_i,
  input  [31:0] dwdata_i,
  output [31:0] drdata_o,
  input   [1:0] dsize_i,
  input          drd_i,
  input          dwr_i
);
```

```
//------------------------------------------------------------------------
// Write operation
//------------------------------------------------------------------------
always @(posedge clk_i) begin
  if (dbe_w[0] && dwr_i)                //*** According to write data size (dsize_i),  marks dbe_w[3:0]
                                //    to select a segmented memory address to overwrite
    mem_r[daddr_i[DEPTH:2]][7:0] <= dwdata_w[7:0];  //    For BYTE mode, select 1 of 4 slots among 32bits
  if (dbe_w[1] && dwr_i)                //    4'b0001  : 4'b0010 : 4'b0100 : 4'b1000
    mem_r[daddr_i[DEPTH:2]][15:8] <= dwdata_w[15:8];    //     #3     #2       #1       slot#0

  // Insert your code
  //{{{
  //if (dbe_w[2] && dwr_i)              //    For HALFWORD mode, select 1 of 2 slots among 32bits
  //  /*Insert your code */  <= dwdata_w[23:16];    //    4'b0011             : 4'b1100
                                //    lower 16 bits overwrite  higher 16 bits overwrite
  //if (dbe_w[3] && dwr_i)              //    For WORD mode
  //  /*Insert your code */  <= dwdata_w[31:24];    //    full 32bits overwrite
  //}}}
end
```

To do …

- Implement riscv_memory.v by completing some missing codes
  - READ operation for instructions
  - READ and WRITE operations for data
- Do a simulation with time = 300ns
- Show the waveform

```
//------------------------------------------
// Instruction memory: READ ONLY
//------------------------------------------
always @(posedge clk_i) begin
  if (~reset_i)
    irdata_r <= 32'h0;
  else begin
    // Insert your code
    //if (ird_i)
    //  irdata_r <= /*Insert your code */
  end
end
// Output ports for Instruction
assign irdata_o = irdata_r;
```

```
//------------------------------------------------------
// Data memory: READ/WRITE
//------------------------------------------------------
// Read operation
//------------------------------------------------------

always @(posedge clk_i) begin
  if (~reset_i)
      drdata_r <= 32'h0;
  else begin
    // Insert your code
  end
end

//------------------------------------------------------
// Write operation
//------------------------------------------------------
always @(posedge clk_i) begin
  if (dbe_w[0] && dwr_i)                //*** According
                                //    to select a segmented
    mem_r[daddr_i[DEPTH:2]][7:0] <= dwdata_w[7:0];  //
  if (dbe_w[1] && dwr_i)                //    4'b0001  :
    mem_r[daddr_i[DEPTH:2]][15:8] <= dwdata_w[15:8];

 // Insert your code
 //{{{
 //if (dbe_w[2] && dwr_i)               //    For HALFWO
 //  /*Insert your code */  <= dwdata_w[23:16];    //
                                //    lower 16 bits overwrit
 //if (dbe_w[3] && dwr_i)               //    For WORD m
 //  /*Insert your code */  <= dwdata_w[31:24];    //
 //}}}
end
```

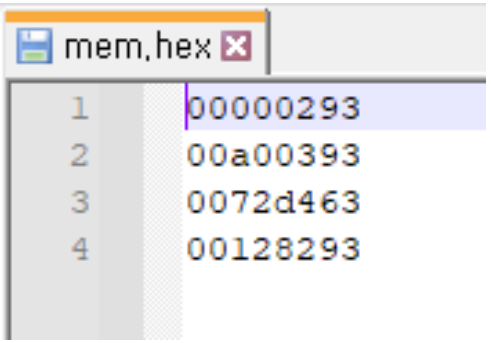
Test bench (riscv_memory_tb.v)

```
module riscv_memory_tb;
        reg reset_i;
        reg clk_i;

        // Input instruction
        reg  [31:0] iaddr_i;
        reg         ird_i;
        reg [31:0] daddr_i;
        reg [31:0] dwdata_i;
        reg [1:0]   dsize_i;
        reg         drd_i;
        reg         dwr_i;
        // Outputs
        wire [31:0] irdata_o;
        wire [31:0] drdata_o;
```

Signals

```
riscv_memory u_memory (
                . iaddr_i (iaddr_i),
                . ird_i(ird_i),
                . daddr_i(daddr_i),
                . dwdata_i (dwdata_i),
                . dsize_i (dsize_i),
                . drd_i (drd_i),
                . dwr_i (dwr_i),
                . irdata_o(irdata_o),
                . drdata_o(drdata_o),
);
```

A memory instance



mem.hex

Test bench: Instruction memory

- Test cases
  - Clock 100MHz (p=10)
  - Access mode = SIZE_WORD
    - Access a 32-bit word
  - Request addresses
    - 0 → 4 → 8 → 12

```
wire [31:0] if_opcode_w;
assign if_opcode_w = irdata_o;

localparam
  SIZE_BYTE = 2'd0,
  SIZE_HALF = 2'd1,
  SIZE_WORD = 2'd2;

parameter p=10;
```

```
initial begin
                clk_i = 1'b0;
                forever #(p/2) clk_i = !clk_i;
end
```

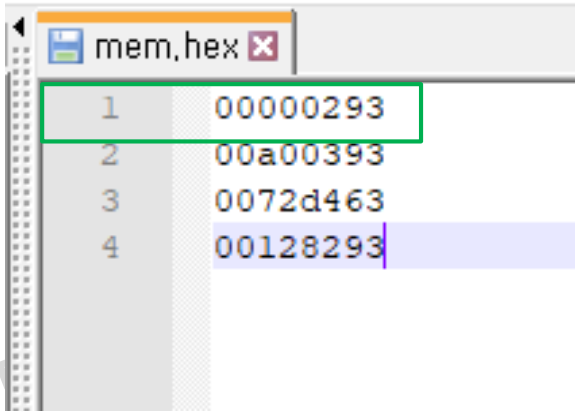
```
initial begin
                reset_i = 1'b0;
                iaddr_i = 0;
                ird_i = 0;
                dsize_i=SIZE_WORD;
                #(4*p) reset_i = 1'b1;
```

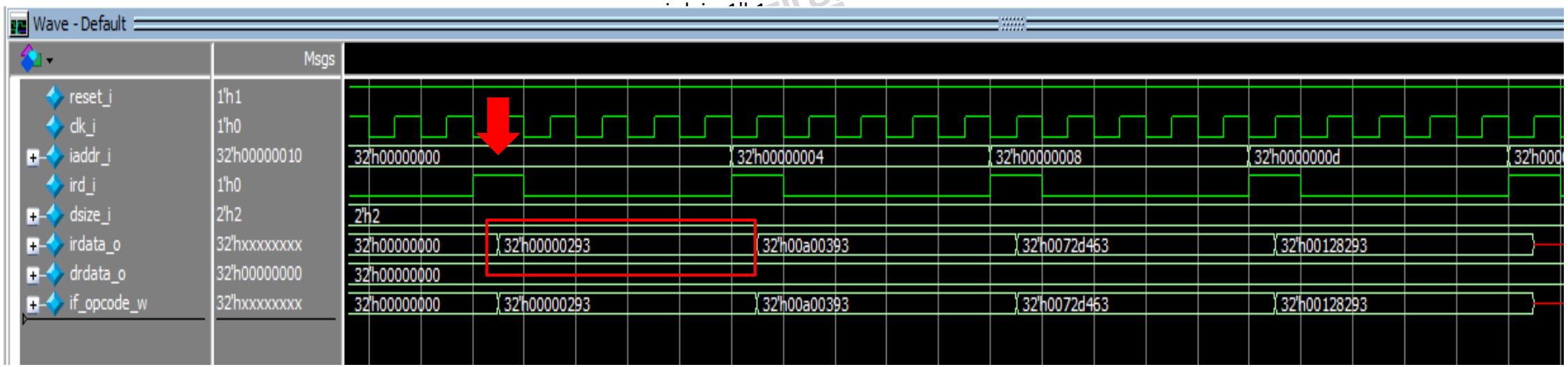
```
#(4*p)          iaddr_i = 32'h0;
                ird_i = 1'b1;
#(p)            ird_i = 1'b0;
#(4*p)          iaddr_i = 32'h4;
                ird_i = 1'b1;
#(p)            ird_i = 1'b0;
#(4*p)          iaddr_i = 32'h8;
                ird_i = 1'b1;
#(p)            ird_i = 1'b0;
#(4*p)          iaddr_i = 32'hC;
                ird_i = 1'b1;
#(p)            ird_i = 1'b0;
#(4*p)          iaddr_i = 32'h10;
                ird_i = 1'b1;
#(p)            ird_i = 1'b0;

end
endmodule
```

Waveform

- Request address = 0
- Return instruction: 0000_0293

```
#(4*p)      iaddr_i = 32'h0;
            ird_i = 1'b1;
#(p)        ird_i = 1'b0;
#(4*p)      iaddr_i = 32'h4;
            ird_i = 1'b1;
#(p)        ird_i = 1'b0;
#(4*p)      iaddr_i = 32'h8;
            ird_i = 1'b1;
#(p)        ird_i = 1'b0;
#(4*p)      iaddr_i = 32'hC;
```

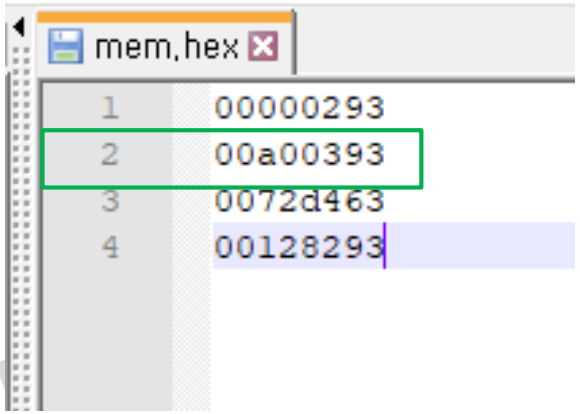


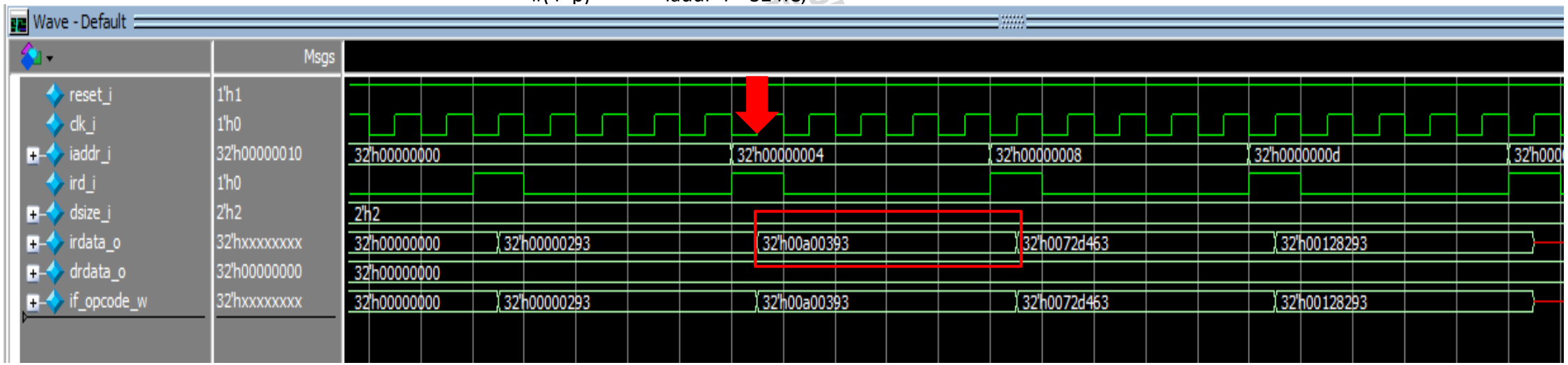


Waveform

- Request address = 4
- Return instruction: 00a0_0393

```
#(4*p)      iaddr_i = 32'h0;
            ird_i = 1'b1;
#(p)        ird_i = 1'b0;
#(4*p)      iaddr_i = 32'h4;
            ird_i = 1'b1;
#(p)        ird_i = 1'b0;
#(4*p)      iaddr_i = 32'h8;
            ird_i = 1'b1;
#(p)        ird_i = 1'b0;
#(4*p)      iaddr  i = 32'hC;
```

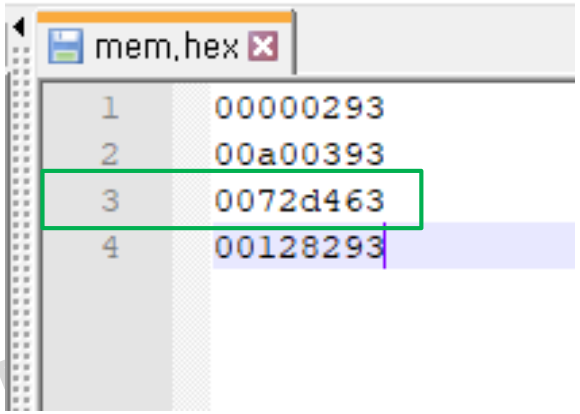


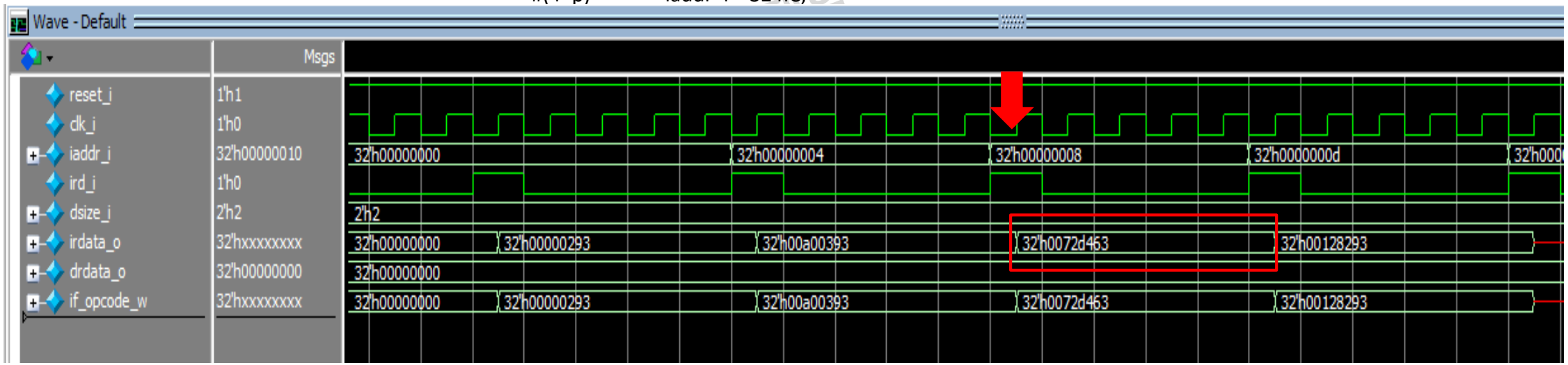


Waveform

- Request address = 8
- Return instruction: 0072_d463

```
#(4*p)      iaddr_i = 32'h0;
            ird_i = 1'b1;
#(p)        ird_i = 1'b0;
#(4*p)      iaddr_i = 32'h4;
            ird_i = 1'b1;
#(p)        ird_i = 1'b0;
#(4*p)      iaddr_i = 32'h8;
            ird_i = 1'b1;
#(p)        ird_i = 1'b0;
#(4*p)      iaddr_i = 32'hC;
```

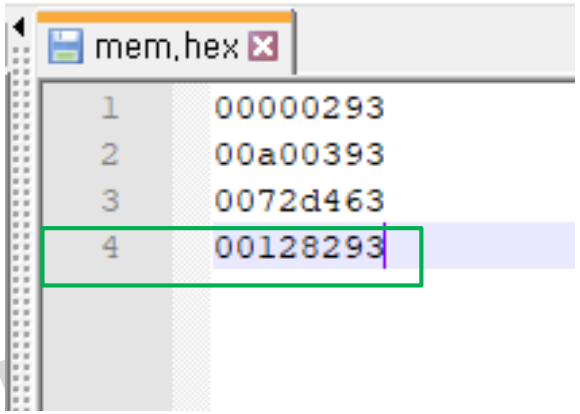


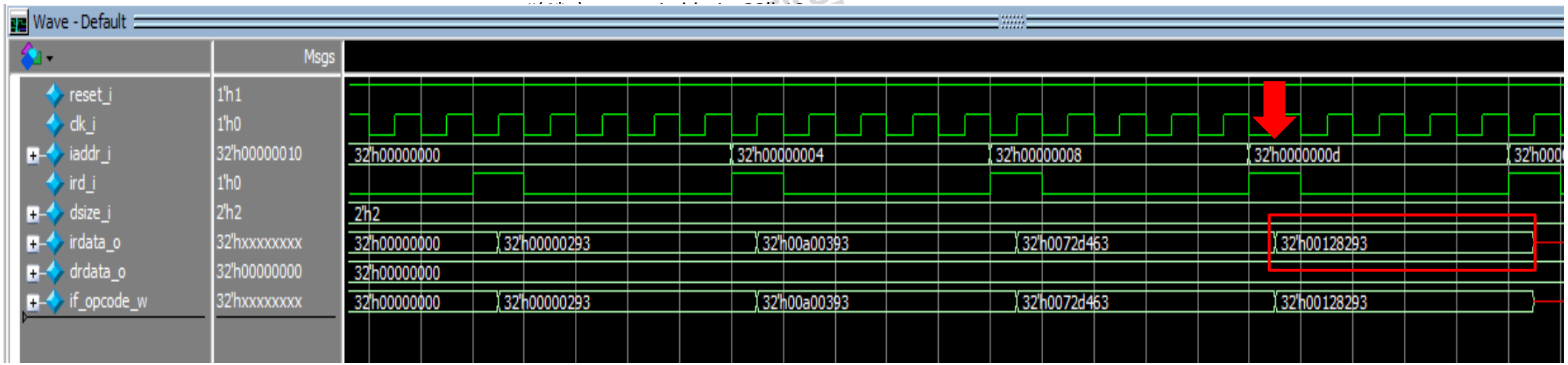


Waveform

- Request address = 12
- Return instruction: 0012_8293

```
#(4*p)      iaddr_i = 32'h0;
            ird_i = 1'b1;
#(p)        ird_i = 1'b0;
#(4*p)      iaddr_i = 32'h4;
            ird_i = 1'b1;
#(p)        ird_i = 1'b0;
#(4*p)      iaddr_i = 32'h8;
            ird_i = 1'b1;
#(p)        ird_i = 1'b0;
#(4*p)      iaddr_i = 32'hC;
            ird_i = 1'b1;
#(p)        ird_i = 1'b0;
```





Road map

RISC-V Simulator

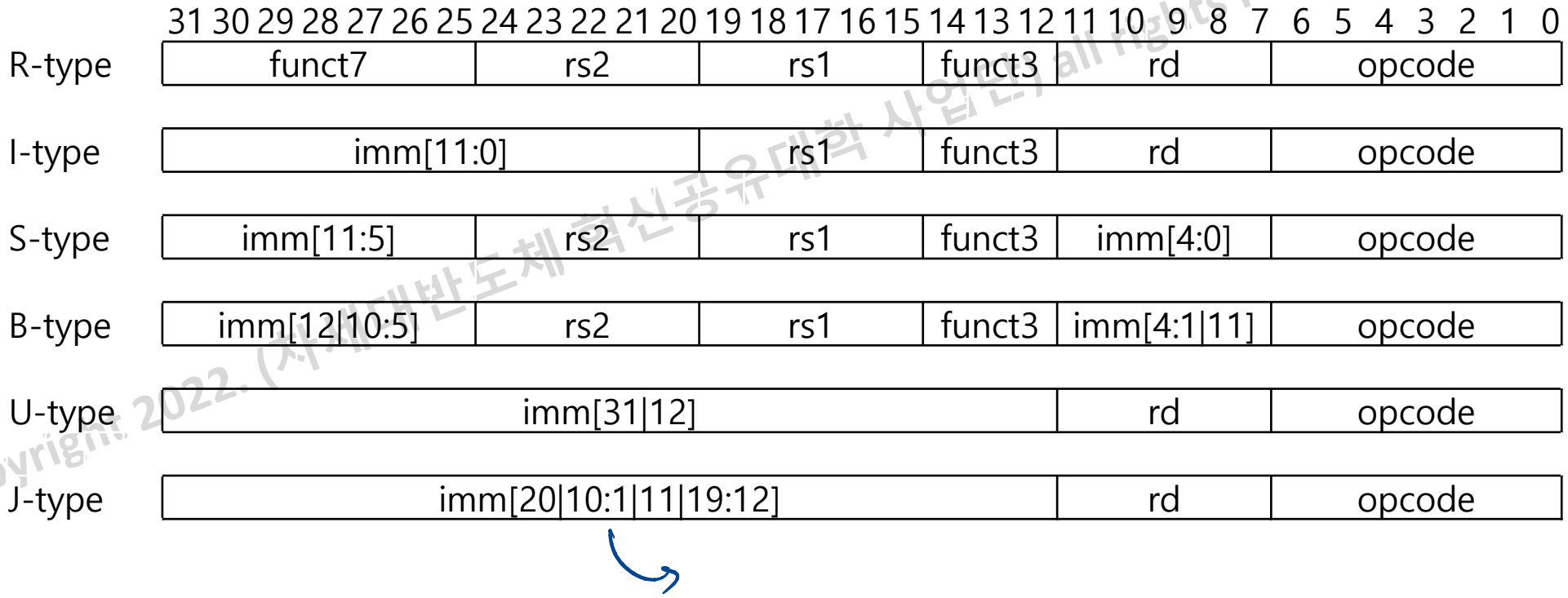
Memory
Instruction memory

RISC-V Instruction format
Decoder

Simple core

RISC-V RV32I Instruction Formats

- 32-bit instruction: Aligned on a four-byte boundary in memory.
- Formats: R, I, S, B, U, J-type
- See RV32I Base Instruction Set



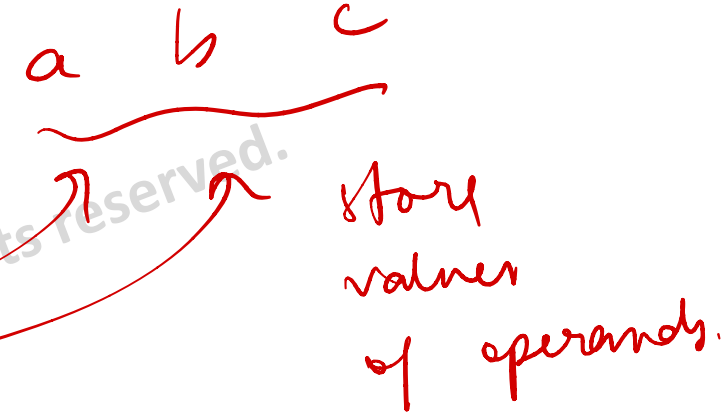
R-type: Register-Register instruction

- R-type (Register-Register Instruction)
  - rs1 and rs2 are the source registers, rd the destination
  - Function types (funct3)
    - ADD/SUB
    - SLT, SLTU: set less than
    - SRL, SLL, SRA: shift logical or arithmetic left or right

	31 30 29 28 27 26 25	24 23 22 21 20	19 18 17 16 15	14 13 12	11 10 9 8 7	6 5 4 3 2 1 0	
R-type	funct7	rs2	rs1	funct3	rd	opcode	
	0000000	rs2	rs1	000	rd	0110011	ADD
	0100000	rs2	rs1	000	rd	0110011	SUB
	0000000	rs2	rs1	001	rd	0110011	SLL
	0000000	rs2	rs1	101	rd	0110011	SRL
	0100000	rs2	rs1	101	rd	0110011	SRA

R-type: Register-Register instructions

- Opcode = 0110011: Define R-type instructions
- funct3, funct7: ALU operations
  - ADD: funct3 == 000 && funct7 == 0000000
  - SUB: funct3 == 000 && funct7 == 0100000
  - SRL: funct3 == 101 && funct7 == 0000000
  - SRA: funct3 == 101 && funct7 == 0100000



	31 30 29 28 27 26 25	24 23 22 21 20	19 18 17 16 15	14 13 12	11 10 9 8 7	6 5 4 3 2 1 0	
R-type	funct7	rs2	rs1	funct3	rd	opcode	
	0000000	rs2	rs1	000	rd	0110011	ADD
	0100000	rs2	rs1	000	rd	0110011	SUB
	0000000	rs2	rs1	001	rd	0110011	SLL
	0000000	rs2	rs1	101	rd	0110011	SRL
	0100000	rs2	rs1	101	rd	0110011	SRA

I-type: Immediate-Register Instructions

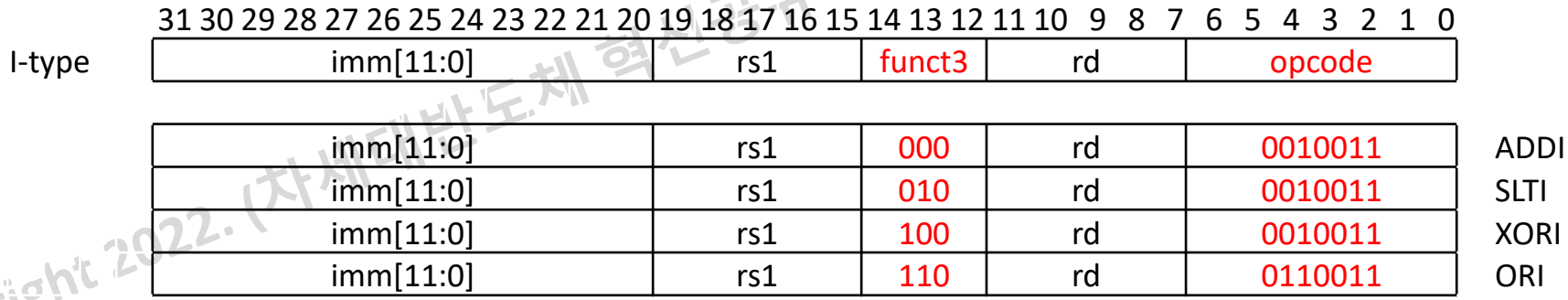
- I-type (Immediate)
  - All immediates in all instructions are sign extended
    - ADDI: adds sign extended 12-bit immediate to rs1
    - SLTI(U): set less than immediate
    - ANDI/ORI/XORI: Logical operations
    - SLLI/SRLI/SRAI: Shifts by constants
- I-type instructions end with I

	31 30 29 28 27 26 25 24 23 22 21 20	19 18 17 16 15	14 13 12	11 10 9 8 7	6 5 4 3 2 1 0	
I-type	imm[11:0]	rs1	funct3	rd	opcode	
	imm[11:0]	rs1	000	rd	0010011	ADDI
	imm[11:0]	rs1	010	rd	0010011	SLTI
	imm[11:0]	rs1	100	rd	0010011	XORI
	imm[11:0]	rs1	110	rd	0110011	ORI

I-type: Immediate-Register Instructions

- Opcode = 0010011: Define I-type instructions
- funct3: ALU operations
  - ADDI: funct3 == 000
  - SLTI: funct3 == 010
  - XORI: funct3 == 100
  - ORI: funct3 == 110

addi x9 x0 64



	31 30 29 28 27 26 25 24 23 22 21 20	19 18 17 16 15	14 13 12	11 10 9 8 7	6 5 4 3 2 1 0	
I-type	imm[11:0]	rs1	funct3	rd	opcode	
	imm[11:0]	rs1	000	rd	0010011	ADDI
	imm[11:0]	rs1	010	rd	0010011	SLTI
	imm[11:0]	rs1	100	rd	0010011	XORI
	imm[11:0]	rs1	110	rd	0110011	ORI

B-type: Control Instruction

- Assuming translations below, compile the "*if*" block

  f → x10 g → x11 h → x12

  i → x13 j → x14

```
if (i == j)                bne x13,x14,Else
   f = g + h;              add x10,x11,x12
else                       j Exit
   f = g – h;        Else: sub x10,x11,x12
                     Exit:
```

B-type: Conditional Transfer Instructions

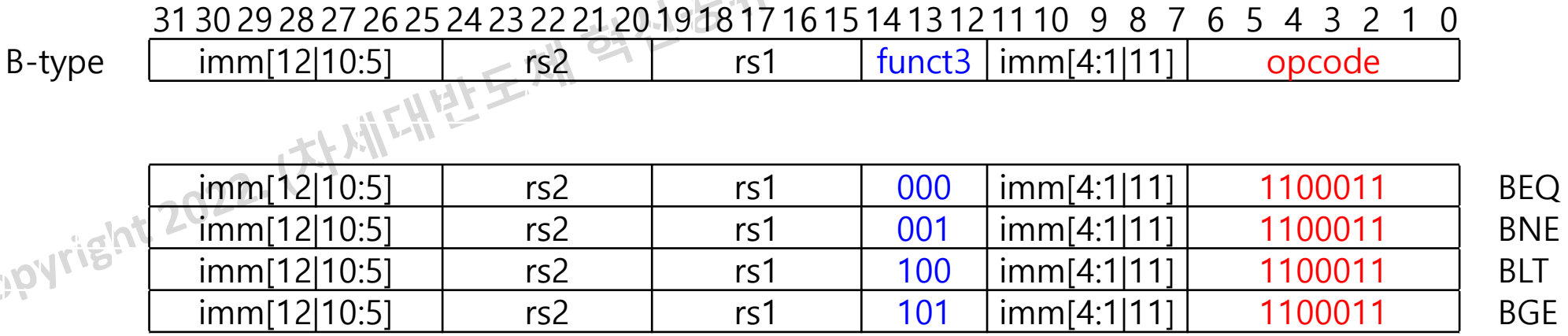
- Conditional Branches: S/B-type and PC+offset target
  - Branches, compare two registers, PC+(immediate<<1) target
    - Signed offset in multiples of two.
  - Branches do not have delay slot

⇒ NO architecturally visible delay slots

	31 30 29 28 27 26 25	24 23 22 21 20	19 18 17 16 15	14 13 12	11 10 9 8 7	6 5 4 3 2 1 0	
B-type	imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	
	imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
	imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
	imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
	imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE

B-type: Conditional Transfer Instructions

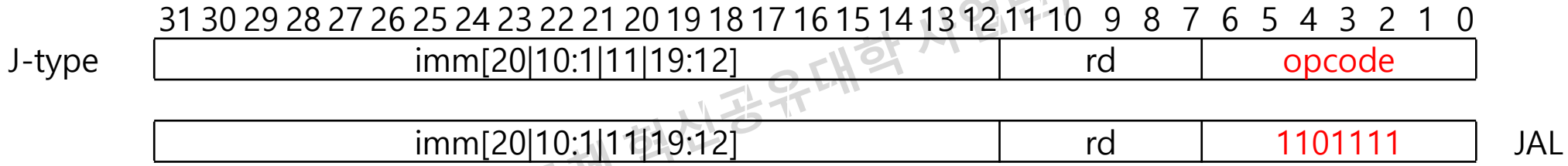
- Opcode = 1100011: Define B-type instructions
- funct3: branch operations
  - Branch if = zero (BEQ): funct3 == 000
  - Branch if ≠ zero (BNE): funct3 == 001
  - Branch if < zero (BLT): funct3 == 100
  - Branch if ≥ zero (BGE): funct3 == 101

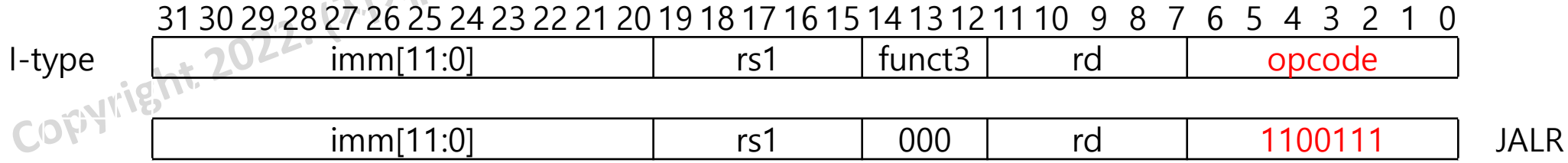


	31 30 29 28 27 26 25	24 23 22 21 20	19 18 17 16 15	14 13 12	11 10 9 8 7	6 5 4 3 2 1 0	
B-type	imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	
	imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
	imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
	imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
	imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE

J-type: Jump and Link instructions

- Unconditional Jumps: *Program counter + offset target*
  - JAL: Jump and link, also writes PC+4 to x1, J-type
    - Offset scaled by 1-bit left shift
    - Can jump to 16-bit instruction boundary (Same for branches)
  - JALR: Jump and link register where Imm (12 bits) + rd1 = target





I-Type: Load instruction

- Load (I-type): Read data from Memory and save it into Register
  - reg[rd] ← MEM(reg[rs1] + imm)
  - Opcode = 0000011, Load byte (LB), load half (LH), load word (LW)
    - Load Byte: funct3 == 000
    - Load Half: funct3 == 001
    - Load Word: funct3 == 010
    - Load Byte (unsigned): funct3 == 100
    - Load Half (unsigned): funct3 == 101

	31 30 29 28 27 26 25 24 23 22 21 20	19 18 17 16 15	14 13 12	11 10 9 8 7	6 5 4 3 2 1 0	
I-type	imm[11:0]	rs1	funct3	rd	opcode	
	imm[11:0]	rs1	000	rd	0000011	LB
	imm[11:0]	rs1	001	rd	0000011	LH
	imm[11:0]	rs1	010	rd	0000011	LW
	imm[11:0]	rs1	100	rd	0000011	LBU
	imm[11:0]	rs1	101	rd	0000011	LHU

S-Type: Store instruction

- Store (S-type)
  - Read data from Register File and save it into Memory
    - MEM(reg[rs1] + imm) ← reg[rs2]
  - Opcode = 0100011
  - Store byte (SB), store half (SH), store word (SW)
    - Store Byte: funct3 == 000
    - Store Haft: funct3 == 001
    - Store Word: funct3 == 010

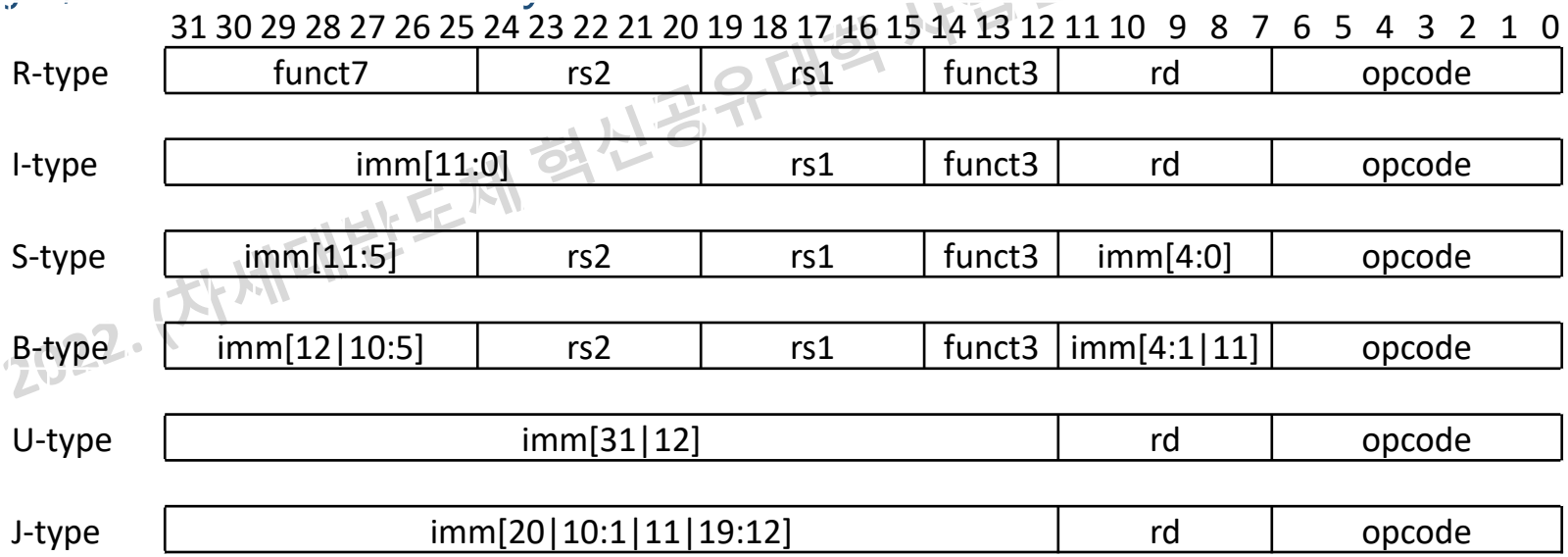
S-type	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	
	imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
	imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
	imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW

Lab 2: RISC-V decoder

- Lab 2: RISC-V decoder
  - Implement a decoder for RISC-V
  - Run simulation
  - Show the output result

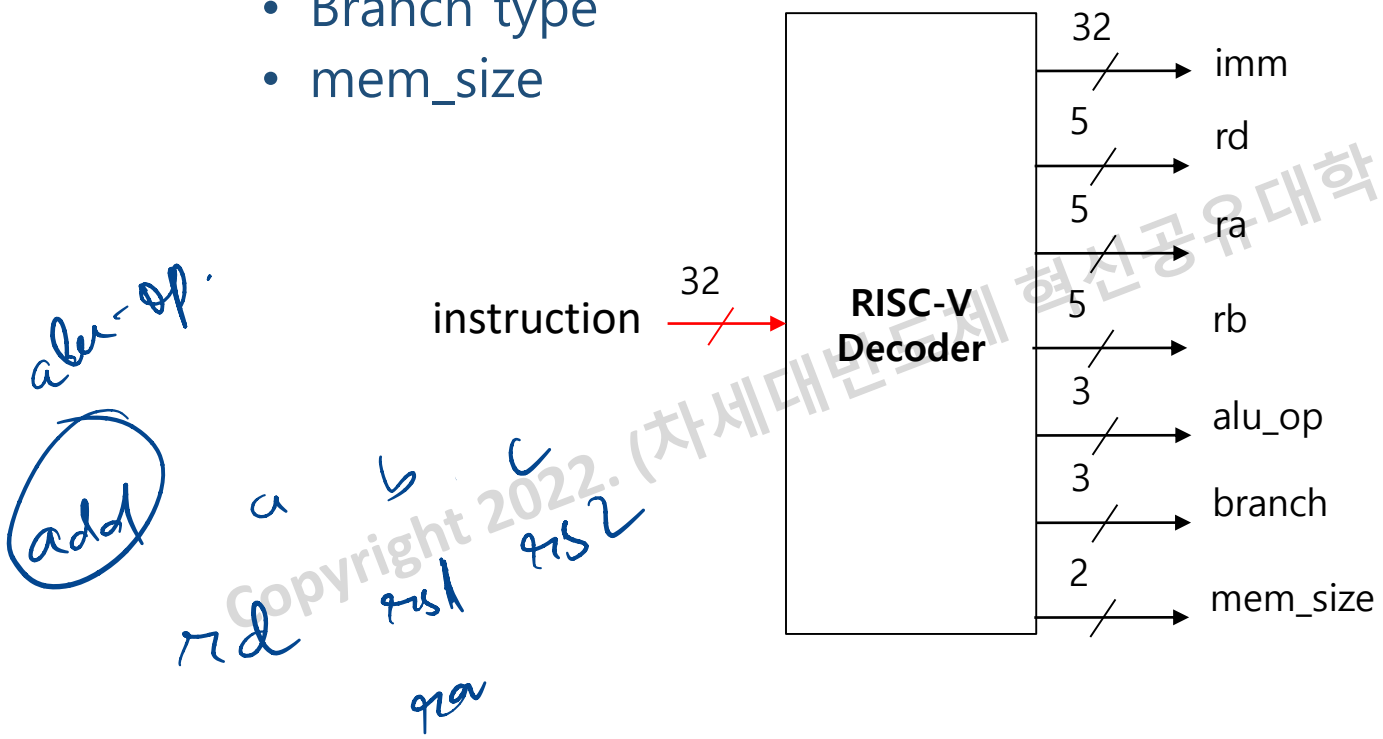
Instruction decode

- Gather data from the fields (decode all necessary instruction data)
  1. read the opcode to determine instruction type and field lengths
  2. read in data from all necessary registers •
     - for add, read two registers
     - for addi, read one register
     - for jal, no reads necessary



Decoder (riscv_decoder.v)

- Input: An instruction coming from Instruction memory
- Outputs:
  - Addresses: ra, rb, rd → register file
  - Operation for ALU: alu_op
  - Branch type
  - mem_size

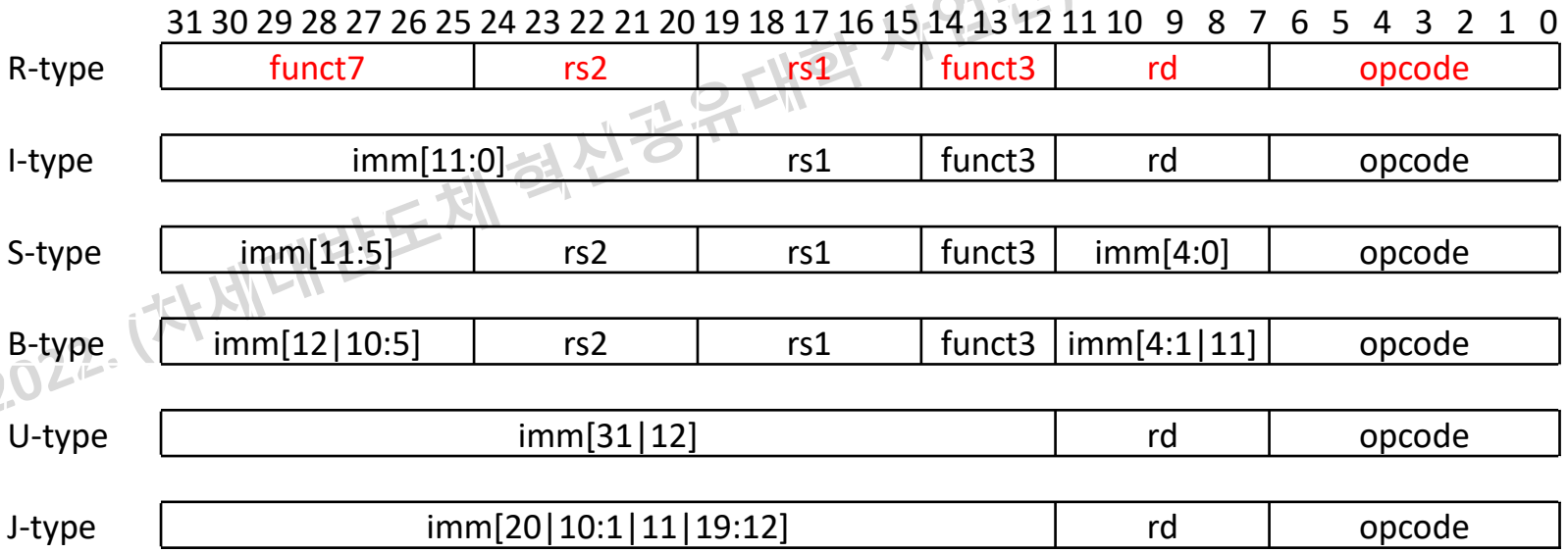



```
module riscv_decoder
(
    // Input instruction
    input  [31:0] if_opcode_w,
    // Outputs
    output [31:0] id_imm_w,
    output [4:0] id_rd_index_w,
    output [4:0] id_ra_index_w,
    output [4:0] id_rb_index_w,
    output [3:0] id_alu_op_w,
    output [2:0] id_branch_w,
    output [1:0] id_mem_size_w,
    //Flags
    output mulh_w,
    output mulhsu_w,
    output div_w,
    output rem_w,
    output sra_w,
    output srai_w,
    output alu_imm_w,
    output jal_w,
    output load_w,
    output store_w,
    output lbu_w,
    output lhu_w,
    output jalr_w,
    output id_illegal_w
);
```

Opcode parser

- Step 1: Decoder extracts alignment regions
  - Opcode = instruction[6:0]
  - Funct7 = instruction[31:25]

```
assign op_w = if_opcode_w[6:0];
...
assign f7_w = if_opcode_w[31:25];
```



Opcode parser

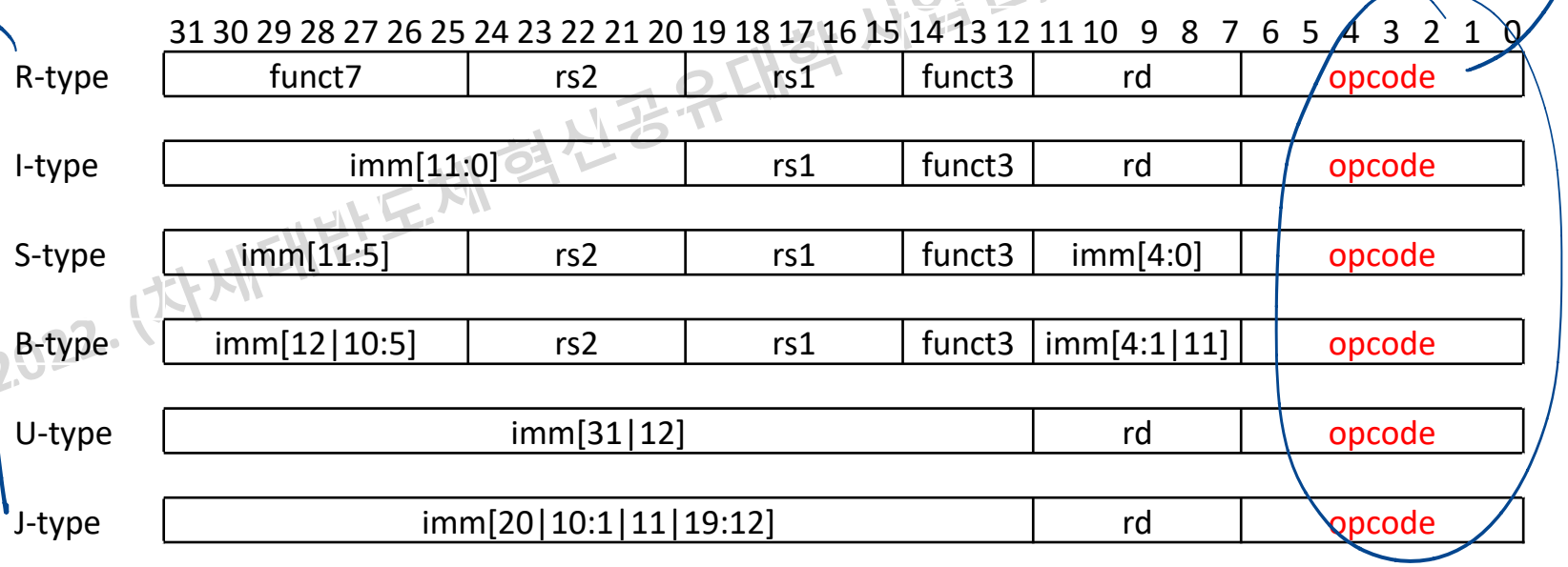
- Step 2: Decoder extracts instruction types **based on Opcode**
  - Branch instructions: 7'b1100011 == op_w
  - ALU instructions: 7'b0110011 == op_w

```
assign op_branch_w  = (7'b1100011 == op_w);
...
assign op_alu_reg_w = (7'b0110011 == op_w);
```

all have diff type opcode

0110011

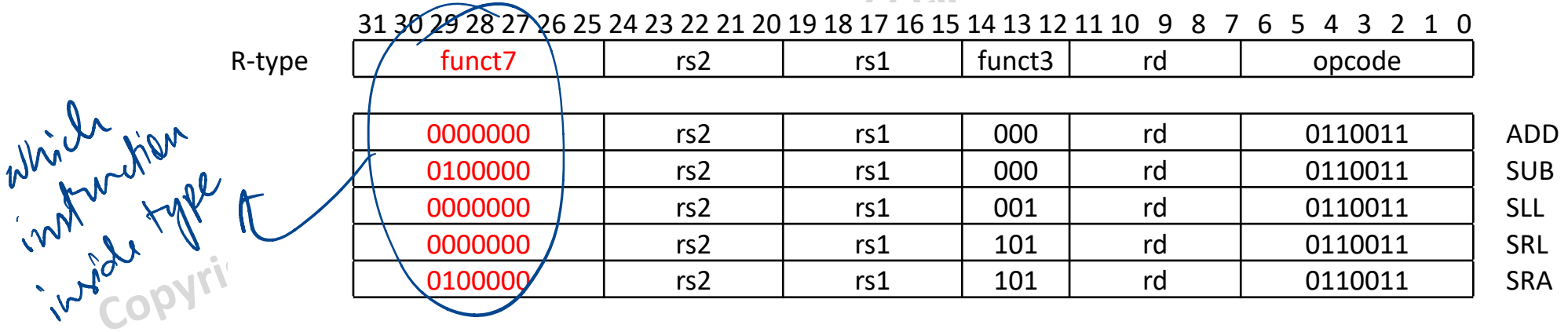
which type



Opcode parser

- Step 3: Decoder extracts instruction types **based on funct7**
  - Main, i.e. ADD
  - Alternative
  - Multiplier

```
assign op_f7_main_w = (7'b0000000 == f7_w);
assign op_f7_alt_w  = (7'b0100000 == f7_w);
assign op_f7_mul_w  = (7'b0000001 == f7_w);
```



	31 30 29 28 27 26 25	24 23 22 21 20	19 18 17 16 15	14 13 12	11 10 9 8 7	6 5 4 3 2 1 0	
R-type	funct7	rs2	rs1	funct3	rd	opcode	
	0000000	rs2	rs1	000	rd	0110011	ADD
	0100000	rs2	rs1	000	rd	0110011	SUB
	0000000	rs2	rs1	001	rd	0110011	SLL
	0000000	rs2	rs1	101	rd	0110011	SRL
	0100000	rs2	rs1	101	rd	0110011	SRA

Opcode parser

- Step 4: Decoder extracts instruction types based on instruction types (opcode) and functionality (funct3, funct7).

```
assign lui_w    = (7'b0110111 == op_w);          flags assign from opcode register field [6:0] and f3 combination
...
assign jalr_w   = (7'b1100111 == op_w) && (3'b000 == f3_w);
```

```
assign beq_w    = op_branch_w  && (3'b000 == f3_w);          branch flags assign from f3 field and branch flag
...
assign bgeu_w   = op_branch_w  && (3'b111 == f3_w);
```

```
assign lb_w     = op_load_w    && (3'b000 == f3_w);          load flags assign from f3 field and load flag
...
assign lhu_w    = op_load_w    && (3'b101 == f3_w);
```

```
assign sb_w     = op_store_w   && (3'b000 == f3_w);          store flags assign from f3 field and store flag
assign sh_w     = op_store_w   && (3'b001 == f3_w);
assign sw_w     = op_store_w   && (3'b010 == f3_w);
```

```
assign addi_w   = op_alu_imm_w && (3'b000 == f3_w);          ALU flags assign from f3 field and alu flag
...
assign remu_w   = op_alu_reg_w && (3'b111 == f3_w) && op_f7_mul_w;
```

Test bench (riscv_decoder_tb.v)

- Test bench module
  - Internal signals
  - A decoder instance

```
module riscv_decoder_tb;
reg rst;
reg clk;
reg  [31:0] if_opcode_w;
wire [31:0] id_imm_w;
wire [4:0] id_rd_index_w;
wire [4:0] id_ra_index_w;
wire [4:0] id_rb_index_w;
wire [3:0] id_alu_op_w;
wire [2:0] id_branch_w;
wire [1:0] id_mem_size_w;
wire mulh_w;
wire mulhsu_w;
wire div_w;
wire rem_w;
wire sra_w;
wire srai_w;
wire alu_imm_w;
wire jal_w;
wire load_w;
wire store_w;
wire lbu_w;
wire lhu_w;
wire jalr_w;
wire id_illegal_w;
```

```
riscv_decoder u_decoder
(
                .if_opcode_w(if_opcode_w),
                .id_imm_w(id_imm_w),
                .id_rd_index_w(id_rd_index_w),
                .id_ra_index_w(id_ra_index_w),
                .id_rb_index_w(id_rb_index_w),
                .id_alu_op_w(id_alu_op_w),
                .id_branch_w(id_branch_w),
                .id_mem_size_w(id_mem_size_w),
                .mulh_w(mulh_w),
                .mulhsu_w(mulhsu_w),
                .div_w(div_w),
                .rem_w(rem_w),
                .sra_w(sra_w),
                .srai_w(srai_w),
                .alu_imm_w(alu_imm_w),
                .jal_w(jal_w),
                .load_w(load_w),
                .store_w(store_w),
                .lbu_w(lbu_w),
                .lhu_w(lhu_w),
                .jalr_w(jalr_w),
                .id_illegal_w(id_illegal_w)
);
```

Test bench

- Clock 100MHz
- Test four instructions

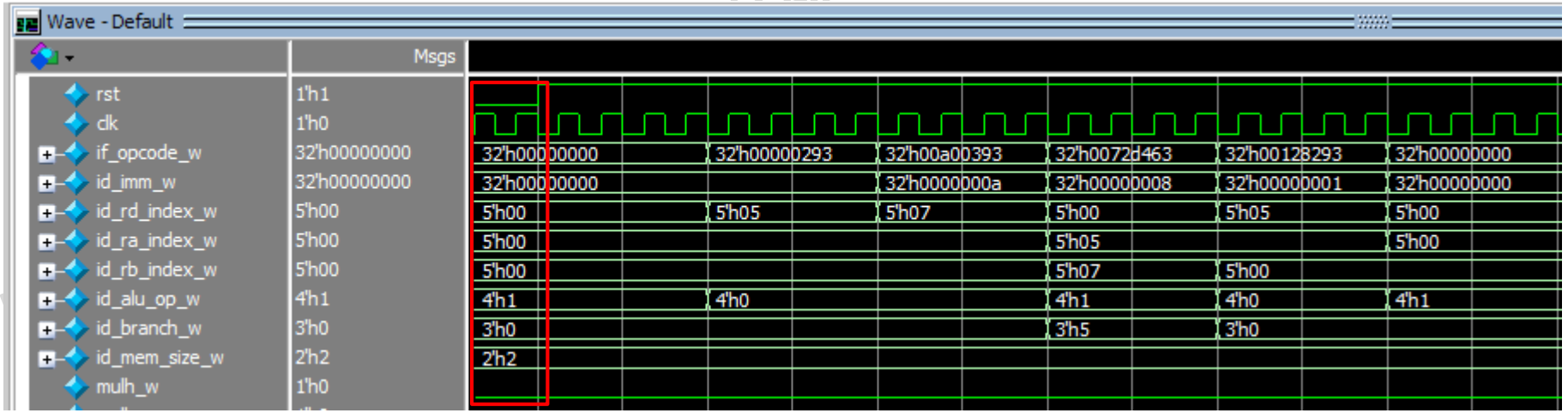
```
initial begin
clk_i = 0;
forever #5 clk_i = ~clk_i;
end
initial begin
                clk_i=1;
                rst_i = 0;
                #20             rst_i = 1;
```

```
                rst = 1'b0;                     // negedge reset on
                if_opcode_w = 0;
                #(4*p) rst = 1'b1;              // negedge reset off

                #(4*p) if_opcode_w = 32'h00000293;
                #(4*p) if_opcode_w = 32'h00a00393;
                #(4*p) if_opcode_w = 32'h0072d463;
                #(4*p) if_opcode_w = 32'h00128293;
                #(4*p) if_opcode_w = 32'h0;
end
endmodule
```

Waveform

- Reset

```
rst = 1'b0;                    // negedge reset on
if_opcode_w = 0;

#(4*p) rst = 1'b1;             // negedge reset off

#(4*p) if_opcode_w = 32'h00000293;
#(4*p) if_opcode_w = 32'h00a00393;
#(4*p) if_opcode_w = 32'h0072d463;
#(4*p) if_opcode_w = 32'h00128293;
#(4*p) if_opcode_w = 32'h0;
```

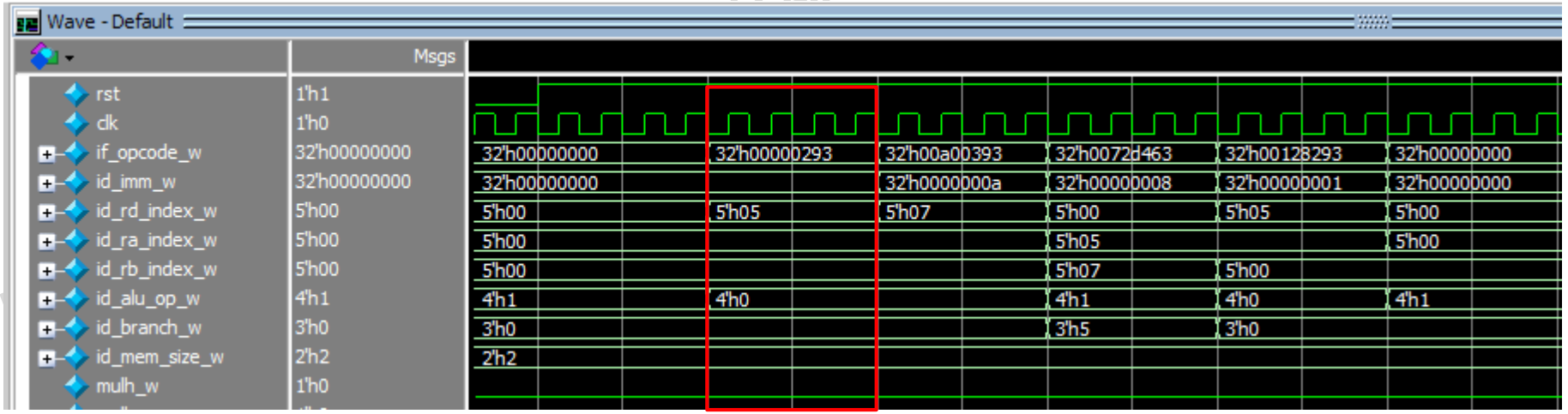


Waveform

- If_opcode_w =32'h0000_0293

```
// Opcoder parser
assign op_w = if_opcode_w[6:0];
assign rd_w = if_opcode_w[11:7];
assign f3_w = if_opcode_w[14:12];
assign ra_w = if_opcode_w[19:15];
assign rb_w = if_opcode_w[24:20];
assign f7_w = if_opcode_w[31:25];
```

```
rst = 1'b0;                    // negedge reset on
if_opcode_w = 0;

#(4*p) rst = 1'b1;             // negedge reset off

#(4*p) if_opcode_w = 32'h00000293;
#(4*p) if_opcode_w = 32'h00a00393;
#(4*p) if_opcode_w = 32'h0072d463;
#(4*p) if_opcode_w = 32'h00128293;
#(4*p) if_opcode_w = 32'h0;
```

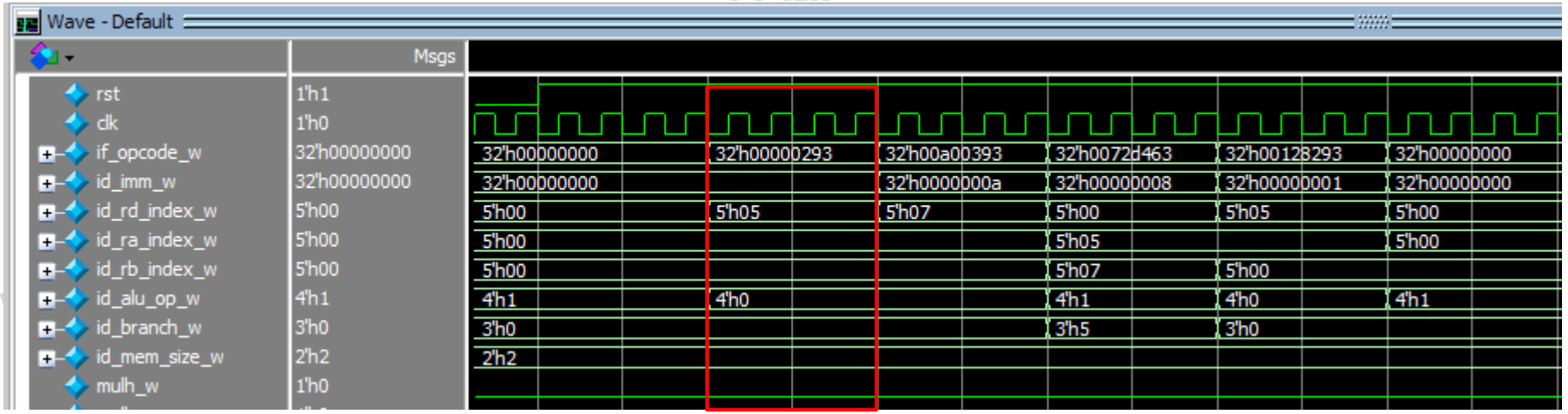


Waveform

- If_opcode_w=00000000000000000000001010010011
- op_w = 0010011
- rd_w = 00101
- f3_w = 000
- ra_w = 00000
- rb_w = 00000
- f7_w = 0000000

```
// Opcoder parser
assign op_w = if_opcode_w[6:0];
assign rd_w = if_opcode_w[11:7];
assign f3_w = if_opcode_w[14:12];
assign ra_w = if_opcode_w[19:15];
assign rb_w = if_opcode_w[24:20];
assign f7_w = if_opcode_w[31:25];
```

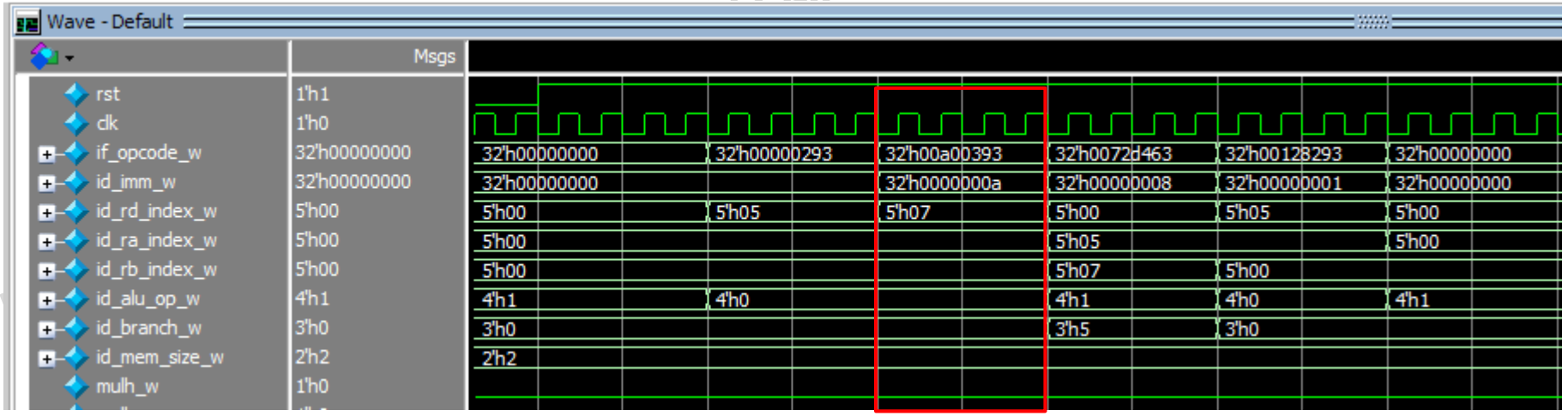
```
rst = 1'b0;                    // negedge reset on
if_opcode_w = 0;

#(4*p) rst = 1'b1;             // negedge reset off

#(4*p) if_opcode_w = 32'h00000293;
#(4*p) if_opcode_w = 32'h00a00393;
#(4*p) if_opcode_w = 32'h0072d463;
#(4*p) if_opcode_w = 32'h00128293;
#(4*p) if_opcode_w = 32'h0;
```



Waveform

- If_opcode_w = 00000000101000000000001110010011
- op_w = 0010011
- rd_w = 00011
- f3_w = 000
- ra_w = 00000
- rb_w = 01010
- f7_w = 0000000

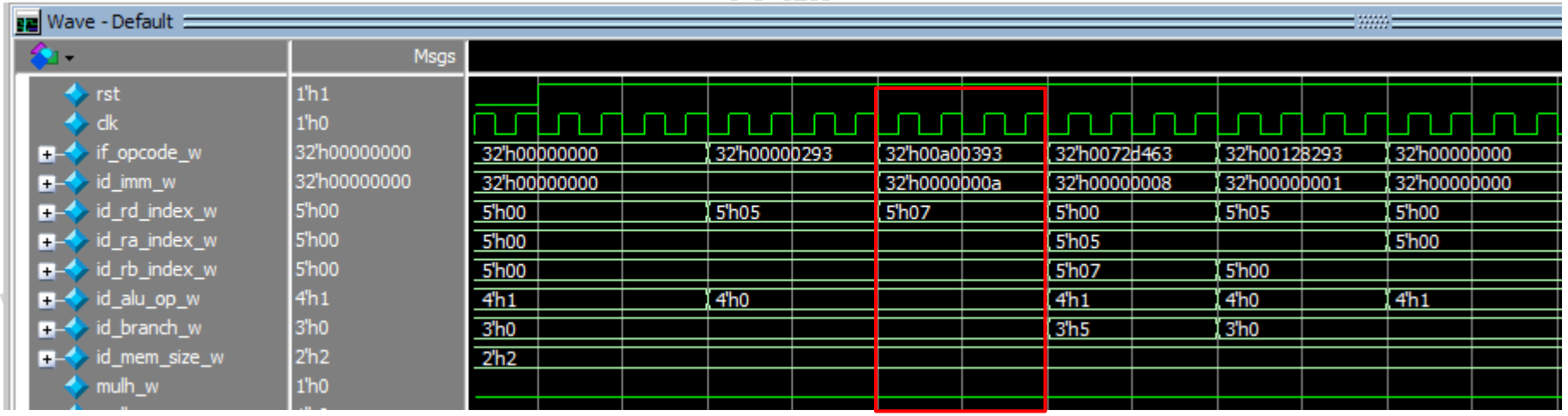
```
rst = 1'b0;                    // negedge reset on
if_opcode_w = 0;

#(4*p) rst = 1'b1;             // negedge reset off

#(4*p) if_opcode_w = 32'h00000293;
#(4*p) if_opcode_w = 32'h00a00393;
#(4*p) if_opcode_w = 32'h0072d463;
#(4*p) if_opcode_w = 32'h00128293;
#(4*p) if_opcode_w = 32'h0;
```



Waveform

- If_opcode_w = 00000000101000000000001110010011
- op_w = 0010011
- rd_w = 00011
- f3_w = 000
- ra_w = 00000
- rb_w = 01010
- f7_w = 0000000

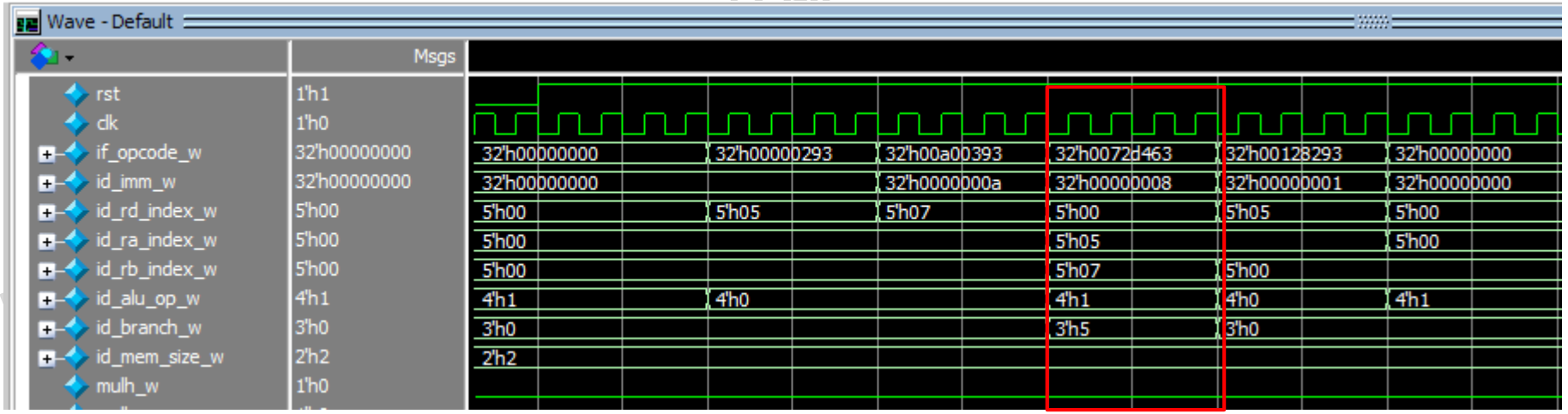
```
rst = 1'b0;                    // negedge reset on
if_opcode_w = 0;

#(4*p) rst = 1'b1;             // negedge reset off

#(4*p) if_opcode_w = 32'h00000293;
#(4*p) if_opcode_w = 32'h00a00393;
#(4*p) if_opcode_w = 32'h0072d463;
#(4*p) if_opcode_w = 32'h00128293;
#(4*p) if_opcode_w = 32'h0;
```



Waveform

- If_opcode_w = 00000000011100101110010001100011
- op_w = 1100011
- rd_w = 01000
- f3_w = 110
- ra_w = 00101
- rb_w = 00111
- f7_w = 0000000

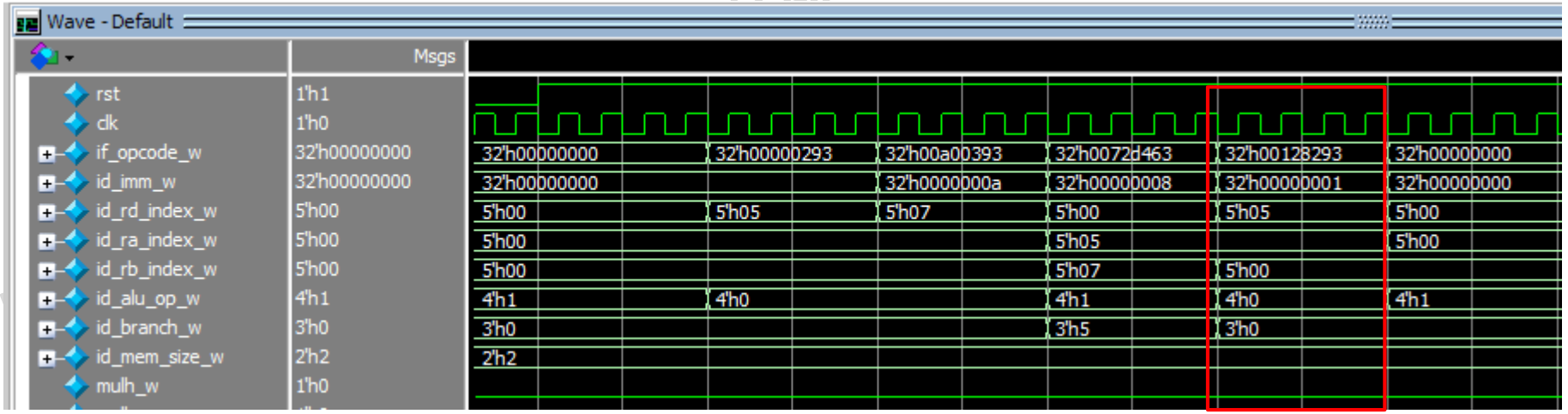
```
rst = 1'b0;                    // negedge reset on
if_opcode_w = 0;

#(4*p) rst = 1'b1;             // negedge reset off

#(4*p) if_opcode_w = 32'h00000293;
#(4*p) if_opcode_w = 32'h00a00393;
#(4*p) if_opcode_w = 32'h0072d463;
#(4*p) if_opcode_w = 32'h00128293;
#(4*p) if_opcode_w = 32'h0;
```



Waveform

- If_opcode_w = 00000000000100101000001010010011
- op_w = 0010011
- rd_w = 00101
- f3_w = 000
- ra_w = 00101
- rb_w = 00001
- f7_w = 0000000

```
rst = 1'b0;                    // negedge reset on
if_opcode_w = 0;

#(4*p) rst = 1'b1;             // negedge reset off

#(4*p) if_opcode_w = 32'h00000293;
#(4*p) if_opcode_w = 32'h00a00393;
#(4*p) if_opcode_w = 32'h0072d463;
#(4*p) if_opcode_w = 32'h00128293;
#(4*p) if_opcode_w = 32'h0;
```



To do …

- Implement riscv_decoder.v by completing the missing codes
- Do simulation and show the waveform

```
//**** Opcoder parser
//     Wire assign from each opcode field
assign op_w = if_opcode_w[6:0];
assign rd_w = if_opcode_w[11:7];
assign f3_w = if_opcode_w[14:12];
assign ra_w = if_opcode_w[/*your code*/];
assign rb_w = if_opcode_w[24:20];
assign f7_w = if_opcode_w[31:25];

//**** flag assign from opcode register field [6:0]
assign op_branch_w  = (7'b1100011 == op_w);
assign op_load_w    = (7'b0000011 == op_w);
assign op_store_w   = (7'b0100011 == op_w);
assign op_alu_imm_w = (7'b0010011 == op_w);
assign op_alu_reg_w = (/*your code*/ == op_w);

//**** flags assign from opcode register field [31:25]
assign op_f7_main_w = (7'b0000000 == f7_w);
assign op_f7_alt_w  = (/*your code*/  == f7_w);
assign op_f7_mul_w  = (7'b0000001 == f7_w);

//**** flags assign from opcode register field [6:0] and f3 cc
assign lui_w    = (7'b0110111 == op_w);
assign auipc_w  = (/*your code*/  == op_w);
assign jal_w    = (7'b1101111 == op_w);
assign jalr_w   = (7'b1100111 == op_w) && (3'b000 == f3_w);
```

Parse the parameters for **jump instructions**

```
//**** branch flags assign from f3 field and branch flag
assign beq_w    = op_branch_w  && (3'b000 == f3_w);
assign bne_w    = op_branch_w  && (/*your code*/ == f3_w);
assign blt_w    = op_branch_w  && (3'b100 == f3_w);
assign bge_w    = op_branch_w  && (3'b101 == f3_w);
assign bltu_w   = op_branch_w  && (3'b110 == f3_w);
assign bgeu_w   = op_branch_w  && (3'b111 == f3_w);
```

Link instruction

```
//**** load flags assign from f3 field and load flag
assign lb_w     = op_load_w    && (3'b000 == f3_w);
assign lh_w     = op_load_w    && (3'b001 == f3_w);
assign lw_w     = op_load_w    && (/*your code*/ == f3_w);
assign lbu_w    = op_load_w    && (3'b100 == f3_w);
assign lhu_w    = op_load_w    && (3'b101 == f3_w);

//**** store flags assign from f3 field and store flag
assign sb_w     = op_store_w   && (/*your code*/== f3_w);
assign sh_w     = op_store_w   && (3'b001 == f3_w);
assign sw_w     = op_store_w   && (3'b010 == f3_w);
```

To do …

- Implement riscv_decoder.v by completing some missing codes
- Do simulation
- Show the waveform

ALU instruction

```
//**** ALU flags assign from f3 field and alu flag
assign addi_w   = op_alu_imm_w && (3'b000 == f3_w);
assign slti_w   = op_alu_imm_w && (3'b010 == f3_w);
assign sltiu_w  = op_alu_imm_w && (3'b011 == f3_w);
assign xori_w   = op_alu_imm_w && (/*your code*/ == f3_w);
assign ori_w    = op_alu_imm_w && (3'b110 == f3_w);
assign andi_w   = op_alu_imm_w && (3'b111 == f3_w);
assign slli_w   = op_alu_imm_w && (/*your code*/ == f3_w) && op_f7_main_w;
assign srli_w   = op_alu_imm_w && (3'b101 == f3_w) && op_f7_main_w;
assign srai_w   = op_alu_imm_w && (3'b101 == f3_w) && op_f7_alt_w;

assign add_w    = op_alu_reg_w && (3'b000 == f3_w) && op_f7_main_w;
assign sub_w    = op_alu_reg_w && (3'b000 == f3_w) && op_f7_alt_w;
assign slt_w    = op_alu_reg_w && (/*your code*/ == f3_w) && op_f7_main_w;
assign sltu_w   = op_alu_reg_w && (3'b011 == f3_w) && op_f7_main_w;
assign xor_w    = op_alu_reg_w && (3'b100 == f3_w) && op_f7_main_w;
assign or_w     = op_alu_reg_w && (3'b110 == f3_w) && op_f7_main_w;
assign and_w    = op_alu_reg_w && (3'b111 == f3_w) && op_f7_main_w;
assign sll_w    = op_alu_reg_w && (3'b001 == f3_w) && op_f7_main_w;
assign srl_w    = op_alu_reg_w && (/*your code*/ == f3_w) && op_f7_main_w;
assign sra_w    = op_alu_reg_w && (3'b101 == f3_w) && op_f7_alt_w;

assign mul_w    = op_alu_reg_w && (3'b000 == f3_w) && op_f7_mul_w;
assign mulh_w   = op_alu_reg_w && (3'b001 == f3_w) && op_f7_mul_w;
assign mulhsu_w = op_alu_reg_w && (/*your code*/ == f3_w) && op_f7_mul_w;
assign mulhu_w  = op_alu_reg_w && (3'b011 == f3_w) && op_f7_mul_w;
assign div_w    = op_alu_reg_w && (3'b100 == f3_w) && op_f7_mul_w;
assign divu_w   = op_alu_reg_w && (3'b101 == f3_w) && op_f7_mul_w;
assign rem_w    = op_alu_reg_w && (3'b110 == f3_w) && op_f7_mul_w;
assign remu_w   = op_alu_reg_w && (/*your code*/ == f3_w) && op_f7_mul_w;
```

Road map

RISC-V Simulator

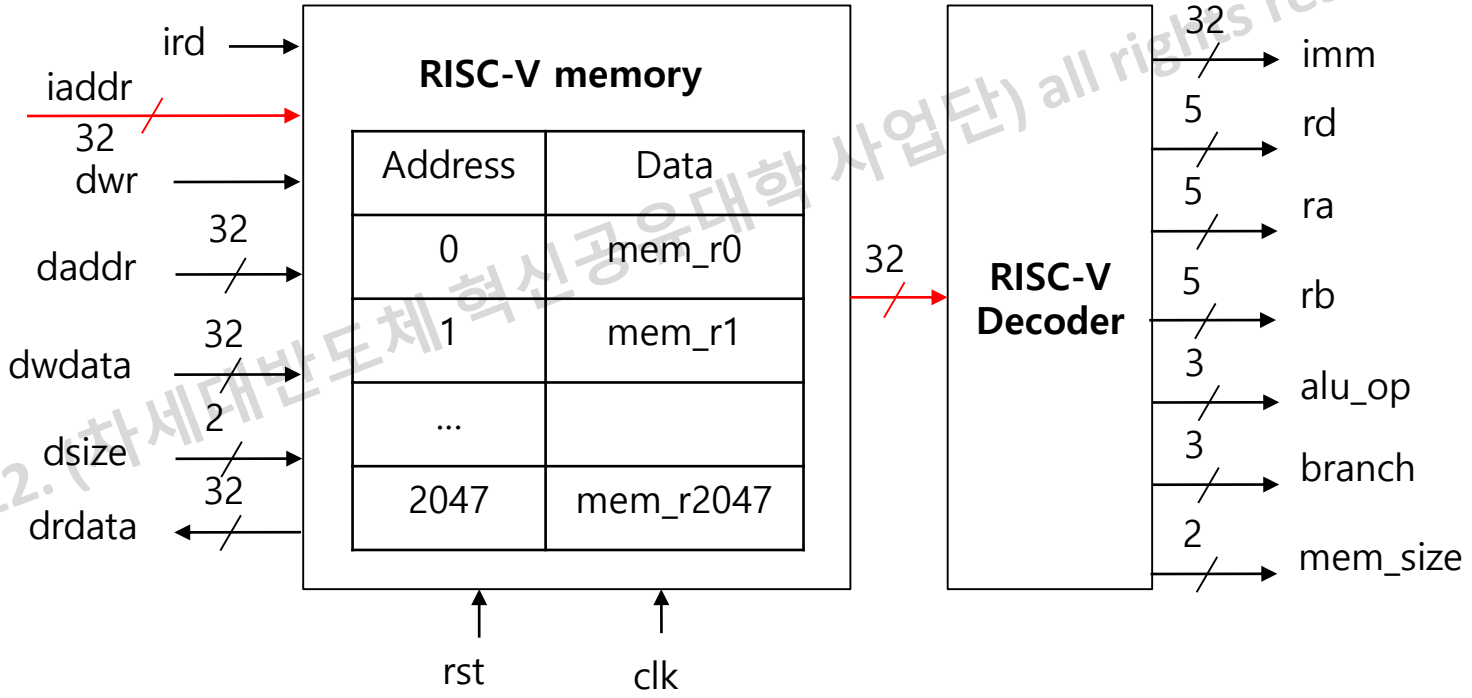
Memory
Instruction/data memory

RISC-V Instruction format
Decoder

Simple core

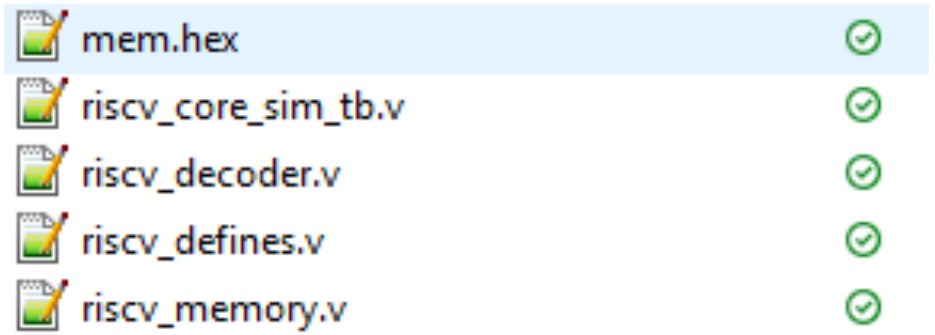
Lab 3: Simple core

- Lab 3: Memory and Decoder
  - Complete the ***connections*** for memory and decoder modules
  - Do simulation
  - Capture the waveform



Code structure

- mem.hex: initialization file for riscv_memory.v
- riscv_defines.v: general definitions in RISC-V (e.g., ALU, branch instructions)
- riscv_core_sim_tb.v: a test bench includes two instances
  - riscv_memory.v: Instruction/data memory model
    - Optionally initialized with mem.hex
  - riscv_decoder.v: RISC-V decoder



mem.hex

- Generate machine code of examples
- Save generate machine codes in mem.hex file
- Replace the given mem.hex file by the new one

```
int A[64];
int sum = 0;
for (int i=0; i<64; i++)
    sum +=  A[i];
```

```
add x9, x8, x0 # x9=&A[0]
add x10, x0, x0 # sum=0
add x11, x0, x0 # i=0
Loop:
  lw x12, 0(x9) # x12=A[i]
  add x10,x10,x12 # sum+=
  addi x9,x9,4  # &A[i++]
  addi x11,x11,1 # i++
  addi x13,x0,64 # x13=64
  blt x11,x13,Loop
```

riscv_core_sim_tb.v

- Internal signals
  - Clock, reset
  - Signals to interface with memory
    - addresses (iaddr, daddr)
    - Request
      - Instruction: Read only (ird)
      - Data: Read (drd) and Write (dwr)
  - Signals to interface with decoder
    - Instruction (if_opcode_w)
    - Control signals

```
module riscv_core_sim_tb ();

    reg reset_i;
    reg clk_i;

    // Input instruction
    reg  [31:0] iaddr_i;
    reg         ird_i;
    reg [31:0] daddr_i;
    reg [31:0] dwdata_i;
    reg [1:0]   dsize_i;
    reg       drd_i;
    reg         dwr_i;
    // Outputs
    wire [31:0] irdata_o;
    wire [31:0] drdata_o;
```

Memory

```
    // Input instruction
    reg  [31:0] if_opcode_w;
    // Outputs
    wire [31:0] id_imm_w;
    wire [4:0] id_rd_index_w;
    wire [4:0] id_ra_index_w;
    wire [4:0] id_rb_index_w;
    wire [3:0] id_alu_op_w;
    wire [2:0] id_branch_w;
    wire [1:0] id_mem_size_w;
```

Decoder

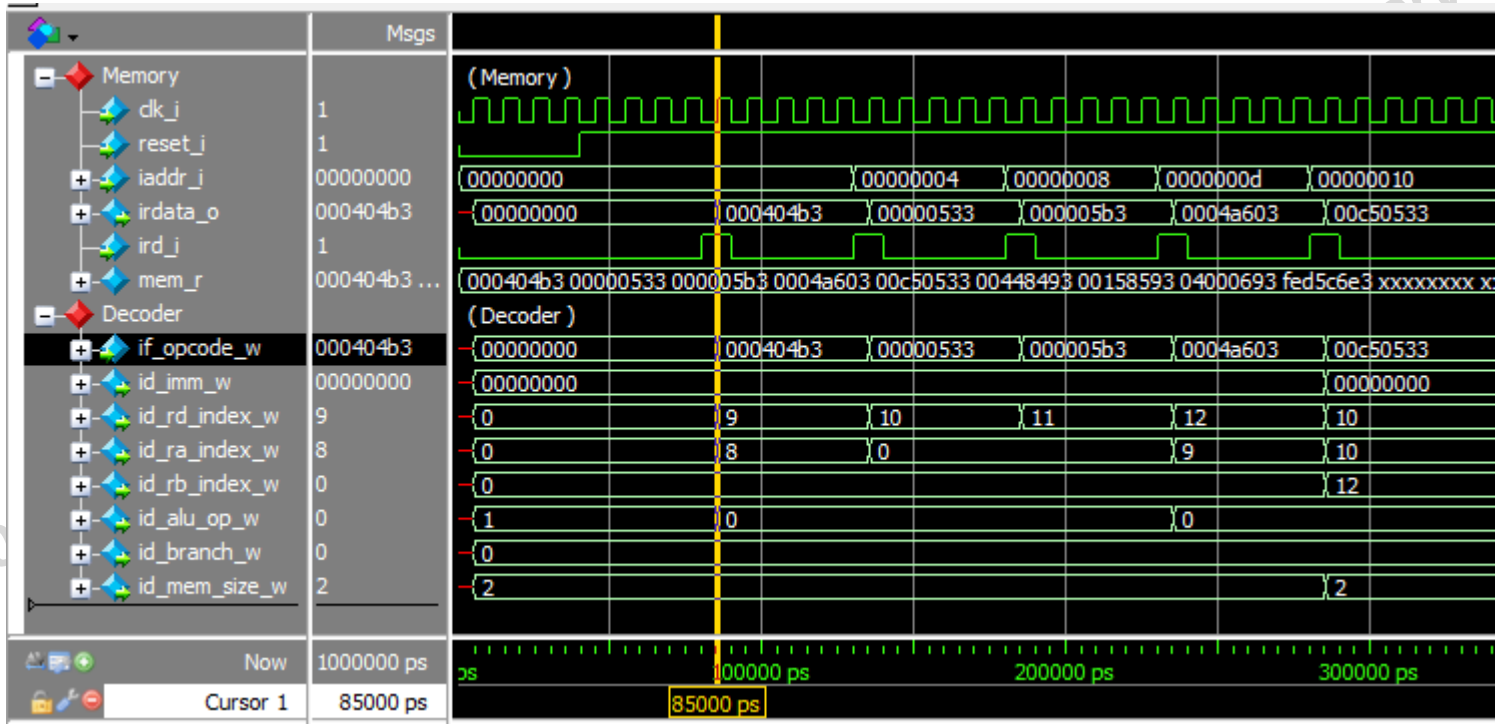
Test bench

- Clock 100MHz
- Read out fives instructions
  - ird_i
  - iaddr_i: 0, 4, 8, 12, 16

```
// Clock and Reset
parameter p=10;
initial begin
 clk_i = 1'b0;
 forever #(p/2) clk_i = !clk_i;
end

// Test cases
initial
begin:stimuli
      reset_i = 1'b0;
      iaddr_i = 0;
      ird_i = 0;
      dsize_i=SIZE_WORD;
      #(4*p) reset_i = 1'b1;

      #(4*p)  iaddr_i = 32'h0;
              ird_i = 1'b1;
      #(p)    ird_i = 1'b0;
              $display("T=%03t ns: %h : %h\n",$realtime/1000, iaddr_i, irdata_o);
      #(4*p)  iaddr_i = 32'h4;
              ird_i = 1'b1;
      #(p)    ird_i = 1'b0;
              $display("T=%03t ns: %h : %h\n",$realtime/1000, iaddr_i, irdata_o);
      #(4*p)  iaddr_i = 32'h8;
              ird_i = 1'b1;
      #(p)    ird_i = 1'b0;
              $display("T=%03t ns: %h : %h\n",$realtime/1000, iaddr_i, irdata_o);
      #(4*p)  iaddr_i = 32'hd;
              ird_i = 1'b1;
      #(p)    ird_i = 1'b0;
              $display("T=%03t ns: %h : %h\n",$realtime/1000, iaddr_i, irdata_o);
      #(4*p)  iaddr_i = 32'h10;
              ird_i = 1'b1;
      #(p)    ird_i = 1'b0;
              $display("T=%03t ns: %h : %h\n",$realtime/1000, iaddr_i, irdata_o);

end
```

Waveform

- Example
  - Assembly code: add x9, x8, x0
  - Machine code (RISC-V): 00040483
  - Decoder results: rd=9 (x9), ra=8 (rs1/x8), rb=0 (rs2/x0), alu_op=0 (ALU_ADD).



To do …

- Code reuse
  - riscv_memory.v in Lab 1
  - riscv_decoder.v in Lab 2
- Complete some missing codes at riscv_core_sim_tb.v
- Do a simulation and show the waveform

```
//------------------------------------------------------------------
// Your Memory module
//------------------------------------------------------------------
//{{{
// Memory
riscv_memory
u_memory (
    .clk_i(clk_i),
    .reset_i(reset_i)
);

// Decoder
riscv_decoder
u_decoder
(
);
//}}}
```

Questions

- How do you connect the four following modules?
  - Memory
  - Decoder
  - ALU
  - Register File

Questions

- How do you connect the four following modules?
  - Memory
  - Decoder
  - ALU
  - Register File