차세대 반도체 디지털 혁신공유대학 사업단

POLARIS

Lecture 2: Introduction to RISC-V, Register File, ALU

Xuan-Truong Nguyen

Recap: Course overview

	#	Topics	Labs
CPU memory	1	Introduction, Basic Digital Logic	ModelSim, Encoder, Counter
	2	Introduction to RISC-V	Register File, Shifter and ALU
	3	Instruction Set Architecture	Instruction memory, decoder
	4	Instructions	Program Counter, Branch Instruction
	5	RISC-V Compiler, GNU Tool Chain	Load/Store Instruction, RISC-V core
Bus and IO	6	AMBA Bus Specification	AHB Master, AHB slave, AHB Bus
	7	IO Devices and Display Panel	LCD Drive, Display Panel
	8	System integration	LCD interface, memory interface, top system
CNN Accelerator	9	Accelerator, Super resolution networks	Matlab, CNN reference S/W, quantization, BRAM
	10	Computing Units of DNN accelerator	MAC, Adder tree, quantization, and activation
	11	Datapath for a DNN accelerator I	Sliding windows, weight/bias/scale buffers
	12	Datapath for a DNN accelerator II	Buffer access, Super resolution NPU
	13	System Integration	BRAM, Function verification, DMA
	14	Optimization	Dual buffering, Pipelining
	15	Final project and Final Exam	Execution time reduction, Memory buffer reduction

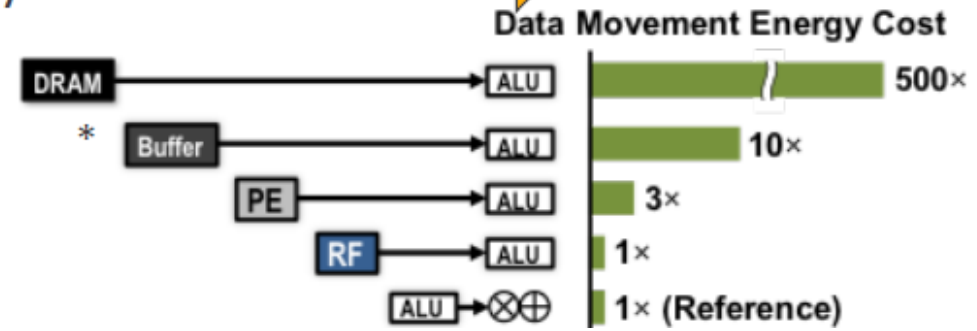
Recap: Challenges with DNN Computations

- **Millions of Parameters (i.e., weights)**
  - Billions of computations ⇨ **Need lots of parallel computations**

DNN Topology	Number of Weights
AlexNet (2012)	3.98M
VGGnet-16 (2014)	28.25M
GoogleNet (2015)	6.77M
Resnet-50 (2016)	23M
DLRM (2019)	540M
Megatron (2019)	8.3B

This makes CPUs inefficient

  - Heavy data movement ⇨ **Need to reduce energy**



This makes GPUs inefficient

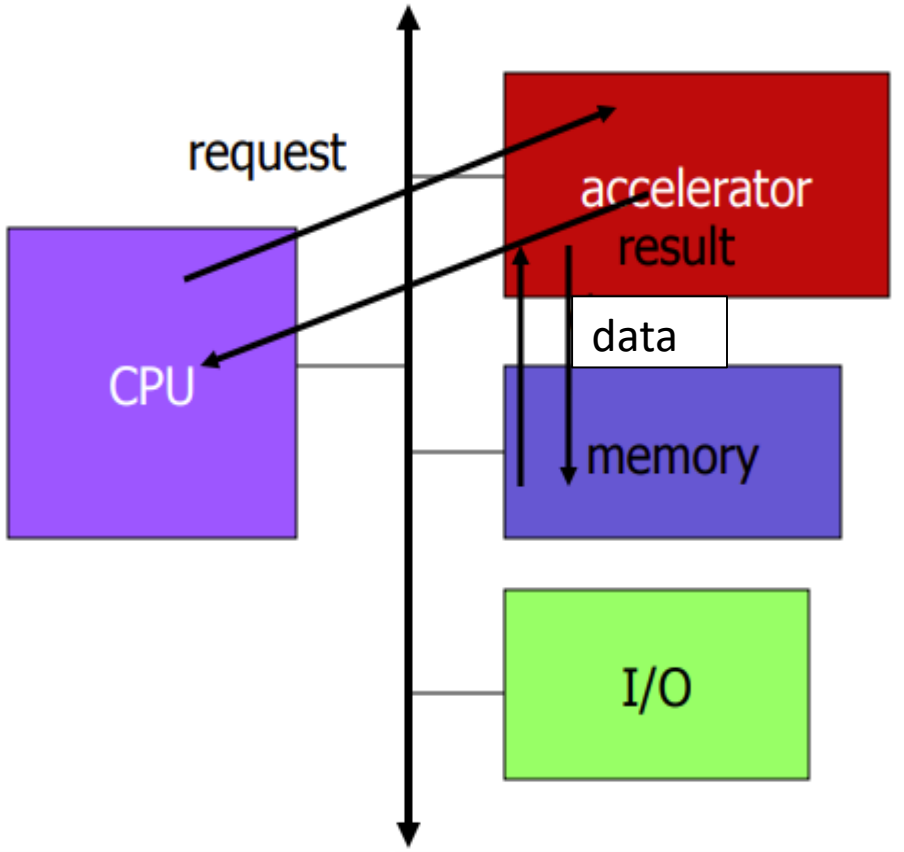
Courtesy: MIT Eyeriss

AI computing system

- Von Neumann architecture
  - A central processing unit (CPU)
  - Memory
  - Input/Output (IO)

- AI or Deep Learning Accelerator
  - Use additional computational units dedicated to some functionality
    - A hardware (H/W) module for specified applications

- Hardware/software co-design
  - Joint design of H/W and S/W architectures



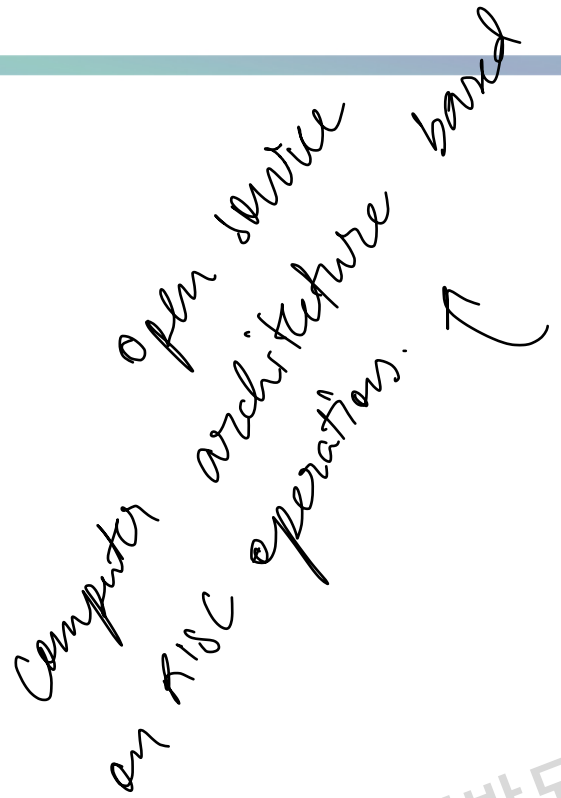
Lecture plan

- Today we will:
  - Study RISC-V
  - Continue reviewing Verilog HDL
    - Describe how to use continuous assignments
    - Describe the data types allowed in Verilog HDL
    - Describe the operation of the operators used in Verilog HDL
    - Describe the operands that may be used associated with a specified operator

    (Credit: Some slides are based on Digital Systems Design and Experiments (https://ocw.snu.ac.kr/node/2390))

  - Labs
    - Lab 1: Register file
    - Lab 2: Shifter, Arithmetic unit
    - Lab 3: Multiplier

Road map



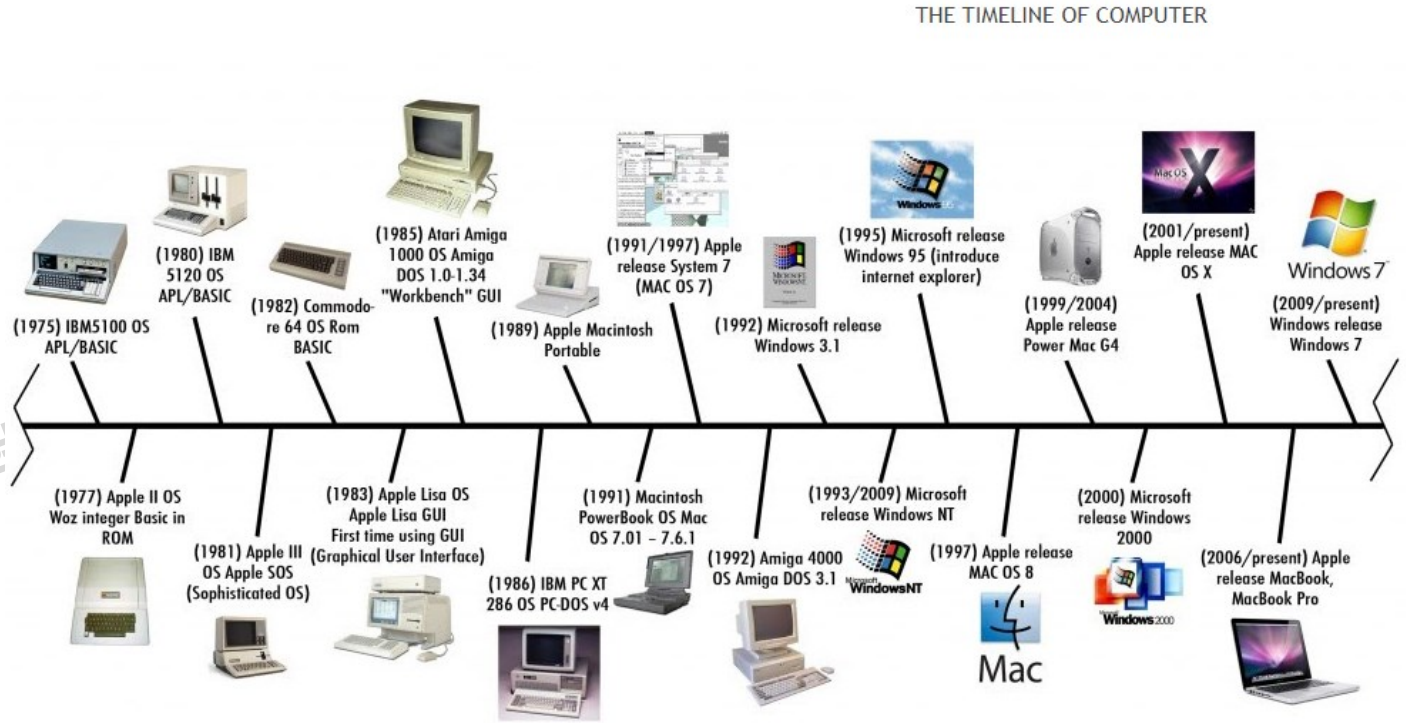
Introduction to RISC-V

Verilog HDL
Dataflow modelling

Labs

The Computer Revolution

- Progress in computer technology
  - Underpinned by Moore's Law
- Makes novel applications feasible
  - Computers in automobiles
  - Cell phones
  - Human genome project
  - World Wide Web
  - Search Engines
- Computers are pervasive in our life.



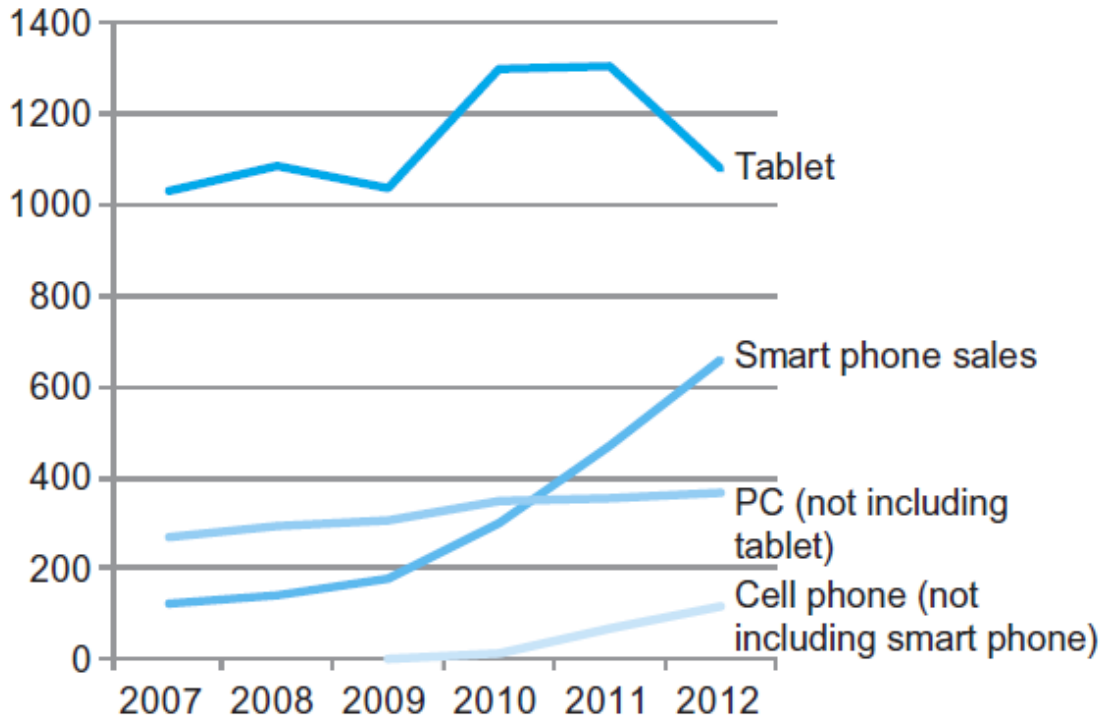
https://sites.google.com/site/ict0512163083/alisyashaheer

Classes of Computers

- Personal computers
  - General purpose, variety of software
  - Subject to cost/performance tradeoff
- Server computers
  - Network-based
  - High capacity, performance, reliability
  - Range from small servers to building-sized
- Supercomputers
  - High-end scientific and engineering calculations
  - Highest capability but represent a small fraction of the overall computer market
- Embedded computers
  - Hidden as components of systems
  - Stringent power/performance/cost constraints

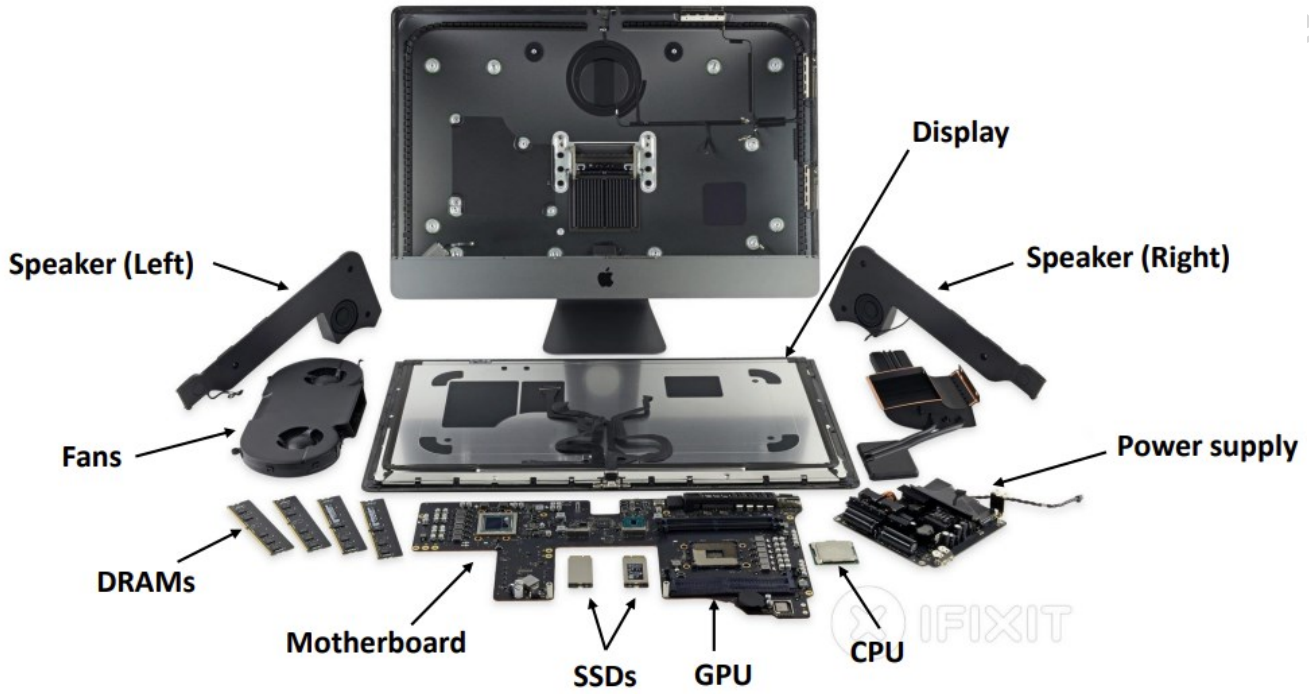
The Post-PC Era

- Personal Mobile Device (PMD)
  - Battery operated
  - Connects to the Internet
  - Hundreds of dollars
  - Smartphones, tablets, electronic glasses
- Cloud computing
  - Warehouse Scale Computers (WSC)
  - Software as a Service (SaaS)
  - A portion of software runs on a PMD, and a portion runs in the Cloud
  - Amazon and Google
- Fog computing and edge computing involve pushing intelligence and processing capabilities down closer to where the data originates from pumps, motors, sensors, relays, etc.



Exploring the Box (iMAC)

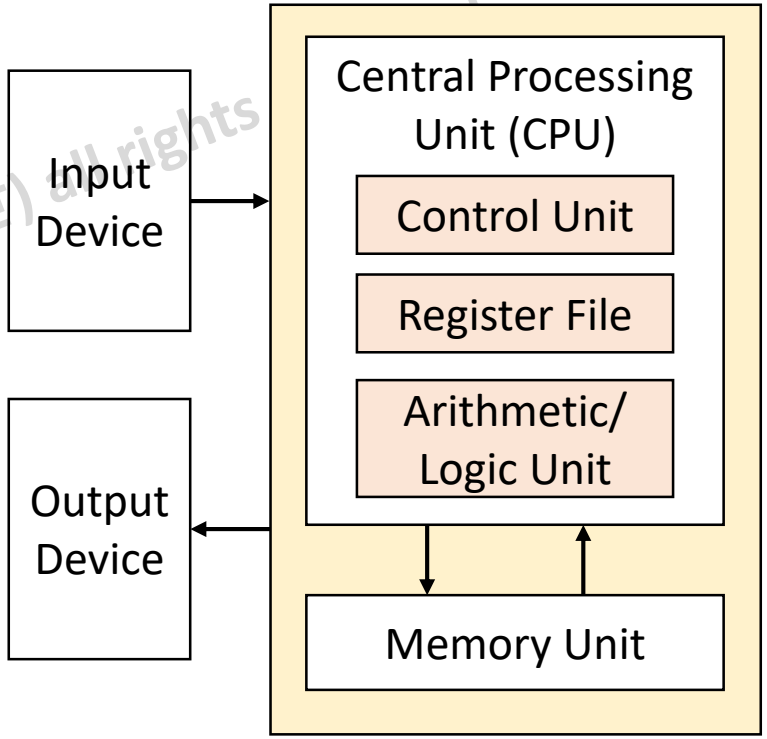
- Central Processing unit (CPU), Graphical Processing Unit (GPU)
- Memory: DRAMs, SSD
- IO, Peripherals: Display, speaker



Source: https://www.ifixit.com/Teardown/iMac+Pro+Teardown/101807

Von Neumann Architecture

- A design architecture for an electronic digital computer with these components:
  - A processing unit that contains
    an *arithmetic logic unit*
    (ALU) and processor registers
    - A control unit that contains
      an *instruction register* and *program counter*
  - Memory that stores data and instructions
  - External mass storage
  - Input and output mechanisms

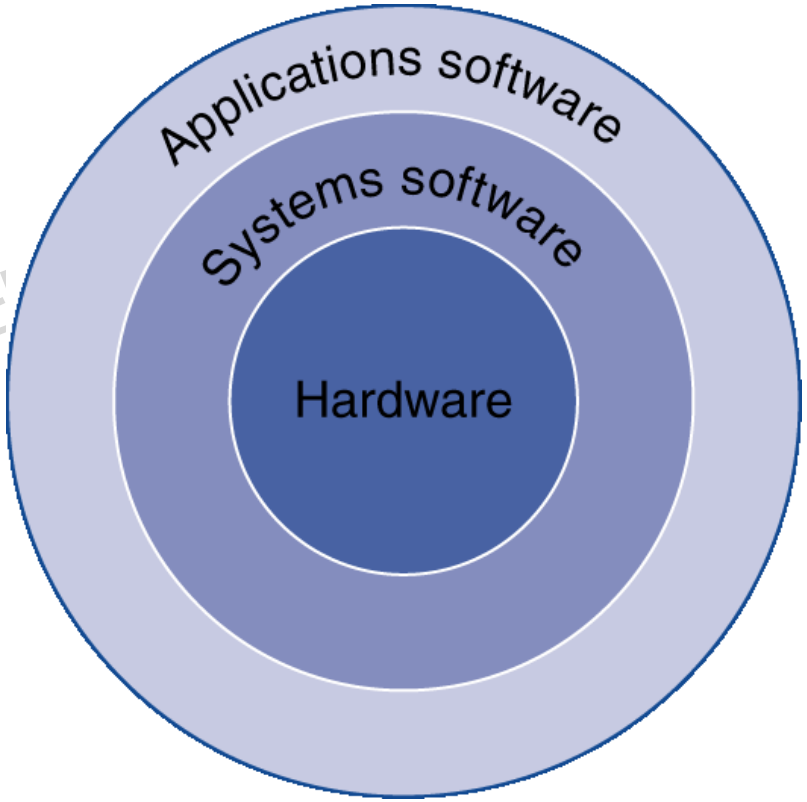


Below Your Program

- Application software
  - Written in high-level language (HLL)

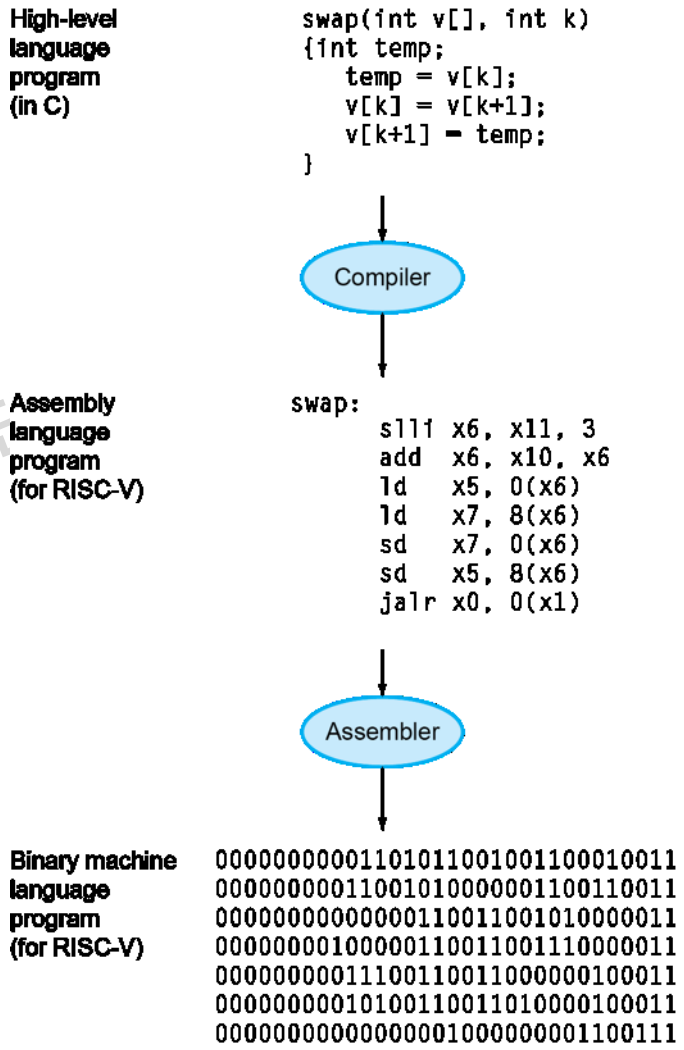
- System software
  - Compiler: translates HLL code to machine code
  - Operating System: service code
    - Handling input/output
    - Managing memory and storage
    - Scheduling tasks & sharing resources

- Hardware
  - Processor, memory, I/O controllers



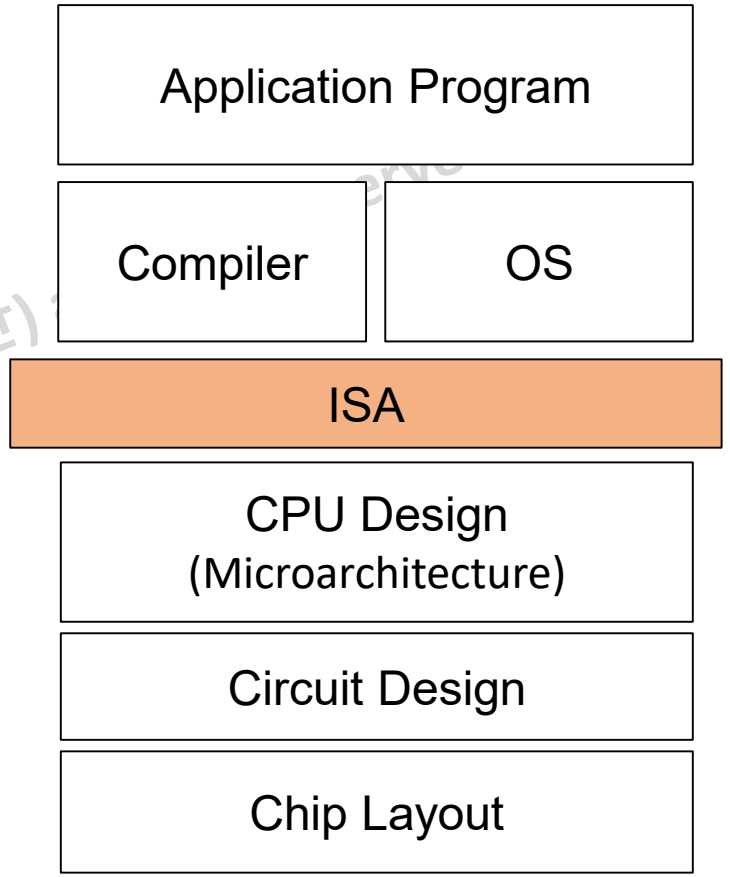
Levels of Program Code

- High-level language
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability
- Assembly language
  - Textual representation of instructions
- Hardware representation
  - Binary digits (bits)
  - Encoded instructions and data



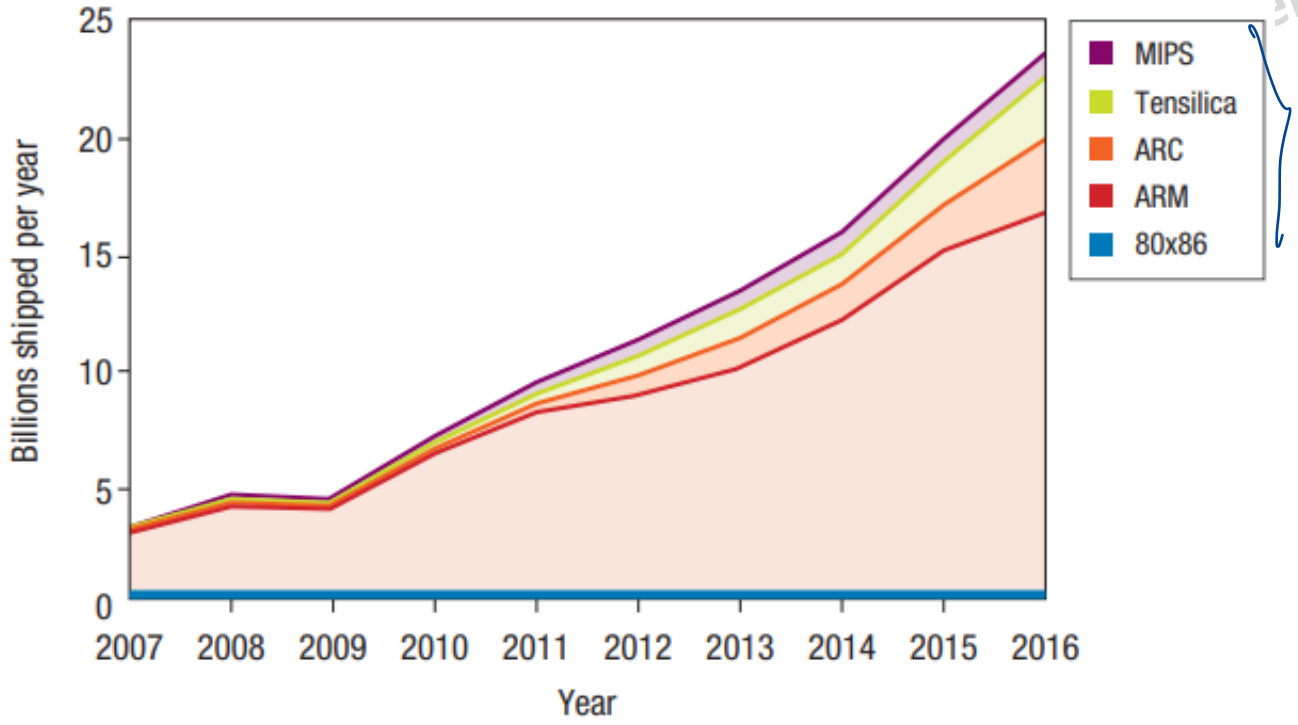
Instruction Set Architecture (ISA)

- The repertoire of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets



Instruction Set Architecture (ISA)

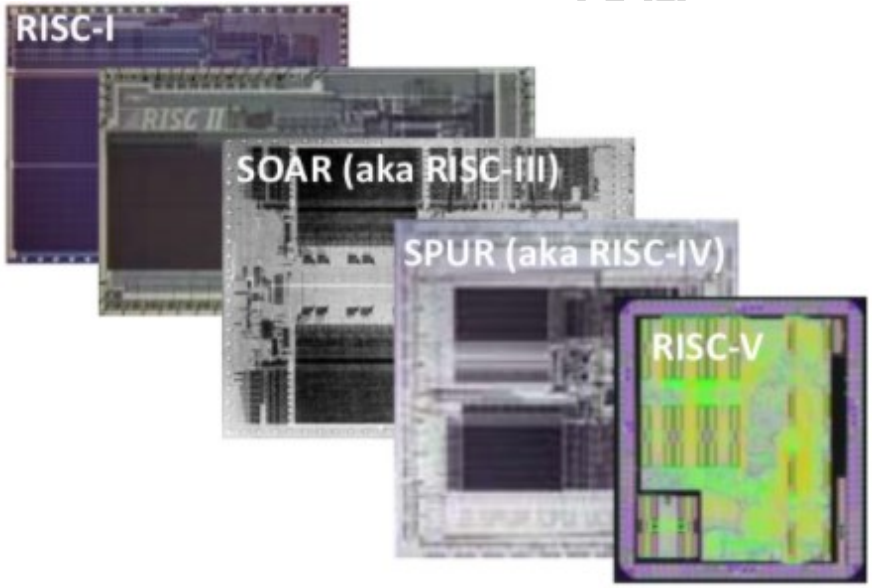
- Most ISAs: X86, ARM, Power, MIPS, SPARC
  - Commercially protected by patents
  - Preventing practical efforts to reproduce the computer systems.



Source: *David Patterson, "Reduced Instruction Set Computers Then and Now," IEEE Computer, 2017*

The RISC-V Instruction Set

- RISC-V (aka "risk-five"): Developed at UC Berkeley as open ISA
  - RISC-I (1981), RISC-II (1983), SOAR (1984), SPUR (1989), RISC-V (2010)
  - Now managed by the RISC-V Foundation (riscv.org)
  - Typical of many modern ISAs
- Similar ISAs have a large share of the embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers,…



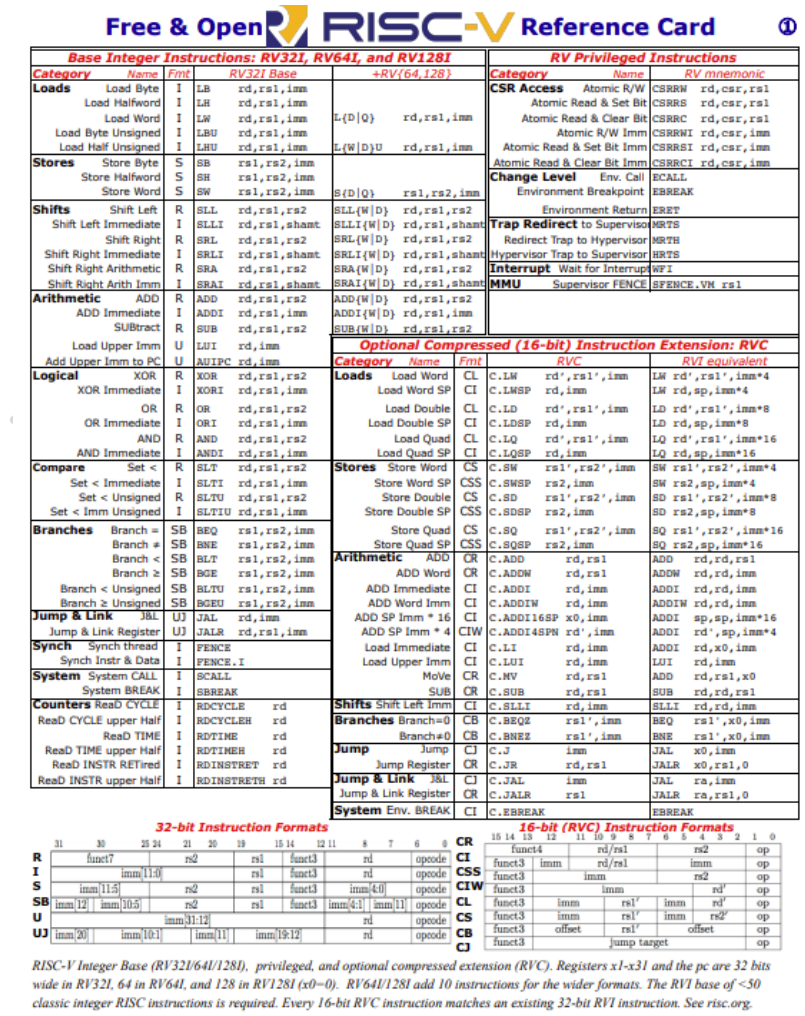
RISC-V Ecosystem

- The RISC-V Foundation (www.riscv.org) was founded in 2015
- An open, collaborative community of S/W and H/W innovators based on the RISC-V ISA.

Source: https://riscv.org/2019/05/getting-started-with-risc-v-china-roadshow-proceedings/ultrasoc-ppt/

Why RISC-V is popular nowadays?

- Why RISC-V?
  - A **free and open** ISA
  - **Simple**: Far smaller than other commercial ISAs
  - **Clean-slate** design
    - Clear separation between user and privileged ISA
    - Avoid micro-architecture or technology-dependent ISA
  - A modular ISA
    - A small standard base ISA, i.e., RV32I
    - Multiple standard extensions
- A two-page card for RISC-V
  - Instruction formats: 16/32/64/128-bit formats
  - Base integer instructions and extensions
    - Load/store
    - Shifts/arithmetic/logical/compare
    - Braches/jump and link



RISC-V Integer Base (RV32I/64I/128I), privileged, and optional compressed extension (RVC). Registers x1-x31 and the pc are 32 bits wide in RV32I, 64 in RV64I, and 128 in RV128I (x0=0). RV64I/128I add 10 instructions for the wider formats. The RVI base of <50 classic integer RISC instructions is required. Every 16-bit RVC instruction matches an existing 32-bit RVI instruction. See risc.org.

Example: Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination

```
add a, b, c  // a gets b + c
```

a = b + c

if you give

add(a, b, c)

- All arithmetic operations have this form

- *Design Principle 1:* Simplicity favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

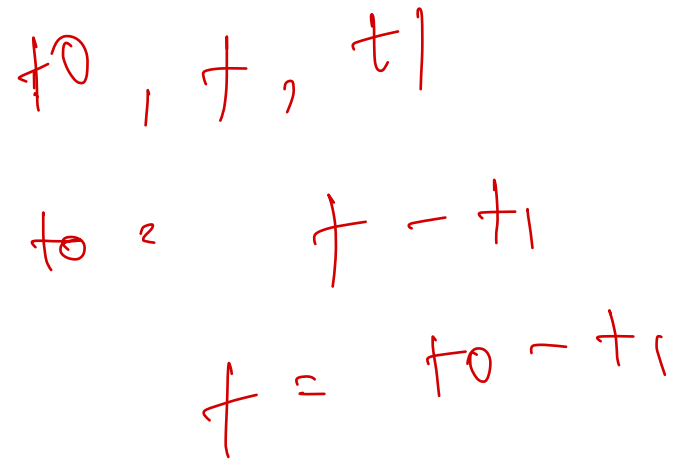
Arithmetic example

- C code:

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

- 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
```



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
  - 32-bit data is called a “word”

- *Design Principle 2:* Smaller is faster
  - c.f. main memory: millions of locations

RISC-V register

- x0: the constant value 0
- x1: return address
- x2: stack pointer
- x3: global pointer
- x4: thread pointer
- x5 – x7, x28 – x31: temporaries
- x8: frame pointer
- x9, x18 – x27: saved registers
- x10 – x11: function arguments/results
- x12 – x17: function arguments

Register Operand Example

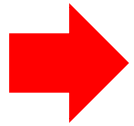
- C code:

  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
```

Road map

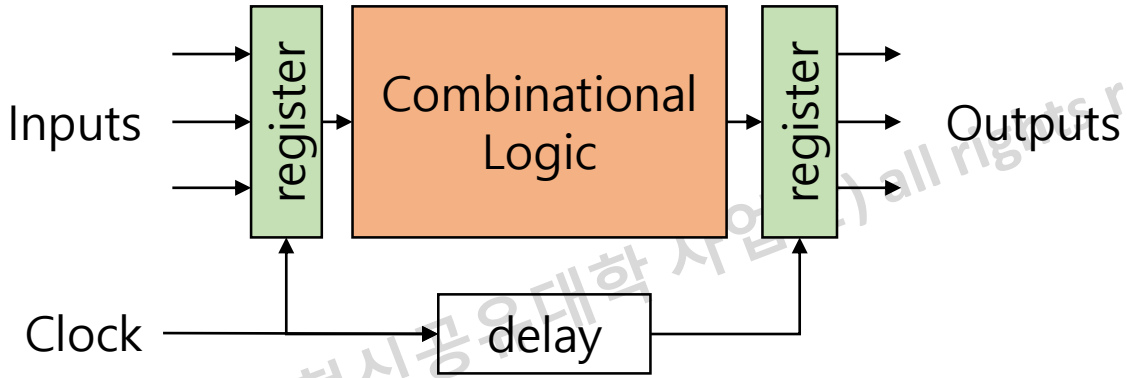
Introduction to RISC-V

Verilog HDL
Dataflow modelling

Labs

Dataflow modelling

- Any digital system can be constructed by *interconnecting registers and a combinational logic* put between them for performing the necessary functions.



- Dataflow provides a powerful way to implement a design.
- Logic synthesis tools can be used to create a gate-level circuit from a dataflow design description.
- RTL (register transfer level) is a combination of dataflow and behavioral modeling.

Assignments

- Two basic forms of assignments
  - Continuous assignment: assign values to **nets**
  - Procedural assignment: assign values to **variables**

- Two additional forms of assignments: procedural continuous assignments
  - assign/deassign
  - force/release

- An assignment consists of two parts:
  - a left-hand side (LHS) and a right-hand side (RHS) separated by = or <=
  - RHS: any expression that evaluates to a value to which the LHS value is to be assigned.
  - LHS: can take one of the forms given on the **Next** page.

Assignments

Table 6-1—Legal left-hand forms in assignment statements

Statement type	Left-hand side
Continuous assignment	Net (vector or scalar) Constant bit-select of a vector net Constant part-select of a vector net Constant indexed part-select of a vector net Concatenation or nested concatenation of any of the above left-hand side
Procedural assignment	Variables (vector or scalar) Bit-select of a vector reg, integer, or time variable Constant part-select of a vector reg, integer, or time variable Indexed part-select of a vector reg, integer, or time variable Memory word Concatenation or nested concatenation of any of the above left-hand side

Continuous Assignments

- Continuous assignment: the most basic statement of dataflow modeling.
  - It is used to drive a value onto a net.
  - It is always active.
  - Provides a way to model combinational logic without specifying an interconnection of gates. Instead, it specifies the logical expression that drives the net.
  - It can only update values of net data types such as wire, triand, etc.
  - This assignment shall occur whenever the value of the right-hand side changes.
- Example:

```
assign {c_out, sum[3:0]} = a[3:0] + b[3:0] + c_in;
```

Continuous Assignments

- A continuous assignment begins with the keyword assign.

```
assign net_lvalue = expression;

assign net1 = expr1,
       net2 = expr2,
       ...,
       netn = exprn;
```


  - net_lvalue is a scalar or vector net or their concatenation.
  - RHS operands can be variables or nets, or function calls.
  - Registers or nets can be scalar or vectors.
  - Delay values can be specified.

Continuous Assignments

- An implicit continuous assignment
  - is the shortcut of declaring a net first and then writing a continuous assignment on the net.
  - is always active.
  - can only have one implicit declaration assignment per net.

```
wire out;                    // net declaration
assign out = in1 & in2; // regular continuous assignment

wire  out = in1 & in2;    // implicit continuous assignment
```

Data types

- Two classes of data types:
  - nets: Nets mean any hardware connection points.
  - variables: Variables represent any data storage elements.

- Variable data types
  - reg
  - integer
  - time
  - real
  - realtime

Variable data types

- A reg variable
  - holds a value between assignments.
  - may be used to model hardware registers.
  - need not actually represent a hardware storage element.

```
reg a, b;                 // reg a, and b are 1-bit reg
reg [7:0] data_a;    // an 8-bit reg, the msb is bit 7
reg [0:7] data_b;    // an 8-bit reg, the msb is bit 0
reg signed [7:0] d; // d is an 8-bit signed reg
```

- The integer variable
  - contains integer values.
  - has at least 32 bits.
  - is treated as a signed reg variable with the least-significant bit (LSB) being bit 0.

```
integer i,j;          // declare two integer variables
integer data[7:0]; // array of integer
```

The time, real, and realtime variables

- The time variable
  - is used for storing and manipulating simulation time quantities.
  - is typically used in conjunction with the $time system task.
  - holds only unsigned value and is at least 64 bits, with the LSB being bit 0.

```
time events;         // hold one time value
time current_time; // hold one time value
```

- The real and realtime variables
  - cannot use range declaration and their initial values are defaulted to zero (0.0).

```
real events;              // declare a real variable
realtime current_time; // hold current time as real
```

Vectors

- A vector (multiple bit width) describes a bundle of signals as a basic unit.
  - [high:low] or [low:high]
  - The leftmost bit is the MSB.
  - Both nets and reg data types can be declared as vectors.
- The default is 1-bit vector or called scalar.

- Bit-Select and Part-Select
  - integer and time can also be accessed by bit-select or part-select.
  - real and realtime are not allowed to be accessed by bit-select or part-select.
  - Constant part select: data_bus[3:0], bus[3]
  - Variable part select:

    [<starting_bit>+:width]: data_bus[8+:8]

    [<starting_bit>-:width]: data_bus[15-:8]

Array and memory elements

- Array and Memory Elements
  - all net and variable data types are allowed to be declared as **multi-dimensional** arrays.
  - an array element can be a scalar or a vector if the element is a net or reg data type.

```
wire a[3:0];            // a scalar wire array of 4 elements
reg  d[7:0];            // a scalar reg array of 8 elements
wire [7:0]  x[3:0];     // an 8-bit wire array of 4 elements
reg  [31:0] y[15:0];    // a 32-bit reg array of 16 elements
integer states [3:0];   // an integer array of 4 elements
time    current[5:0];   // a time array of 6 elements
```

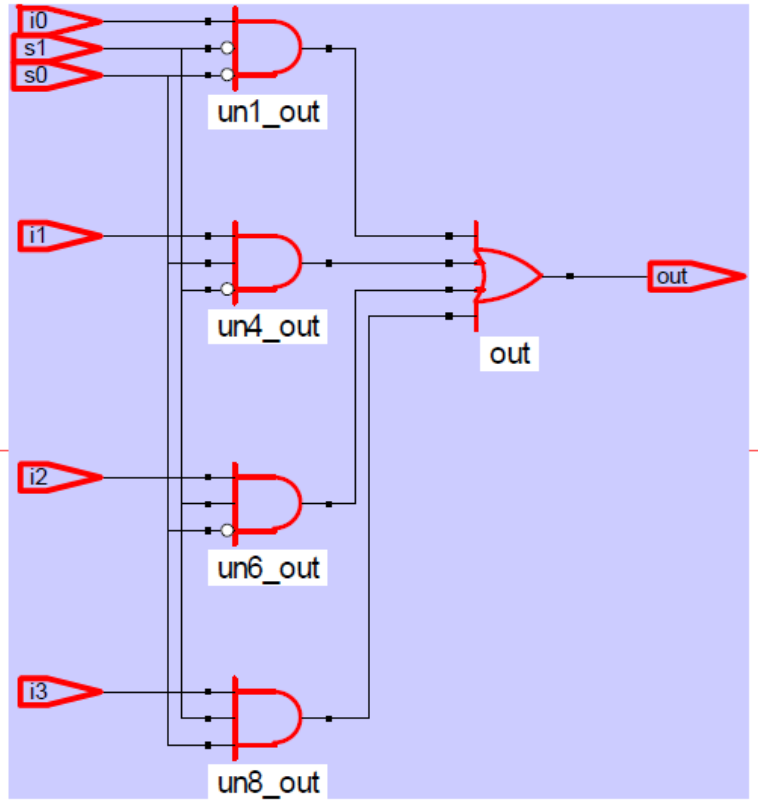
Bitwise operations

- Bitwise operators
  - They perform a bit-by-bit operation on two operands.
  - A z is treated as x in bit-wise operation.
  - The shorter operand is zero-extended to match the length of the longer operand.

Symbol	Operation
~	Bitwise negation
&	Bitwise and
|	Bitwise or
^	Bitwise exclusive or
~^, ^~	Bitwise exclusive nor

Example: 4-to-1 MUX

```
module mux41_dataflow(i0, i1, i2, i3, s1, s0, out);
// Port declarations
input i0, i1, i2, i3;
input s1, s0;
output out;
// Using basic and, or , not logic operators.
assign out =    (~s1 & ~s0 & i0) |
                (~s1 & s0 & i1)  |
                (s1 & ~s0 & i2)  |
                (s1 & s0 & i3) ;
endmodule
```



Arithmetic Operators

- Arithmetic operators
  - If any operand bit has a value x, then the result is x.
  - The operators + and –can also used as unary operators to represent signed numbers.
  - Modulus operators produce the remainder from the division of two numbers.
  - In Verilog HDL, 2's complement is used to represent negative numbers.

Symbol	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponent (power)
%	Modulus

Concatenation and Replication operations

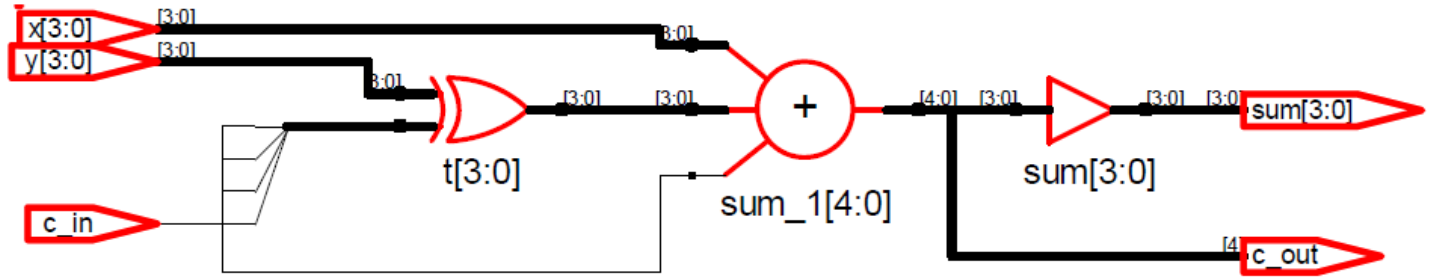
- Concatenation operators
  - The operands must be sized.
  - Operands can be scalar nets or registers, vector nets or registers, bit-select, part-select, or sized constants.
  - Example: y = {a, b[0], c[1]};

Symbol	Operation
{ , }	Concatenation
{const_expr{}}	Replication

- Replication operators
  - They specify how many times to replicate the number inside the braces.
  - Example: y = {a, 4{b[0]}, c[1]};

Example: A 4-bit two's complement adder

```
module twos_adder(x, y, c_in, sum, c_out);
// I/O port declarations
input    [3:0] x, y;    // declare as a 4-bit array
input    c_in;
output   [3:0] sum;   // declare as a 4-bit array
output   c_out;
wire     [3:0] t;       // outputs of xor gates

// Specify the function of a two's complement adder
   assign t = y ^ {4{c_in}};
   assign {c_out, sum} = x + t + c_in;
endmodule
```



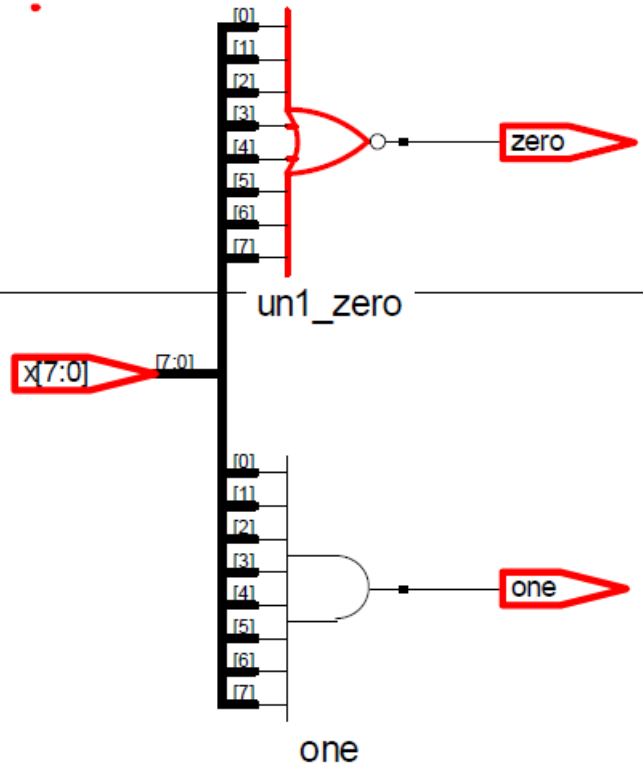
Reduction operators

- Reduction operators
  - perform only on one vector operand.
  - carry out a bit-wise operation on a single vector operand and yield a 1-bit result.
  - work bit by bit from right to left.

Symbol	Operation
&	Reduction and
~&	Reduction nand
|	Reduction or
~|	Reduction nor
^	Reduction exclusive or
~^, ^~	Reduction exclusive nor

Example: An All-Bit-Zero/One detector

```
module all_bit_01_detector_reduction(x, zero,one);
// I/O port declarations
input    [7:0] x;
output   zero, one;
// dataflow modeling
   assign zero = ~(|x);  // all-bit zero detector
   assign one = &x;      // all-bit one detector
endmodule
```



Logical, relational Operators

- Logical operators
  - They always evaluate to a 1-bit value, 0, 1, or x.
  - If any operand bit is x or z, it is equivalent to x and treated as a false condition by simulators.

Symbol	Operation
!	Logical negation
&&	Logical and
||	Logical or

- Relational operators
  - They return logical value 1 if the expression is true and 0 if the expression is false.
  - The expression takes a value x if there are any unknown (x) or z bits in the operands.

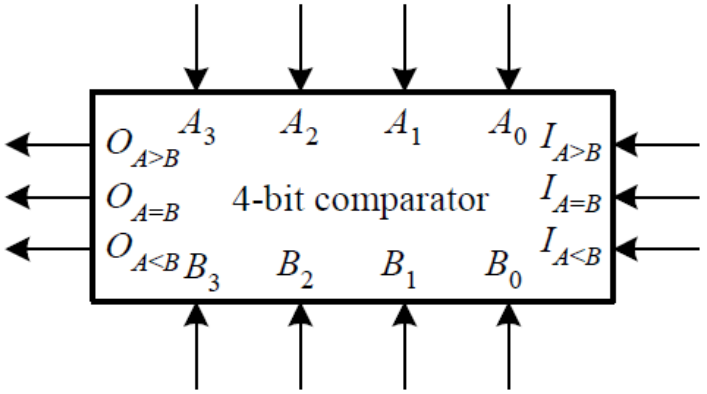
Symbol	Operation
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal

Equality operations

- Equality operators
  - compare the two operands bit by bit, with zero filling if the operands are of unequal length.
  - return logical value 1 if the expression is true and 0 if the expression is false.
- The operators (==, !=) yield an x if either operand has x or z in its bits.
- The operators (===, !==) yield a 1 if the two operands match exactly and 0 if the two operands not match exactly.

Symbol	Operation
==	Logical equality
!=	Logical inequality
===	Case equality
!==	Case inequality

Example: A 4-b Magnitude Comparator




```
module four_bit_comparator(Iagtb, Iaeqb, Ialtb, a, b, Oagtb, Oaeqb, Oaltb);
// I/O port declarations
input    [3:0] a, b;
input    Iagtb, Iaeqb, Ialtb;
output  Oagtb, Oaeqb, Oaltb;
// dataflow modeling using relation operators
   assign Oaeqb = (a == b) && (Iaeqb == 1);  // equality
   assign Oagtb = (a > b) || ((a == b)&& (Iagtb == 1));  // greater than
   assign Oaltb = (a < b) || ((a == b)&& (Ialtb == 1));   // less than
endmodule
```

Shift operators

- Logical shift operators
  - >> operator: logical right shift
  - << operator: logical left shift
  - The vacant bit positions are filled with zeros.
- Arithmetic shift operators
  - >>> operator: arithmetic right shift
  - The vacant bit positions are filled with the MSBs (sign bits).
  - <<< operator: arithmetic left shift
  - The vacant bit positions are filled with zeros.

Symbol	Operation
>>	Logical right shift
<<	Logical left shift
>>>	Arithmetic right shift
<<<	Arithmetic left shift

Example: Shift operators

```
// example to illustrate logic and arithmetic shifts
module arithmetic_shift(x,y,z);
input  signed [3:0] x;
output [3:0] y;
output signed [3:0] z;
assign y = x >> 1;   // logical right shift
assign z = x >>> 1; // arithmetic right shift
endmodule
```

Note that: net variables x and z must be declared with the keyword signed. Replaced net variable with unsigned net (i.e., remove the keyword signed) and see what happens.

The Conditional Operator

- Conditional Operator

  Usage: condition_expr ? true_expr: false_expr;

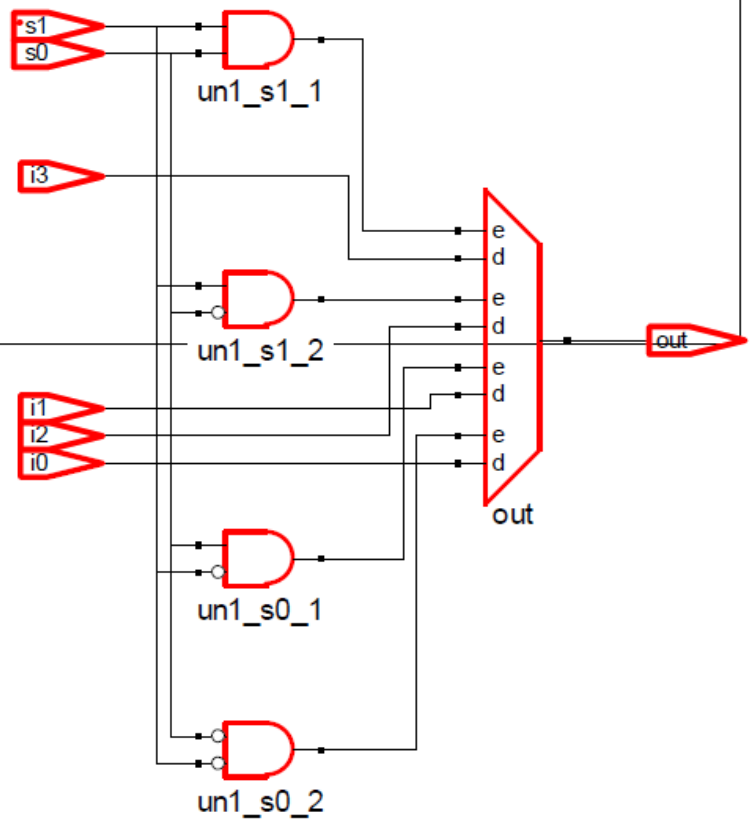
  - The condition_expr is evaluated first.
  - If the result is true then the true_expr is executed; otherwise the false_expr is evaluated.
    - if (condition_expr) true_expr;
    - else false_expr;

- a 2-to-1 multiplexer

```
assign out = selection ? in_1: in_0;
```

Example: A 4-to-1 MUX

```
module mux4_to_1_cond (i0, i1, i2, i3, s1, s0, out);
// Port declarations from the I/O diagram
input   i0, i1, i2, i3;
input   s1, s0;
output out;
// Using conditional operator (?:)
assign out = s1 ? ( s0 ? i3 : i2) : (s0 ? i1 : i0) ;
endmodule
```



Road map

Introduction to RISC-V

Verilog HDL
Dataflow modelling

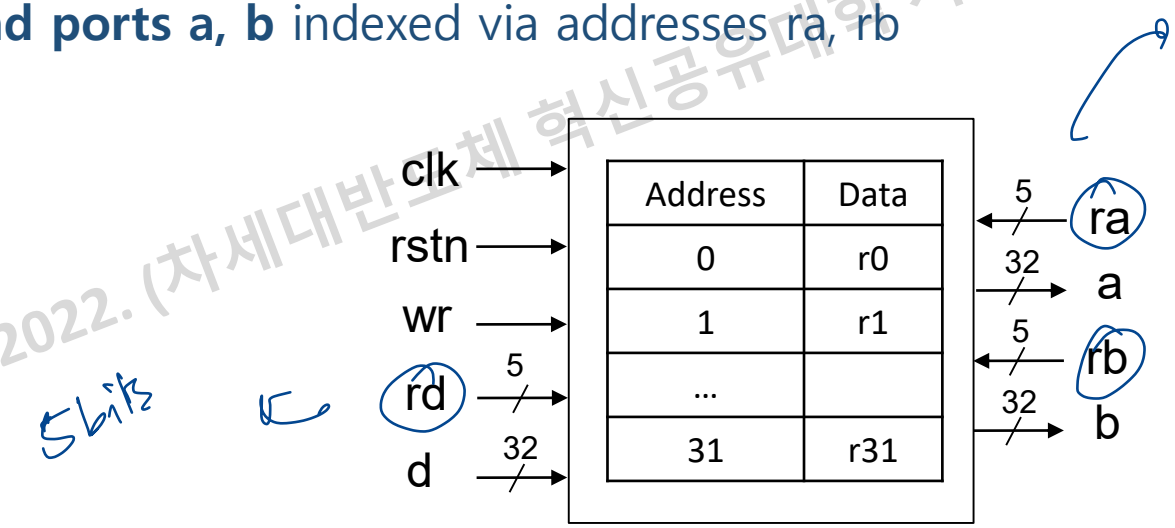
Labs

Lab 1: Register file

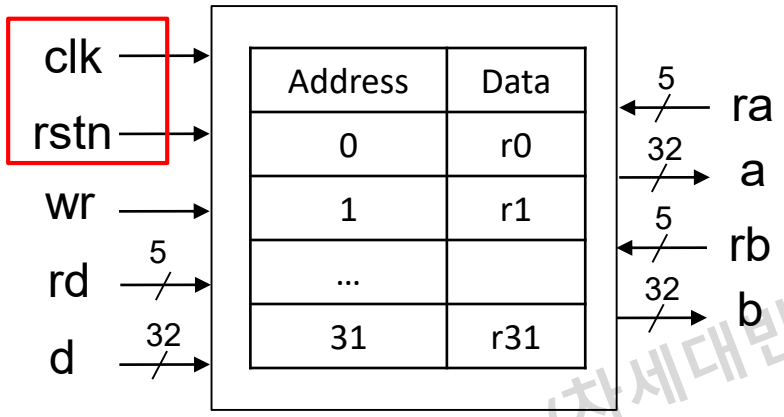
- Lab 1: Design Register file
  - Implement a register file module
  - Create a test bench
  - Run simulation
  - Show the output result

Register file (riscv_regfile.v)

- 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



Clock and reset ports (riscv_regfile.v)

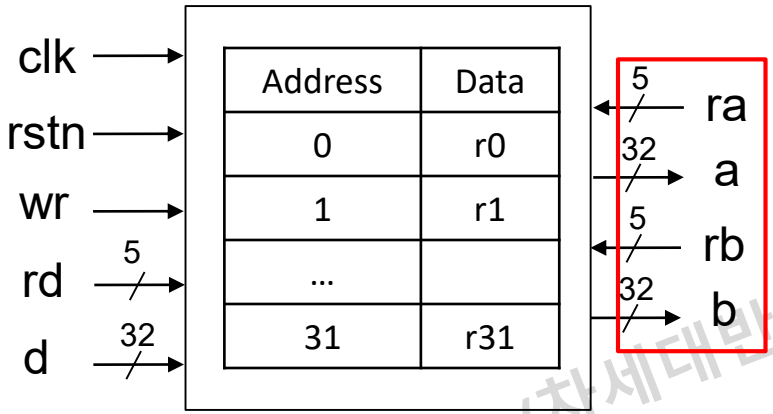



```
module riscv_regfile                ** Input / output ports definition
(                                      : module input/output could be connected to the other module
                                       : Input controlled by designer testbench for simulation.
                                       : output monitored by testbench for funtional verification.
          input    clk_i
         ,input    rstn_i
         ,input  [ 31:0]  rd0_value_i    // data to write
         ,input  [  4:0]  rd0_i          // address to write
         ,input           wr             // write enable signal
         ,input  [  4:0]  ra0_i          // address to read
         ,input  [  4:0]  rb0_i          // address to read
         ,output [ 31:0]  ra0_value_o  // read data from address ra0_i
         ,output [ 31:0]  rb0_value_o  // read data from address rb0_i
);
```

Write ports (riscv_regfile.v)

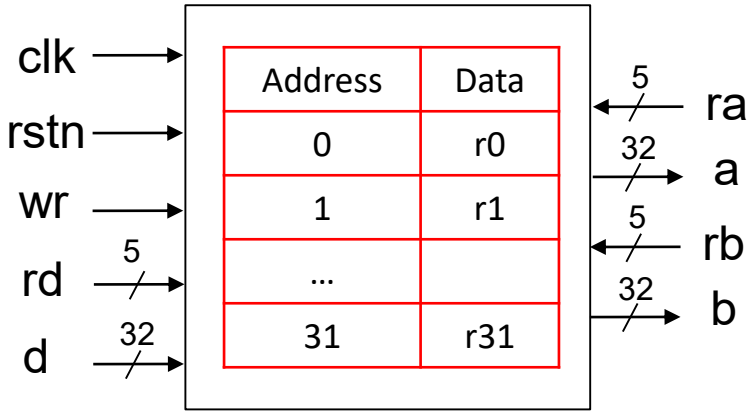

```
module riscv_regfile          ** Input / output ports definition
(                                : module input/output could be connected to the other module
                                 : Input controlled by designer testbench for simulation.
                                 : output monitored by testbench for funtional verification.
          input   clk_i
          ,input  rstn_i
          ,input  [ 31:0]  rd0_value_i   // data to write
          ,input  [  4:0]  rd0_i         // address to write
          ,input           wr            // write enable signal
          ,input  [  4:0]  ra0_i         // address to read
          ,input  [  4:0]  rb0_i         // address to read
          ,output [ 31:0]  ra0_value_o  // read data from address ra0_i
          ,output [ 31:0]  rb0_value_o  // read data from address rb0_i
);
```

Read ports (riscv_regfile.v)




```
module riscv_regfile          ** Input / output ports definition
(                                : module input/output could be connected to the other module
                                 : Input controlled by designer testbench for simulation.
                                 : output monitored by testbench for funtional verification.
        input   clk_i
       ,input   rstn_i
       ,input  [ 31:0]  rd0_value_i   // data to write
       ,input  [  4:0]  rd0_i         // address to write
       ,input           wr            // write enable signal
       ,input  [  4:0]  ra0_i         // address to read
       ,input  [  4:0]  rb0_i         // address to read
       ,output [ 31:0]  ra0_value_o  // read data from address ra0_i
       ,output [ 31:0]  rb0_value_o  // read data from address rb0_i
);
```

Internal signals (riscv_regfile.v)




```
reg [31:0] reg_r1_q;
reg [31:0] reg_r2_q;
...
reg [31:0] reg_r31_q;
```

internally used registers definition

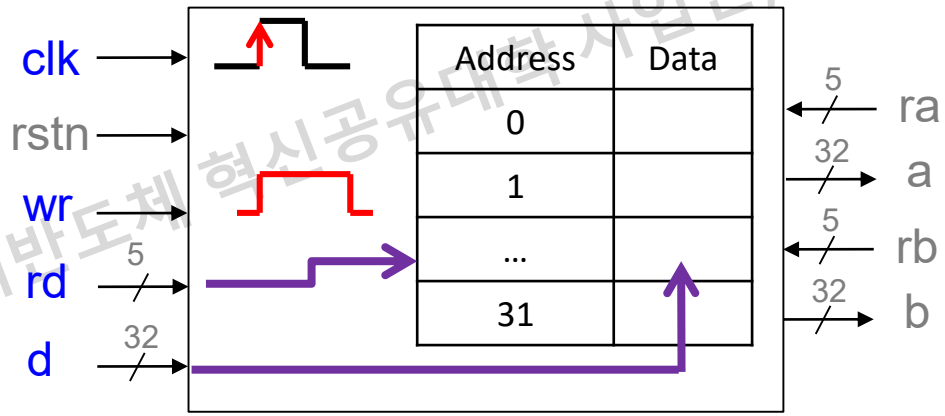
```
wire [31:0] x0_zero_w = 32'b0;
wire [31:0] x1_ra_w   = reg_r1_q;
...
wire [31:0] x31_t6_w  = reg_r31_q;
```

wire assignments using different names

#	Name	Usage
r0	zero	Hard-wired zero
r1	ra	Return address
r2	sp	Stack pointer
r3	gp	Global pointer
r4	tp	Thread pointer
r5~r7	t0~t2	Temporaries (Caller-save registers)
r8	s0/fp	Saved registers/ Frame pointer
r9	s1	Saved registers
r10~r17	a0~a8	Function arguments
r18~r26	s2~s11	Temporaries (Caller-save registers)
r28~r31	t3~t6	Temporaries (Caller-save registers)

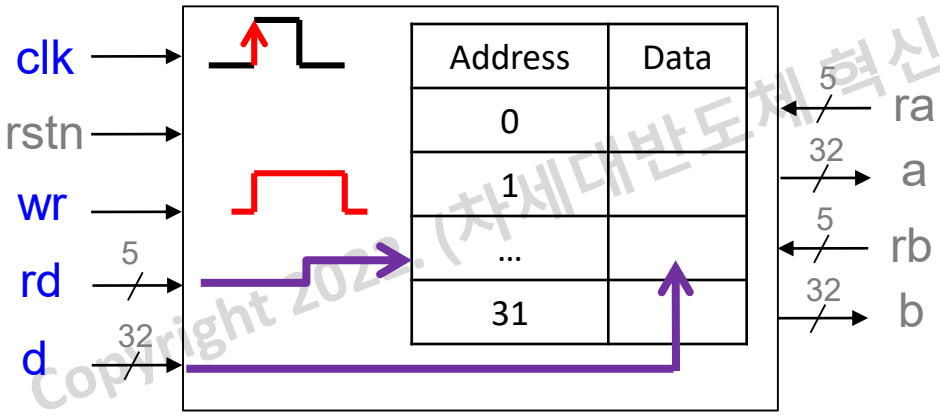
Write operation

- Sequential logic
- Pseudocode
  - If wr == 1 && positive edge of clk
    - regs[rd] ← d



Write operation (riscv_regfile.v)

- Use an "always" block to represent write operations
  - Reset: assign a register to "0"
  - Use "if"
    - To check the write enable flag
    - To check the register address



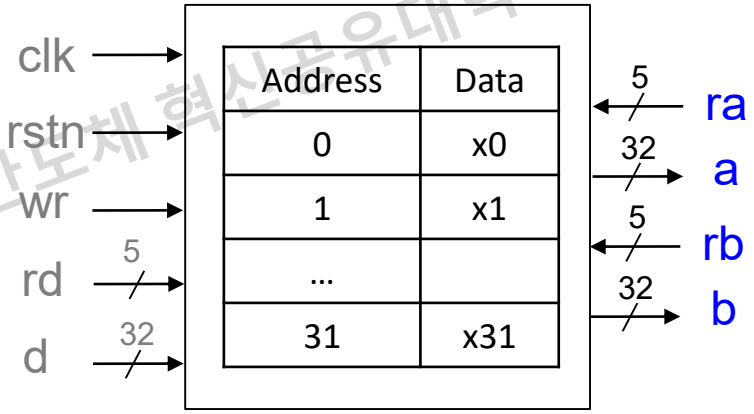

```
always @ (posedge clk_i or negedge rstn_i) begin
  if (~rstn_i) begin
     reg_r1_q      <= 32'h00000000;
     reg_r2_q      <= 32'h00000000;
      ...
     reg_r31_q     <= 32'h00000000;
  end
  else begin
     if (wr) begin
                  if    (rd0_i == 5'd1) reg_r1_q <= rd0_value_i;
                  if    (rd0_i == 5'd2) reg_r2_q <= rd0_value_i;
                  ...
                  if    (rd0_i == 5'd31) reg_r31_q <= rd0_value_i;
     end
  end
end
```

reset value "0" for initialization

write 32bits new data to new address when write enabled

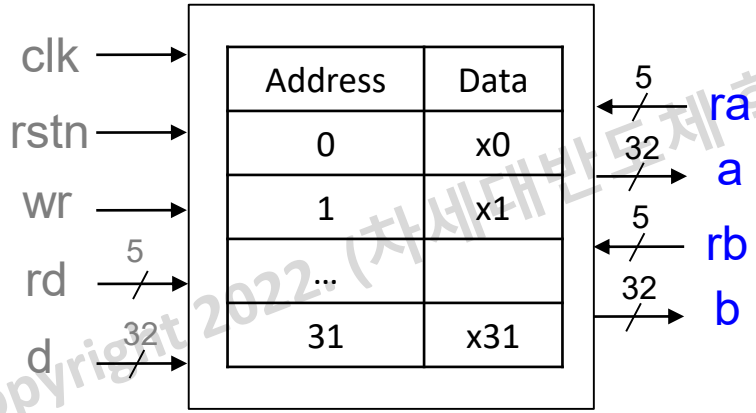
Read operation (riscv_regfile.v)

- Combinational Logic
- Pseudocode
  - Input: addresses ra and rb
  - Internal signals: regs[0:31]
  - Output
    - a ← regs[ra]
    - b ← regs[rb]



Read operation (riscv_regfile.v)

- There are two read ports which use two address ra and rb.
- Use a "case" structure to check a given read address




```
reg [31:0] ra0_value_r;
reg [31:0] rb0_value_r;
always @ * begin
    case (ra0_i)
    5'd1: ra0_value_r = reg_r1_q;
    5'd2: ra0_value_r = reg_r2_q;
    ...
    5'd31: ra0_value_r = reg_r31_q;
    default : ra0_value_r = 32'h00000000;
    endcase

    case (rb0_i)
    5'd1: rb0_value_r = reg_r1_q;
    5'd2: rb0_value_r = reg_r2_q;
    ...
    5'd31: rb0_value_r = reg_r31_q;
    default : rb0_value_r = 32'h00000000;
    endcase
  end

  assign ra0_value_o = ra0_value_r;
  assign rb0_value_o = rb0_value_r;

endmodule     //module description ends with "endmodule"
```

- asynchronous read by a given read address
- default value set to zero

read data outputs based on an address

signal assignment for output ports

Test bench (riscv_regfile_tb.v)

- riscv_regfile_tb.v
  - Signals
  - Unit under test: riscv_regfile.v
  - Clock module
  - Reset module
  - Test cases

```
`timescale 1ns / 100ps
module riscv_regfile_tb;
                wire [31:0] ra0_value_o, rb0_value_o;   //**** module output
                reg [4:0] ra0_i, rb0_i;    //**** module inputs
                reg [4:0] rd0_i;
                reg [31:0] rd0_value_i;
                reg wr, clk, rstn;
```

```
riscv_regfile u_riscv_regfile
 (              .clk_i(clk),
                .rstn_i(rstn),
                .ra0_i(ra0_i),
                .rb0_i(rb0_i),
                .rd0_i(rd0_i),
                .rd0_value_i(rd0_value_i),
                .wr(wr),
                .ra0_value_o(ra0_value_o),
                .rb0_value_o(rb0_value_o)
);
```

Unit under test

```
initial forever
                #5 clk = ~clk;
initial begin
                clk=1;
                rstn = 0;
                #20             rstn = 1;
```

Clock and reset signals generation

Test bench (riscv_regfile_tb.v)

- Test case
  - Write operation
    - Set values for a write address rd
    - Set a write-enable signal (wr)
  - Read operation
    - Set values for the read addresses ra and rb
    - Monitor the outputs at ra_value, rb_value

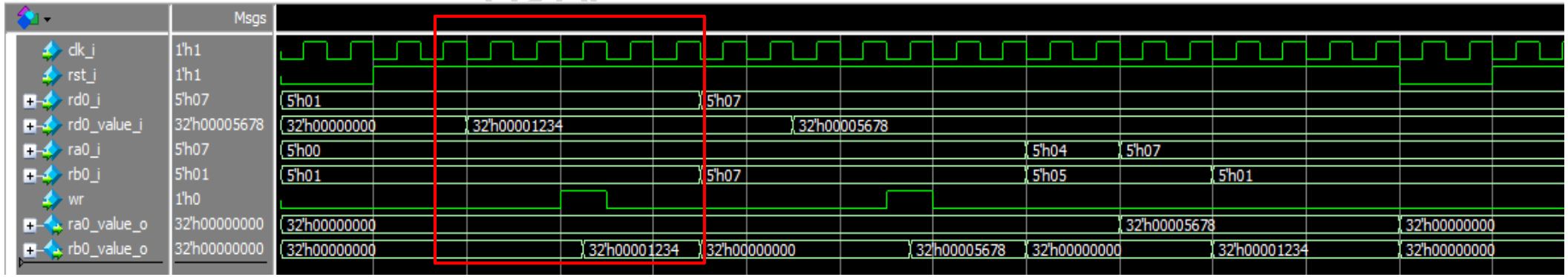
```
    ra0_i = 5'b00000;                   // Select R0
    rb0_i = 5'b00001;                   // Select R1

    #20     rd0_value_i = 32'h1234;   // data to write
    #20     wr = 1;                     // write enable
            rd0_i =  5'b00001;          // addressing R1 to write
    #10     wr = 0;                          write operations

    #20     rb0_i = 5'b00111;           // addressing R7 to read
            rd0_i = 5'b00111;           // new address to write
    #20     rd0_value_i = 32'h5678;   // new data to write
    #20     wr = 1;                     // write enable
    #10     wr = 0;                     // disabled
```

```
    #20     ra0_i = 5'b00100;           // Read R4
            rb0_i = 5'b00101;           // Read R5
                                                read operations
    #20     ra0_i = 5'b00111;           // Read R7
    #20     rb0_i = 5'b00001;           // Read R1

    #20
    #20     rstn = 0;
    #20     rstn = 1;
  end
endmodule                                   // testbench module ends
```

Waveform

- Write operation
  - Write to R1
    - rd0_valule ← new value
    - rd0 ← new address (R1)

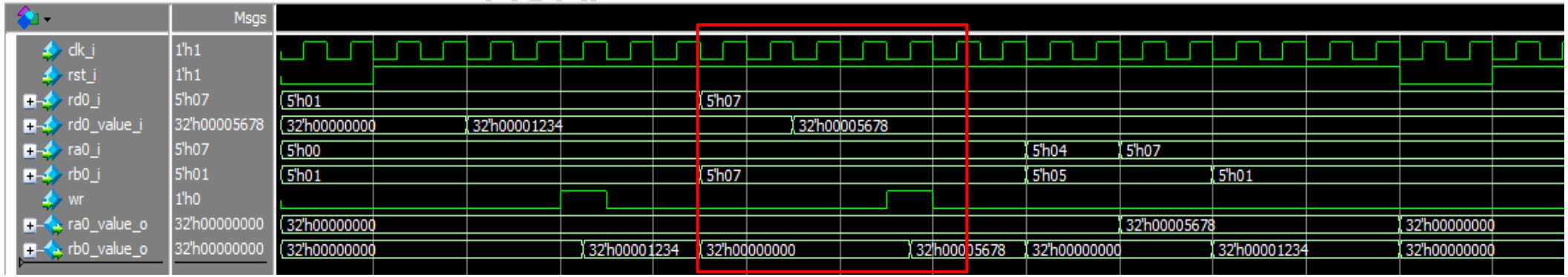
```
// WRITE
      ra0_i = 5'b00000;                  // Select R0
      rb0_i = 5'b00001;                  // Select R1

      #20     rd0_value_i = 32'h1234;
      #20     wr = 1;                    // write to R1
              rd0_i =  5'b00001;
      #10     wr = 0;

      #20     rb0_i = 5'b00111;          // Select R7
              rd0_i =  5'b00111;
      #20     rd0_value_i = 32'h5678;

      #20     wr = 1;                    // write to R7
      #10     wr = 0;
```



Waveform

- Write operation
  - Write to R7
    - rd0_valule ← new value
    - rd0 ← new address (R7)

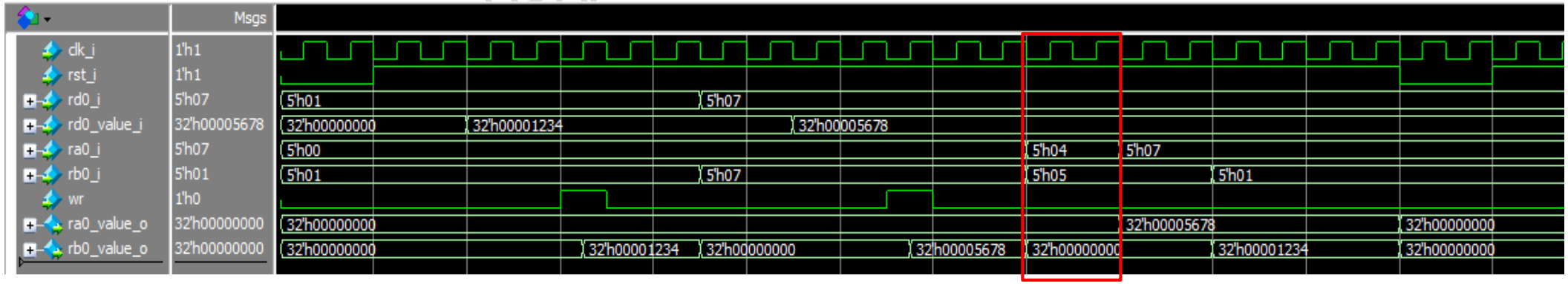
```
// WRITE
        ra0_i = 5'b0000;                    // Select R0
        rb0_i = 5'b0001;                    // Select R1

        #20     rd0_value_i = 32'h1234;
        #20     wr = 1;                     // write to R1
                rd0_i =  5'b0001;
        #10     wr = 0;

        #20     rb0_i = 5'b0111;            // Select R7
                rd0_i =  5'b0111;
        #20     rd0_value_i = 32'h5678;

        #20     wr = 1;                     // write to R7
        #10     wr = 0;
```



Waveform

- Read operation
  - Read R4, R5
    - ra0_i ← new address (R4)
    - rb0_i ← new address (R5)

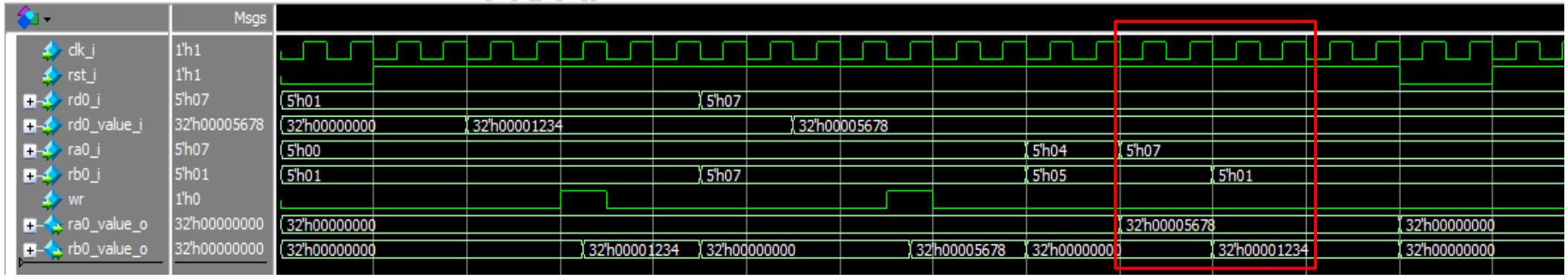
```
// READ
                    //////////// READ //////////
      #20     ra0_i = 5'b00100;          // Read R4
              rb0_i = 5'b00101;          // Read R5
      #20     ra0_i = 5'b00111;          // Read R7
      #20     rb0_i = 5'b00001;          // Read R1
              ////////////////////////////
      #20
      #20     rstn = 0;
      #20     rstn = 1;
```



Waveform

- Read operation
  - Read R7, R1
    - ra0_i ← new address (R7)
    - rb0_i ← new address (R1)

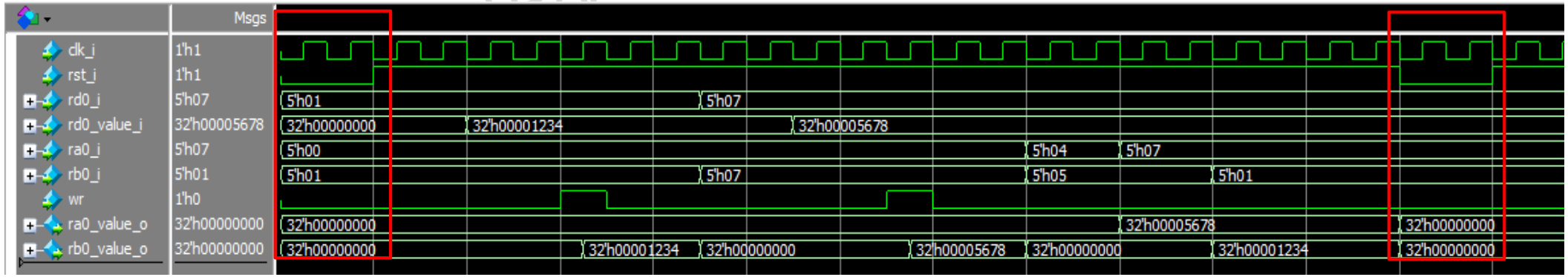
```
// READ
                    //////////// READ //////////
        #20         ra0_i = 5'b00100;           // Read R4
                    rb0_i = 5'b00101;           // Read R5
        #20         ra0_i = 5'b00111;           // Read R7
        #20         rb0_i = 5'b00001;           // Read R1
                    ////////////////////////////
        #20
        #20         rstn = 0;
        #20         rstn = 1;
```



Reset

- Often occurs when starting a system
  - ~ restart a PC
  - Registers are assigned to default values
- Reset
  - Reset register values
    - R7, R1

```
// READ
                    //////////// READ //////////
        #20         ra0_i = 5'b00100;          // Read R4
                    rb0_i = 5'b00101;          // Read R5
        #20         ra0_i = 5'b00111;          // Read R7
        #20         rb0_i = 5'b00001;          // Read R1
                    ////////////////////////////
        #20
        #20         rstn = 0;
        #20         rstn = 1;
```



TODO: …

- Implement riscv_regfile.v by completing the missing codes
  - Write operation
- Do a simulation with time = 300ns
- Show the output waveform

```
always @ (posedge clk_i or negedge rstn_i)

//-----------------------------------------
//**** reset value "0" for initialization
//     : all defined registers set to zero
//-----------------------------------------
    if (!rstn_i) begin
        reg_r1_q        <= 32'h00000000;
        reg_r2_q        <= 32'h00000000;
        reg_r3_q        <= 32'h00000000;
        /*Insert your code */
        //reg_r4_q
        //reg_r5_q
        //reg_r6_q
        //reg_r7_q
        /*******************/
        reg_r8_q        <= 32'h00000000;
        reg_r9_q        <= 32'h00000000;
        reg_r10_q       <= 32'h00000000;
        reg_r11_q       <= 32'h00000000;
        reg_r12_q       <= 32'h00000000;
        reg_r13_q       <= 32'h00000000;
        reg_r14_q       <= 32'h00000000;
        reg_r15_q       <= 32'h00000000;
        reg_r16_q       <= 32'h00000000;
        reg_r17_q       <= 32'h00000000;
        reg_r18_q       <= 32'h00000000;
        reg_r19_q       <= 32'h00000000;
        reg_r20_q       <= 32'h00000000;
        reg_r21_q       <= 32'h00000000;
        reg_r22_q       <= 32'h00000000;
        reg_r23_q       <= 32'h00000000;
        reg_r24_q       <= 32'h00000000;
        reg_r25_q       <= 32'h00000000;
        reg_r26_q       <= 32'h00000000;
        reg_r27_q       <= 32'h00000000;
        reg_r28_q       <= 32'h00000000;
        reg_r29_q       <= 32'h00000000;
        reg_r30_q       <= 32'h00000000;
        reg_r31_q       <= 32'h00000000;
    end
```

```
    else begin
        if(wr) begin
            if      (rd0_i == 5'd1) reg_r1_q <= rd0_value_i;
            if      (rd0_i == 5'd2) reg_r2_q <= rd0_value_i;
            if      (rd0_i == 5'd3) reg_r3_q <= rd0_value_i;
            if      (rd0_i == 5'd4) reg_r4_q <= rd0_value_i;
            if      (rd0_i == 5'd5) reg_r5_q <= rd0_value_i;
            if      (rd0_i == 5'd6) reg_r6_q <= rd0_value_i;
            if      (rd0_i == 5'd7) reg_r7_q <= rd0_value_i;
            if      (rd0_i == 5'd8) reg_r8_q <= rd0_value_i;
            if      (rd0_i == 5'd9) reg_r9_q <= rd0_value_i;
            if      (rd0_i == 5'd10) reg_r10_q <= rd0_value_i;
            if      (rd0_i == 5'd11) reg_r11_q <= rd0_value_i;
            if      (rd0_i == 5'd12) reg_r12_q <= rd0_value_i;
            if      (rd0_i == 5'd13) reg_r13_q <= rd0_value_i;
            if      (rd0_i == 5'd14) reg_r14_q <= rd0_value_i;
            if      (rd0_i == 5'd15) reg_r15_q <= rd0_value_i;
            /*Insert your code */
            //if      (rd0_i == 5'd16)
            //if      (rd0_i == 5'd17)
            //if      (rd0_i == 5'd18)
            //if      (rd0_i == 5'd19)
            /*******************/
            if      (rd0_i == 5'd20) reg_r20_q <= rd0_value_i;
            if      (rd0_i == 5'd21) reg_r21_q <= rd0_value_i;
            if      (rd0_i == 5'd22) reg_r22_q <= rd0_value_i;
            if      (rd0_i == 5'd23) reg_r23_q <= rd0_value_i;
            if      (rd0_i == 5'd24) reg_r24_q <= rd0_value_i;
            if      (rd0_i == 5'd25) reg_r25_q <= rd0_value_i;
            if      (rd0_i == 5'd26) reg_r26_q <= rd0_value_i;
            if      (rd0_i == 5'd27) reg_r27_q <= rd0_value_i;
            if      (rd0_i == 5'd28) reg_r28_q <= rd0_value_i;
            if      (rd0_i == 5'd29) reg_r29_q <= rd0_value_i;
            if      (rd0_i == 5'd30) reg_r30_q <= rd0_value_i;
            if      (rd0_i == 5'd31) reg_r31_q <= rd0_value_i;
        end
    end
```

TODO: …

- Implement riscv_regfile.v by completing the missing codes
  - Read operation
- Do a simulation with time = 300ns
- Show the output waveform

```
reg [31:0] ra0_value_r;
reg [31:0] rb0_value_r;
always @ *
begin
    case (ra0_i)                                    //***
                                                    //
    5'd1: ra0_value_r = reg_r1_q;                   //****
    5'd2: ra0_value_r = reg_r2_q;
    5'd3: ra0_value_r = reg_r3_q;
    5'd4: ra0_value_r = reg_r4_q;
    5'd5: ra0_value_r = reg_r5_q;
    5'd6: ra0_value_r = reg_r6_q;
    5'd7: ra0_value_r = reg_r7_q;
    5'd8: ra0_value_r = reg_r8_q;
    5'd9: ra0_value_r = reg_r9_q;
    /*Insert your code */
    //5'd10:
    //5'd11:
    //5'd12:
    //5'd13:
    /********************/
    5'd14: ra0_value_r = reg_r14_q;
    5'd15: ra0_value_r = reg_r15_q;
    5'd16: ra0_value_r = reg_r16_q;
    5'd17: ra0_value_r = reg_r17_q;
    5'd18: ra0_value_r = reg_r18_q;
    5'd19: ra0_value_r = reg_r19_q;
    5'd20: ra0_value_r = reg_r20_q;
    5'd21: ra0_value_r = reg_r21_q;
    5'd22: ra0_value_r = reg_r22_q;
    5'd23: ra0_value_r = reg_r23_q;
    5'd24: ra0_value_r = reg_r24_q;
    5'd25: ra0_value_r = reg_r25_q;
    5'd26: ra0_value_r = reg_r26_q;
    5'd27: ra0_value_r = reg_r27_q;
    5'd28: ra0_value_r = reg_r28_q;
    5'd29: ra0_value_r = reg_r29_q;
    5'd30: ra0_value_r = reg_r30_q;
    5'd31: ra0_value_r = reg_r31_q;
    default : ra0_value_r = 32'h00000000;
    endcase                                         //
```

```
case (rb0_i)
5'd1: rb0_value_r = reg_r1_q;
5'd2: rb0_value_r = reg_r2_q;
5'd3: rb0_value_r = reg_r3_q;
5'd4: rb0_value_r = reg_r4_q;
5'd5: rb0_value_r = reg_r5_q;
5'd6: rb0_value_r = reg_r6_q;
/*Insert your code */
//5'd7:
//5'd8:
//5'd9:
//5'd10:
/********************/
5'd11: rb0_value_r = reg_r11_q;
5'd12: rb0_value_r = reg_r12_q;
5'd13: rb0_value_r = reg_r13_q;
5'd14: rb0_value_r = reg_r14_q;
5'd15: rb0_value_r = reg_r15_q;
5'd16: rb0_value_r = reg_r16_q;
5'd17: rb0_value_r = reg_r17_q;
5'd18: rb0_value_r = reg_r18_q;
5'd19: rb0_value_r = reg_r19_q;
5'd20: rb0_value_r = reg_r20_q;
5'd21: rb0_value_r = reg_r21_q;
5'd22: rb0_value_r = reg_r22_q;
5'd23: rb0_value_r = reg_r23_q;
5'd24: rb0_value_r = reg_r24_q;
5'd25: rb0_value_r = reg_r25_q;
5'd26: rb0_value_r = reg_r26_q;
5'd27: rb0_value_r = reg_r27_q;
5'd28: rb0_value_r = reg_r28_q;
5'd29: rb0_value_r = reg_r29_q;
5'd30: rb0_value_r = reg_r30_q;
5'd31: rb0_value_r = reg_r31_q;
default : rb0_value_r = 32'h00000000;
endcase
```

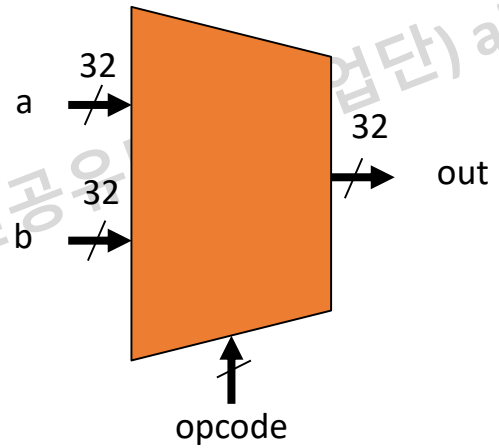
Lab 2: Arithmetic Logic Unit (ALU)

- Lab 2: Design an ALU
  - Implement an ALU module
  - Run simulation
  - Show the output result

ALU structure

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

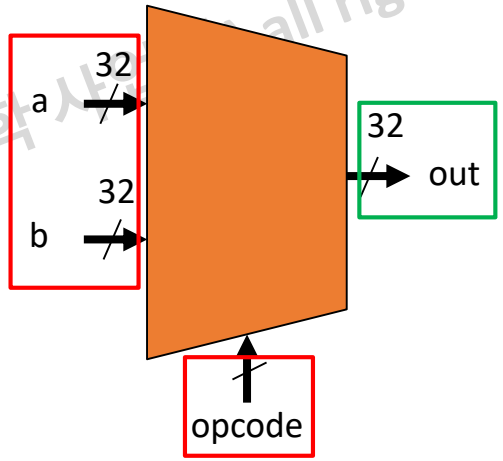
- Pseudocode
  - Case (opcode)
    - ADD
    - SUB
    - AND
    - OR
    - SLL
    - SRL
    - .....
  - Endcase



Port declaration (riscv_alu.v)

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

```
module riscv_alu
(
    // Inputs
    input  [ 3:0]  alu_op_i,
    input  [31:0]  alu_a_i,
    input  [31:0]  alu_b_i,

    // Outputs
    output [31:0]  alu_p_o
);
```



Logics (riscv_alu.v)

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

      .....
  - Endcase

```
always @ (alu_op_i or alu_a_i or alu_b_i or sub_res_w)
begin
   shift_right_fill_r = 16'b0;
   shift_right_1_r = 32'b0;
   ...
   shift_left_1_r = 32'b0;
   ...
   case (alu_op_i)
   `ALU_SHL :                    // Shift Left    shift left operation by 32bit operand b cases (shift left 1 to 31)
     begin
        if (alu_b_i[0] == 1'b1)
           shift_left_1_r = {alu_a_i[30:0],1'b0};
        else
           shift_left_1_r = alu_a_i;
           ...
     end
```

Shift operation

- Shift operations
  - Left: out = a << b
  - Right: out = a >> b
- Both a and b are 32-bit registers
- Question: What is the number of **possible outcomes**?
  - 0, 1, 2, …, 2^{32}-1

Shift operation

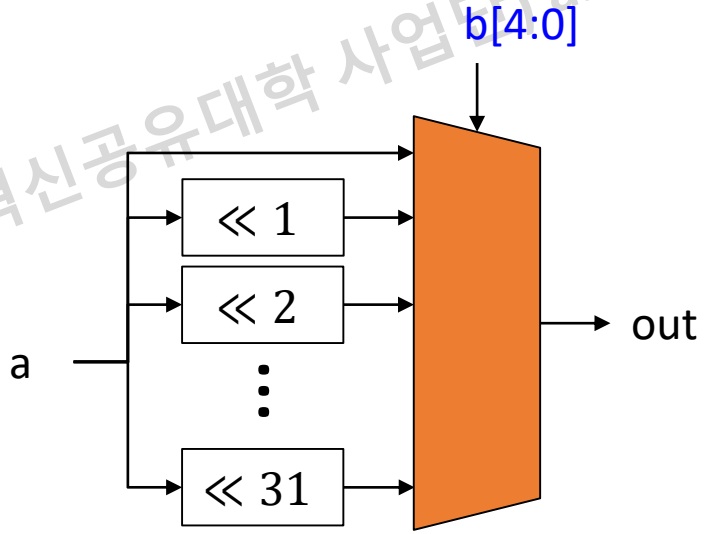
- Shift operations
  - Left: out = a << b
  - Right: out = a >> b
- Both a and b are 32-bit registers
- Question: What is the number of **possible outcomes**?
  - 0, 1, 2, …, ~~$2^{32}-1$~~ 31 because a is a **32-bit** number

Shift operation: Naïve implementation

- Shift operations
  - Left: out = a << b
  - Right: out = a >> b
- Both a and b are 32-bit registers
- Question: What is the number of possible outcomes?
  - 0, 1, 2, …, ~~$2^{32}-1$~~ 31 because a is a **32-bit** number

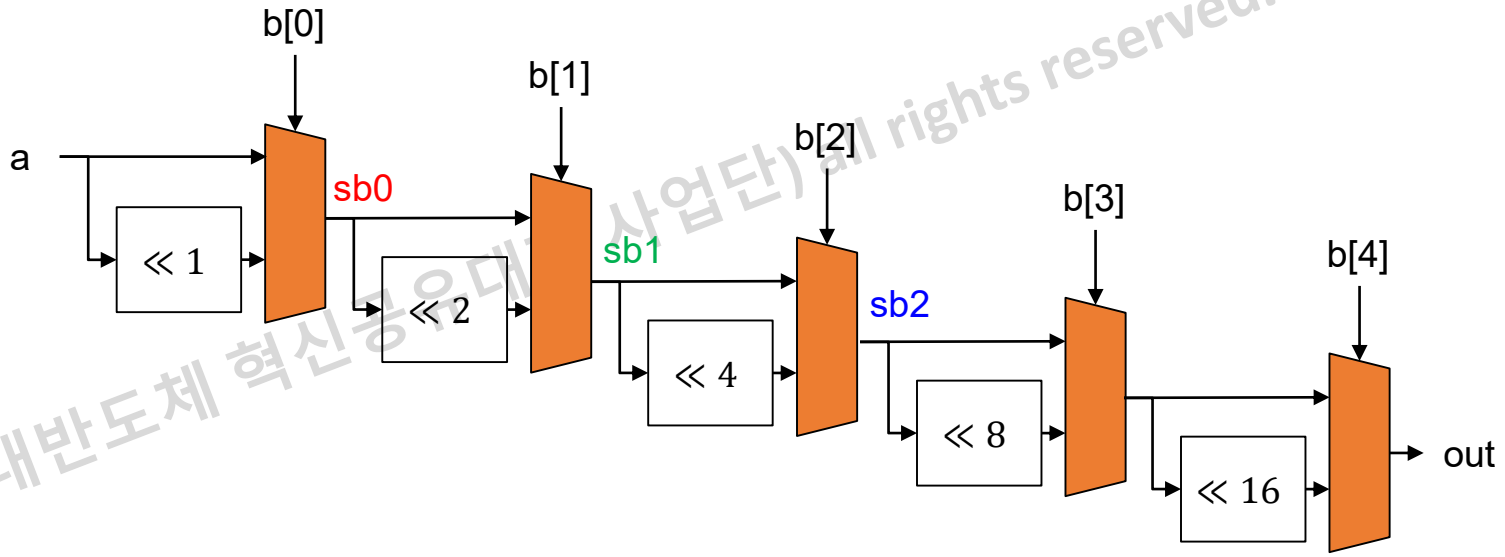
⇒b[31:5] != 0 → out = 0

- Pseudocode
  - Case (b[4:0])
    - 0
    - 1
    - …
    - 31
  - End case



Shift operation: Barrel implementation

- b in {0, 1, 2, …, 31} => b[4:0]
- Pseudocode for Barrel shift implementation
  - If b[0] == 1
    sb0 ← (a<<1)
  - Else
    sb0 ← a
  - If b[1] == 1
    sb1 ← (sb0<<2)
  - Else
    sb1 ← sb0
  - If b[2] == 1
    sb2 ← (sb1<<4)
  - Else
    sb2 ← sb1
    …



TODO: …

- Implement riscv_alu.v codes by completing some missing codes
- Do a simulation with time = 400ns
- Show the waveform

```
//------------------------------------------------
// Shift Right Logical and Shift Right arithmetic
//------------------------------------------------
`ALU_SRL,`ALU_SRA :
begin
    // Arithmetic shift? Fill with 1's if MSB set
    if (alu_a_i[31] == 1'b1 && alu_op_i == `ALU_SRA)
        shift_right_fill_r = 16'b1111111111111111;
    else
        shift_right_fill_r = 16'b0000000000000000;

    // Insert your code here
    //{{{
    //if (alu_b_i[0] == 1'b1)
    //    shift_right_1_r = /*Insert your code */
    //else
    //    shift_right_1_r = /*Insert your code */

    //if (alu_b_i[1] == 1'b1)
    //    shift_right_2_r = /*Insert your code */
    //else
    //    shift_right_2_r = /*Insert your code */
    //}}}
```

```
case (alu_op_i)
    //------------------------------------------------
    // Shift Left Logical
    //------------------------------------------------
    `ALU_SLL :
    begin
        if (alu_b_i[0] == 1'b1)
            shift_left_1_r = {alu_a_i[30:0],1'b0};
        else
            shift_left_1_r = alu_a_i;

        if (alu_b_i[1] == 1'b1)
            shift_left_2_r = {shift_left_1_r[29:0],2'b00};
        else
            shift_left_2_r = shift_left_1_r;

        // Insert your code here
        //{{{
        //if (alu_b_i[2] == 1'b1)
        //    shift_left_4_r = /*Insert your code */
        //else
        //    shift_left_4_r = /*Insert your code */

        //if (alu_b_i[3] == 1'b1)
        //    shift_left_8_r = /*Insert your code */
        //else
        //    shift_left_8_r = /*Insert your code */
        //}}}
        if (alu_b_i[4] == 1'b1)
            result_r = {shift_left_8_r[15:0],16'b0000000000000000};
        else
            result_r = shift_left_8_r;
    end
```

TODO: …

- Implement riscv_alu.v codes by completing some missing codes
- Do a simulation with time = 400ns
- Show the waveform

```
//------------------------------------------------
// Logical
//------------------------------------------------
`ALU_AND :
begin
    result_r      = (alu_a_i & alu_b_i);
end
`ALU_OR  :
begin
    //result_r      = /*Insert your code */
end
`ALU_XOR :
begin
    //result_r      = /*Insert your code */
end
//------------------------------------------------
// Comparision
//------------------------------------------------
`ALU_SLTU :  // Unsigned number
begin
    result_r      = (alu_a_i < alu_b_i) ? 32'h1 : 32'h0;
end
`ALU_SLT :   // Signed numbers
begin
    //Insert your code
    //{{{
    //if (alu_a_i[31] != alu_b_i[31])
    //    result_r  = /*Insert your code */
    //else
    //    result_r  = sub_res_w[31] ? 32'h1 : 32'h0;
    //}}}
end
```

Operation definition (riscv_defines.v)

- Operation definition
  - Shift, shift arithmetic
  - Arithmetic operations: ADD, SUB.
  - Logic operations: AND, OR, XOR
  - Compare operations: Less than, less than with signed.

```
`define ALU_ADD   4'd0      // Addition
`define ALU_SUB   4'd1      // Subtraction
`define ALU_AND   4'd2      // AND
`define ALU_OR    4'd3      // OR
`define ALU_XOR   4'd4      // XOR
`define ALU_SLT   4'd5      // Compare, Signed
`define ALU_SLTU  4'd6      // Compare, Unsigned
`define ALU_SLL   4'd7      // Shift Left Logical
`define ALU_SRL   4'd8      // Shift Right Logical
`define ALU_SRA   4'd9      // Shift Right Arithmetic
`define ALU_MULL  4'd10     // Multiplier Upper Half
`define ALU_MULH  4'd11     // Multiplier Upper Half
`define ALU_DIV   4'd12     // Divider, division
`define ALU_REM   4'd13     // Divider, quotient
`define ALU_AUIPC 4'd14     // Add Upper Imm to PC
`define ALU_IDLE  4'd15
```

Test bench (riscv_alu_tb.v)

```
module riscv_alu_tb ();
        reg rstn;
        reg clk;
        reg [3:0] alu_op_i;
        reg [31:0] alu_a_i
        reg [31:0] alu_b_i;
        wire [31:0] alu_p_o;
```

```
riscv_alu u_alu (
        .alu_op_i(alu_op_i),
        .alu_a_i(alu_a_i),
        .alu_b_i(alu_b_i),
        .alu_p_o(alu_p_o)
);
```

module instance for test bench

```
        parameter p=10;
        initial
        begin
                clk = 1'b0;
                forever #p clk = !clk;
        end

        initial begin
                rstn = 1'b0;            // negedge reset on
                #(4*p) rst = 1'b1;      // negedge reset off
        end
```

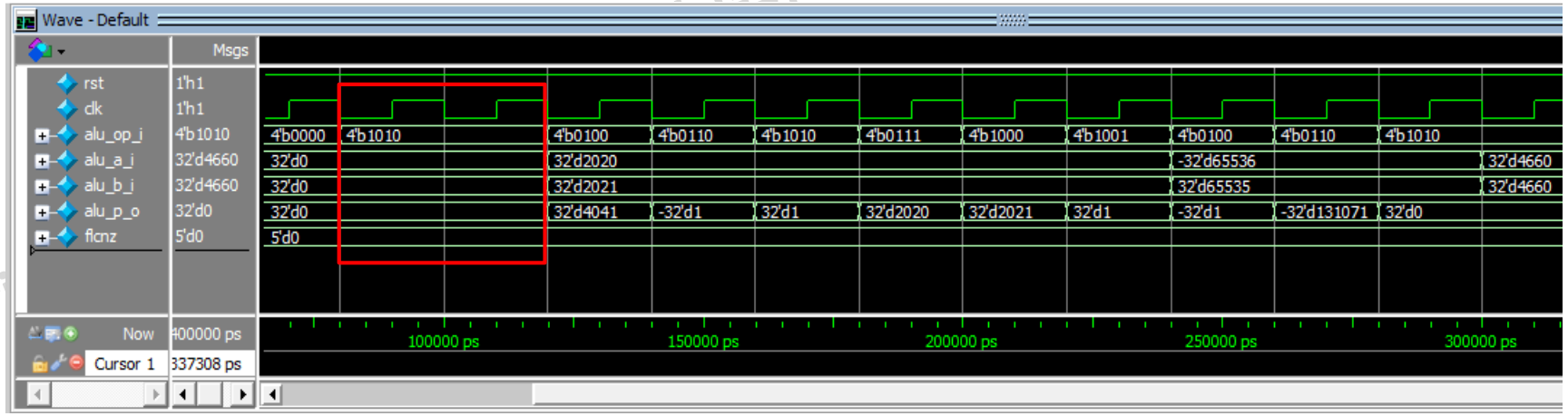
Test cases

```
initial begin
        #(8*p)      alu_a_i = 32'h0;
                    alu_b_i = 32'h0;
                    alu_op_i = `ALU_LESS_THAN;

        #(4*p)      alu_a_i = 32'd2020;
                    alu_b_i = 32'd2021;
                    alu_op_i = `ALU_ADD;
        #(2*p)      alu_op_i = `ALU_SUB;

        #(2*p)      alu_op_i = `ALU_AND;
        #(2*p)      alu_op_i = `ALU_OR;
        #(2*p)      alu_op_i = `ALU_XOR;

 end

endmodule
```

Waveform

- Insert operands (from Register File)
- Insert operations
  - Comparison

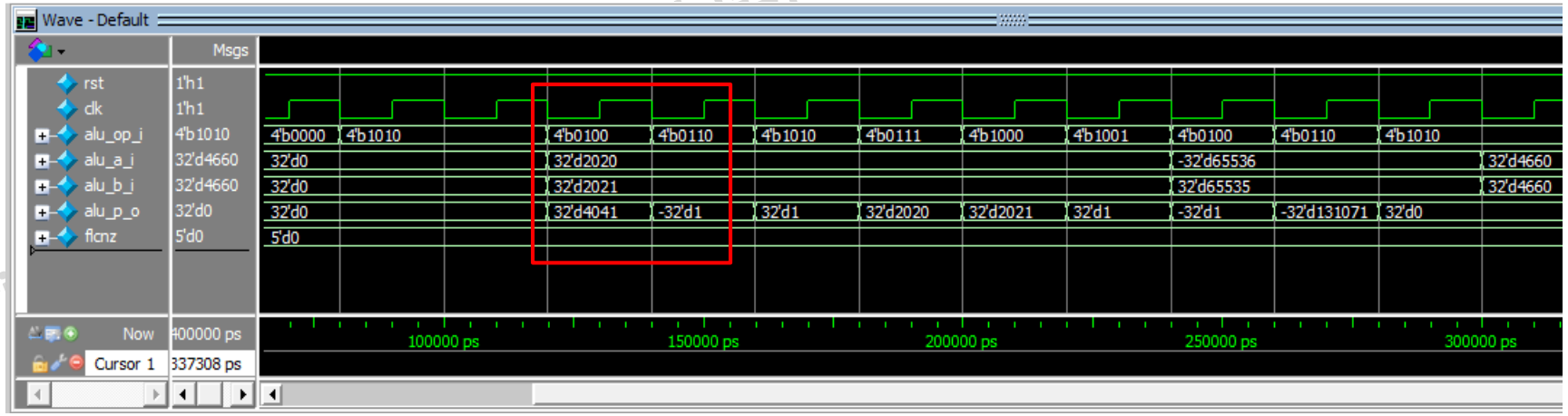
```
#(8*p)
      alu_a_i = 32'h0;
      alu_b_i = 32'h0;
      alu_op_i = `ALU_LESS_THAN;
#(4*p)
      alu_a_i = 32'd2020;
      alu_b_i = 32'd2021;
      alu_op_i = `ALU_ADD;
#(2*p) alu_op_i = `ALU_SUB;
#(2*p) alu_op_i = `ALU_AND;
#(2*p) alu_op_i = `ALU_OR;
#(2*p) alu_op_i = `ALU_XOR;
```



Waveform

- Insert operands (from Register File)
- Insert operations
  - Add
  - Subtraction

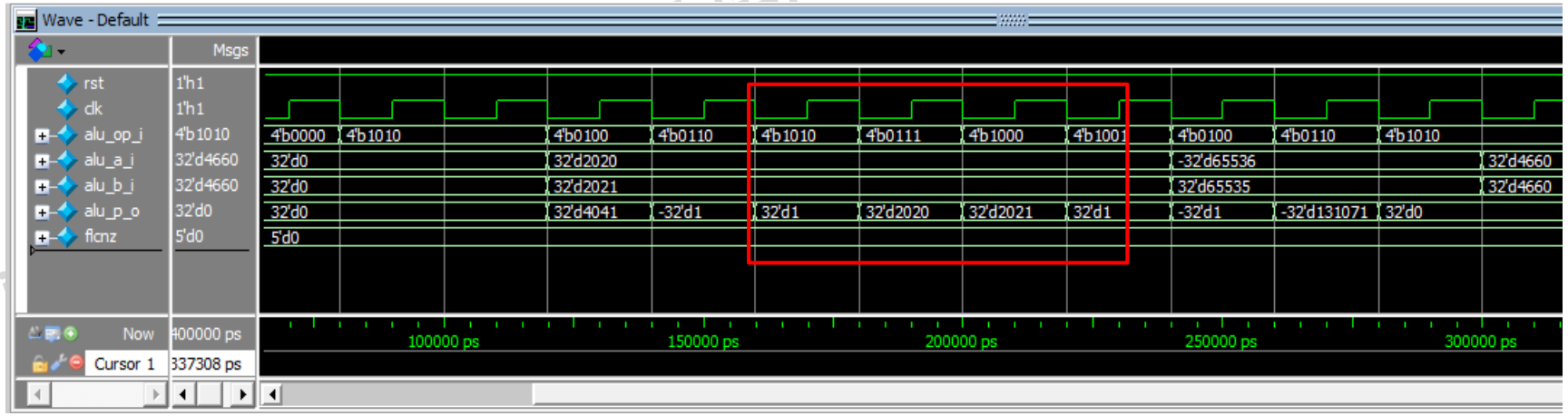
```
#(8*p)
      alu_a_i = 32'h0;
      alu_b_i = 32'h0;
      alu_op_i = `ALU_LESS_THAN;
#(4*p)
      alu_a_i = 32'd2020;
      alu_b_i = 32'd2021;
      alu_op_i = `ALU_ADD;
#(2*p) alu_op_i = `ALU_SUB;
#(2*p) alu_op_i = `ALU_AND;
#(2*p) alu_op_i = `ALU_OR;
#(2*p) alu_op_i = `ALU_XOR;
```



Waveform

- Insert operands (from Register File)
- Insert operations
  - AND
  - OR
  - XOR

```
#(8*p)
    alu_a_i = 32'h0;
    alu_b_i = 32'h0;
    alu_op_i = `ALU_LESS_THAN;
#(4*p)
    alu_a_i = 32'd2020;
    alu_b_i = 32'd2021;
    alu_op_i = `ALU_ADD;
#(2*p) alu_op_i = `ALU_SUB;
#(2*p) alu_op_i = `ALU_AND;
#(2*p) alu_op_i = `ALU_OR;
#(2*p) alu_op_i = `ALU_XOR;
```



Test bench

- Display the results of test cases.
  - $display: a function to print out a message
  - $realtime: a function to show a time stamp

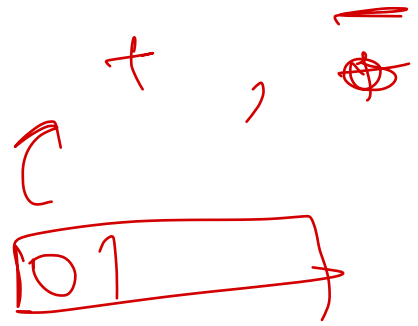
```
//------------------------------------------------------------------------------
// ALU operations
//------------------------------------------------------------------------------
#(8*p)
        alu_a_i = 32'h0000_2222;
        alu_b_i = 32'h0000_2222;
        alu_op_i = `ALU_SLTU;
#(p)    $display("T=%03t ns: %h < %h : %h (Unsigned)\n",$realtime/1000, alu_a_i, alu_b_i,alu_p_o);

// Addition
#(4*p)
        alu_a_i = 32'h0000_1111;
        alu_b_i = 32'h0000_1111;
        alu_op_i = `ALU_ADD;
#(p)    $display("T=%03t ns: %h + %h = %h\n",$realtime/1000, alu_a_i, alu_b_i,alu_p_o);

// Subtraction
#(2*p) alu_op_i = `ALU_SUB;
#(p)    $display("T=%03t ns: %h - %h = %h\n",$realtime/1000, alu_a_i, alu_b_i,alu_p_o);

// Comparison
#(2*p) alu_op_i = `ALU_SLTU;
#(p)    $display("T=%03t ns: %h < %h = %h\n",$realtime/1000, alu_a_i, alu_b_i,alu_p_o);
```

Test bench

- Display the results of test cases.
  - $display: a function to print out a message
  - $realtime: a function to show a time stamp

```
//------------------------------------------------------------------------------
// ALU operations
//------------------------------------------------------------------------------
#(8*p)
        alu_a_i = 32'h0000
        alu_b_i = 32'h0000
        alu_op_i = `ALU_SI
#(p)    $display("T=%03t r

// Addition
#(4*p)
        alu_a_i = 32'h0000
        alu_b_i = 32'h0000
        alu_op_i = `ALU_AD
#(p)    $display("T=%03t r

// Subtraction
#(2*p) alu_op_i = `ALU_SU
#(p)    $display("T=%03t r

// Comparison
#(2*p) alu_op_i = `ALU_SI
#(p)    $display("T=%03t n
```

```
VSIM 33> run 1000ns
# T=290 ns: 00002222 < 00002222 : 00000000 (Unsigned)
#
# T=340 ns: 00001111 + 00001111 = 00002222
#
# T=370 ns: 00001111 - 00001111 = 00000000
#
# T=400 ns: 00001111 < 00001111 = 00000000
#
# T=450 ns: 00001110 & 00000111 = 00000110
#
# T=480 ns: 00001110 | 00000111 = 00001111
#
# T=510 ns: 00001110 ^ 00000111 = 00001001
```

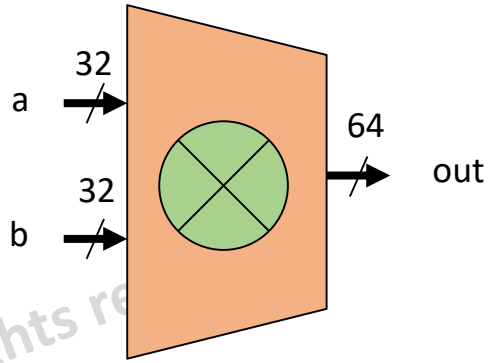


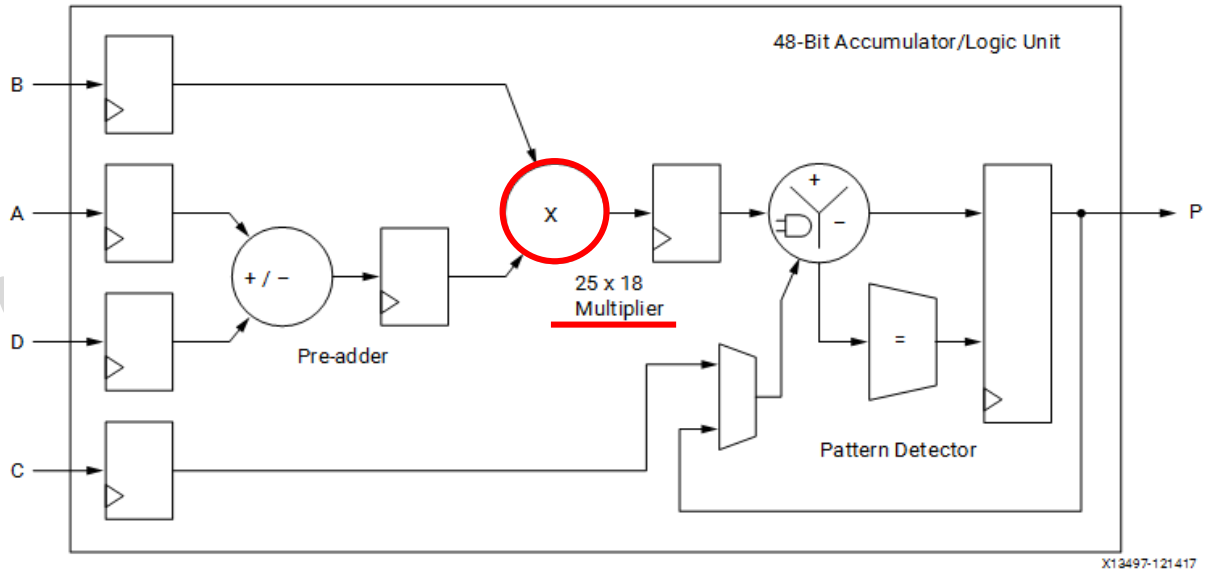
Lab 3: Multiplier

- Lab 3: Design a sequential multiplier
  - Implement a sequential multiplier module
  - Create a test bench
  - Run simulation
  - Show the output result

Multiplier

- Multiplier out = a × b
  - a and b are 32-bit integers; out is a 64-bit integer.
- How to implement a multiplier in H/W?
  - Simply use a "*" operation
    - Dedicated multiplier circuit (DSP)
  - Design a customized multiplier





Xilinx: The structure of the DSP48E block

Multiplier (riscv_multiplier.v)

```
module riscv_multiplier
(
                    // Inputs
                    input clk_i,
                    input rstn_i,
                    input [3:0] id_alu_op_r,
                    input id_a_signed_r,
                    input id_b_signed_r,
                    input [31:0] id_ra_value_r,     // 32bits multiplicand register a
                    input [31:0] id_rb_value_r,     // 32bits multiplier register b

                    // Outputs
                    output [63:0] mul_res_w,                    // 64bits multiplier out
                    output ex_stall_mul_w
);
```

```
reg             mul_busy_r;
reg             mul_ready_r;
reg   [4:0]     mul_count_r;
reg  [63:0]     mul_res_r;
wire            mul_request_w;
wire [32:0]     mul_sum_w;
```

Multiplier

- Multiplier out = a × b
  - a, b are 32-bit integers; out is a 64-bit integer.

- Decompose a multiplier into multiple sequential adders and shifters
  - t = 1: out = b[0] × a
  - t = 2: out = ((b[1] × a)<<1) + out

  ……
  - t = 32: out = ((b[31] × a)<<31) + out

Sequential multiplier

- Pseudocode

```
mul_sum_w = { 1'b0, mul_res_r[63:32] } + { 1'b0, mul_res_r[0] ? b : 32'h0 }
if mul_busy
        count ← count – 1
         mul_res_r ← { mul_sum_w, mul_res_r[31:1] };
        if count == 0
            mul_busy  ← 0
        end
end
else if (mul_ready_r)
       mul_ready_r <= 1'b0;
end
else if mul_req
        count ← 31
        mul_busy ← 1
        mul_res_r ← { 32'h0, a};
end
```

Sequential multiplier

- Pseudocode

```
mul_sum_w = { 1'b0, mul_res_r[63:32] } + { 1'b0, mul_res_r[0] ? b : 32'h0 }
if mul_busy
        count ← count – 1
         mul_res_r ← { mul_sum_w, mul_res_r[31:1] };
        if count == 0
            mul_busy  ← 0
        end
end
else if (mul_ready_r)
       mul_ready_r <= 1'b0;
end
else if mul_req
        count ← 31
        mul_busy ← 1
        mul_res_r ← { 32'h0, a};
end
```

- Start a multiplication request
  - Initialize a counter
  - Indicate that it is busy
  - Initialize the accumulate result

Sequential multiplier

- Pseudocode

```
mul_sum_w = { 1'b0, mul_res_r[63:32] } + { 1'b0, mul_res_r[0] ? b : 32'h0 }
if mul_busy
        count ← count – 1
         mul_res_r ← { mul_sum_w, mul_res_r[31:1] };
        if count == 0
            mul_busy  ← 0
        end
end
else if (mul_ready_r)
       mul_ready_r <= 1'b0;
end
else if mul_req
        count ← 31
        mul_busy ← 1
        mul_res_r ← { 32'h0, a};
end
```

- Do multiplication
- Update a counter
- Update a busy flag

Explanation

```
assign mul_sum_w = { 1'b0, mul_res_r[63:32] } + { 1'b0, mul_res_r[0] ? mul_b_w : 32'h0 };
assign mul_res_w = mul_negative_w ? -mul_res_r : mul_res_r;

always @(posedge clk_i, neg) begin
        if (~reset_i) begin
                mul_busy_r  <= 1'b0;
                …
        end
        else begin
                if (mul_busy_r) begin
                        mul_count_r <= mul_count_r - 5'd1;
                        …
                end
                else if (mul_ready_r) begin
                        mul_ready_r <= 1'b0;
                end
                else if (mul_request_w) begin
                        mul_count_r <= 5'd31;
                        …
                end
        end
end
```

```
assign mul_request_w  = (`ALU_MULL == id_alu_op_r || `ALU_MULH == id_alu_op_r);
assign ex_stall_mul_w = mul_request_w && !mul_ready_r;

endmodule
```

TODO: …

- Implement riscv_multiplier.v by completing some missing codes
- Do a simulation with time = 1000ns
- Show the waveform

```
always @(posedge clk_i or negedge rstn_i) begin
  if (~rstn_i) begin
    mul_busy_r  <= 1'b0;
    mul_ready_r <= 1'b0;
    mul_count_r <= 5'd0;
    mul_res_r   <= 64'h0;
  end else begin
    if (mul_busy_r) begin
      //Insert your code
      //mul_count_r <= /*Insert your code */
      mul_res_r   <= { mul_sum_w, mul_res_r[31:1] };

      if (mul_count_r == 5'd0) begin
        mul_busy_r  <= 1'b0;
        // Insert your code
        // mul_ready_r <= /*Insert your code */
      end

    end else if (mul_ready_r) begin                    //***
      mul_ready_r <= 1'b0;

    end else if (mul_request_w) begin
      mul_count_r <= 5'd31;                            //***
      mul_busy_r  <= 1'b1;                             //***
      // Insert your code
      //mul_res_r   <= /*Insert your code */           //***
    end
  end
end
```

Test bench (riscv_multiplier_tb.v)

```
module multiplier_tb;
        reg rstn;
        reg clk;
        reg [3:0] alu_op_i;
        reg [31:0] alu_a_i, alu_b_i;
        wire [63:0] alu_p_o;
        wire ex_stall_mul_w;
        reg a_signed, b_signed;
```

```
parameter p=10;
initial begin
                clk = 1'b0;
                forever #(p/2) clk = !clk;
end
```

Fast

```
riscv_multiplier_fast
u_riscv_multiplier_fast(
    ./*input */clk_i(clk),
    ./*input */rstn_i(rstn),
    ./*input [3:0]  */id_alu_op_r(alu_op_i),
    ./*input        */id_a_signed_r(a_signed),
    ./*input        */id_b_signed_r(b_signed),
    ./*input [31:0] */id_ra_value_r(alu_a_i),
    ./*input [31:0] */id_rb_value_r(alu_b_i),
    ./*output [63:0] */mul_res_w(alu_p_o_f),
    ./*output       */ex_stall_mul_w(ex_stall_mul_w_f)
);
```

Slow

```
// Sequential multipler: 32 cycle delay
riscv_multiplier
u_riscv_multiplier(
    ./*input */clk_i(clk),
    ./*input */rstn_i(rstn),
    ./*input [3:0]  */id_alu_op_r(alu_op_i),
    ./*input        */id_a_signed_r(a_signed),
    ./*input        */id_b_signed_r(b_signed),
    ./*input [31:0] */id_ra_value_r(alu_a_i),
    ./*input [31:0] */id_rb_value_r(alu_b_i),
    ./*output [63:0] */mul_res_w(alu_p_o),
    ./*output       */ex_stall_mul_w(ex_stall_mul_w)
);
```

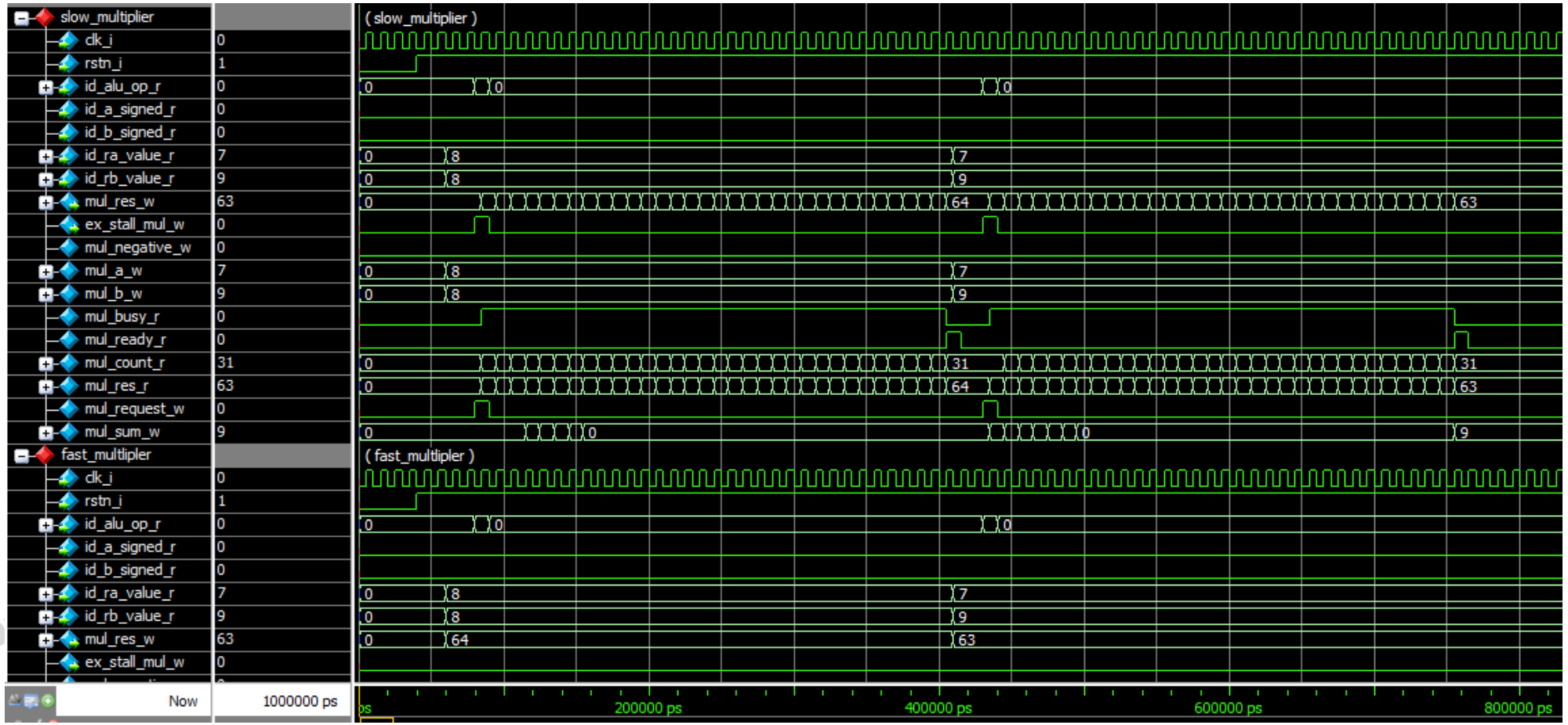
Test cases (riscv_multiplier_tb.v)

```
initial begin
        rstn = 1'b0;                        // negedge reset on
        alu_a_i = 0;
        alu_b_i = 0;
        alu_op_i = 0;
        a_signed = 0;
        b_signed = 0;
        alu_op_i = `ALU_ADD;
        #(4*p) rstn = 1'b1;                 // negedge reset off
        #(2*p)          a_signed = 0;
                        b_signed = 0;
                        alu_a_i = 32'h00000008;
                        alu_b_i = 32'h00000008;
        #(2*p)          alu_op_i = `ALU_MULL;

        #(p)            alu_op_i = ` ALU_ADD;

        #(32*p)         alu_a_i = 32'h00000007;
                        alu_b_i = 32'h00000009;
        #(2*p)          alu_op_i = `ALU_MULL;

        #(p)            alu_op_i = ` ALU_ADD;
 end

endmodule
```

- Test case 1: do 8*8
- Test case 2: do 7*9

Waveform (Slow multiplier)

- It takes 32 cycles to complete a multiplier



Questions

1. (Shifter) Compare two shifter implementations.
2. (Multiplier) How many cycles does a sequential (slow) multiplier take to complete a multiplication?

Why?

- 32 cycles
- Does the multiplier work correctly if *a new multiplication is issued during computation*?
- Does the multiplier work correctly if *the inputs are updated during computation*?