차세대 반도체 디지털 혁신공유대학 사업단

POLARIS

Lecture 5: Load and store instructions, RISC-V core

Xuan-Truong Nguyen

Lecture plan

- Today we will:
  - Review the previous lecture and demonstrate how to work with the baseline code
  - Study and implement load and store instructions
  - Integrate all components to build a complete RISC-V core
  - Study the GNU toolchain for RISC-V
  - Study optimization

Road map

Review

Load, store instructions

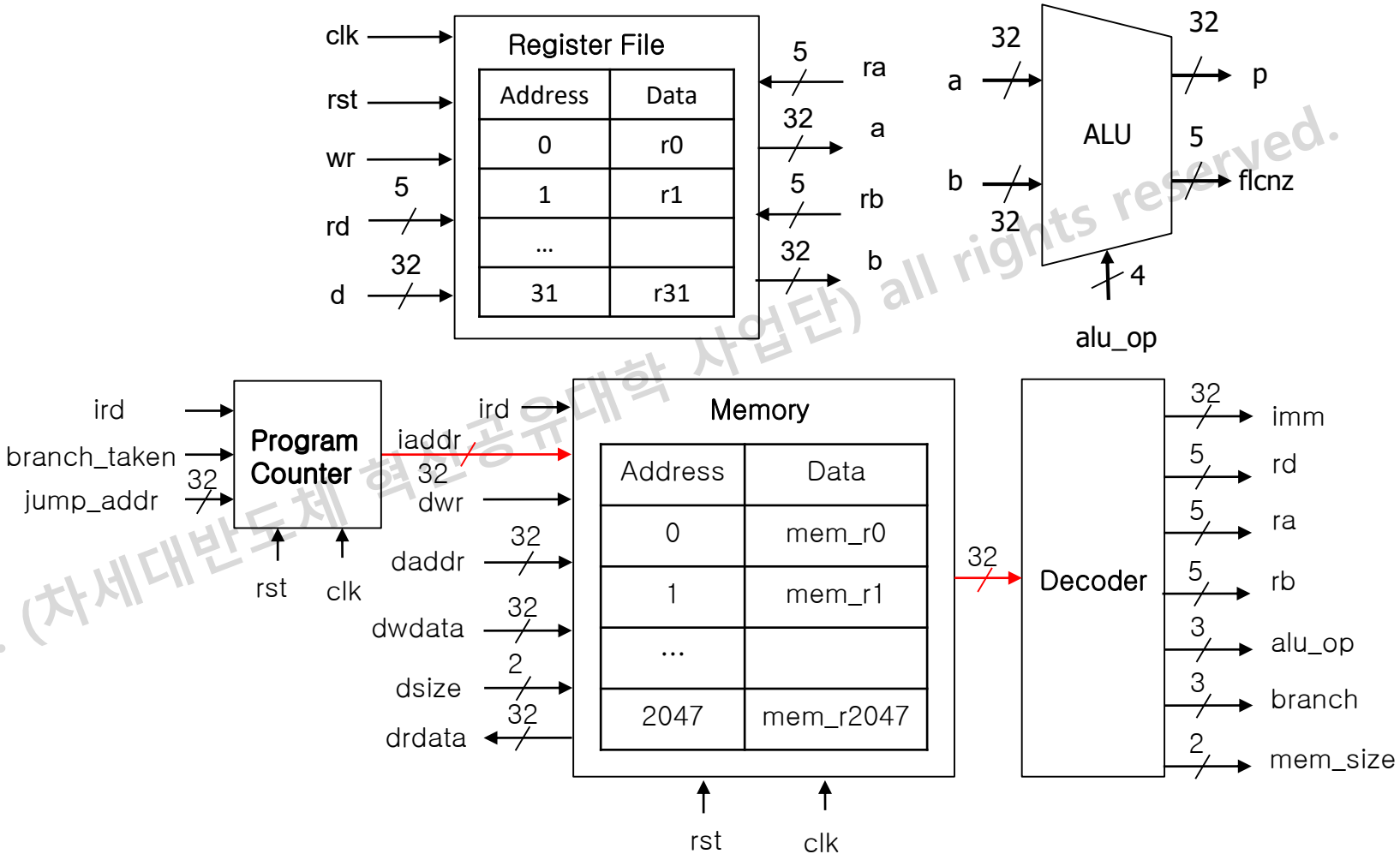
RISC-V core integration

RISC-V GNU Compiler

Optimization

Recap: Simplified RISC-V

- Five modules
  - Program counter
  - Memory
  - Decoder
  - ALU
  - Register File

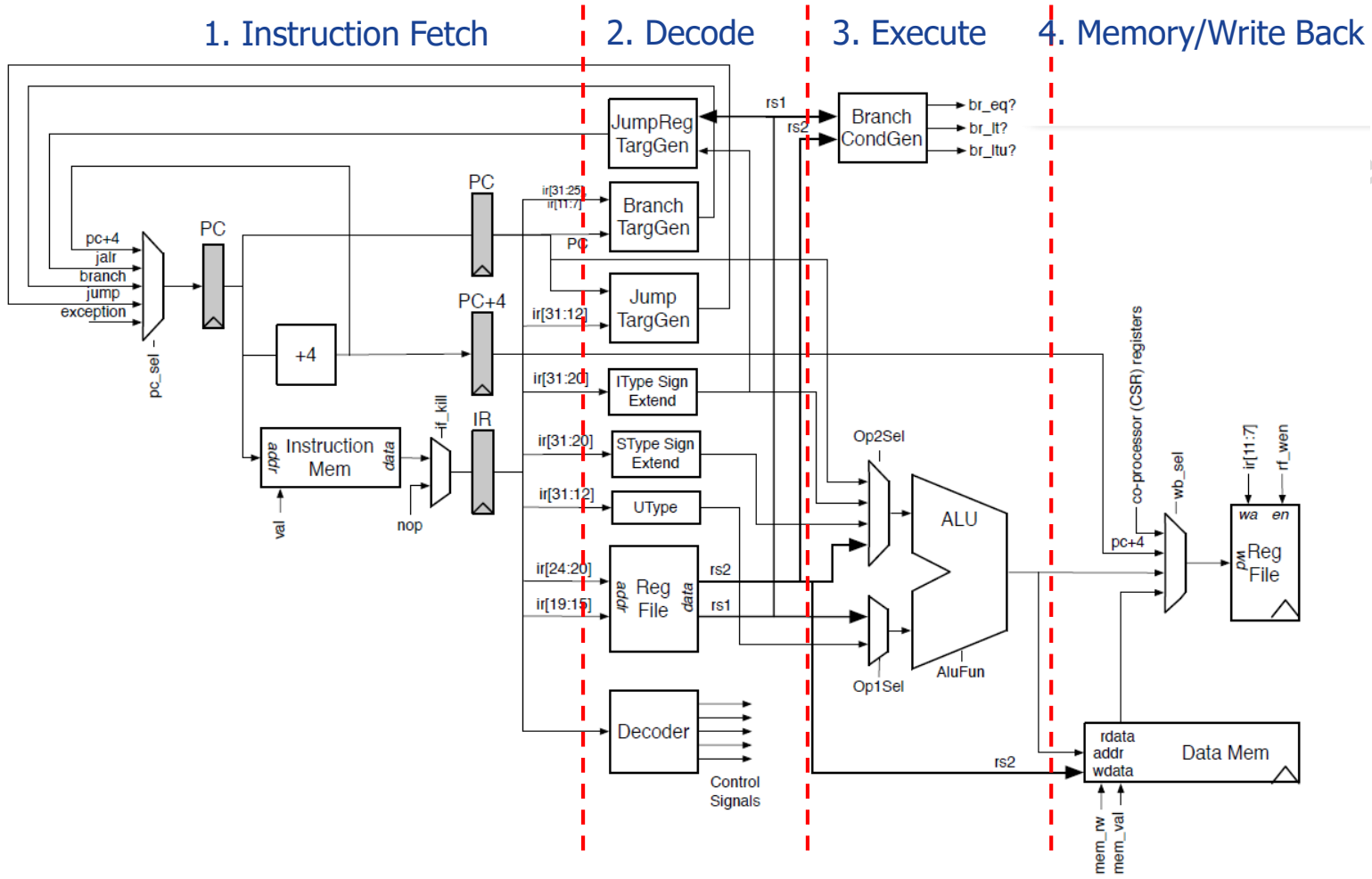


Dummy vs TODO

- Dummy code/connections
  - Maintain the dataflow among modules, i.e., PC, Decoder, ALU, and RF.
  - Some logic is incomplete.
- TODO
  - Some logic is missing, so they need to be filled out.

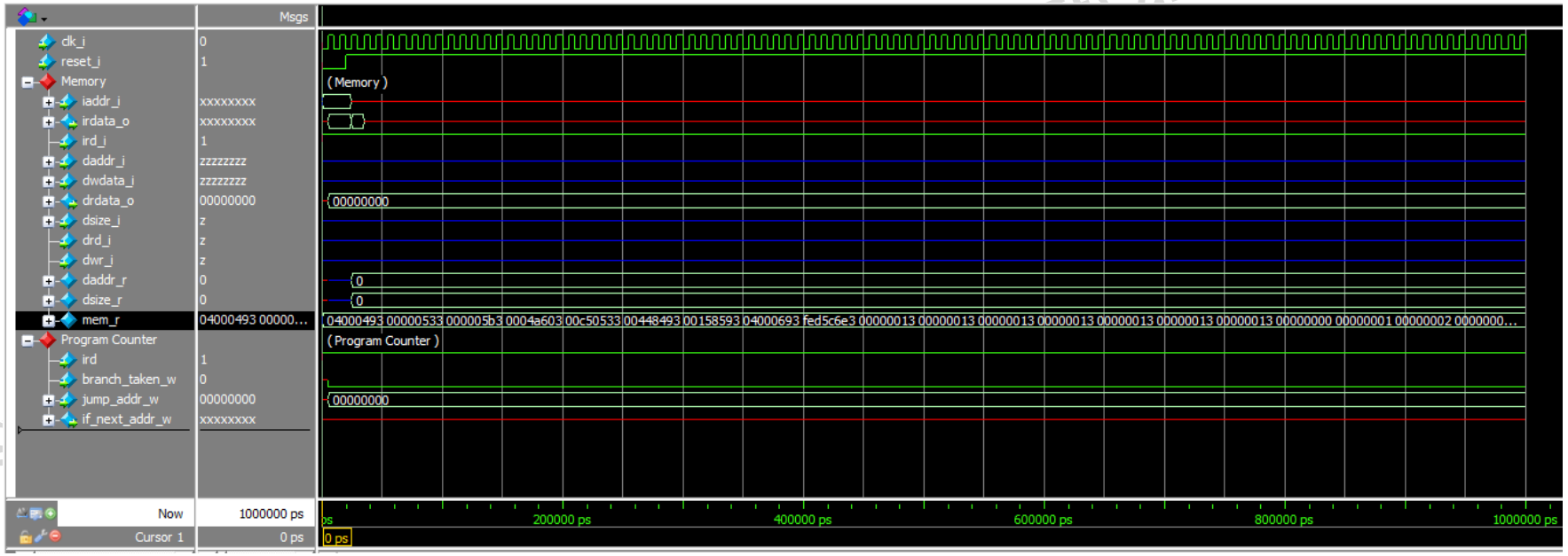
- How to work on the baseline code?

RISC-V core and Memory



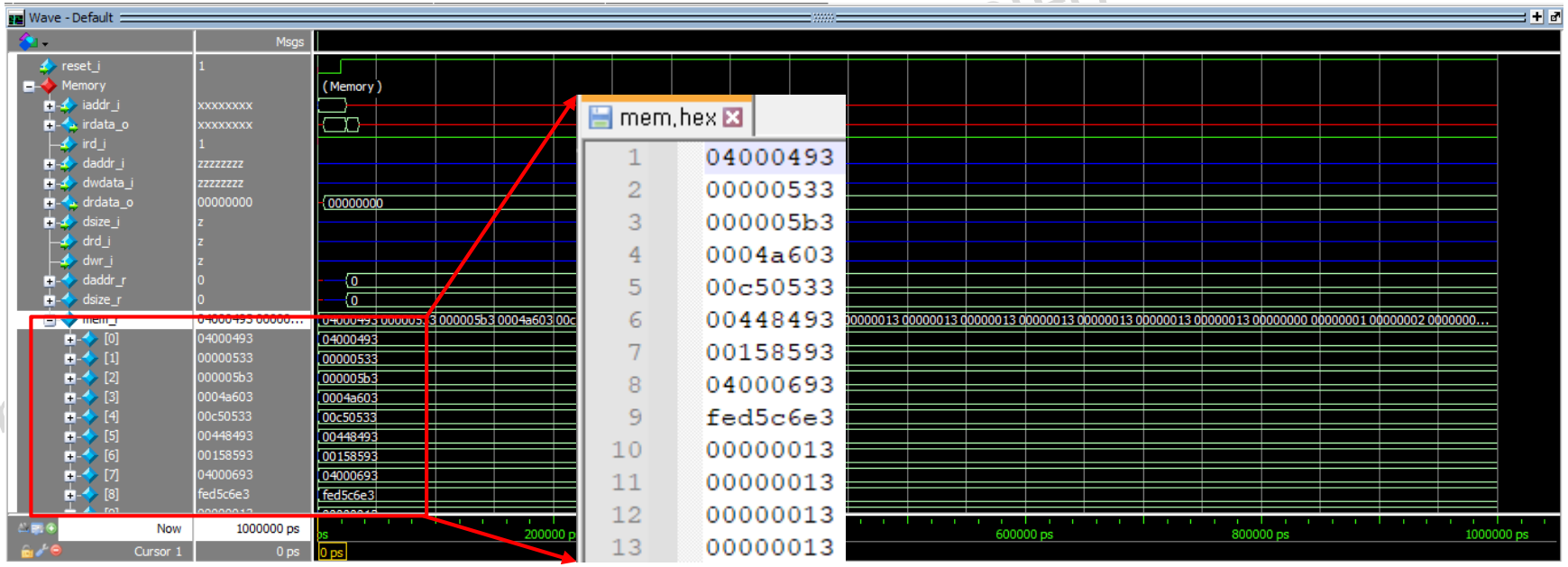
Instruction Fetch

- Compile the baseline code successfully.
- Do a simulation
  - Add signals from Memory and PC.
  - Select signals → Right-click → Group → Define "name" → OK.
  - Simulation time = 1000ns



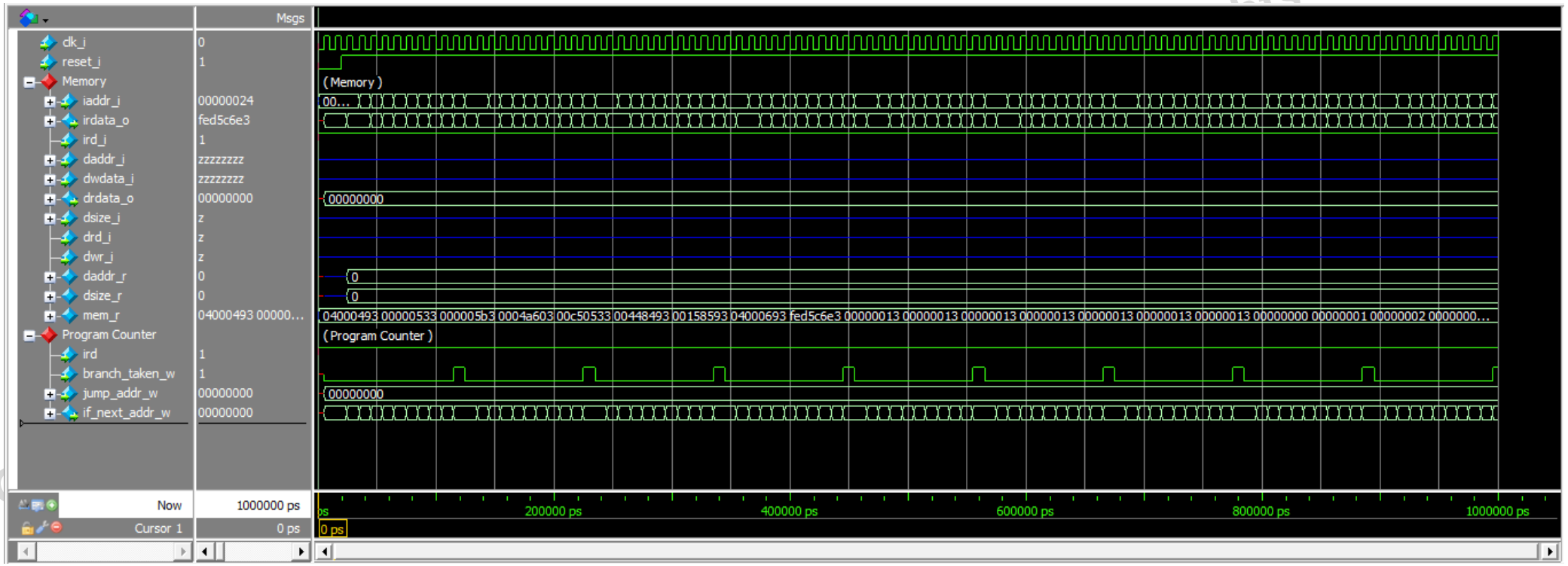
Instruction fetch

- Do a simulation
  - Check initialized values in mem_r
  - mem_r[0]= 04000493 0000: 0000_0000_0000_0000 (00 or 0)
  - mem_r[1]= 00000533 0004: 0000_0000_0000_0100 (01 or 1)
  - mem_r[2]= 000005b3 0008: 0000_0000_0000_1000 (10 or 2)



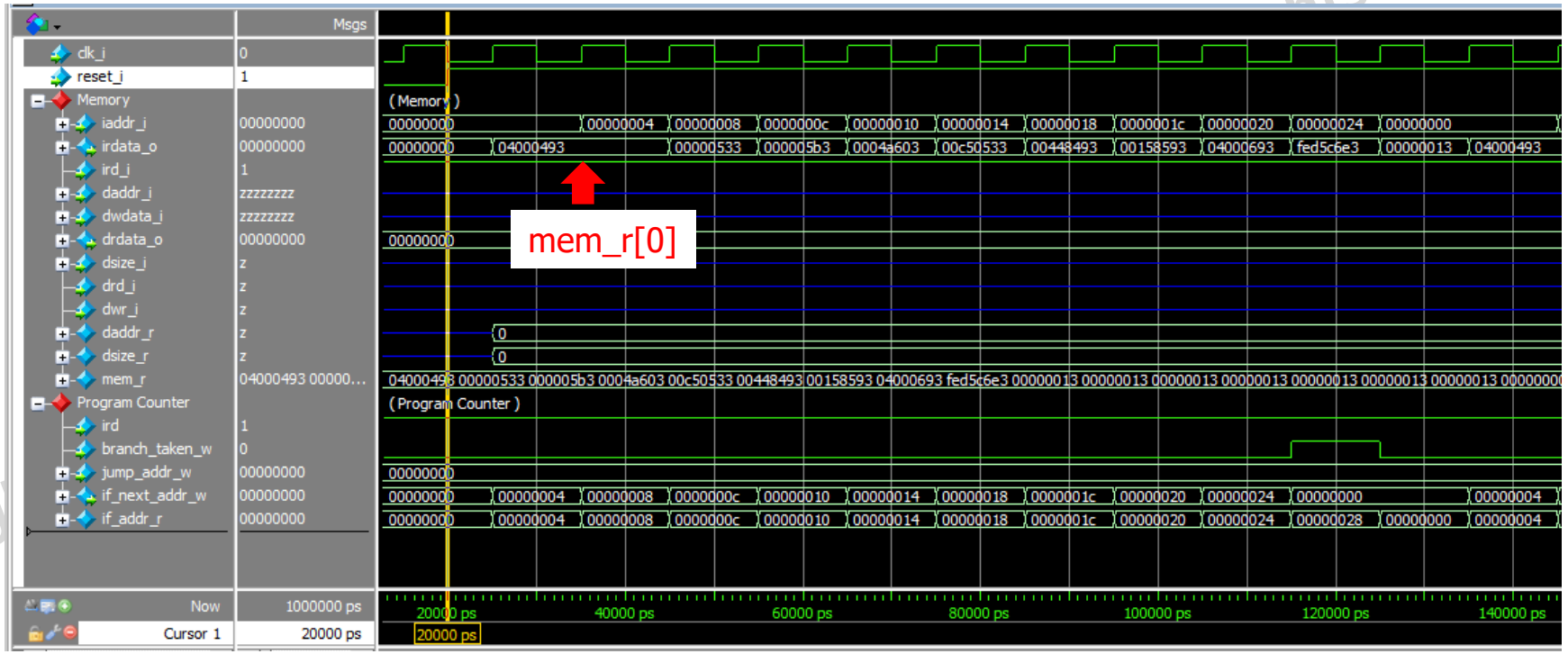
Instruction fetch

- Complete Program Counter (riscv_pc.v)
  - Reset: if_addr_r ← RESET_SP
  - Jump: if_addr_r ← jum_addr_w
  - Others: if_addr_r ← if_addr_r + 4



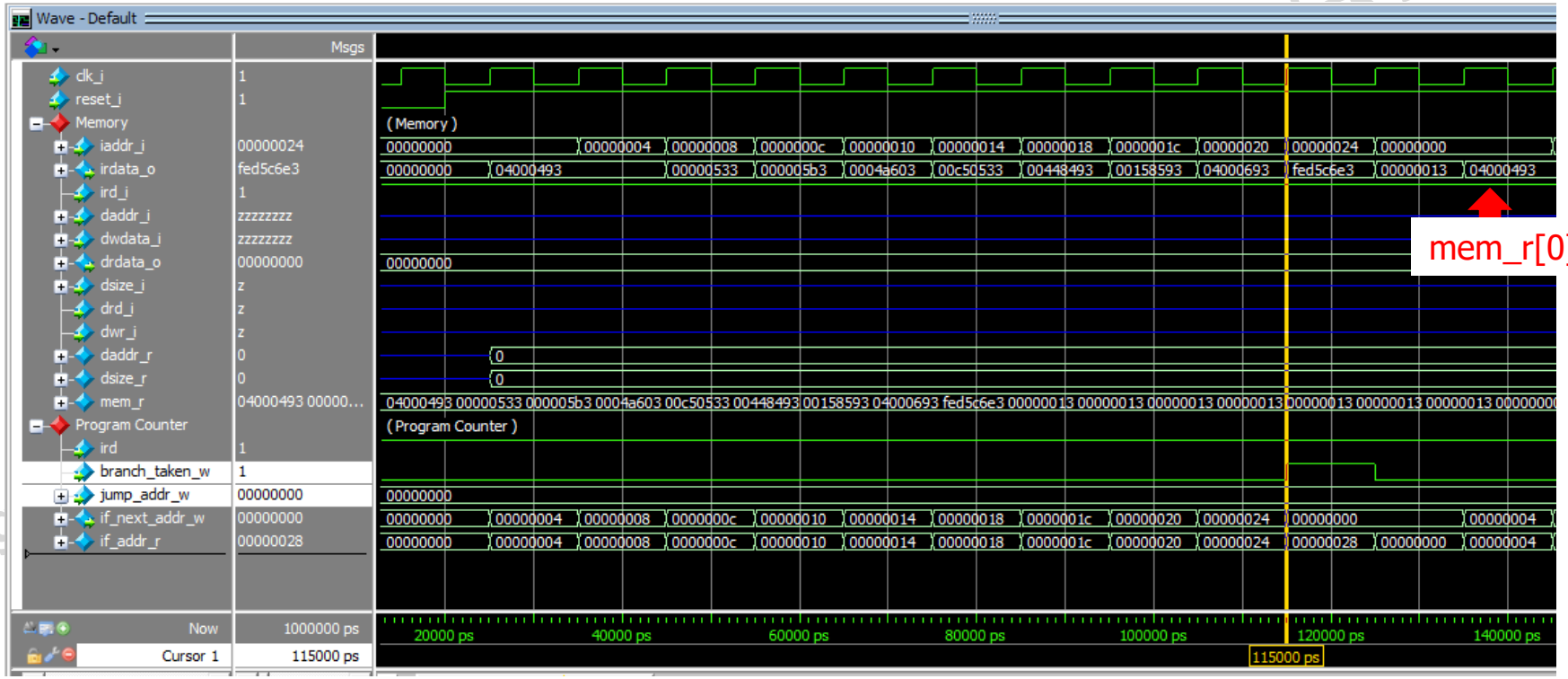
Instruction fetch

- Complete Program Counter (riscv_pc.v)
  - Reset: if_addr_r ← RESET_SP
  - Jump: if_addr_r ← jum_addr_w
  - Others: if_addr_r ← if_addr_r + 4



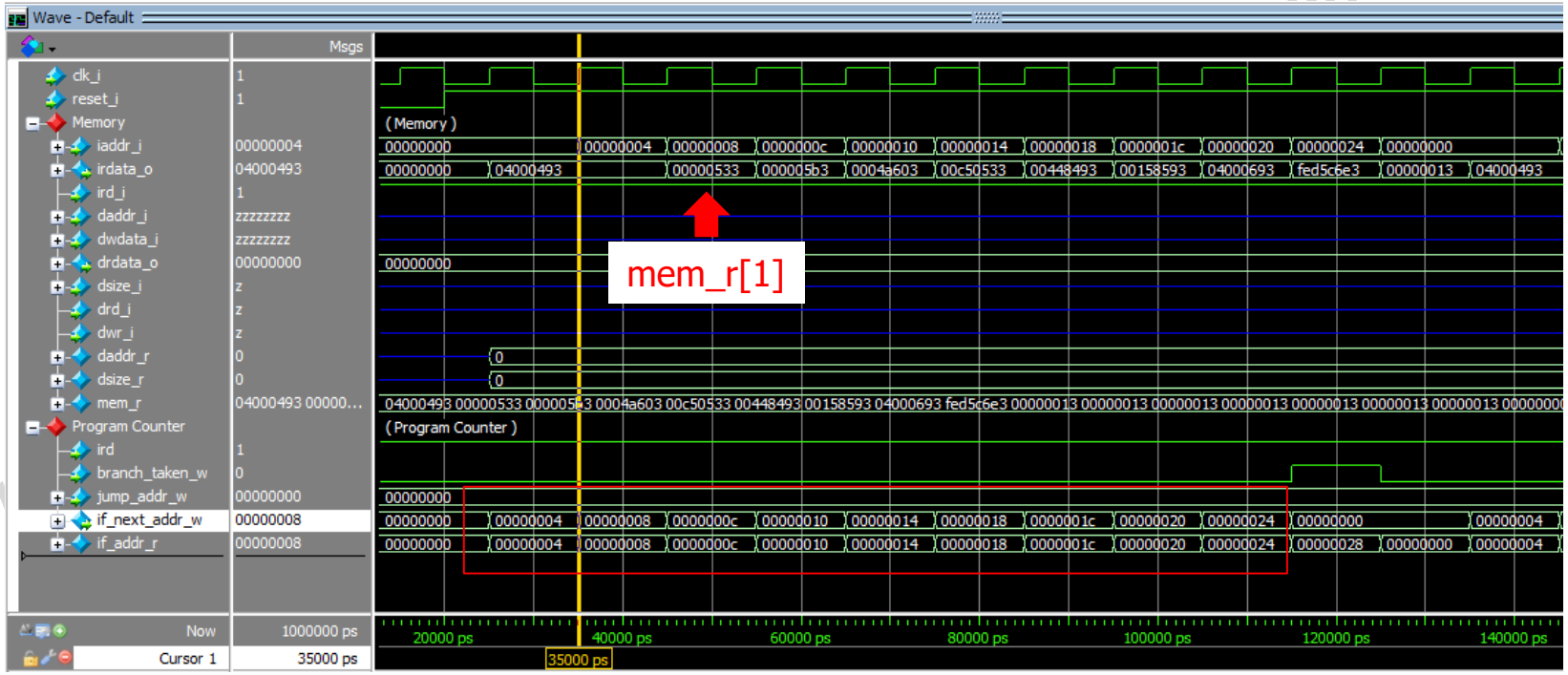
Instruction fetch

- Complete Program Counter (riscv_pc.v)
  - Reset: if_addr_r ← RESET_SP
  - Jump: if_addr_r ← jum_addr_w
  - Others: if_addr_r ← if_addr_r + 4



Instruction fetch

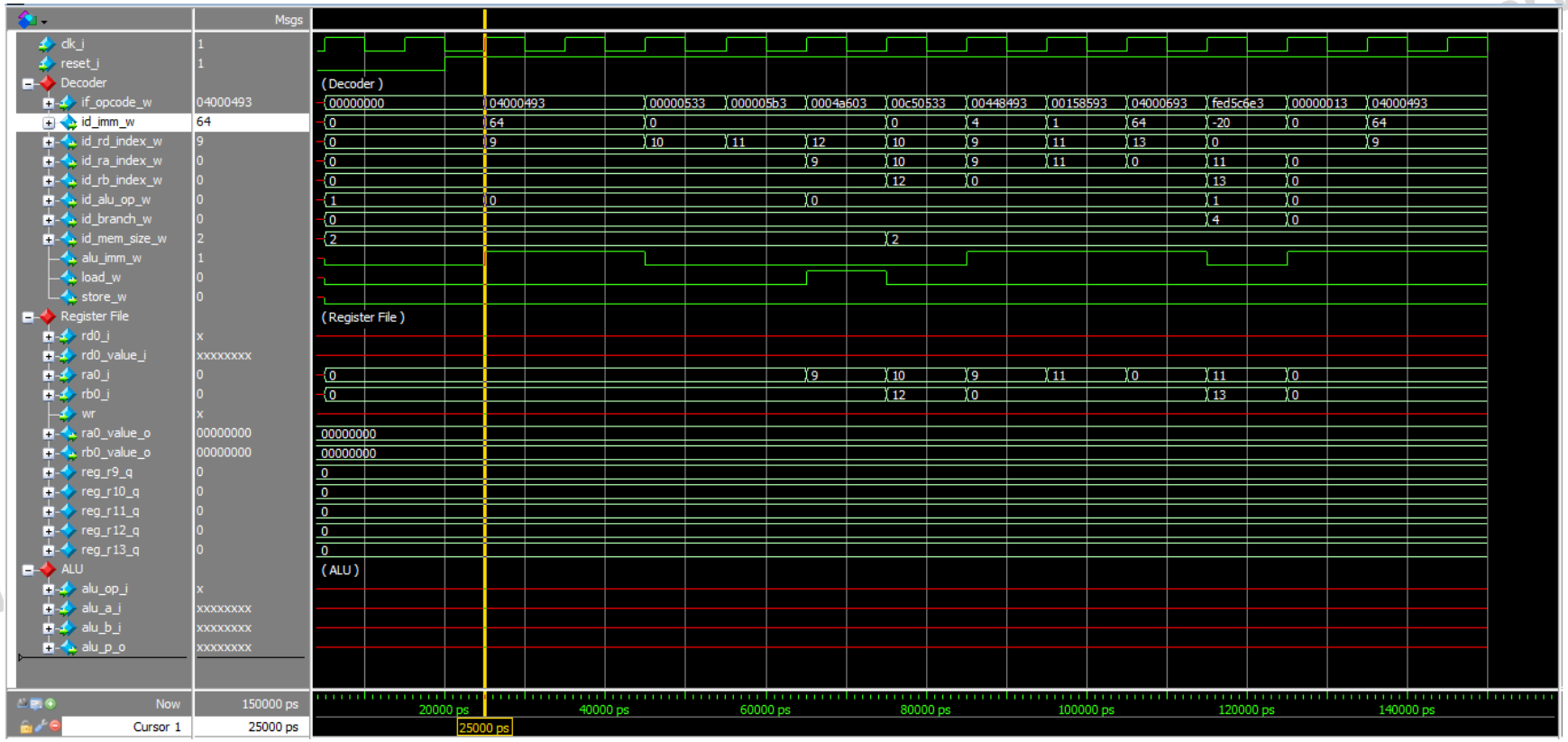
- Complete Program Counter (riscv_pc.v)
  - Reset: if_addr_r ← RESET_SP
  - Jump: if_addr_r ← jum_addr_w
  - Others: if_addr_r ← if_addr_r + 4



Decode

- Extract information in an instruction register: Arithmetic Instruction
  - if_opcode_w = 0400_0493 (addi x9, x0, 64)

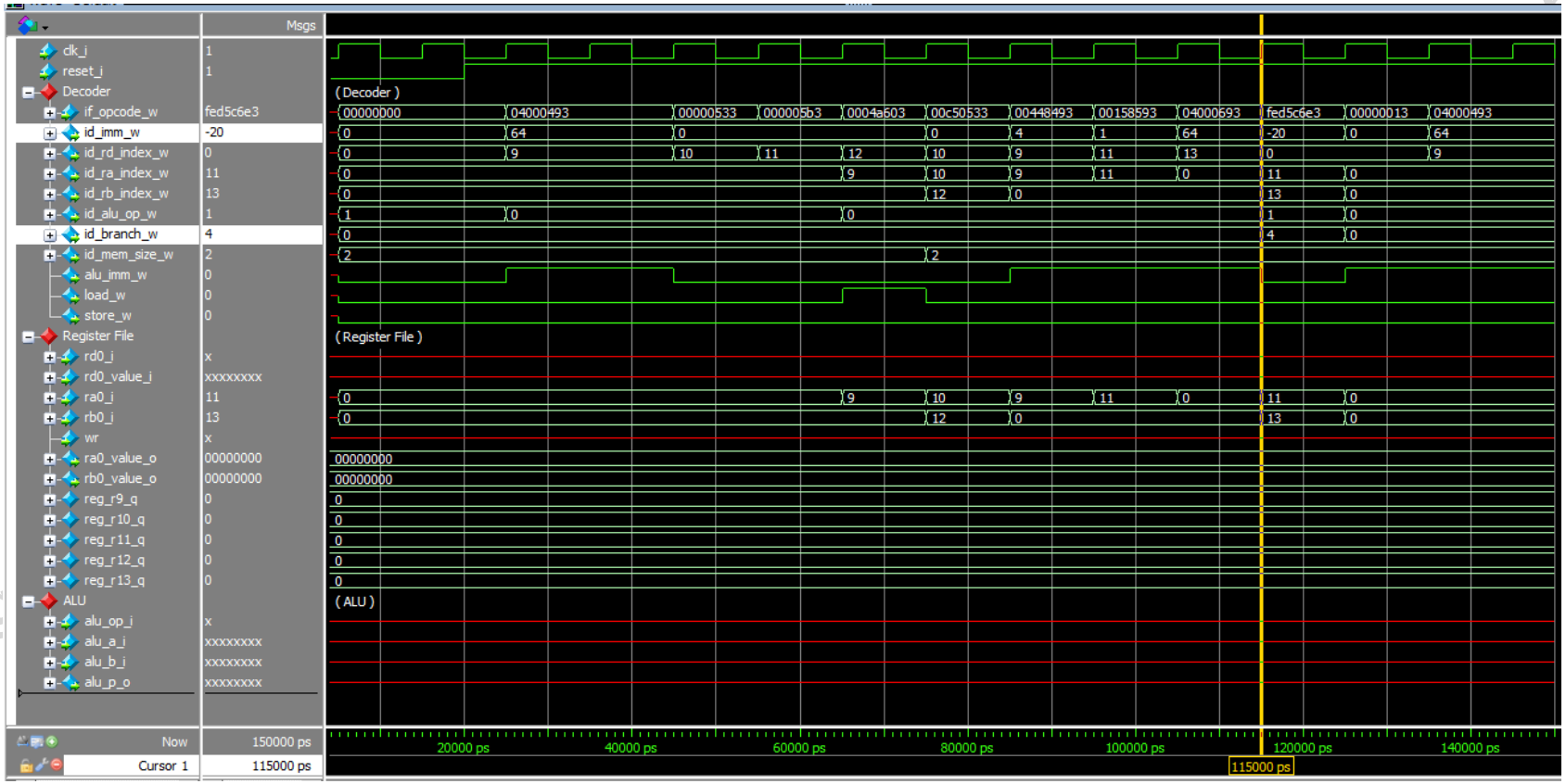
  ⇒ alu_imm_w = 64, id_rd = 9, id_alu_op = 0 (ADDi).



Decode

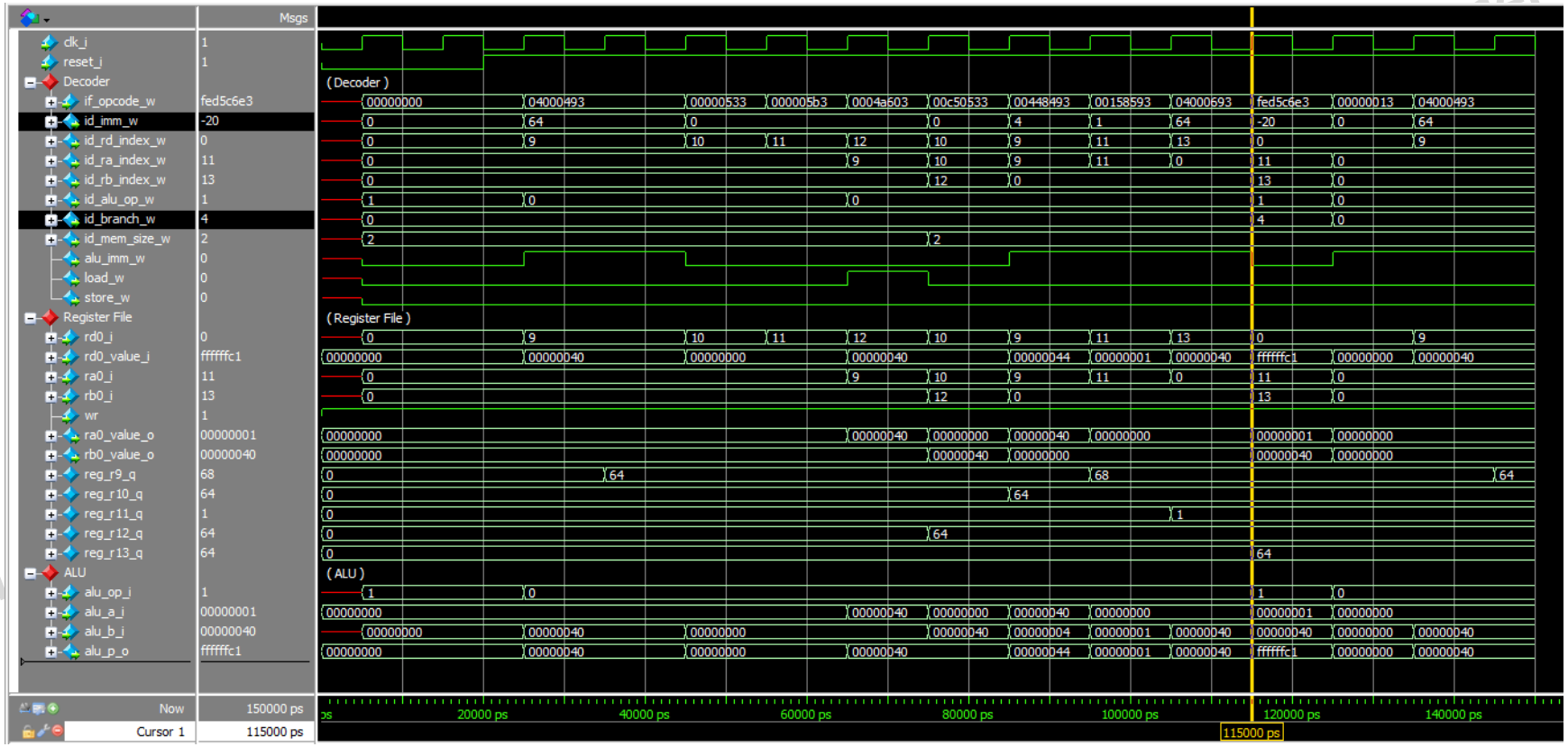
- Extract information in an instruction register: Branch-Less-Than (BLT)
  - if_opcode_w = fed5_c6e3 (blt x11, x13, -20)

  ⇒ alu_imm_w = -20, id_ra = 11, id_rb = 13, id_branch = 4 (BLT).



Execution

- Uncomment the code to connect Decoder to RF and ALU
- Do a simulation
  - Simulation time = 150ns



Dummy

- Dummy code for "branch if less than (BR_LT)"
  - branch_taken_w = 1'b1

  ⇒ Set the flag to HIGH for "branch if less than (BR_LT)"

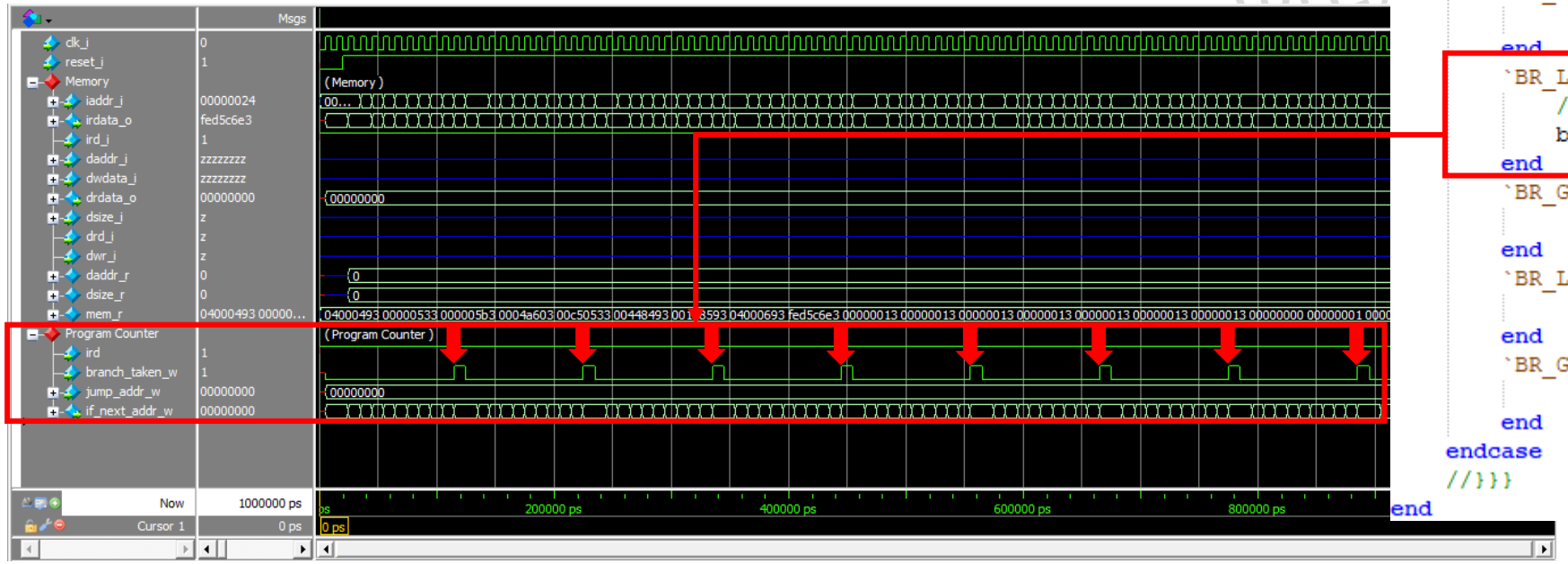
  ⇒ Dummy: Branch-Less-Than is an unconditional instruction

Address	Instruction	Basic code	Original code	Comments
0000	00004093	addi x9 x0 64	addi x9, x0, 64	# x9=&A[0]
0004	00000533	add x10 x0 x0	add x10, x0, x0	# sum=0
0008	000005b3	add x11 x0 x0	add x11, x0, x0	# i=0
000C	0004a603	lw x12 0(x9)	lw x12, 0(x9)	# x12=A[i]
0010	00c50533	add x10 x10 x12	add x10,x10,x12	# sum+=
0014	00448493	addi x9 x9 4	addi x9,x9,4	# &A[i++]
0018	00158593	addi x11 x11 1	addi x11,x11,1	# i++
001C	04000693	addi x13 x0 64	addi x13,x0,64	# x13=64
0020	fed5c6e3	blt x11 x13 -20	blt x11,x13,Loop	# Branch

```
/* TODO: Branch, Jump and Link instructions */
always @ (*) begin
    branch_taken_w = 1'b0;
    jump_addr_w = 32'h0;
    // Insert your code
    //{{{
    case(id_branch_w)
        `BR_JUMP: begin
        end
        `BR_EQ: begin

        end
        `BR_NE: begin

        end
        `BR_LT: begin
            // Dummy Branch
            branch_taken_w = 1'b1;
        end
        `BR_GE: begin

        end
        `BR_LTU: begin

        end
        `BR_GEU: begin

        end
    endcase
    //}}}
end
```

Dummy

- Dummy code/connections
  - Maintain the dataflow among modules
  - Some logic is incomplete.

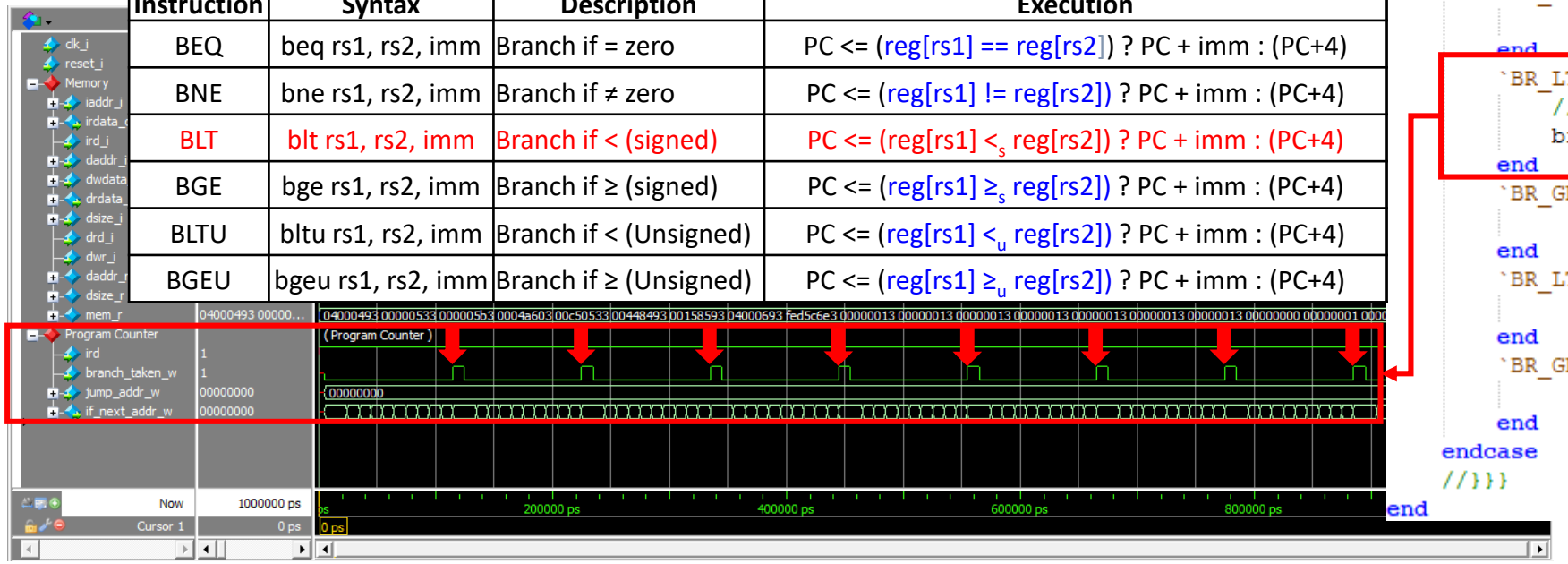



```
/* TODO: Branch, Jump and Link instructions */
always @ (*) begin
    branch_taken_w = 1'b0;
    jump_addr_w = 32'h0;
    // Insert your code
    //{{{
    case(id_branch_w)
        `BR_JUMP: begin
        end
        `BR_EQ: begin

        end
        `BR_NE: begin

        end
        `BR_LT: begin
            // Dummy Branch
            branch_taken_w = 1'b1;
        end
        `BR_GE: begin

        end
        `BR_LTU: begin

        end
        `BR_GEU: begin

        end
    endcase
    //}}}
end
```

How about TODO?

- TODO
  - Some logic is missing, so it needs to be filled out.
  - Example: Complete “branch if less than (BR_LT)”

Instruction	Syntax	Description	Execution
BEQ	beq rs1, rs2, imm	Branch if = zero	PC <= (reg[rs1] == reg[rs2]) ? PC + imm : (PC+4)
BNE	bne rs1, rs2, imm	Branch if ≠ zero	PC <= (reg[rs1] != reg[rs2]) ? PC + imm : (PC+4)
BLT	blt rs1, rs2, imm	Branch if < (signed)	PC <= (reg[rs1] $<_s$ reg[rs2]) ? PC + imm : (PC+4)
BGE	bge rs1, rs2, imm	Branch if ≥ (signed)	PC <= (reg[rs1] $\geq_s$ reg[rs2]) ? PC + imm : (PC+4)
BLTU	bltu rs1, rs2, imm	Branch if < (Unsigned)	PC <= (reg[rs1] $<_u$ reg[rs2]) ? PC + imm : (PC+4)
BGEU	bgeu rs1, rs2, imm	Branch if ≥ (Unsigned)	PC <= (reg[rs1] $\geq_u$ reg[rs2]) ? PC + imm : (PC+4)




```
/* TODO: Branch, Jump and Link instructions */
always @ (*) begin
    branch_taken_w = 1'b0;
    jump_addr_w = 32'h0;
    // Insert your code
    //{{{
    case(id_branch_w)
        `BR_JUMP: begin
        end
        `BR_EQ: begin

        end
        `BR_NE: begin

        end
        `BR_LT: begin
            // Dummy Branch
            branch_taken_w = 1'b1;
        end
        `BR_GE: begin

        end
        `BR_LTU: begin

        end
        `BR_GEU: begin

        end
    endcase
    //}}}
end
```

To do: Calculate a target jump address

- Calculate a target address
  - blt x11, x13, -20
  - if_opcode_w = fed5_c6e3
    - alu_imm_w = -20,
    - id_ra = 11
    - id_rb = 13
    - id_branch = 4 (BLT).

⇒ jump_addr_w = imm + PC

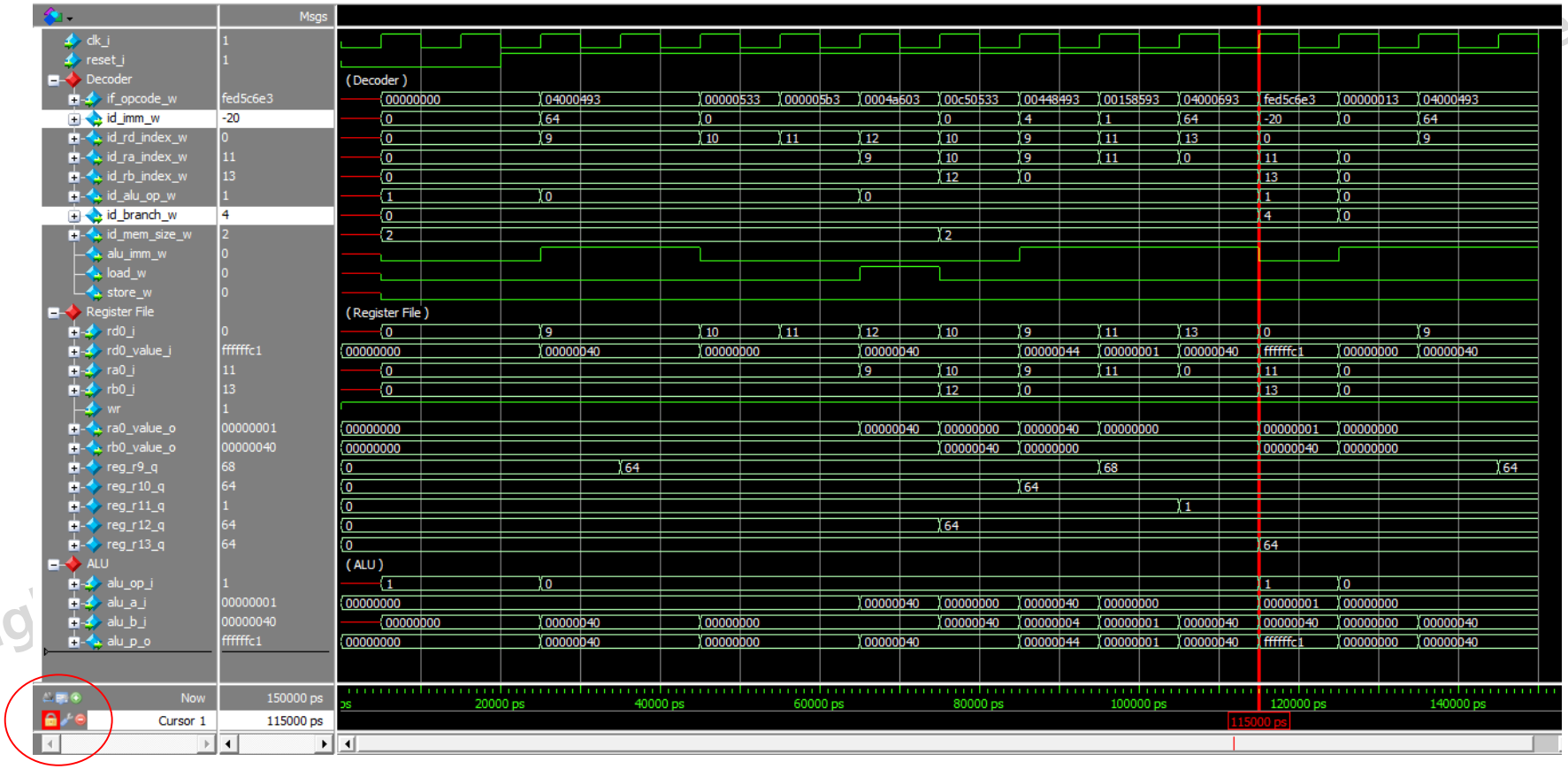
⇒ What is the current PC?

Instruction	Syntax	Description	Execution
BEQ	beq rs1, rs2, imm	Branch if = zero	PC <= (reg[rs1] == reg[rs2]) ? PC + imm : (PC+4)
BNE	bne rs1, rs2, imm	Branch if ≠ zero	PC <= (reg[rs1] != reg[rs2]) ? PC + imm : (PC+4)
BLT	blt rs1, rs2, imm	Branch if < (signed)	PC <= (reg[rs1] $<_s$ reg[rs2]) ? PC + imm : (PC+4)
BGE	bge rs1, rs2, imm	Branch if ≥ (signed)	PC <= (reg[rs1] $\geq_s$ reg[rs2]) ? PC + imm : (PC+4)
BLTU	bltu rs1, rs2, imm	Branch if < (Unsigned)	PC <= (reg[rs1] $<_u$ reg[rs2]) ? PC + imm : (PC+4)
BGEU	bgeu rs1, rs2, imm	Branch if ≥ (Unsigned)	PC <= (reg[rs1] $\geq_u$ reg[rs2]) ? PC + imm : (PC+4)

```
/* TODO: Branch, Jump and Link instructions */
always @ (*) begin
    branch_taken_w = 1'b0;
    jump_addr_w = 32'h0;
    // Insert your code
    //{{{
    case(id_branch_w)
        `BR_JUMP: begin
        end
        `BR_EQ: begin

        end
        `BR_NE: begin

        end
        `BR_LT: begin
            // Dummy Branch
            branch_taken_w = 1'b1;
        end
        `BR_GE: begin

        end
        `BR_LTU: begin

        end
        `BR_GEU: begin

        end
    endcase
    //}}}
end
```

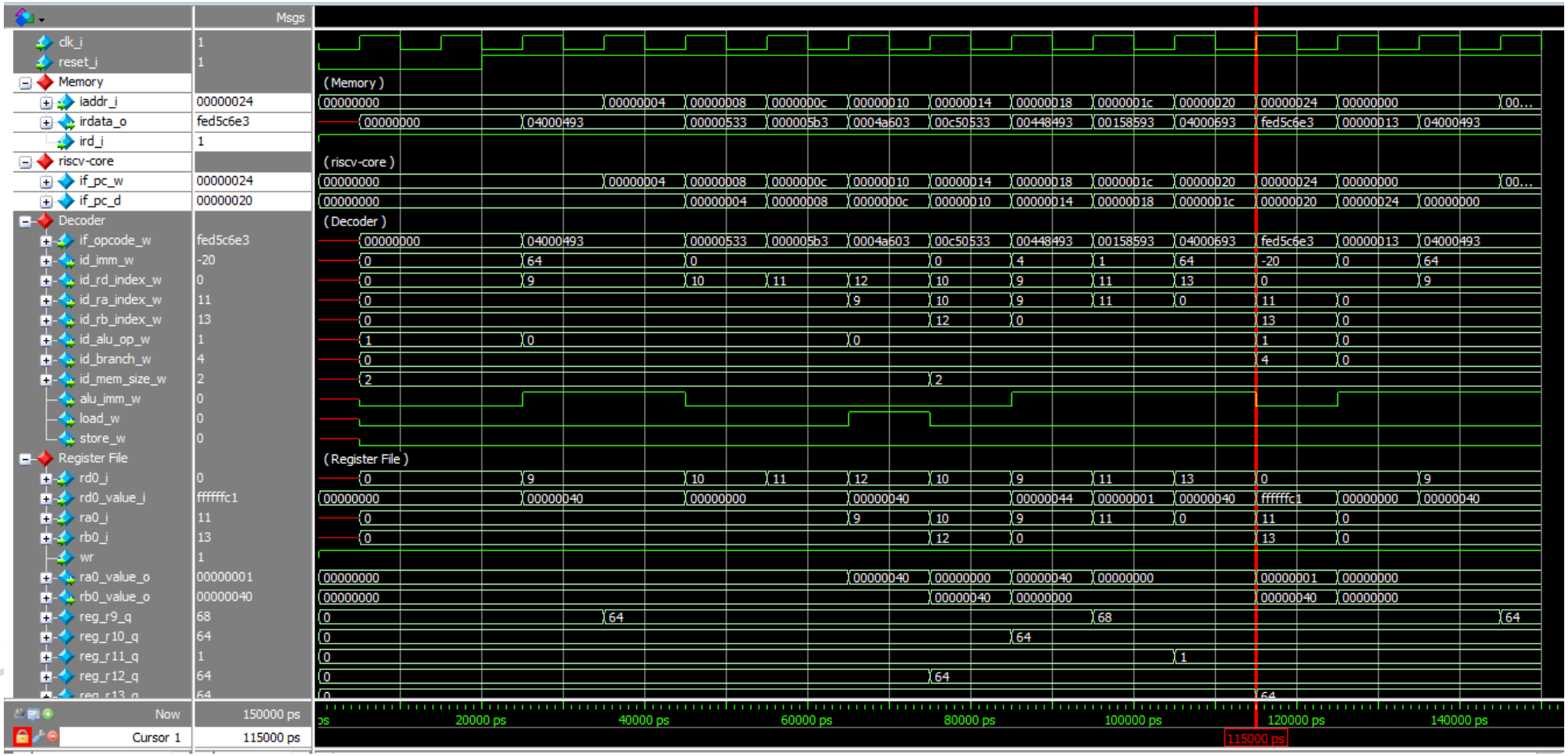
Conditional Branch

- Our goal is to find the current program counter
  - Lock a branch state



Conditional Branch

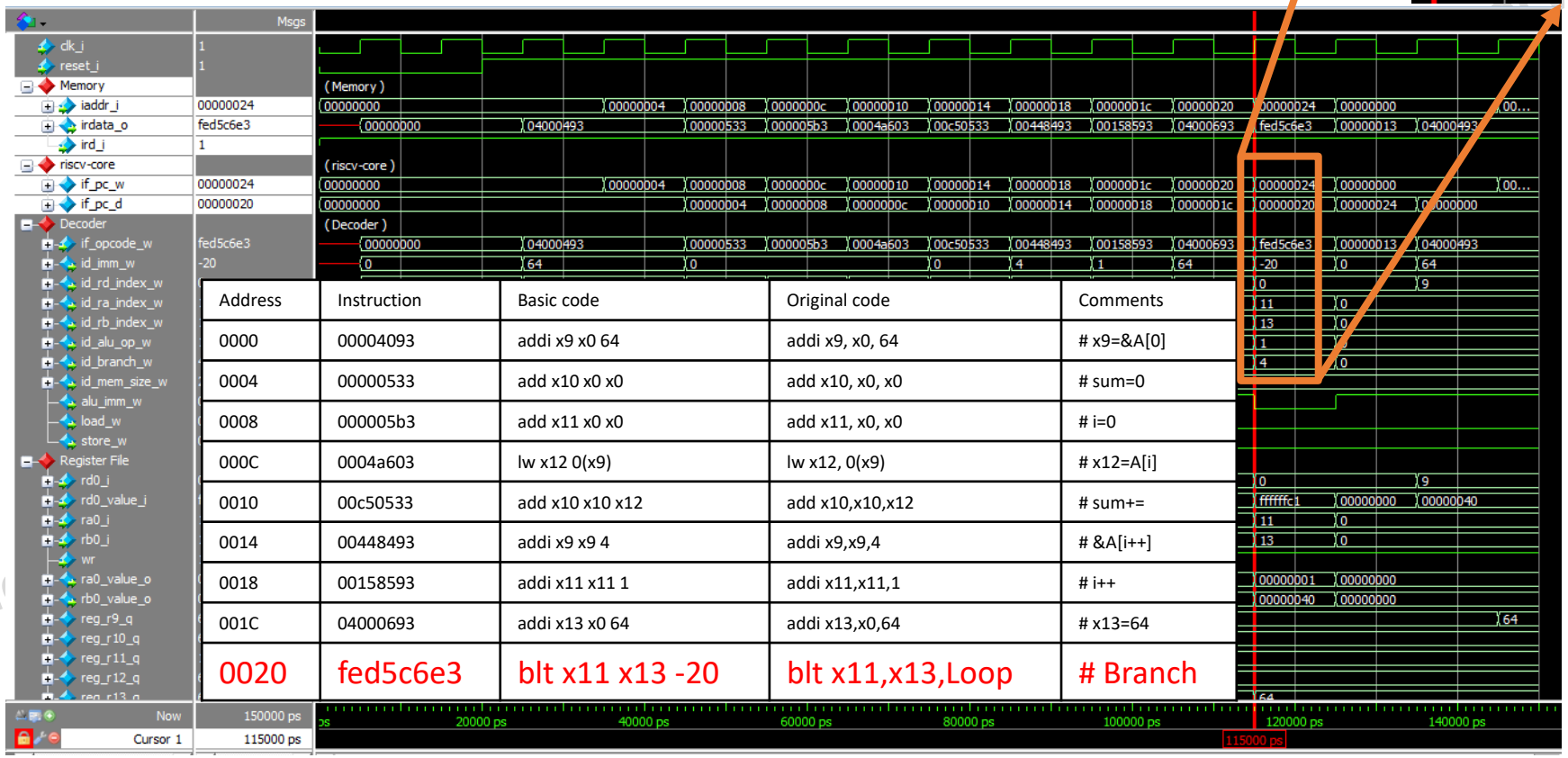
- Add signals of Memory and PC core that are related to Program Counter



Conditional Branch

- Our goal is to find the current PC
  - Current PC = 0020 or if_pc_d

⇒ jump_addr_w = id_imm_w + if_pc_d



Address	Instruction	Basic code	Original code	Comments
0000	00004093	addi x9 x0 64	addi x9, x0, 64	# x9=&A[0]
0004	00000533	add x10 x0 x0	add x10, x0, x0	# sum=0
0008	000005b3	add x11 x0 x0	add x11, x0, x0	# i=0
000C	0004a603	lw x12 0(x9)	lw x12, 0(x9)	# x12=A[i]
0010	00c50533	add x10 x10 x12	add x10,x10,x12	# sum+=
0014	00448493	addi x9 x9 4	addi x9,x9,4	# &A[i++]
0018	00158593	addi x11 x11 1	addi x11,x11,1	# i++
001C	04000693	addi x13 x0 64	addi x13,x0,64	# x13=64
0020	fed5c6e3	blt x11 x13 -20	blt x11,x13,Loop	# Branch

To do: Calculate a target jump address

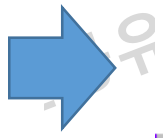
- Calculate a target address
  - blt x11, x13, -20
  - if_opcode_w = fed5_c6e3
    - alu_imm_w = -20,
    - id_ra = 11
    - id_rb = 13
    - id_branch = 4 (BLT).

⇒ jump_addr_w = if_pc_d + id_imm_w

```
always @ (*) begin
    branch_taken_w = 1'b0;
    jump_addr_w = 32'h0;
    case(id_branch_w)
        `BR_JUMP: begin
        end
        `BR_EQ: begin

        end
        `BR_NE: begin
        // Insert your code
        //{{{
        //}}}
        end
        `BR_LT: begin
        // Insert your code
        //{{{
            // Dummy Branch
            branch_taken_w = 1'b1;
            jump_addr_w = if_pc_d + id_imm_w;
        //}}}
        end
        `BR_GE: begin

        end
        `BR_LTU: begin

        end
        `BR_GEU: begin

        end
    endcase
end
```

Verification

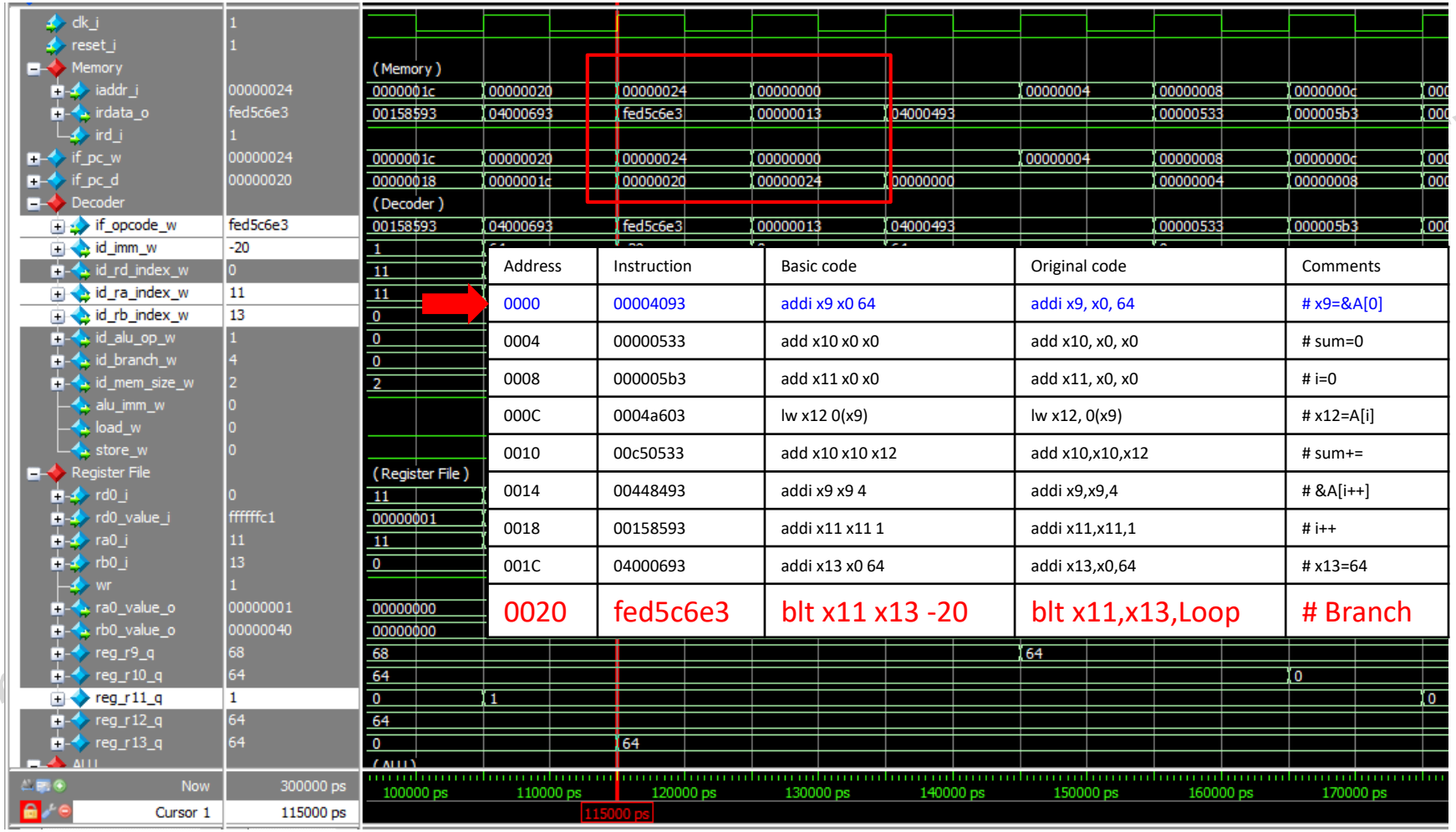
```
/* TODO: Branch, Jump and Link instructions */
always @ (*) begin
    branch_taken_w = 1'b0;
    jump_addr_w = 32'h0;
    // Insert your code
    //{{{
    case(id_branch_w)
        `BR_JUMP: begin
        end
        `BR_EQ: begin

        end
        `BR_NE: begin

        end
        `BR_LT: begin
            // Dummy Branch
            branch_taken_w = 1'b1;
        end
        `BR_GE: begin

        end
        `BR_LTU: begin

        end
        `BR_GEU: begin

        end
    endcase
    //}}}
end
```



```
always @ (*) begin
    branch_taken_w = 1'b0;
    jump_addr_w = 32'h0;
    case(id_branch_w)
        `BR_JUMP: begin
        end
        `BR_EQ: begin

        end
        `BR_NE: begin
        // Insert your code
        //{{{
        //}}}
        end
        `BR_LT: begin
        // Insert your code
        //{{{
            // Dummy Branch
            branch_taken_w = 1'b1;
            jump_addr_w = if_pc_d + id_imm_w;
        //}}}
        end
        `BR_GE: begin

        end
        `BR_LTU: begin

        end
        `BR_GEU: begin

        end
    endcase
end
```

Waveform: Before modification (Baseline)

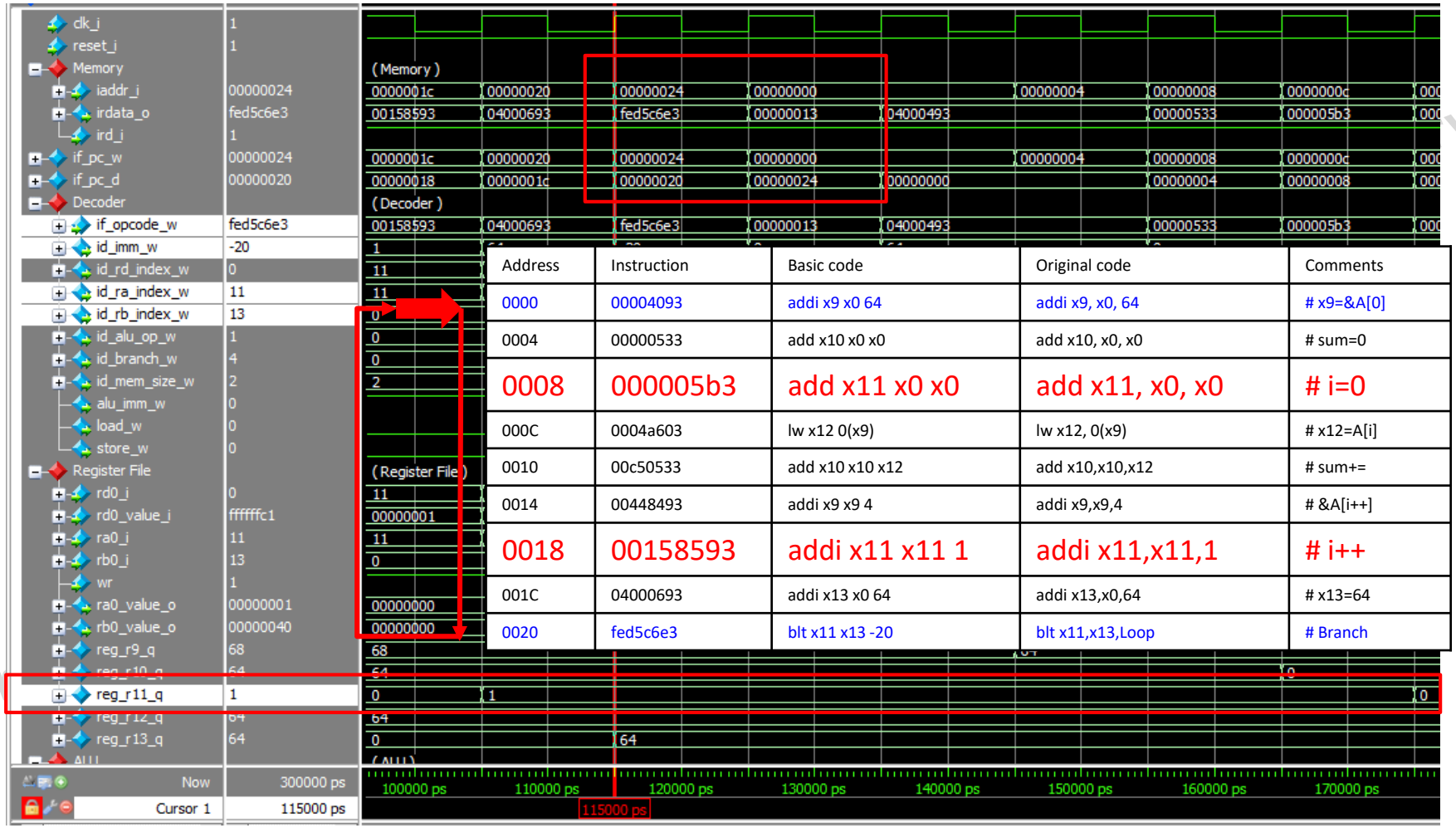
- Jump address = 0000



Address	Instruction	Basic code	Original code	Comments
0000	00004093	addi x9 x0 64	addi x9, x0, 64	# x9=&A[0]
0004	00000533	add x10 x0 x0	add x10, x0, x0	# sum=0
0008	000005b3	add x11 x0 x0	add x11, x0, x0	# i=0
000C	0004a603	lw x12 0(x9)	lw x12, 0(x9)	# x12=A[i]
0010	00c50533	add x10 x10 x12	add x10,x10,x12	# sum+=
0014	00448493	addi x9 x9 4	addi x9,x9,4	# &A[i++]
0018	00158593	addi x11 x11 1	addi x11,x11,1	# i++
001C	04000693	addi x13 x0 64	addi x13,x0,64	# x13=64
0020	fed5c6e3	blt x11 x13 -20	blt x11,x13,Loop	# Branch

Waveform: Before modification (Baseline)

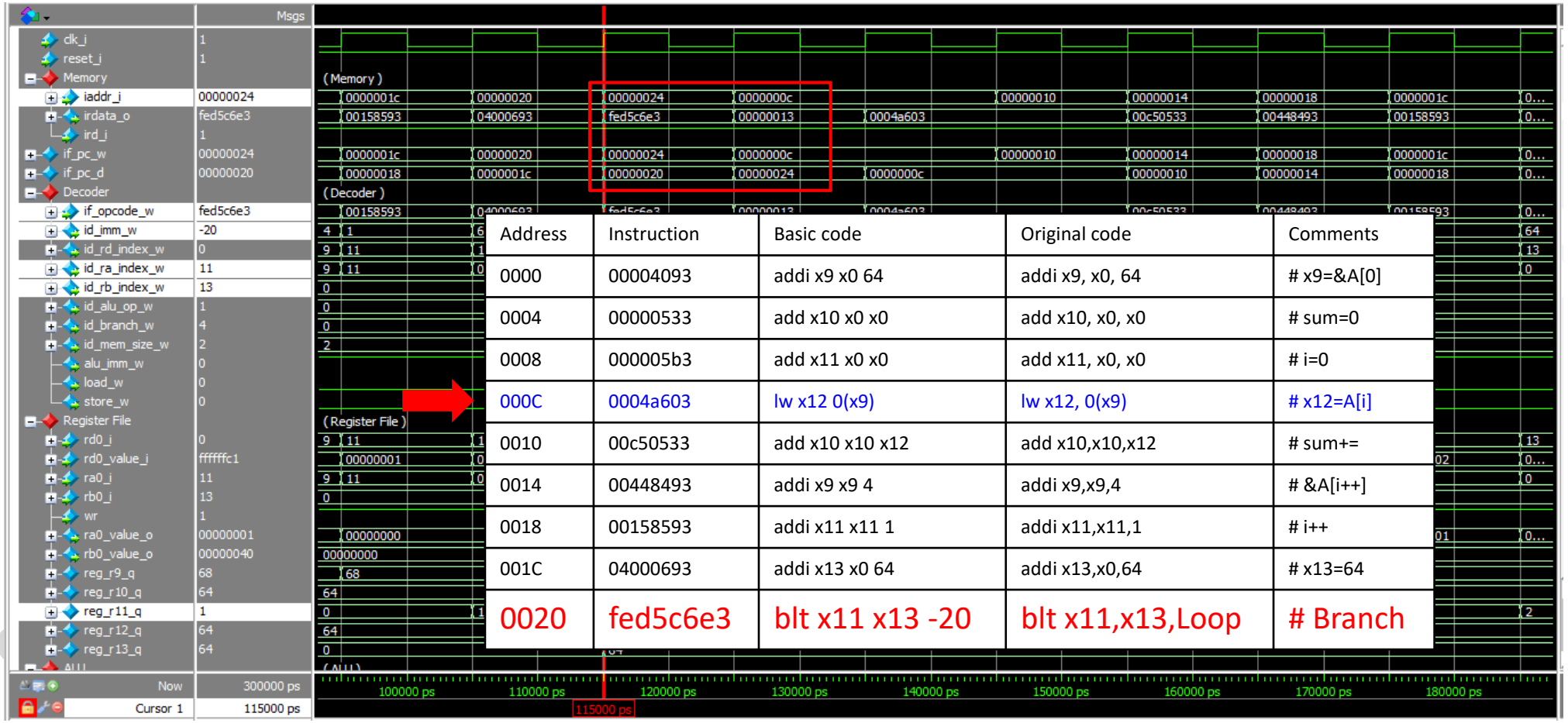
- Jump address = 0000: r11=0; then r11=r11+1=1 → r11 has two values 0, 1



Address	Instruction	Basic code	Original code	Comments
0000	00004093	addi x9 x0 64	addi x9, x0, 64	# x9=&A[0]
0004	00000533	add x10 x0 x0	add x10, x0, x0	# sum=0
0008	000005b3	add x11 x0 x0	add x11, x0, x0	# i=0
000C	0004a603	lw x12 0(x9)	lw x12, 0(x9)	# x12=A[i]
0010	00c50533	add x10 x10 x12	add x10,x10,x12	# sum+=
0014	00448493	addi x9 x9 4	addi x9,x9,4	# &A[i++]
0018	00158593	addi x11 x11 1	addi x11,x11,1	# i++
001C	04000693	addi x13 x0 64	addi x13,x0,64	# x13=64
0020	fed5c6e3	blt x11 x13 -20	blt x11,x13,Loop	# Branch

Waveform: After modification

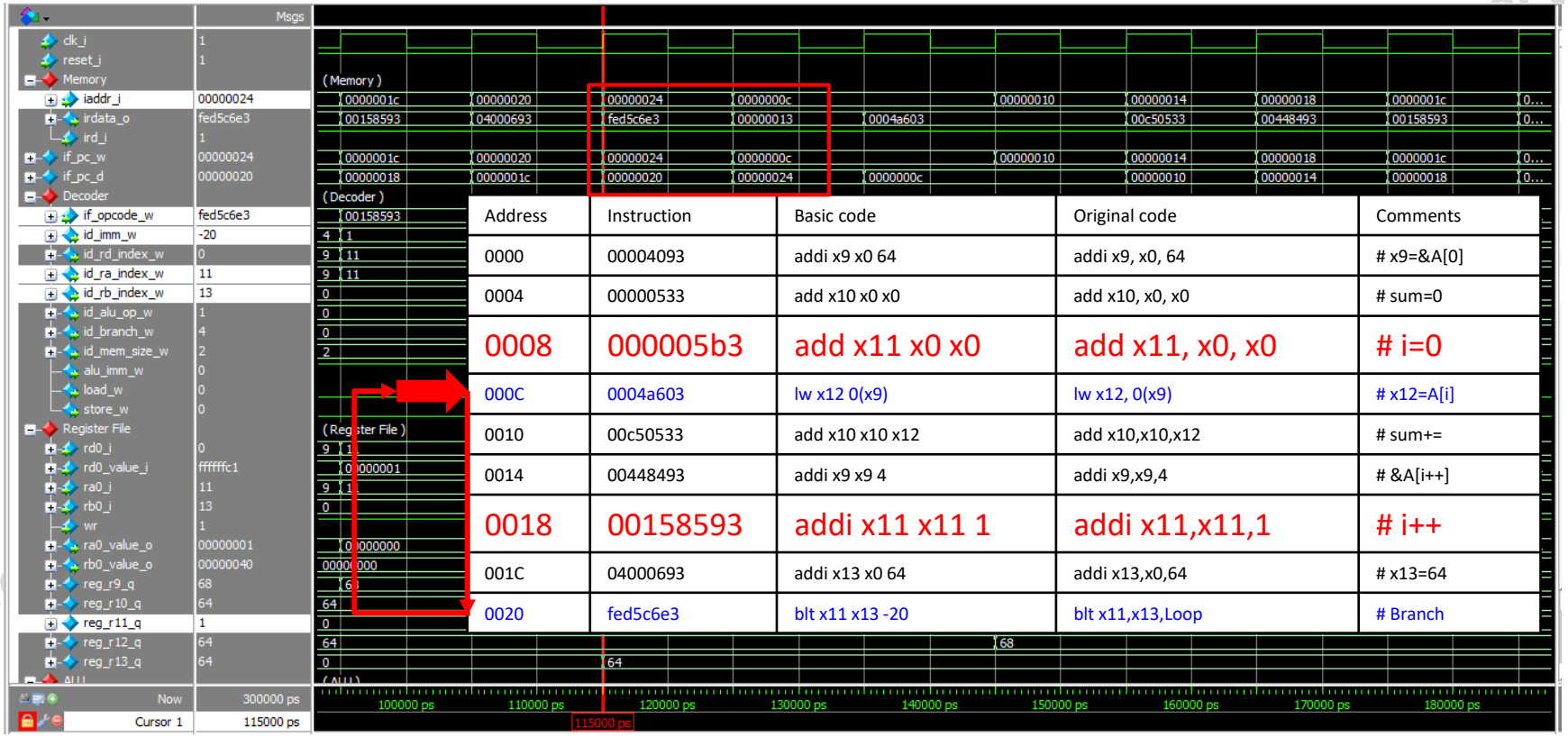
- Jump address = 000C



Address	Instruction	Basic code	Original code	Comments
0000	00004093	addi x9 x0 64	addi x9, x0, 64	# x9=&A[0]
0004	00000533	add x10 x0 x0	add x10, x0, x0	# sum=0
0008	000005b3	add x11 x0 x0	add x11, x0, x0	# i=0
000C	0004a603	lw x12 0(x9)	lw x12, 0(x9)	# x12=A[i]
0010	00c50533	add x10 x10 x12	add x10,x10,x12	# sum+=
0014	00448493	addi x9 x9 4	addi x9,x9,4	# &A[i++]
0018	00158593	addi x11 x11 1	addi x11,x11,1	# i++
001C	04000693	addi x13 x0 64	addi x13,x0,64	# x13=64
0020	fed5c6e3	blt x11 x13 -20	blt x11,x13,Loop	# Branch

Waveform: After modification

- 0008: r11=0 (Initialization)
- Jump address = 000C
  - 0018: r11=r11+1 → 1, 2, ….



Address	Instruction	Basic code	Original code	Comments
0000	00004093	addi x9 x0 64	addi x9, x0, 64	# x9=&A[0]
0004	00000533	add x10 x0 x0	add x10, x0, x0	# sum=0
0008	000005b3	add x11 x0 x0	add x11, x0, x0	# i=0
000C	0004a603	lw x12 0(x9)	lw x12, 0(x9)	# x12=A[i]
0010	00c50533	add x10 x10 x12	add x10,x10,x12	# sum+=
0014	00448493	addi x9 x9 4	addi x9,x9,4	# &A[i++]
0018	00158593	addi x11 x11 1	addi x11,x11,1	# i++
001C	04000693	addi x13 x0 64	addi x13,x0,64	# x13=64
0020	fed5c6e3	blt x11 x13 -20	blt x11,x13,Loop	# Branch

To do: Calculate an enable signal

- Calculate an enable signal for a branch instruction
  - blt x11, x13, -20
  - if_opcode_w = fed5_c6e3
    - alu_imm_w = -20,
    - id_ra = 11
    - id_rb = 13
    - id_branch = 4 (BLT).

⇒ Dummy: Branch-Less-Than is an unconditional instruction

Instruction	Syntax	Description	Execution
BEQ	beq rs1, rs2, imm	Branch if = zero	PC <= (reg[rs1] == reg[rs2]) ? PC + imm : (PC+4)
BNE	bne rs1, rs2, imm	Branch if ≠ zero	PC <= (reg[rs1] != reg[rs2]) ? PC + imm : (PC+4)
BLT	blt rs1, rs2, imm	Branch if < (signed)	PC <= (reg[rs1] $<_s$ reg[rs2]) ? PC + imm : (PC+4)
BGE	bge rs1, rs2, imm	Branch if ≥ (signed)	PC <= (reg[rs1] $\geq_s$ reg[rs2]) ? PC + imm : (PC+4)
BLTU	bltu rs1, rs2, imm	Branch if < (Unsigned)	PC <= (reg[rs1] $<_u$ reg[rs2]) ? PC + imm : (PC+4)
BGEU	bgeu rs1, rs2, imm	Branch if ≥ (Unsigned)	PC <= (reg[rs1] $\geq_u$ reg[rs2]) ? PC + imm : (PC+4)

```
always @ (*) begin
    branch_taken_w = 1'b0;
    jump_addr_w = 32'h0;
    case(id_branch_w)
        `BR_JUMP: begin
        end
        `BR_EQ: begin

        end
        `BR_NE: begin
        // Insert your code
        //{{{
        //}}}
        end
        `BR_LT: begin
        // Insert your code
        //{{{
            // Dummy Branch
            branch_taken_w = 1'b1;
            jump_addr_w = if_pc_d + id_imm_w;
        //}}}
        end
        `BR_GE: begin

        end
        `BR_LTU: begin

        end
        `BR_GEU: begin

        end
    endcase
end
```

To do: Calculate an enable signal

```
case(id_branch_w)
    `BR_JUMP: begin
    end
    `BR_EQ: begin

    end
    `BR_NE: begin
    // Insert your code
    //{{{
    //}}}
    end
    `BR_LT: begin
    // Insert your code
    //{{{
        // Dummy Branch
        branch_taken_w = ($signed(ra_value_r) < $signed(rb_value_r)) ? 1'b1 : 1'b0;
        jump_addr_w = if_pc_d + id_imm_w;
    //}}}
```

Calculate an enable signal for a branch

blt x11, x13, -20

if_opcode_w = fed5_c6e3

alu_imm_w = -20,

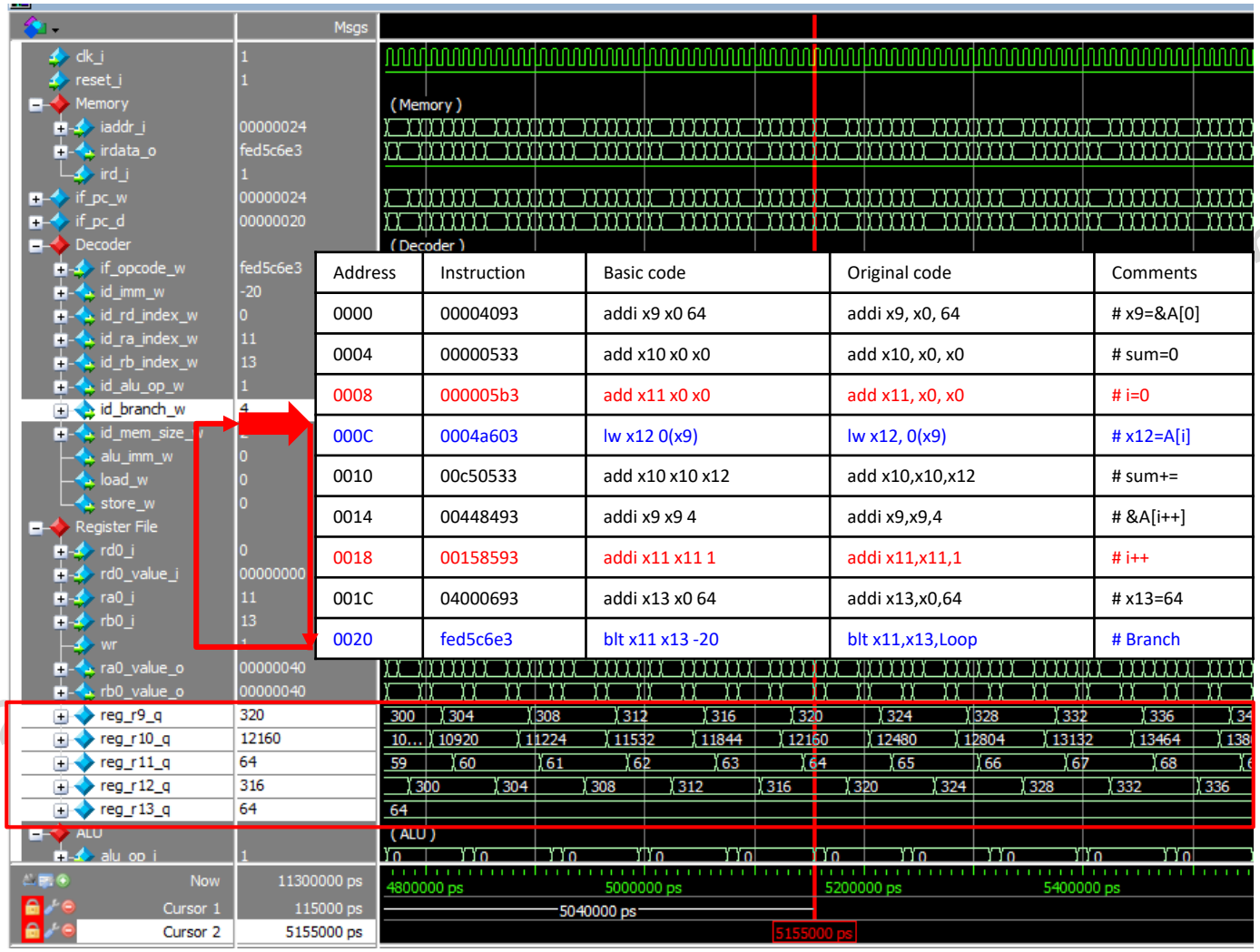
id_ra = 11

id_rb = 13

id_branch = 4 (BLT).

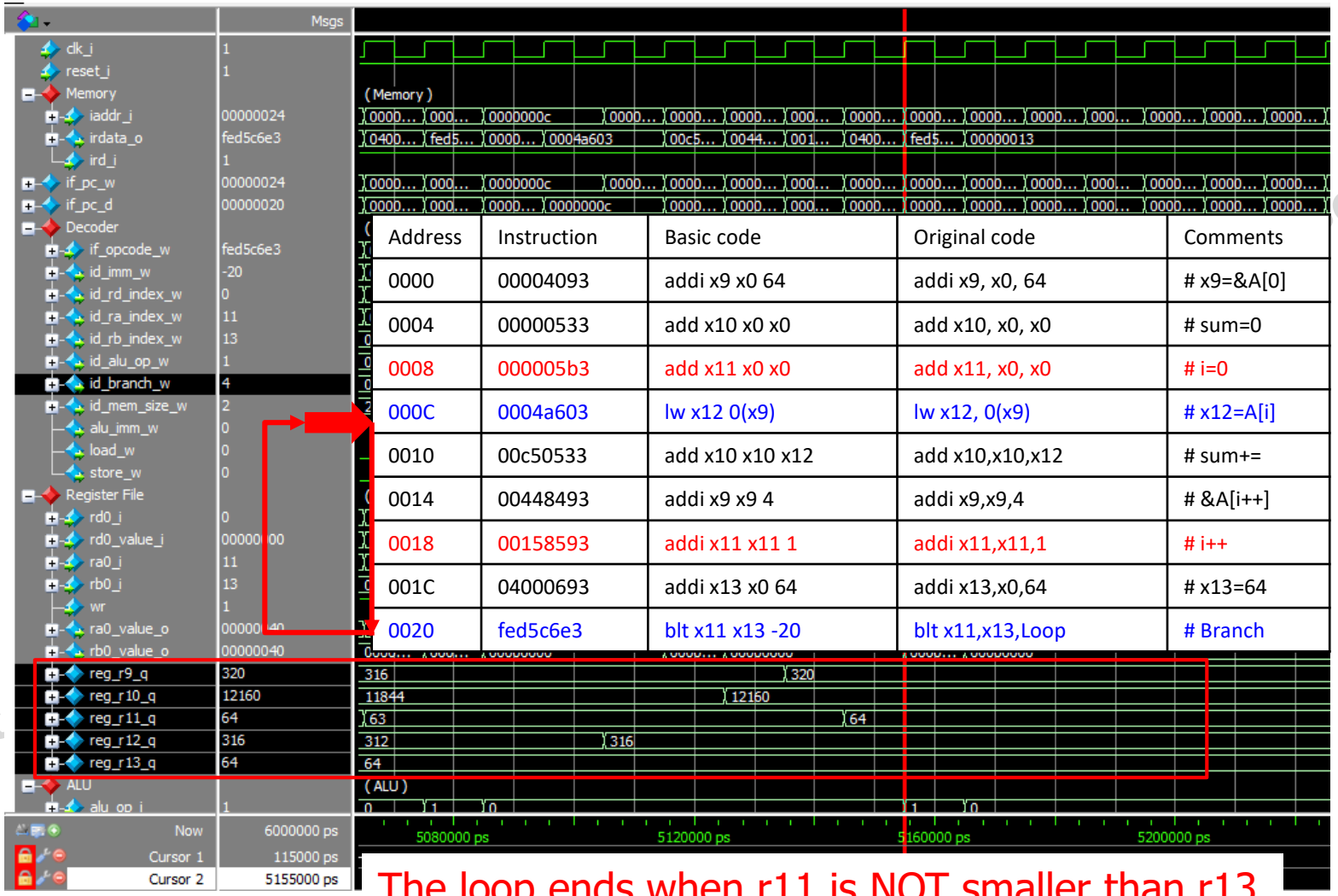
Instruction	Syntax	Description	Execution
BEQ	beq rs1, rs2, imm	Branch if = zero	PC <= (reg[rs1] == reg[rs2]) ? PC + imm : (PC+4)
BNE	bne rs1, rs2, imm	Branch if ≠ zero	PC <= (reg[rs1] != reg[rs2]) ? PC + imm : (PC+4)
BLT	blt rs1, rs2, imm	Branch if < (signed)	PC <= (reg[rs1] $<_s$ reg[rs2]) ? PC + imm : (PC+4)
BGE	bge rs1, rs2, imm	Branch if ≥ (signed)	PC <= (reg[rs1] $\geq_s$ reg[rs2]) ? PC + imm : (PC+4)
BLTU	bltu rs1, rs2, imm	Branch if < (Unsigned)	PC <= (reg[rs1] $<_u$ reg[rs2]) ? PC + imm : (PC+4)
BGEU	bgeu rs1, rs2, imm	Branch if ≥ (Unsigned)	PC <= (reg[rs1] $\geq_u$ reg[rs2]) ? PC + imm : (PC+4)

Waveform: Before modification



Address	Instruction	Basic code	Original code	Comments
0000	00004093	addi x9 x0 64	addi x9, x0, 64	# x9=&A[0]
0004	00000533	add x10 x0 x0	add x10, x0, x0	# sum=0
0008	000005b3	add x11 x0 x0	add x11, x0, x0	# i=0
000C	0004a603	lw x12 0(x9)	lw x12, 0(x9)	# x12=A[i]
0010	00c50533	add x10 x10 x12	add x10,x10,x12	# sum+=
0014	00448493	addi x9 x9 4	addi x9,x9,4	# &A[i++]
0018	00158593	addi x11 x11 1	addi x11,x11,1	# i++
001C	04000693	addi x13 x0 64	addi x13,x0,64	# x13=64
0020	fed5c6e3	blt x11 x13 -20	blt x11,x13,Loop	# Branch

Waveform: After modification



Address	Instruction	Basic code	Original code	Comments
0000	00004093	addi x9 x0 64	addi x9, x0, 64	# x9=&A[0]
0004	00000533	add x10 x0 x0	add x10, x0, x0	# sum=0
0008	000005b3	add x11 x0 x0	add x11, x0, x0	# i=0
000C	0004a603	lw x12 0(x9)	lw x12, 0(x9)	# x12=A[i]
0010	00c50533	add x10 x10 x12	add x10,x10,x12	# sum+=
0014	00448493	addi x9 x9 4	addi x9,x9,4	# &A[i++]
0018	00158593	addi x11 x11 1	addi x11,x11,1	# i++
001C	04000693	addi x13 x0 64	addi x13,x0,64	# x13=64
0020	fed5c6e3	blt x11 x13 -20	blt x11,x13,Loop	# Branch

The loop ends when r11 is NOT smaller than r13

Road map

Review

Load, store instructions

RISC-V core integration

RISC-V GNU Compiler

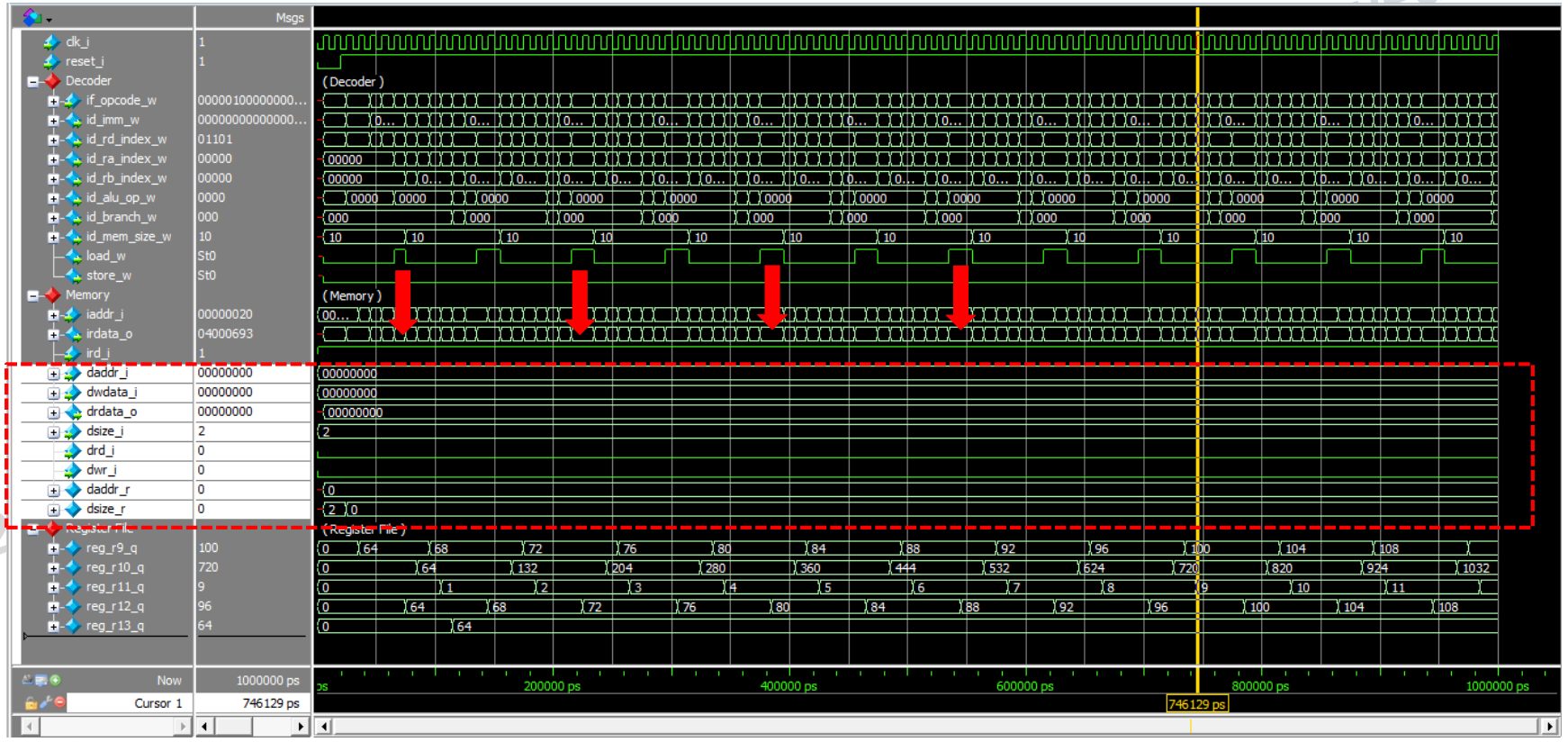
Optimization

Motivation: Load instruction

- Baseline code: Have load instructions, but there is no access to data memory.

⇒ Dummy connections for LOAD

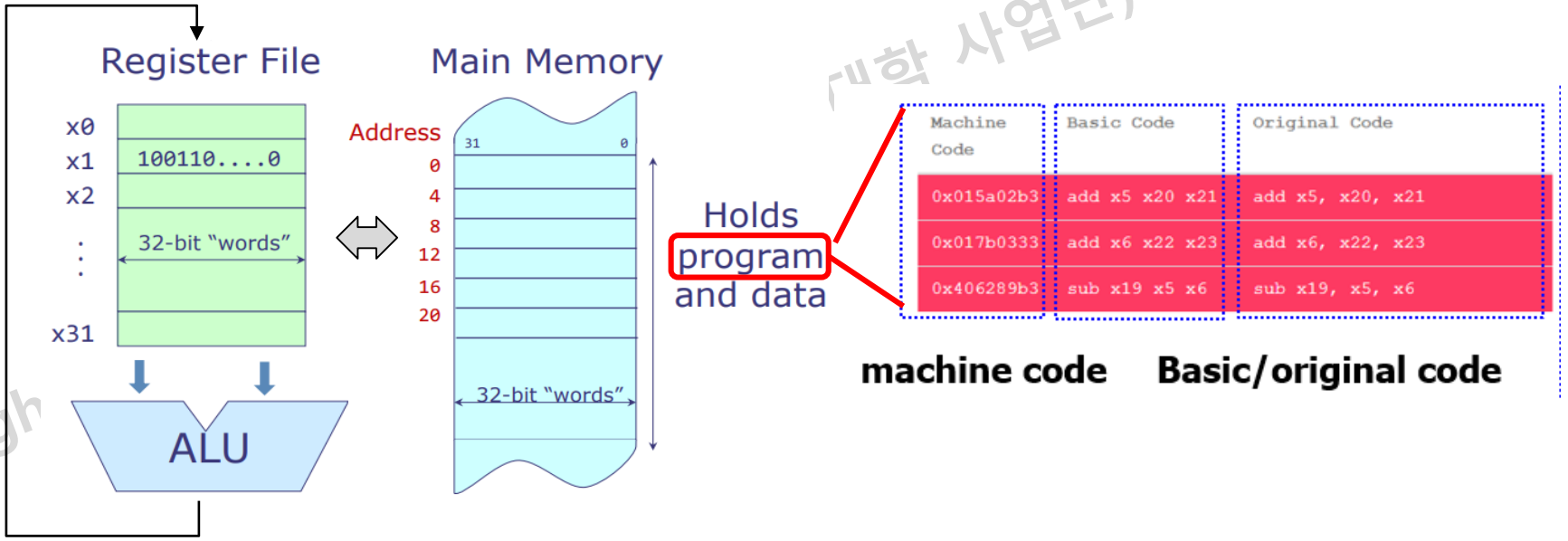
```
1   04000493
2   00000533
3   000005b3
4   0004a603
5   00c50533
6   00448493
7   00158593
8   04000693
9   fed5c6e3
10  00000013
11  00000013
12  00000013
13  00000013
14  00000013
15  00000013
16  00000013
17  00000000
18  00000001
19  00000002
20  00000003
```



Recap: Components of MicroProcessor

- Machine language can directly represent this structure.
  - Computing component (Arithmetic Logic Unit (ALU))
  - Main memory: Hold program and data

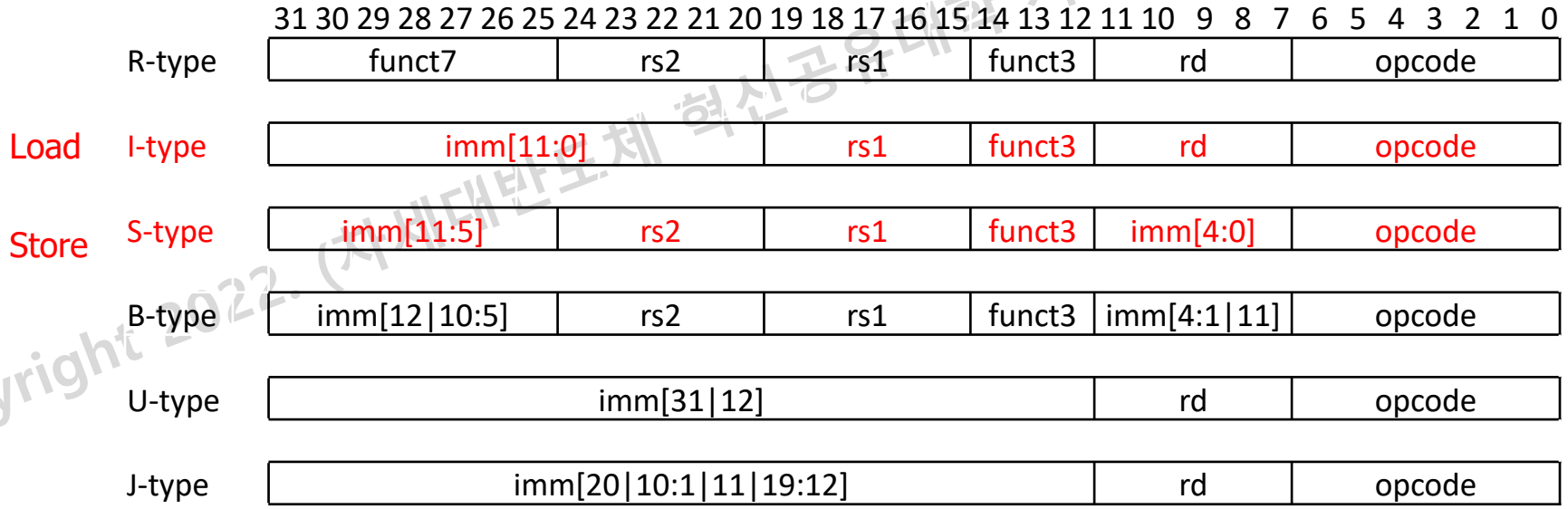
⇒ We manually made a program file while working with Venus



Recap: RISC-V Core 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
  - Last week: **Instruction ports** of memory
  - This week: **Data ports** of memory

  ⇒ Load and store instructions



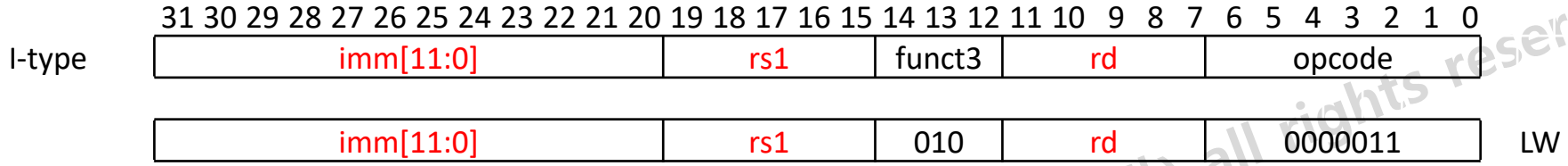
Load instruction

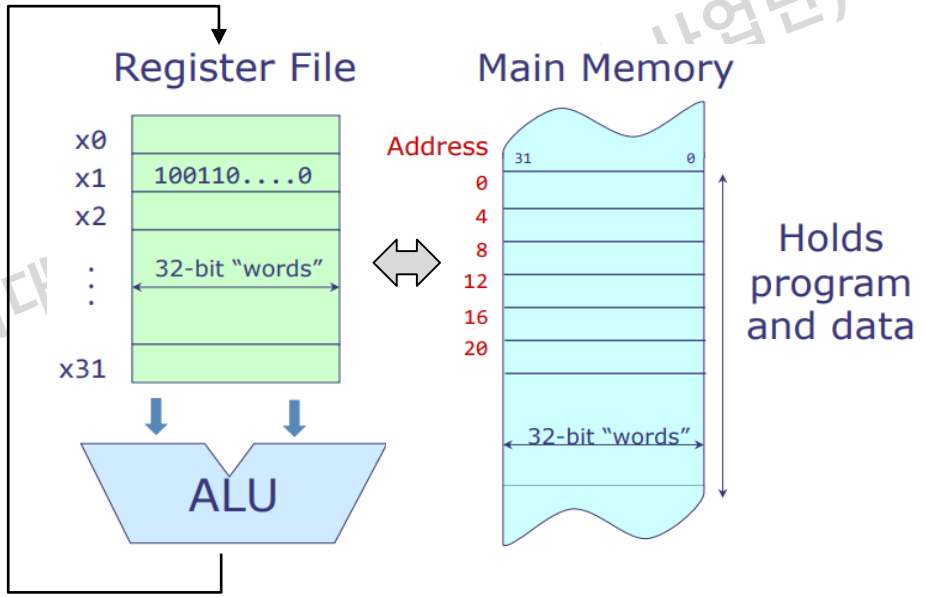
- Load (I-type)
  - Opcode = 0000011
  - Load byte (LB), load half (LH), load word (LW)
    - Load Byte: funct3 == 000
    - Load Half: funct3 == 001
    - Load Word: funct3 == 010
    - Load Byte (unsigned): funct3 == 100
    - Load Half (unsigned): funct3 == 101

	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

Load instruction

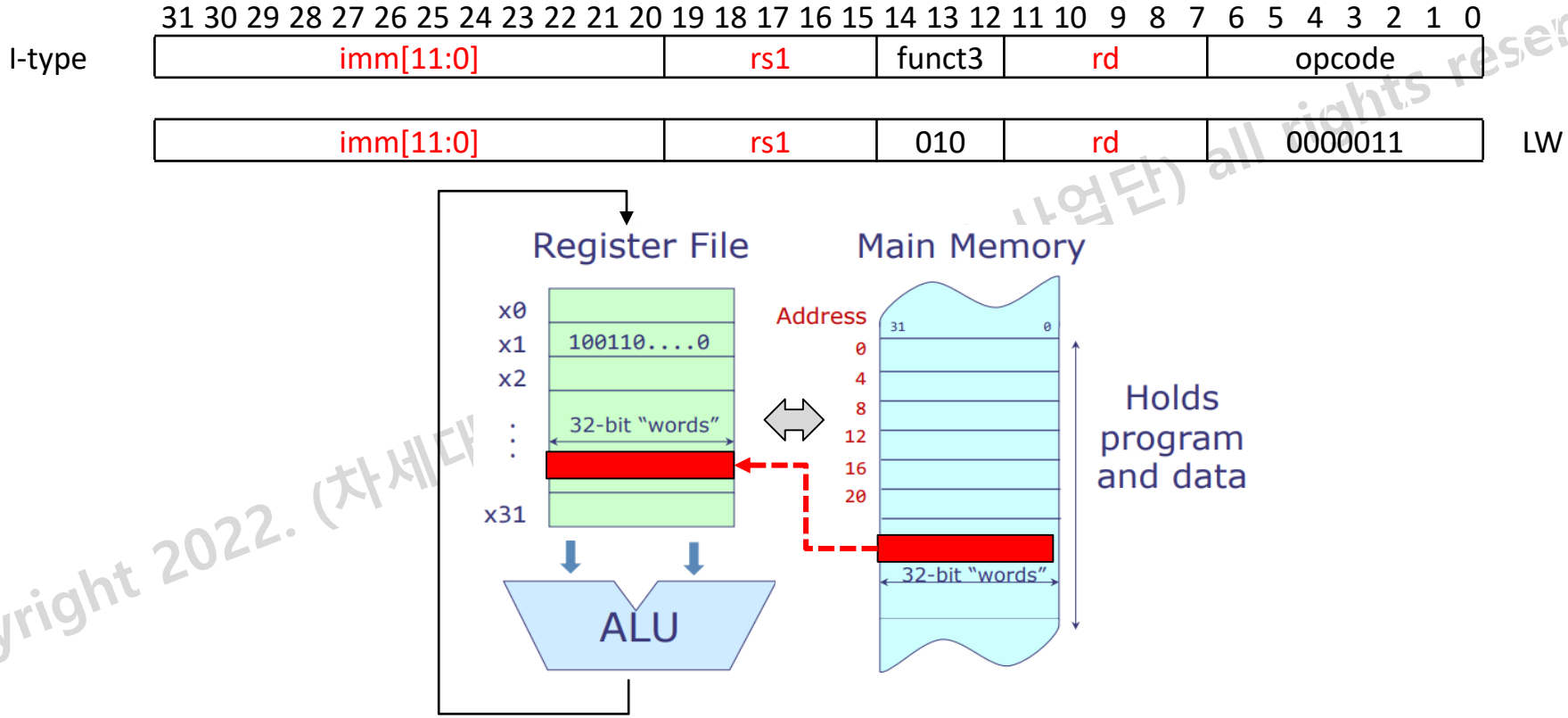
- Load: Read data from Memory and save it into Register
  - reg[rd] ← MEM(reg[rs1] + imm)





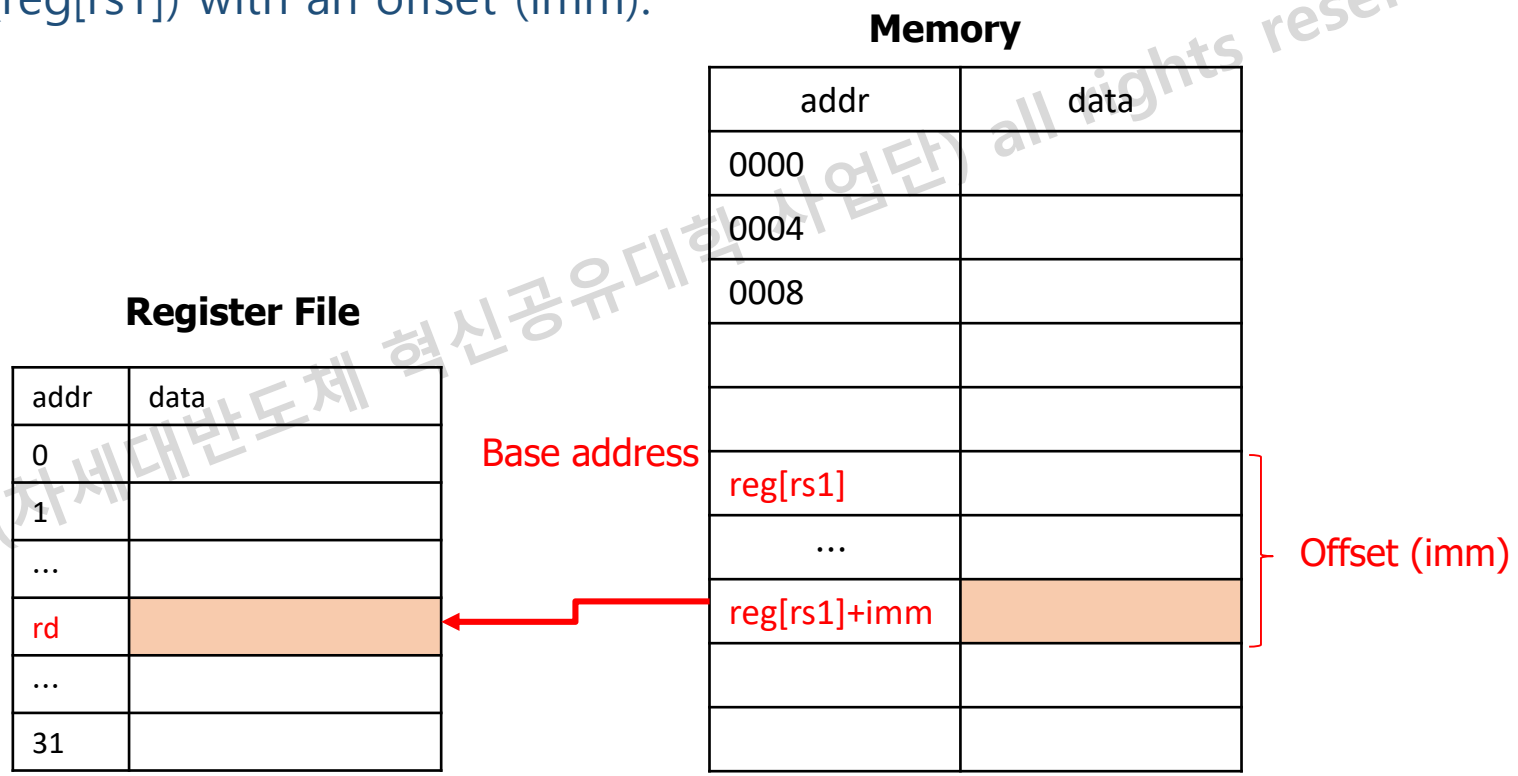
Load instruction

- Load: Read data from Memory and save it into Register
  - reg[rd] ← MEM(reg[rs1] + imm)



Load instruction: Addressing

- Load: Read data from Memory and save it into Register
  - reg[rd] ← MEM(reg[rs1] + imm)
  - Memory
    - Base address (reg[rs1]) with an offset (imm).
  - Register File
    - Address rd



Store instruction

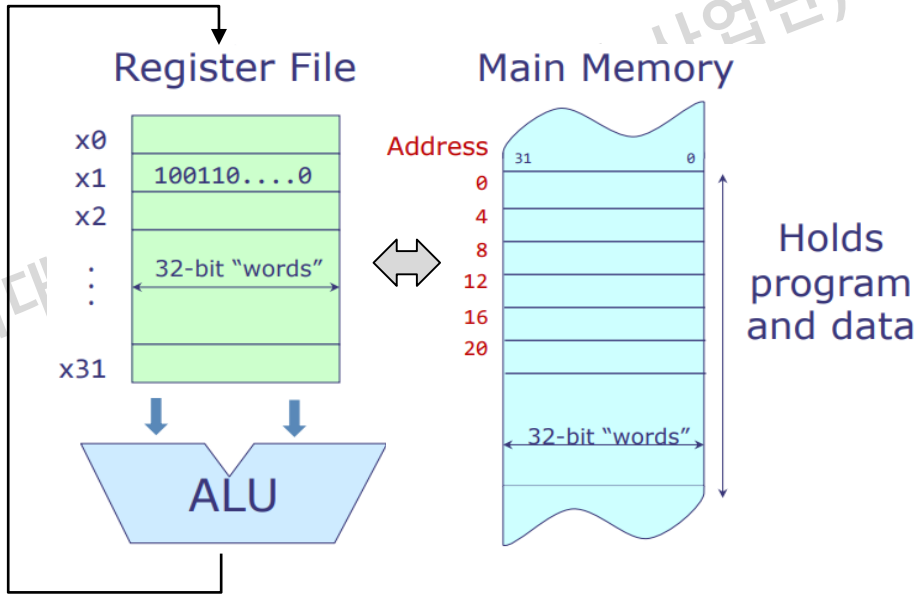
- Store (S-type)
  - Opcode = 0100011
  - Store byte (SB), store half (SH), store word (SW)
    - Store Byte: funct3 == 000
    - Store Haft: funct3 == 001
    - Store Word: funct3 == 010

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

Store instruction

- Store: Read data from Register File and save it into Memory
  - MEM(reg[rs1] + imm) ← reg[rs2]

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

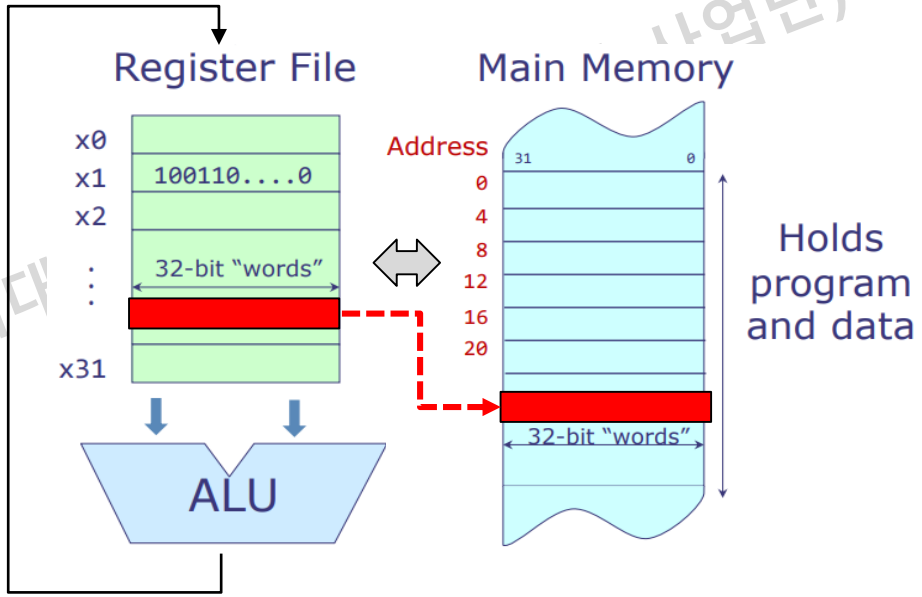


Store instruction

- Store: Read data from Register File and save it into Memory
  - MEM(reg[rs1] + imm) ← reg[rs2]

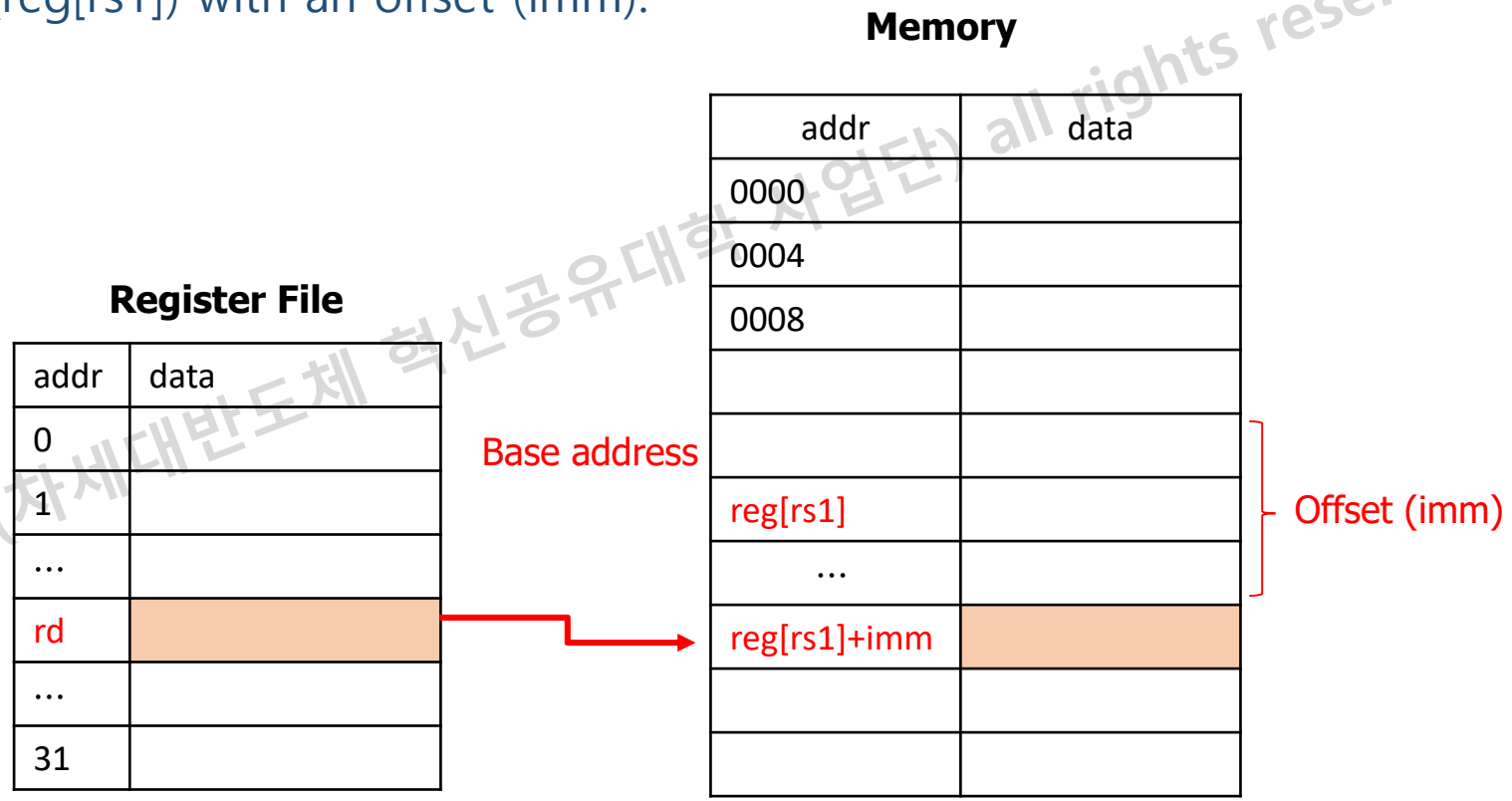
S-type	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
	imm[11:5]	rs2	rs1	010	imm[4:0]	0100011

SW



Store instruction: Addressing

- Store: Read data from Register File and save it into Memory
  - MEM(reg[rs1] + imm) ← reg[rs2]
  - Memory
    - Base address (reg[rs1]) with an offset (imm).
  - Register File
    - Address rs2



Example

- A simple program calculates a sum of all elements in an array.

```
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
```

Load instruction

Example

- Register x9 stores the base address (0040 or 64)

Address	Instruction	Basic code	Original code	Comments
0000	00004093	addi x9 x0 64	addi x9, x0, 64	# x9=&A[0]
0004	00000533	add x10 x0 x0	add x10, x0, x0	# sum=0
0008	000005b3	add x11 x0 x0	add x11, x0, x0	# i=0
000C	0004a603	lw x12 0(x9)	lw x12, 0(x9)	# x12=A[i]
0010	00c50533	add x10 x10 x12	add x10,x10,x12	# sum+=
0014	00448493	addi x9 x9 4	addi x9,x9,4	# &A[i++]
0018	00158593	addi x11 x11 1	addi x11,x11,1	# i++
001C	04000693	addi x13 x0 64	addi x13,x0,64	# x13=64
0020	fed5c6e3	blt x11 x13 -20	blt x11,x13,Loop	# Branch

Program

addr	data
0000	00004093
0004	00000533
0008	000005b3
…	…
…	…
0040	00000000
0044	00000001
0048	00000002
…	…

Memory

Example

- lw x12 0(x9)
  - Read Mem[x9=64]: Base address at x9 (=60 (0x0040)offset = 0
  - Save it to x12

Address	Instruction	Basic code	Original code	Comments
0000	00004093	addi x9 x0 64	addi x9, x0, 64	# x9=&A[0]
0004	00000533	add x10 x0 x0	add x10, x0, x0	# sum=0
0008	000005b3	add x11 x0 x0	add x11, x0, x0	# i=0
000C	0004a603	lw x12 0(x9)	lw x12, 0(x9)	# x12=A[i]
0010	00c50533	add x10 x10 x12	add x10,x10,x12	# sum+=
0014	00448493	addi x9 x9 4	addi x9,x9,4	# &A[i++]
0018	00158593	addi x11 x11 1	addi x11,x11,1	# i++
001C	04000693	addi x13 x0 64	addi x13,x0,64	# x13=64
0020	fed5c6e3	blt x11 x13 -20	blt x11,x13,Loop	# Branch

Program

addr	data
0000	00004093
0004	00000533
0008	000005b3
…	…
…	…
0040	00000000
0044	00000001
0048	00000002
…	…

Memory

Example

- Move to the next element in an array
  - addi x9, x9, 4

Address	Instruction	Basic code	Original code	Comments
0000	00004093	addi x9 x0 64	add x9, x0, 64	# x9=&A[0]
0004	00000533	add x10 x0 x0	add x10, x0, x0	# sum=0
0008	000005b3	add x11 x0 x0	add x11, x0, x0	# i=0
000C	0004a603	lw x12 0(x9)	lw x12, 0(x9)	# x12=A[i]
0010	00c50533	add x10 x10 x12	add x10,x10,x12	# sum+=
0014	00448493	addi x9 x9 4	addi x9,x9,4	# &A[i++]
0018	00158593	addi x11 x11 1	addi x11,x11,1	# i++
001C	04000693	addi x13 x0 64	addi x13,x0,64	# x13=64
0020	fed5c6e3	blt x11 x13 -20	blt x11,x13,Loop	# Branch

Program

addr	data
0000	00004093
0004	00000533
0008	000005b3
…	…
…	…
0040	00000000
0044	00000001
0048	00000002
…	…

Memory

Example

- lw x12 x9
  - Read Mem[x9=68], Base address at x9 (=68 (0x0044), offset = 0
  - Save it to x12

Address	Instruction	Basic code	Original code	Comments
0000	00004093	addi x9 x0 64	addi x9, x0, 64	# x9=&A[0]
0004	00000533	add x10 x0 x0	add x10, x0, x0	# sum=0
0008	000005b3	add x11 x0 x0	add x11, x0, x0	# i=0
000C	0004a603	lw x12 0(x9)	lw x12, 0(x9)	# x12=A[i]
0010	00c50533	add x10 x10 x12	add x10,x10,x12	# sum+=
0014	00448493	addi x9 x9 4	addi x9,x9,4	# &A[i++]
0018	00158593	addi x11 x11 1	addi x11,x11,1	# i++
001C	04000693	addi x13 x0 64	addi x13,x0,64	# x13=64
0020	fed5c6e3	blt x11 x13 -20	blt x11,x13,Loop	# Branch

Program

addr	data
0000	00004093
0004	00000533
0008	000005b3
…	…
…	…
0040	00000000
0044	00000001
0048	00000002
…	…

Memory

Lab 1: Load, store instructions

- Lab 1:
  - Implement load and store instructions
  - Run a simulation
  - Show the output result

RISC-V

- RISC-V Core is connected to Memory
  - Instruction ports
  - Data port

  ⇒ There is no signal from ALU, register file, decoder, program counter.

```
module riscv_core_sim #(
  parameter     PC_SIZE  = 32,
  parameter     RESET_SP = 32'h0000
)(
  input         clk_i,      // Clock
  input         reset_i,    // Reset
  output        lock_o,     // Lock

  output [31:0] iaddr_o,    // Instruction from Memory
  input  [31:0] irdata_i,   // Instruction address
  output        ird_o,      // Read request

  output  [31:0] daddr_o,   // Read/Write address
  output  [31:0] dwdata_o,  // Write Data
  input   [31:0] drdata_i,  // Read data
  output   [1:0] dsize_o,
  output         drd_o,     // Write request
  output         dwr_o      // Read/Write Enable
);
```

Instruction Program

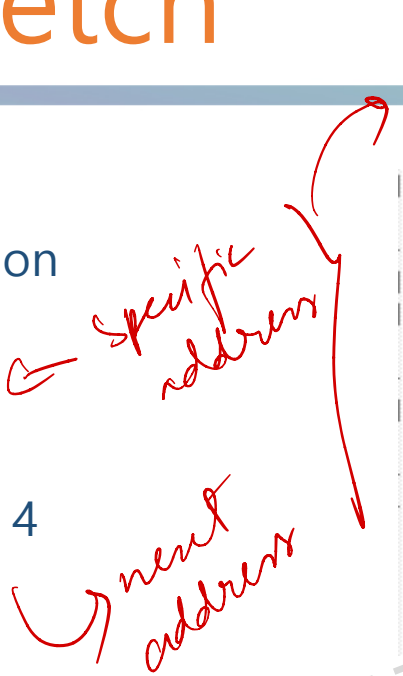
Data

mem.hex | riscv

```
1   04000493
2   00000533
3   000005b3
4   0004a603
5   00c50533
6   00448493
7   00158593
8   04000693
9   fed5c6e3
10
11
12
13
14
15
16  00000000
17  00000001
18  00000002
19  00000003
20  00000004
21  00000005
22  00000006
23  00000007
24  00000008
```

Instruction Fetch

- Program counter
  - Store the next instruction
  - If (branch_taken_w)
    - PC ← jump_addr
  - Else
    - PC ← curr_addr + 4



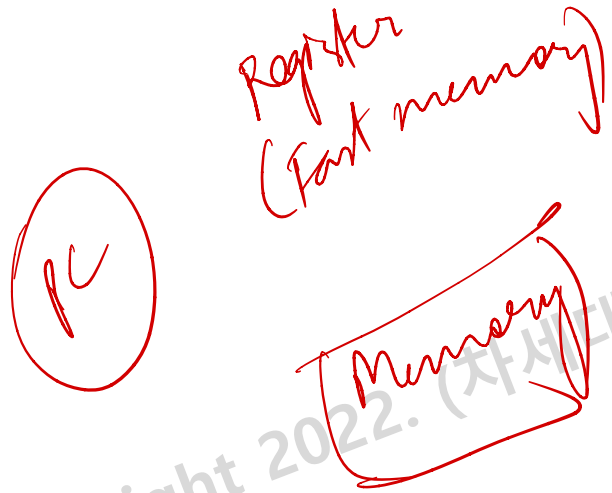

```
//------------------------------------------------------
// TODO: Instruction Fetch
//------------------------------------------------------
always @ (posedge clk_i or negedge reset_i) begin
    if (~reset_i) begin
        if_pc_w <= RESET_SP;
    end
    else begin
        if_pc_w <= if_next_addr_w;
    end
end
assign iaddr_o = if_pc_w;
assign ird_o   = 1'b1;
assign lock_o  = 1'b0;
assign if_opcode_w = irdata_i;
```

```
//------------------------------------------------------
// Program Counter
//------------------------------------------------------
riscv_pc #(.RESET_SP(RESET_SP))
u_pc(
    ./*input         */clk_i(clk_i),
    ./*input         */reset_i(reset_i),
    ./*input         */ird(ird_o),
    ./*input         */branch_taken_w(branch_taken_w),
    ./*input [31:0] */jump_addr_w(jump_addr_w),
    ./*output [31:0]*/if_next_addr_w(if_next_addr_w)
);
//------------------------------------------------------
```

Instruction Fetch

- Program counter
  - Store the next instruction

- For an instruction address,
  the core fetches an instruction from memory.




```
//------------------------------------------------------
// TODO: Instruction Fetch
//------------------------------------------------------
always @ (posedge clk_i or negedge reset_i) begin
    if (~reset_i) begin
        if_pc_w <= RESET_SP;
    end
    else begin
        if_pc_w <= if_next_addr_w;
    end
end
assign iaddr_o = if_pc_w;
assign ird_o   = 1'b1;
assign lock_o  = 1'b0;
assign if_opcode_w = irdata_i;
```

```
//------------------------------------------------------
// Program Counter
//------------------------------------------------------
riscv_pc #(.RESET_SP(RESET_SP))
u_pc(
    ./*input         */clk_i(clk_i),
    ./*input         */reset_i(reset_i),
    ./*input         */ird(ird_o),
    ./*input         */branch_taken_w(branch_taken_w),
    ./*input [31:0] */jump_addr_w(jump_addr_w),
    ./*output [31:0]*/if_next_addr_w(if_next_addr_w)
);
//------------------------------------------------------
```

Decode

- For an instruction, Decoder extracts all information including:
- Addresses of source and dest. registers for Register File (ra, rb, rd).
- ALU opcode (alu_op)
- Immediate (imm)
- Branch, jump & link types (branch)

```
//-----------------------------------------------------
// TODO: Instruction Fetch
//-----------------------------------------------------
always @ (posedge clk_i or negedge reset_i) begin
    if (~reset_i) begin
        if_pc_w <= RESET_SP;
    end
    else begin
        if_pc_w <= if_next_addr_w;
    end
end
assign iaddr_o = if_pc_w;
assign ird_o   = 1'b1;
assign lock_o  = 1'b0;
assign if_opcode_w = irdata_i;
```

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

Register file

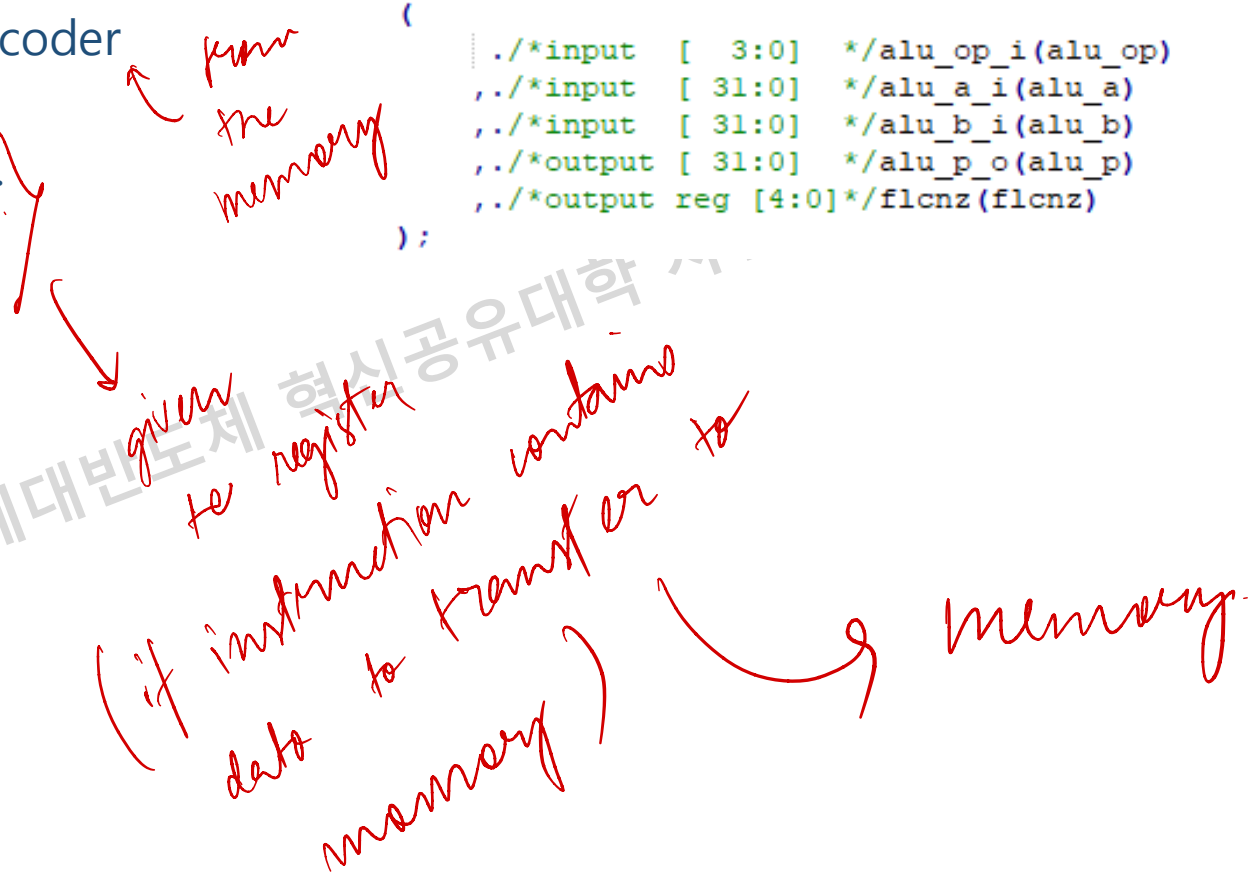
- Given addresses of source and dest. registers (ra, rb, rd), Register File can
  - Output a value of a register.
  - Store or update a new value of a register

```
//------------------------------------------------------
// Register File
//------------------------------------------------------
riscv_regfile
u_regfile
(
     ./*input            */clk_i(clk_i)
    ,./*input            */rst_i(reset_i)
    ,./*input  [  4:0]   */rd0_i(rd_index_w)
    ,./*input  [ 31:0]   */rd0_value_i(rd_value_w)
    ,./*input  [  4:0]   */ra0_i(id_ra_index_w)
    ,./*input  [  4:0]   */rb0_i(id_rb_index_w)
    ,./*input            */wr(rd_we_w)
    ,./*output [ 31:0]   */ra0_value_o(ra_value_r)
    ,./*output [ 31:0]   */rb0_value_o(rb_value_r)
);

// Dummy register file ports
always@(*) begin
    rd_index_w = 5'h0;
    rd_value_w = 32'h0;
    rd_we_w    = 1'b0;
    // Insert your code
    //{{{
    //rd_index_w = id_rd_index_w;
    //rd_value_w = alu_p;
    //rd_we_w    = 1'b1;
    //}}}
end
```

ALU

- ALU calculates the results
  - Inputs:
    - Registers from Register File
    - Operation from Decoder
  - Output
    - ALU output (alu_p).
    - Flags (flcnz).

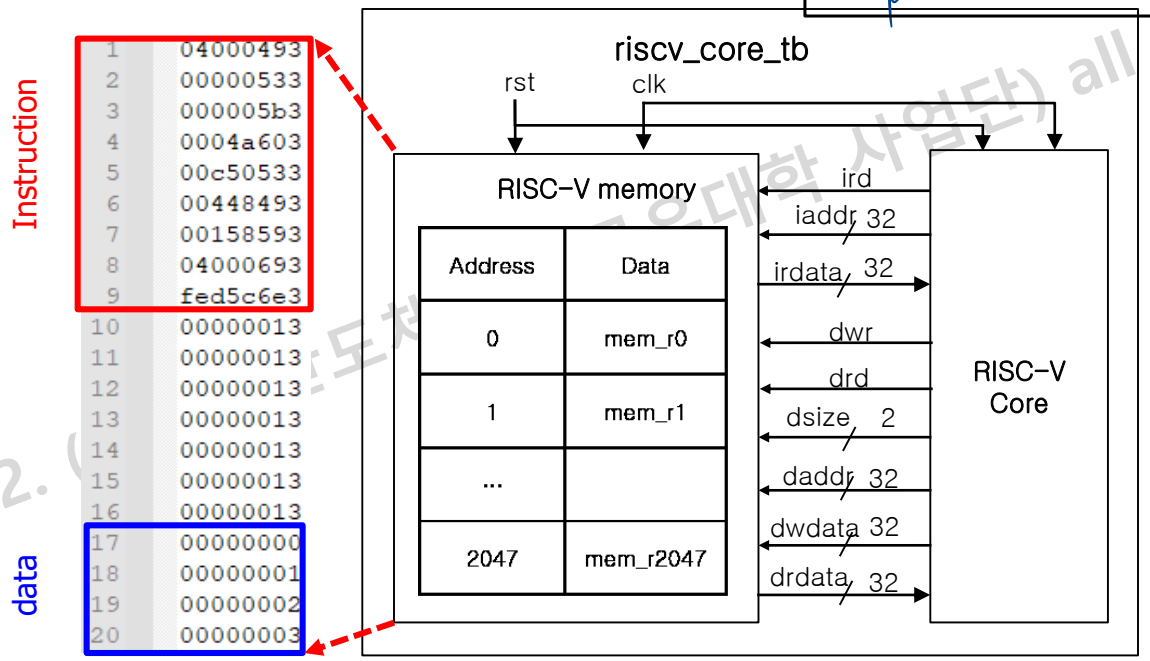
```
//-----------------------------------------------------
// ALU
//-----------------------------------------------------
riscv_alu
u_alu
(
     ./*input  [  3:0]  */alu_op_i(alu_op)
    ,./*input  [ 31:0]  */alu_a_i(alu_a)
    ,./*input  [ 31:0]  */alu_b_i(alu_b)
    ,./*output [ 31:0]  */alu_p_o(alu_p)
    ,./*output reg [4:0]*/flcnz(flcnz)
);
```



Test bench (riscv_core_sim_tb.v)

- Test bench includes
  - A memory with initialized instructions and data.
  - A RISC-V core
- For test cases, only clock and reset are required

mem.hex file already provides all the data.



Test bench

- Test bench includes
  - A memory
  - A RISC-V core

```
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;
        ...
```

```
riscv_memory #(.FIRMWARE("mem.hex"))
u_riscv_memory
(
  ./*input          */clk_i(clk_i),
  ./*input          */reset_i(reset_i),
  ./*input  [31:0]  */iaddr_i(iaddr),
  ./*output [31:0]  */irdata_o(irdata),
  ./*input          */ird_i(ird),
  ./*input  [31:0]  */daddr_i(daddr),
  ./*input  [31:0]  */dwdata_i(dwdata),
  ./*output [31:0]  */drdata_o(drdata),
  ./*input   [1:0]  */dsize_i(dsize),
  ./*input          */drd_i(drd),
  ./*input          */dwr_i(dwr)
);

riscv_core_sim
u_riscv_core_sim
(
  ./*input          */clk_i(clk_i),
  ./*input          */reset_i(reset_i),
  ./*output         */lock_o(lock),
  ./*output [31:0]  */iaddr_o(iaddr),
  ./*input  [31:0]  */irdata_i(irdata),
  ./*output         */ird_o(ird),
  ./*output [31:0]  */daddr_o(daddr),
  ./*output [31:0]  */dwdata_o(dwdata),
  ./*input  [31:0]  */drdata_i(drdata),
  ./*output  [1:0]  */dsize_o(dsize),
  ./*output         */drd_o(drd),
  ./*output         */dwr_o(dwr)
);
```

Test bench

- Test bench includes
  - A memory
  - A RISC-V core
- For test cases, only clock and reset are required

```
// CLOCK
initial begin
clk_i = 0;
forever #5 clk_i = ~clk_i;
end

// Testcase
initial
begin
                reset_i = 1'b0;

                #20 reset_i = 1'b1;
                // Reset

end
```

```
riscv_memory #(.FIRMWARE("mem.hex"))
u_riscv_memory
(
   .*input          */clk_i(clk_i),
   .*input          */reset_i(reset_i),
   .*input  [31:0] */iaddr_i(iaddr),
   .*output [31:0] */irdata_o(irdata),
   .*input          */ird_i(ird),
   .*input  [31:0] */daddr_i(daddr),
   .*input  [31:0] */dwdata_i(dwdata),
   .*output [31:0] */drdata_o(drdata),
   .*input   [1:0] */dsize_i(dsize),
   .*input          */drd_i(drd),
   .*input          */dwr_i(dwr)
);

riscv_core_sim
u_riscv_core_sim
(
   .*input          */clk_i(clk_i),
   .*input          */reset_i(reset_i),
   .*output         */lock_o(lock),
   .*output [31:0] */iaddr_o(iaddr),
   .*input  [31:0] */irdata_i(irdata),
   .*output         */ird_o(ird),
   .*output [31:0] */daddr_o(daddr),
   .*output [31:0] */dwdata_o(dwdata),
   .*input  [31:0] */drdata_i(drdata),
   .*output  [1:0] */dsize_o(dsize),
   .*output         */drd_o(drd),
   .*output         */dwr_o(dwr)
);
```

To do …

- Reuse the register file and ALU from Lecture 2, decoder and memory from Lecture 3
- Reuse your Program Counter (riscv_pc.v) in Lecture 4

```
always @(posedge clk_i or negedge reset_i) begin
  if (~reset_i) begin
    //Your code
  end
  else begin
    if (ird) begin
        // Your code
        //{{{

        //}}}
    end
  end
end
```

- Uncomment the following codes in riscv_core_sim:
  - Connect Register File and Decoder to ALU

```
/* TODO: ALU */
always@(*) begin
    alu_op = `ALU_ADD;
    alu_a = 32'h0;
    alu_b = 32'h0;

    // Insert your code
    //{{
    //alu_op = id_alu_op_w;
    //alu_a  = ra_value_r;
    //alu_b  = rb_value_r;
    //}}
end
```

```
// Dummy register file ports
always@(*) begin
    rd_index_w = 5'h0;
    rd_value_w = 32'h0;
    rd_we_w    = 1'b0;
    // Insert your code
    //{{{
    //rd_index_w = id_rd_index_w;
    //rd_value_w = alu_p;
    //rd_we_w    = 1'b1;
    //}}}
end
```

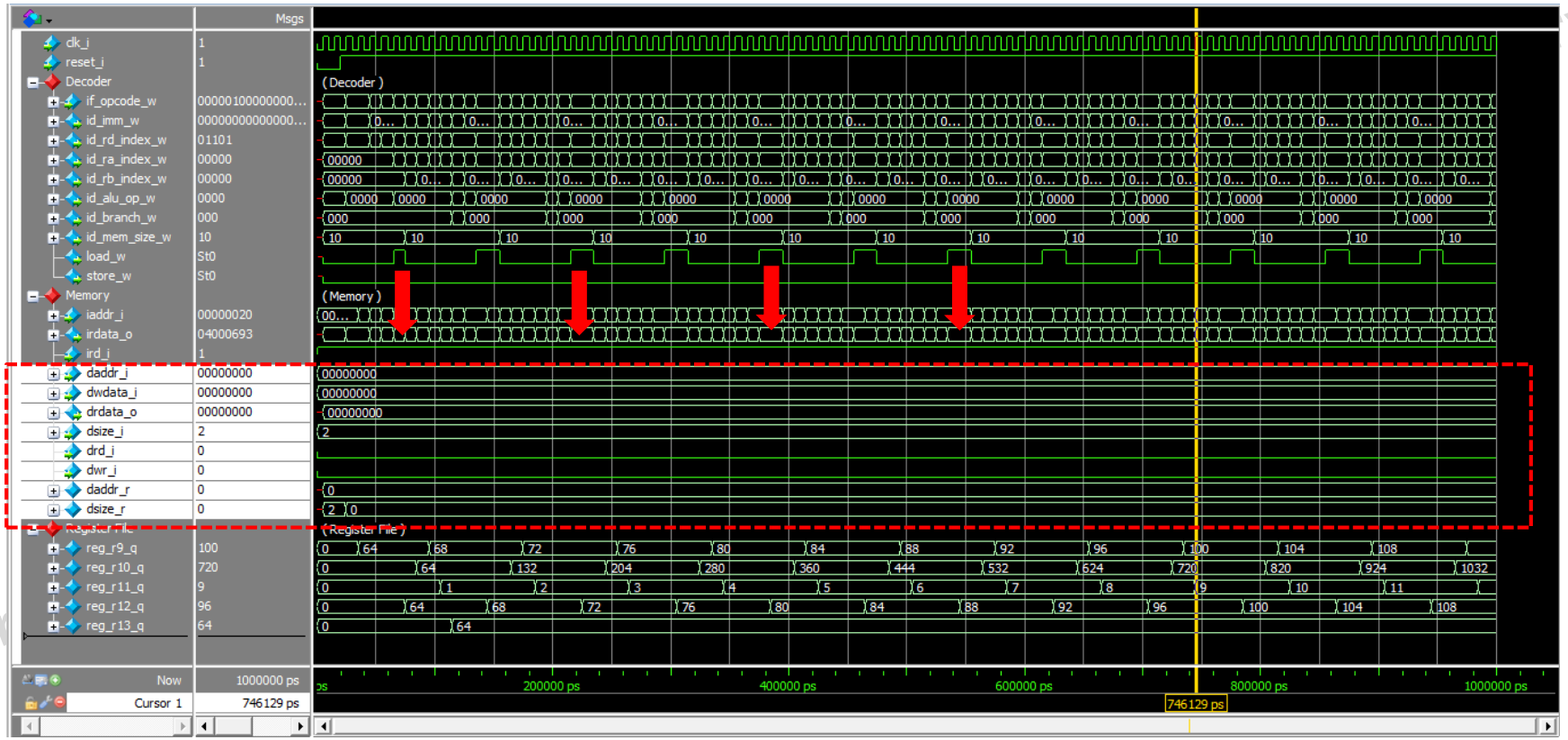
To do ...

- Calculate a target address and enable signals for branch instructions

```
/* TODO: Branch, Jump and Link instructions */
always @ (*) begin
    branch_taken_w = 1'b0;
    jump_addr_w = 32'h0;
    // Insert your code
    //{{{
    case(id_branch_w)
        `BR_JUMP: begin
        end
        `BR_EQ: begin

        end
        `BR_NE: begin

        end
        `BR_LT: begin
            // Dummy Branch
            branch_taken_w = 1'b1;
        end
        `BR_GE: begin

        end
        `BR_LTU: begin

        end
        `BR_GEU: begin

        end
    endcase
    //}}}
end
```

Waveform: Baseline

- Baseline code:
  - Have load instructions, but there is no access to data memory.



To do …

- Insert your code
  - To access data memory

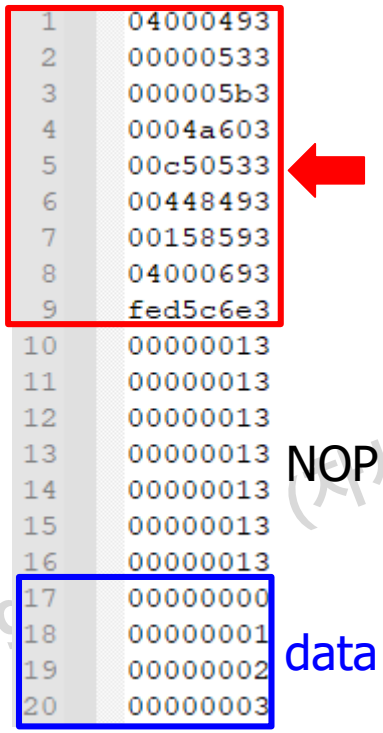
```
// Dummy part
assign  daddr_o = 32'h0;
assign  dwdata_o = 32'h0;
assign  dsize_o = `SIZE_WORD;
assign  drd_o = 1'b0;
assign  dwr_o = 1'b0;

// Your code
//assign daddr_o  = /*Insert your code*/;
//assign dwdata_o = /*Insert your code*/;
//assign dsize_o  = /*Insert your code*/;
//assign drd_o    = /*Insert your code*/&& (mem_stall_r == 1'b0);
//assign dwr_o    = /*Insert your code*/&& (mem_stall_r == 1'b0);
```

Explanation

- Signals
  - daddr: READ/WRITE address
    - Used for both LOAD/STORE instructions
  - dwdata: WRITE data for a store instruction
  - drdata: READ data for a load instruction
  - dsize: load/store types, i.e. byte, half, and word,
  - drd: Enable signal for a load
  - dwr: Enable signal for a store

Memory mapping

- Memory: instruction, data and unused segments.
  - Instruction's base address: 0000
  - Data's base address: 0040 (64)

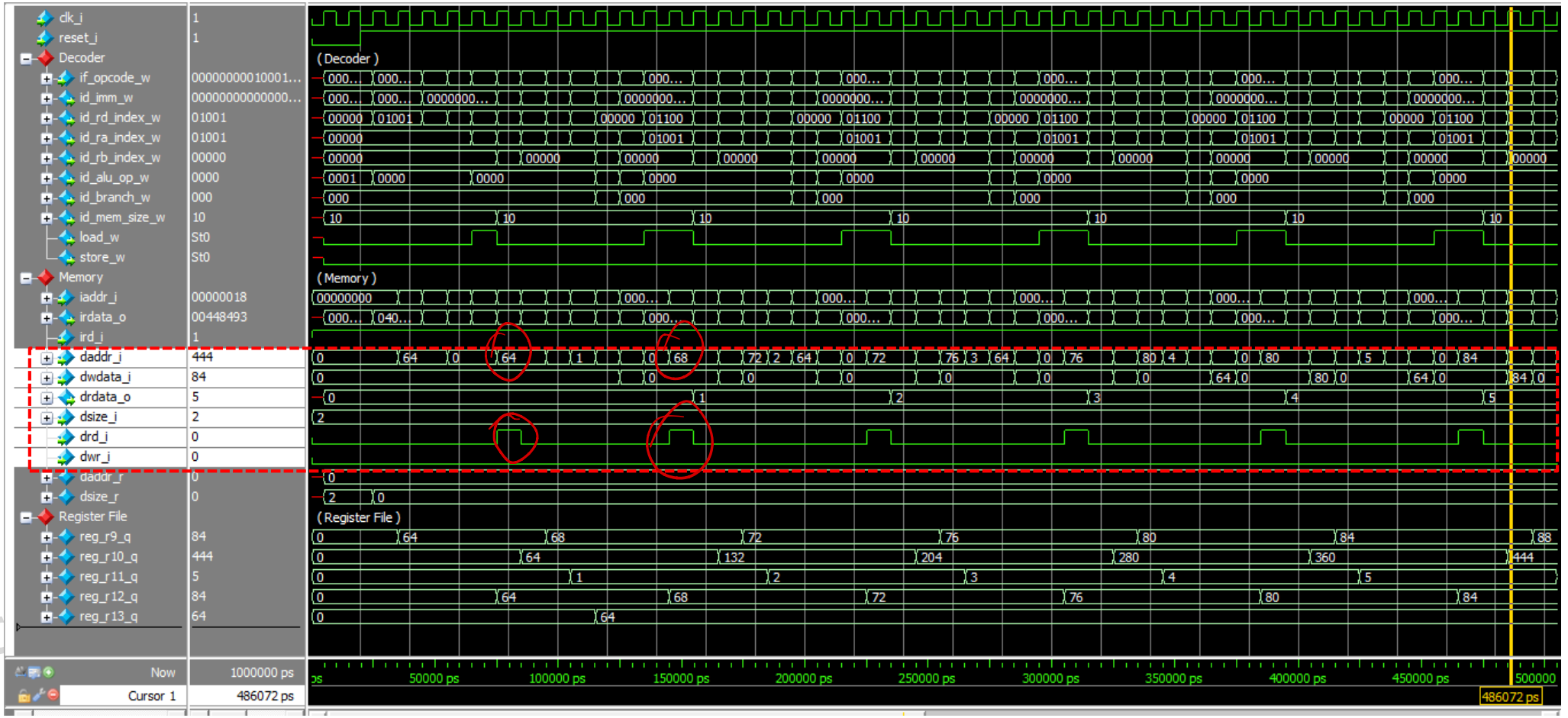


instruction

Address	Instruction	Basic code	Original code	Comments
0000	00004093	addi x9 x0 64	addi x9, x0, 64	# x9=&A[0]
0004	00000533	add x10 x0 x0	add x10, x0, x0	# sum=0
0008	000005b3	add x11 x0 x0	add x11, x0, x0	# i=0
000C	0004a603	lw x12 0(x9)	lw x12, 0(x9)	# x12=A[i]
0010	00c50533	add x10 x10 x12	add x10,x10,x12	# sum+=
0014	00448493	addi x9 x9 4	addi x9,x9,4	# &A[i++]
0018	00158593	addi x11 x11 1	addi x11,x11,1	# i++
001C	04000693	addi x13 x0 64	addi x13,x0,64	# x13=64
0020	fed5c6e3	blt x11 x13 -20	blt x11,x13,Loop	# Branch

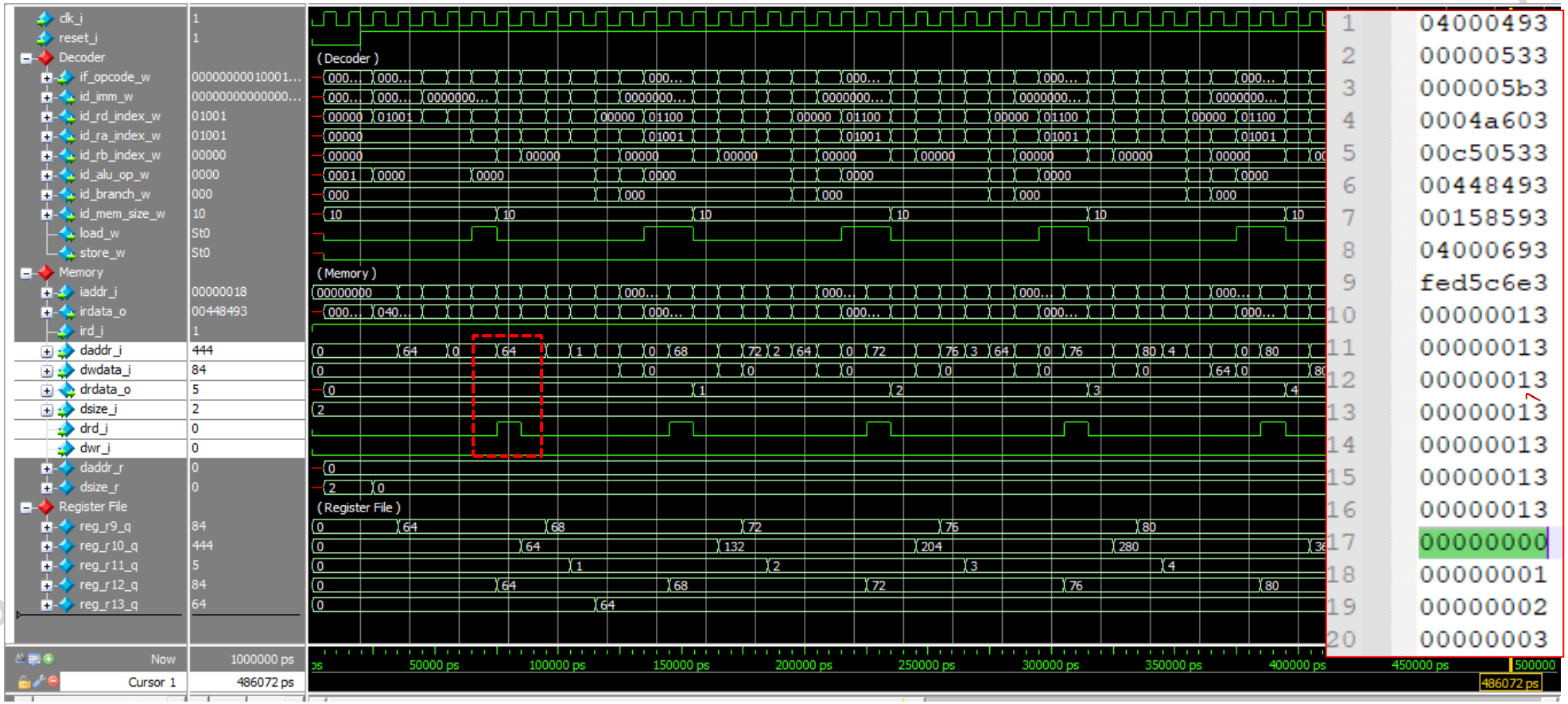
Waveform

- RISC-V Core accesses the data memory



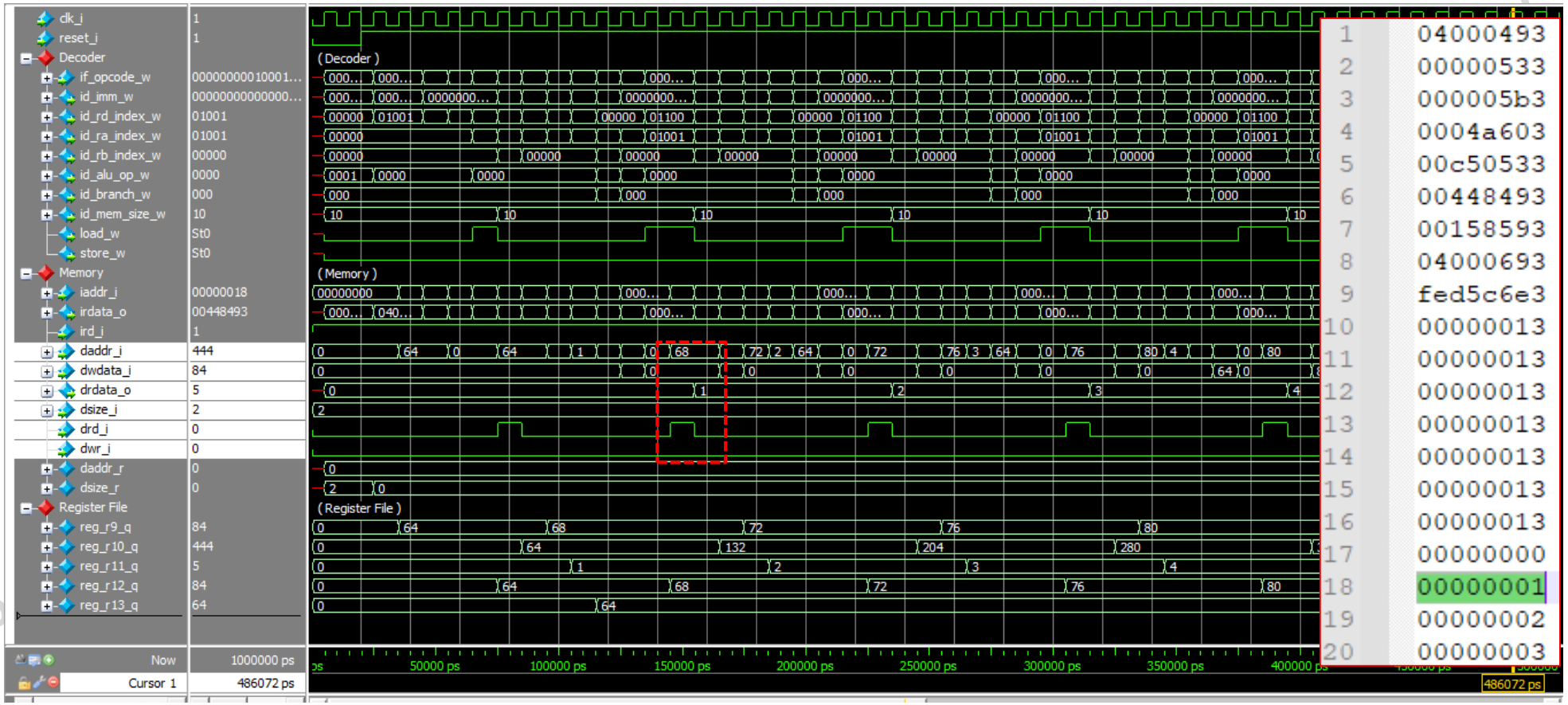
Waveform

- RISC-V Core accesses the data memory: daddr = 64
  - drdata = 0



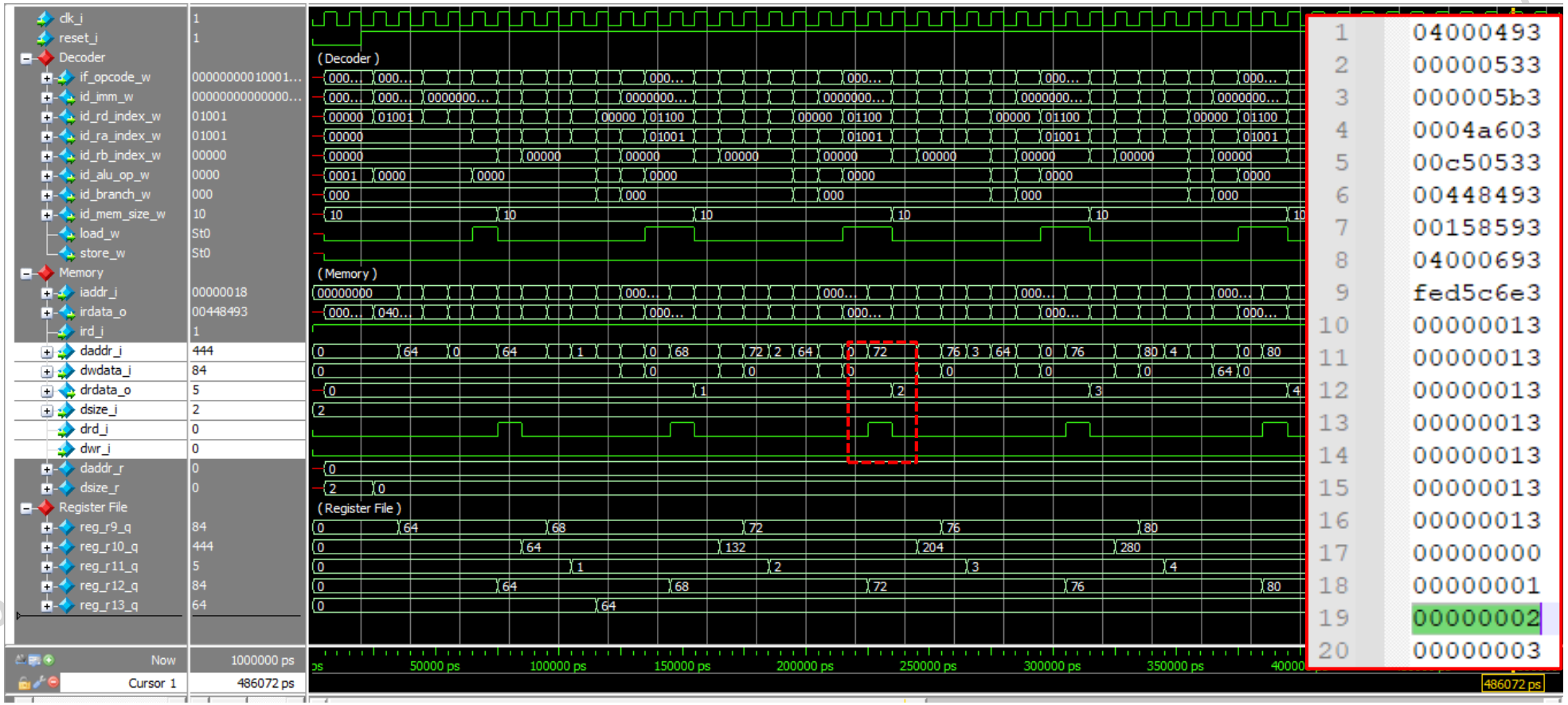
Waveform

- RISC-V Core accesses the data memory: daddr = 68
  - drdata = 1



Waveform

- RISC-V Core accesses the data memory: daddr = 72
  - drdata = 2



To do …

- Complete the missing codes
  - Reuse riscv_pc.v
  - Uncomment the codes in risv_core_sim.v
- Do a simulation
- Capture the waveform

Road map

Review

Load, store instructions

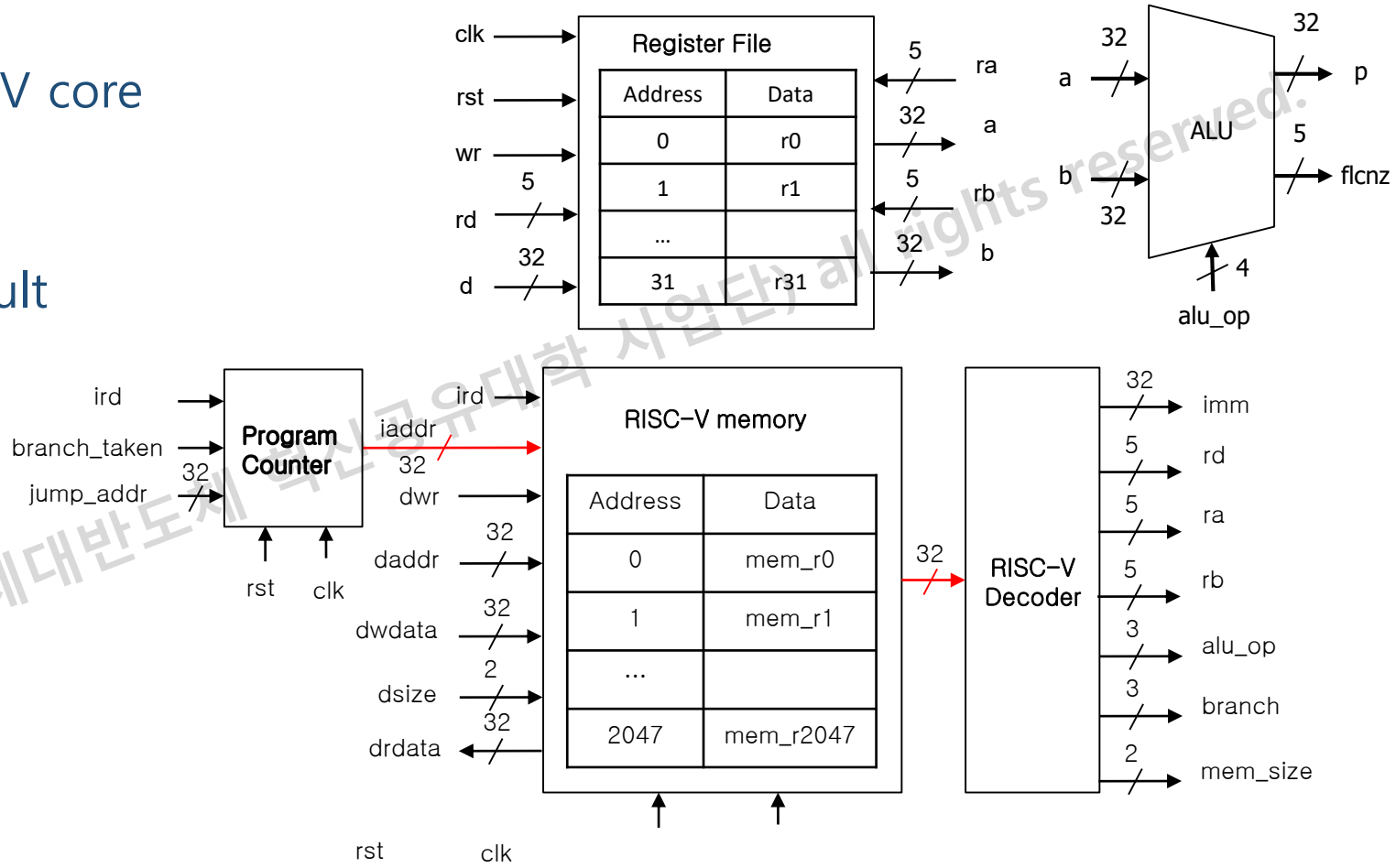
RISC-V core integration

RISC-V GNU Compiler

Optimization

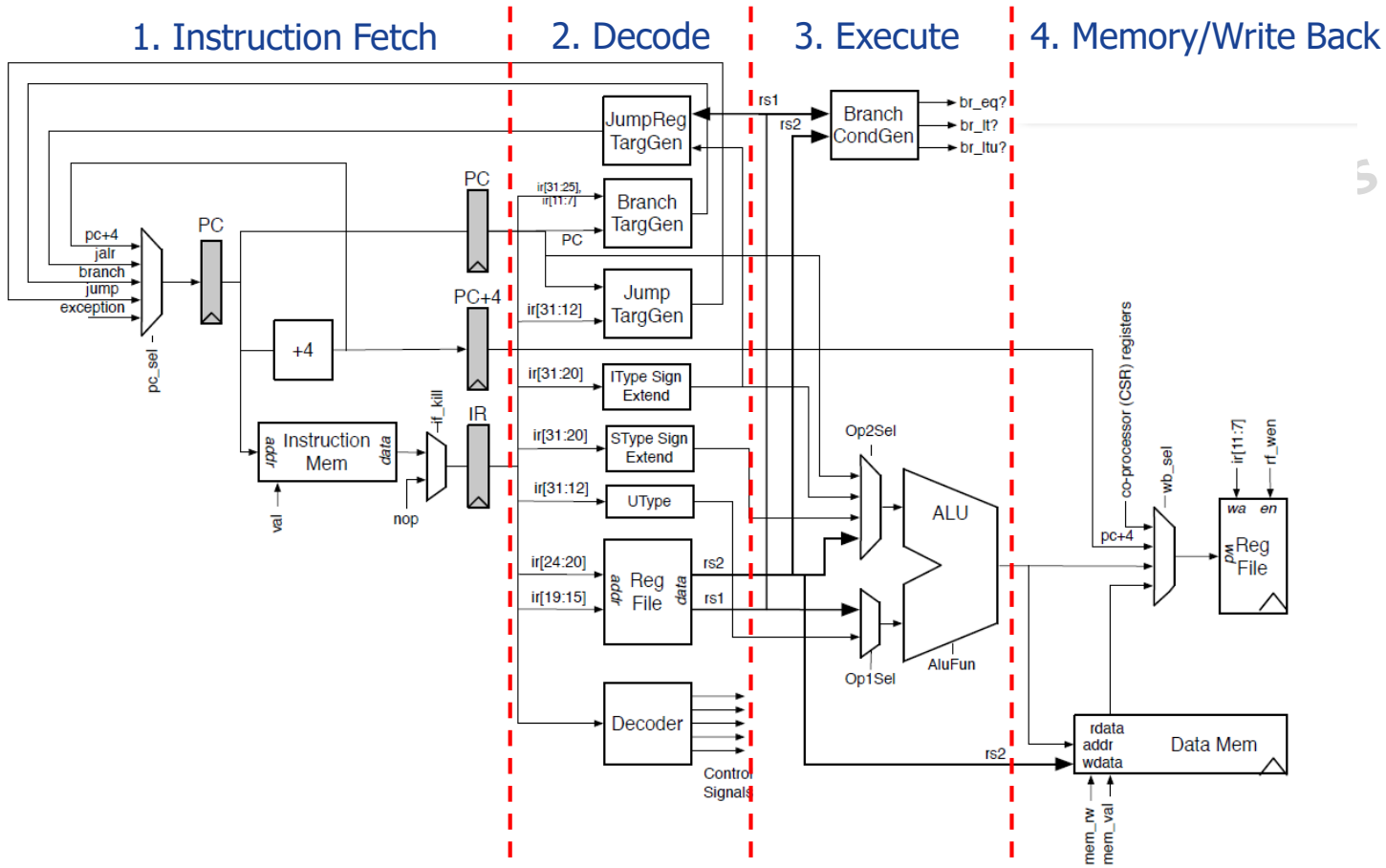
Lab 2: RISC-V core integration

- Lab 2:
  - Implement the RISC-V core
  - Do a simulation
  - Show the output result



RISC-C core and Memory

- Multi-stage CPU



RISC-V Core

- Almost complete code of a RISC-V core

```
module riscv_core #(
  parameter     PC_SIZE  = 32,
  parameter     RESET_SP = 32'h1000
)(
  input         clk_i,      // Clock
  input         reset_i,    // Reset
  output        lock_o,     // Lock

  output [31:0] iaddr_o,    // Instruction from Memory
  input  [31:0] irdata_i,   // Instruction address
  output        ird_o,      // Read request

  output  [31:0] daddr_o,   // Read/Write address
  output  [31:0] dwdata_o,  // Write Data
  input   [31:0] drdata_i,  // Read data
  output   [1:0] dsize_o,
  output         drd_o,     // Write request
  output         dwr_o      // Read/Write Enable
);
```

RISC-V Core

- A state machine to handle byte, half or word modes
- Support 16-bit and 32-bit instructions

```
//-----------------------------------------------------------------
// State machine
//-----------------------------------------------------------------
always @(posedge clk_i or negedge reset_i) begin
  if (~reset_i) begin
    if_state_r <= ST_RESET;
    if_buf_r <= 16'h0;
  end
  else begin
    if (branch_taken_w) begin
        if_state_r <= jump_addr_w[1] ? ST_HIGH : ST_LOW;
    end
    else if (id_exec_w) begin
        case (if_state_r)
    end
  end
end
//-----------------------------------------------------------------
// Instruction Fetch
//-----------------------------------------------------------------
assign ird_request_w =
    (ST_RESET == if_state_r) ||
    (ST_HIGH == if_state_r) ||
    (ST_LOW == if_state_r && if_lo_is_rv_w) ||
    (ST_LOW == if_state_r && if_hi_is_rv_w && if_lo_is_rvc_w) ||
    (ST_UNALIGNED == if_state_r && if_hi_is_rv_w);

assign iaddr_o = if_next_addr_w;
assign ird_o   = branch_taken_w || (ird_request_w && id_exec_w);
assign lock_o  = id_lock_r;

assign if_lo_is_rv_w  = (2'b11 == irdata_i[1:0]);
assign if_lo_is_rvc_w = !if_lo_is_rv_w;
assign if_hi_is_rv_w  = (2'b11 == irdata_i[17:16]);
assign if_hi_is_rvc_w = !if_hi_is_rv_w;

assign if_rv_w    = (ST_UNALIGNED == if_state_r) || (ST_LOW == if_state_r && if_lo_is_rv_w);
assign if_valid_w = !((ST_RESET == if_state_r) || (ST_HIGH == if_state_r && if_hi_is_rv_w));
assign if_rv_op_w  = (ST_UNALIGNED == if_state_r) ? { irdata_i[15:0], if_buf_r } : irdata_i;
assign if_rvc_op_w = (ST_HIGH == if_state_r) ? irdata_i[31:16] : irdata_i[15:0];


always @(posedge clk_i or negedge reset_i) begin

assign if_next_pc_w = if_pc_r + (if_rv_w ? 'd4 : 'd2);
assign if_rvc_dec_w = 32'h0;
assign if_opcode_w = if_rv_w ? if_rv_op_w : if_rvc_dec_w;
```

Instruction Fetch

```
//-----------------------------------------------------------------
// Instruction Fetch
//-----------------------------------------------------------------
assign ird_request_w =
    (ST_RESET == if_state_r) ||
    (ST_HIGH == if_state_r) ||
    (ST_LOW == if_state_r && if_lo_is_rv_w) ||
    (ST_LOW == if_state_r && if_hi_is_rv_w && if_lo_is_rvc_w) ||
    (ST_UNALIGNED == if_state_r && if_hi_is_rv_w);

assign iaddr_o = if_next_addr_w;
assign ird_o   = branch_taken_w || (ird_request_w && id_exec_w);
assign lock_o  = id_lock_r;

assign if_lo_is_rv_w  = (2'b11 == irdata_i[1:0]);
assign if_lo_is_rvc_w = !if_lo_is_rv_w;
assign if_hi_is_rv_w  = (2'b11 == irdata_i[17:16]);
assign if_hi_is_rvc_w = !if_hi_is_rv_w;

assign if_rv_w    = (ST_UNALIGNED == if_state_r) || (ST_LOW == if_state_r && if_lo_is_rv_w);
assign if_valid_w = !((ST_RESET == if_state_r) || (ST_HIGH == if_state_r && if_hi_is_rv_w));
assign if_rv_op_w  = (ST_UNALIGNED == if_state_r) ? { irdata_i[15:0], if_buf_r } : irdata_i;
assign if_rvc_op_w = (ST_HIGH == if_state_r) ? irdata_i[31:16] : irdata_i[15:0];


always @(posedge clk_i or negedge reset_i) begin

assign if_next_pc_w = if_pc_r + (if_rv_w ? 'd4 : 'd2);
assign if_rvc_dec_w = 32'h0;
assign if_opcode_w = if_rv_w ? if_rv_op_w : if_rvc_dec_w;
```

Request signals to Instruction MEM

16-bit and 32-bit instructions

Decode

- Extracts all information, including:
  - Addresses of source and destination. Registers for Register File (ra, rb, rd).
  - ALU opcode (alu_op)
  - Immediate (imm)
  - Branch, jump & link types
- Pipeline registers

```
end else if (id_ready_w) begin
   id_pc_r          <= if_pc_r;
   id_rd_index_r    <= id_rd_index_w;
   id_imm_r         <= id_imm_w;
   id_a_signed_r    <= mulh_w || mulhsu_w || div_w || rem_w || sra_w || srai_w;
   id_b_signed_r    <= mulh_w || div_w || rem_w;
   id_op_imm_r      <= alu_imm_w || jal_w || load_w || store_w;
   id_alu_op_r      <= id_alu_op_w;
   id_mem_rd_r      <= load_w;
   id_mem_wr_r      <= store_w;
   id_mem_signed_r  <= !lbu_w && !lhu_w;
   id_mem_size_r    <= id_mem_size_w;
   id_branch_r      <= id_branch_w;
   id_reg_jump_r    <= jalr_w;
   id_ra_value_r    <= ra_value_r;
   id_rb_value_r    <= rb_value_r;
   id_lock_r        <= id_illegal_w;
end
```

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

Execution

- ALU and flcnz
  - flcnz is computed in riscv_core.v
  - Inputs are alu_a, alu_b and alu_op
  - Output is ex_alu_res_w

- Select input for ALU
  - Immediate-type: id_imm_r
  - Register-type: id_rb_value_r from RF

```
//-----------------------------------------------
// ALU
//-----------------------------------------------
riscv_alu
u_alu
(
    ./*input  [  3:0]  */alu_op_i(id_alu_op_r)
    ,./*input  [ 31:0]  */alu_a_i(adder_opa_w)
    ,./*input  [ 31:0]  */alu_b_i(adder_opb_w)
    ,./*output [ 31:0]  */alu_p_o(ex_alu_res_w)
    ,./*output reg [4:0]*/flcnz(flcnz)
);
```

```
//-----------------------------------------------------------------
// Execution
//-----------------------------------------------------------------
assign alu_opb_w     = id_op_imm_r ? id_imm_r : id_rb_value_r;
assign adder_sub_w   = (`ALU_SUB == id_alu_op_r || `ALU_SLT == id_alu_op_r || `ALU_SLTU == id_alu_op_r);
assign adder_opa_w   = id_ra_value_r;
assign adder_opb_w   = adder_sub_w ? ~alu_opb_w : alu_opb_w;
assign adder_cin_w   = adder_sub_w ? 1'b1 : 1'b0;
assign { adder_c_w, adder_out_w } = { 1'b0, adder_opa_w } + { 1'b0, adder_opb_w } + adder_cin_w;
assign adder_n_w = adder_out_w[31];
assign adder_v_w = (adder_opa_w[31] == adder_opb_w[31] && adder_out_w[31] != adder_opb_w[31]);
assign adder_z_w = (32'h0 == adder_out_w);
```

Execution

- Jump address
- Jump signal enable
  - Based on flcnz

```
always @ (*) begin
    branch_taken_w = 1'b0;
    jump_addr_w = 32'h0;

    jump_addr_w = (id_reg_jump_r ? id_ra_value_r : id_pc_r) + id_imm_r;
    case(id_branch_r)
        `BR_JUMP: begin
            branch_taken_w = 1'b1;
        end
        `BR_EQ: begin
            branch_taken_w = adder_z_w;
        end
        `BR_NE: begin
            /*Insert your code*/
        end
        `BR_LT: begin
            branch_taken_w = (adder_n_w != adder_v_w);
        end
        `BR_GE: begin
            /*Insert your code*/
        end
        `BR_LTU: begin
```

Instruction	Syntax	Description	Execution
BEQ	beq rs1, rs2, imm	Branch if = zero	PC <= (reg[rs1] == reg[rs2]) ? PC + imm : (PC+4)
BNE	bne rs1, rs2, imm	Branch if ≠ zero	PC <= (reg[rs1] != reg[rs2]) ? PC + imm : (PC+4)
BLT	blt rs1, rs2, imm	Branch if < (signed)	PC <= (reg[rs1] $<_s$ reg[rs2]) ? PC + imm : (PC+4)
BGE	bge rs1, rs2, imm	Branch if ≥ (signed)	PC <= (reg[rs1] $\geq_s$ reg[rs2]) ? PC + imm : (PC+4)
BLTU	bltu rs1, rs2, imm	Branch if < (Unsigned)	PC <= (reg[rs1] $<_u$ reg[rs2]) ? PC + imm : (PC+4)
BGEU	bgeu rs1, rs2, imm	Branch if ≥ (Unsigned)	PC <= (reg[rs1] $\geq_u$ reg[rs2]) ? PC + imm : (PC+4)

Memory

- Load instruction
  - Load address (daddr), size (dsize), and enable signal (rd) for a load
  - Incoming data (drdata) is kept in mem_rdata

```
//-----------------------------------------------------------------
// Memory
//-----------------------------------------------------------------
assign daddr_o  = ex_alu_res_r;
assign dwdata_o = ex_mem_data_r;
assign dsize_o  = ex_mem_size_r;
assign drd_o    = ex_mem_rd_r && (mem_stall_r == 1'b0);
assign dwr_o    = ex_mem_wr_r && (mem_stall_r == 1'b0);

assign mem_rdata_w =
  (`SIZE_BYTE == ex_mem_size_r) ? { {24{ex_mem_signed_r & drdata_i[7]}}, drdata_i[7:0] } :
  (`SIZE_HALF == ex_mem_size_r) ? { {16{ex_mem_signed_r & drdata_i[15]}}, drdata_i[15:0] } : drdata_i;


always @(posedge clk_i) begin
  if (~reset_i)
    mem_stall_r <= 1'b0;
  else
    mem_stall_r <= mem_stall_w;
end

assign mem_access_w = (ex_mem_rd_r || ex_mem_wr_r);
assign mem_stall_w  = mem_stall_r ? 1'b0 : mem_access_w;
```

Memory

- Store instruction
  - Store address (daddr), size (dsize), and enable signal (wr) for a store
  - Out-coming data (dwdata)

```
//-----------------------------------------------------------------
// Memory
//-----------------------------------------------------------------
assign daddr_o  = ex_alu_res_r;
assign dwdata_o = ex_mem_data_r;
assign dsize_o  = ex_mem_size_r;
assign drd_o    = ex_mem_rd_r && (mem_stall_r == 1'b0);
assign dwr_o    = ex_mem_wr_r && (mem_stall_r == 1'b0);

assign mem_rdata_w =
  (`SIZE_BYTE == ex_mem_size_r) ? { {24{ex_mem_signed_r & drdata_i[7]}}, drdata_i[7:0] } :
  (`SIZE_HALF == ex_mem_size_r) ? { {16{ex_mem_signed_r & drdata_i[15]}}, drdata_i[15:0] } : drdata_i;


always @(posedge clk_i) begin
  if (~reset_i)
    mem_stall_r <= 1'b0;
  else
    mem_stall_r <= mem_stall_w;
end

assign mem_access_w = (ex_mem_rd_r || ex_mem_wr_r);
assign mem_stall_w  = mem_stall_r ? 1'b0 : mem_access_w;
```

Write back

- Given addresses of source and dest. registers (ra, rb, rd), Register File can
  - Output a value of a register.
  - Store or update a new value of a register

Load instruction

ALU result

```
//-----------------------------------------------------------------
// Write back
//-----------------------------------------------------------------
assign  rd_index_w = ex_rd_index_r;
assign  rd_value_w = mem_access_w ? mem_rdata_w : ex_alu_res_r;
assign  rd_we_w    = (ex_rd_index_r != 5'd0) && (mem_stall_w == 1'b0);

riscv_regfile
u_regfile
(
     ./*input           */clk_i(clk_i)
    ,./*input           */rst_i(reset_i)
    ,./*input   [  4:0] */rd0_i(rd_index_w)
    ,./*input   [ 31:0] */rd0_value_i(rd_value_w)
    ,./*input   [  4:0] */ra0_i(id_ra_index_w)
    ,./*input   [  4:0] */rb0_i(id_rb_index_w)
    ,./*input           */wr(rd_we_w)
    ,./*output [ 31:0]  */ra0_value_o(ra_value_r)
    ,./*output [ 31:0]  */rb0_value_o(rb_value_r)
);
```

Test bench

- Test bench includes
  - A memory
  - A RISC-V core

```
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;
        ...
```

```
riscv_memory #(.FIRMWARE("mem.hex"))
u_riscv_memory
(
  ./*input          */clk_i(clk_i),
  ./*input          */reset_i(reset_i),
  ./*input  [31:0]  */iaddr_i(iaddr),
  ./*output [31:0]  */irdata_o(irdata),
  ./*input          */ird_i(ird),
  ./*input  [31:0]  */daddr_i(daddr),
  ./*input  [31:0]  */dwdata_i(dwdata),
  ./*output [31:0]  */drdata_o(drdata),
  ./*input   [1:0]  */dsize_i(dsize),
  ./*input          */drd_i(drd),
  ./*input          */dwr_i(dwr)
);

riscv_core_sim
u_riscv_core_sim
(
  ./*input          */clk_i(clk_i),
  ./*input          */reset_i(reset_i),
  ./*output         */lock_o(lock),
  ./*output [31:0]  */iaddr_o(iaddr),
  ./*input  [31:0]  */irdata_i(irdata),
  ./*output         */ird_o(ird),
  ./*output [31:0]  */daddr_o(daddr),
  ./*output [31:0]  */dwdata_o(dwdata),
  ./*input  [31:0]  */drdata_i(drdata),
  ./*output  [1:0]  */dsize_o(dsize),
  ./*output         */drd_o(drd),
  ./*output         */dwr_o(dwr)
);
```

Test bench

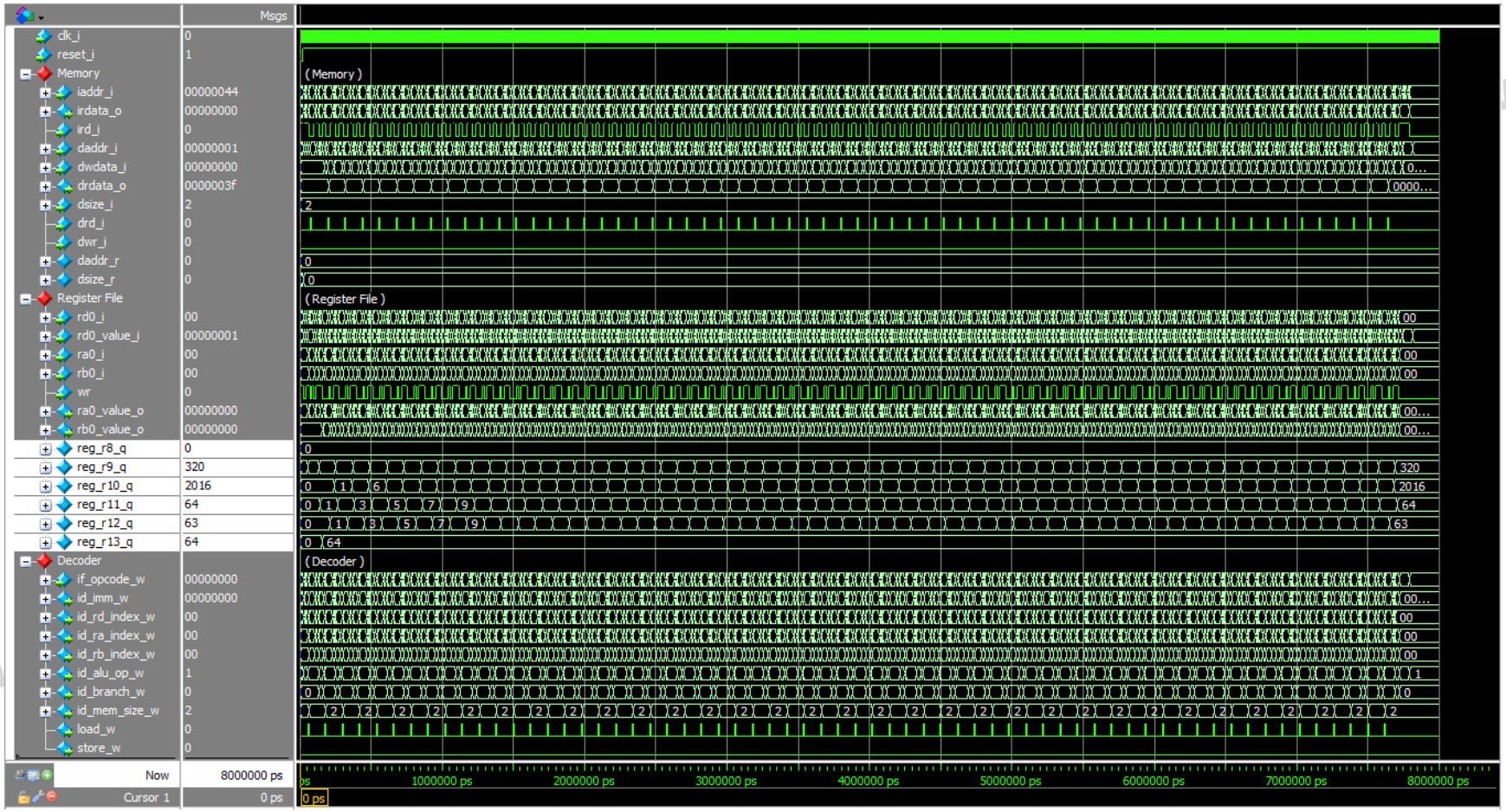
- Test bench includes
  - A memory
  - A RISC-V core
- For test cases, only clock and reset are required

```
// CLOCK
initial begin
clk_i = 0;
forever #5 clk_i = ~clk_i;
end

// Testcase
initial
begin
                reset_i = 1'b0;

                #20 reset_i = 1'b1;
                // Reset

end
```

```
riscv_memory #(.FIRMWARE("mem.hex"))
u_riscv_memory
(
   .clk_i(clk_i),
   .reset_i(reset_i),
   .iaddr_i(iaddr),
   .irdata_o(irdata),
   .ird_i(ird),
   .daddr_i(daddr),
   .dwdata_i(dwdata),
   .drdata_o(drdata),
   .dsize_i(dsize),
   .drd_i(drd),
   .dwr_i(dwr)
);

riscv_core_sim
u_riscv_core_sim
(
   .clk_i(clk_i),
   .reset_i(reset_i),
   .lock_o(lock),
   .iaddr_o(iaddr),
   .irdata_i(irdata),
   .ird_o(ird),
   .daddr_o(daddr),
   .dwdata_o(dwdata),
   .drdata_i(drdata),
   .dsize_o(dsize),
   .drd_o(drd),
   .dwr_o(dwr)
);
```

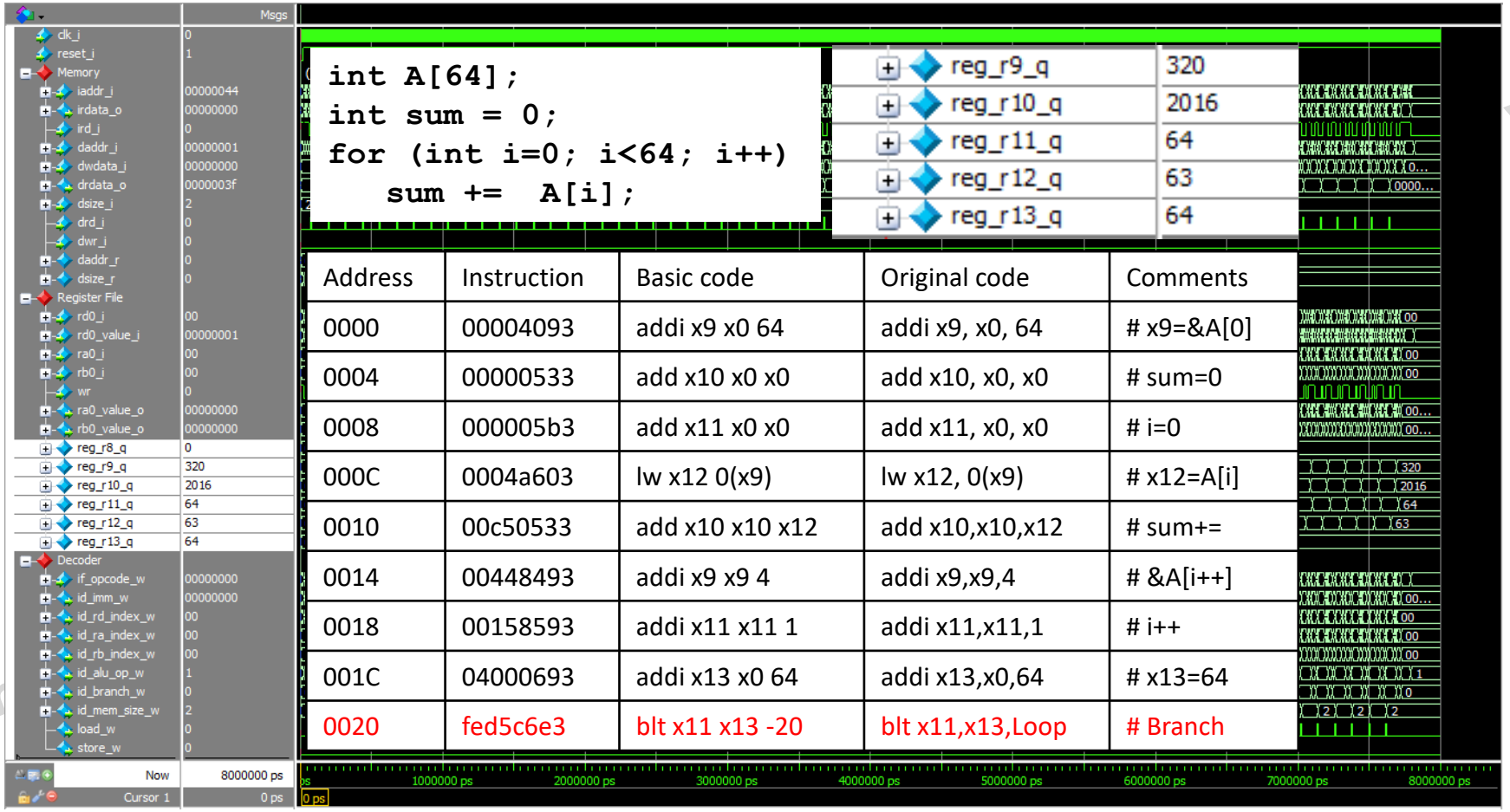
Waveform

- Do a simulation with simulation time = 8,000 ns



Waveform

- Explain the final values of r9, r10, r11, r13.



Address	Instruction	Basic code	Original code	Comments
0000	00004093	addi x9 x0 64	addi x9, x0, 64	# x9=&A[0]
0004	00000533	add x10 x0 x0	add x10, x0, x0	# sum=0
0008	000005b3	add x11 x0 x0	add x11, x0, x0	# i=0
000C	0004a603	lw x12 0(x9)	lw x12, 0(x9)	# x12=A[i]
0010	00c50533	add x10 x10 x12	add x10,x10,x12	# sum+=
0014	00448493	addi x9 x9 4	addi x9,x9,4	# &A[i++]
0018	00158593	addi x11 x11 1	addi x11,x11,1	# i++
001C	04000693	addi x13 x0 64	addi x13,x0,64	# x13=64
0020	fed5c6e3	blt x11 x13 -20	blt x11,x13,Loop	# Branch

To do …

- Change the memory file (mem_bne.hex)
- Modify the test bench to execute the memory file

C code	Assembly Code	
int A[64];	addi x9, x0, 64	# x9=&A[0]
int sum = 0;	add x10, x0, x0	# sum=0
for (int i=0; i<64; i++)	add x11, x0, x0	# i=0
sum += A[i];	Loop:	
	lw x12, 0(x9)	# x12=A[i]
	add x10, x10, x12	# sum+= A[i]
	addi x9, x9, 4	# &A[i++]
	addi x11, x11, 1	# i++
	addi x13, x0, 64	# x13=64
	blt x11, x13, Loop	
int A[64];	addi x9, x0, 64	# x9=&A[0]
int sum = 0;	add x10, x0, x0	# sum=0
for (int i=0; i≠64; i++)	add x11, x0, x0	# i=0
sum += A[i];	Loop:	
	lw x12, 0(x9)	# x12=A[i]
	add x10, x10, x12	# sum+= A[i]
	addi x9, x9, 4	# &A[i++]
	addi x11, x11, 1	# i++
	addi x13, x0, 64	# x13=64
	bne x11, x13, Loop	

To do …

- Complete the missing codes.
- Do a simulation
- Capture the output result.

```
always @ (*) begin
    branch_taken_w = 1'b0;
    jump_addr_w = 32'h0;

    jump_addr_w = (id_reg_jump_r ? id_ra_value_r : id_pc_r) + id_imm_r;
    case(id_branch_r)
        `BR_JUMP: begin
            branch_taken_w = 1'b1;
        end
        `BR_EQ: begin
            branch_taken_w = adder_z_w;
        end
        `BR_NE: begin
            /*Insert your code*/
        end
        `BR_LT: begin
            branch_taken_w = (adder_n_w != adder_v_w);
        end
        `BR_GE: begin
            /*Insert your code*/
        end
        `BR_LTU: begin
            branch_taken_w = !adder_c_w;
        end
        `BR_GEU: begin
            branch_taken_w = adder_c_w;
        end
    endcase
end
```

To do (cont') ...

- Generate a memory file for the modified assembly code using a NOT-EQUAL operation, called mem_bne.hex.
- Modify a test bench to use a new memory file (riscv_core_bne_tb.v).
- Make code to calculate a branch enable signal for Branch-Not-Equal (BNE).
- Do a simulation
- Capture the output waveform.

Road map

Review

Load, store instructions

RISC-V core integration

RISC-V GNU Compiler

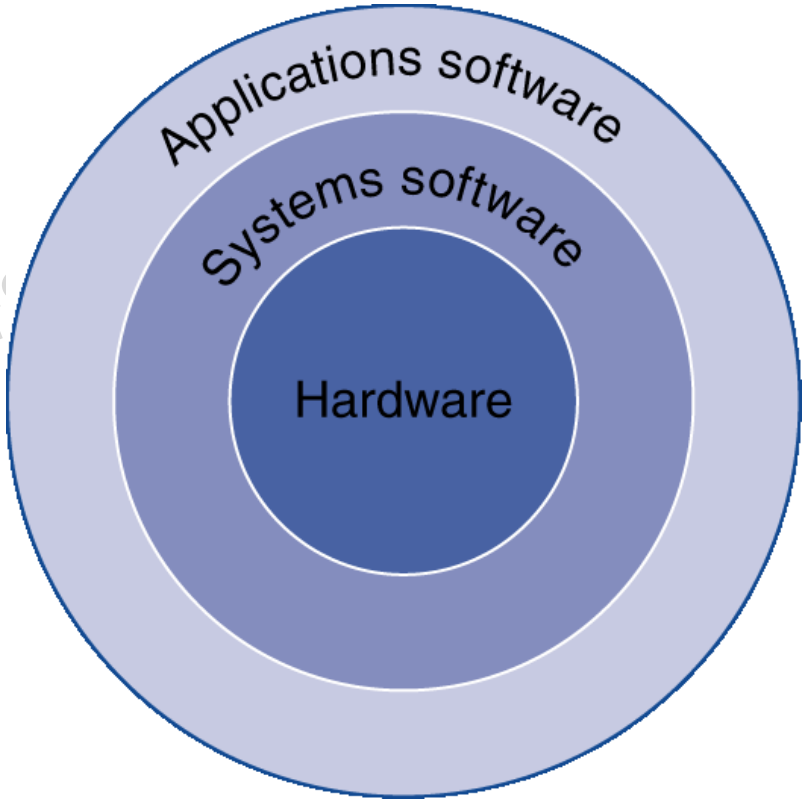
Optimization

Below Your Program

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

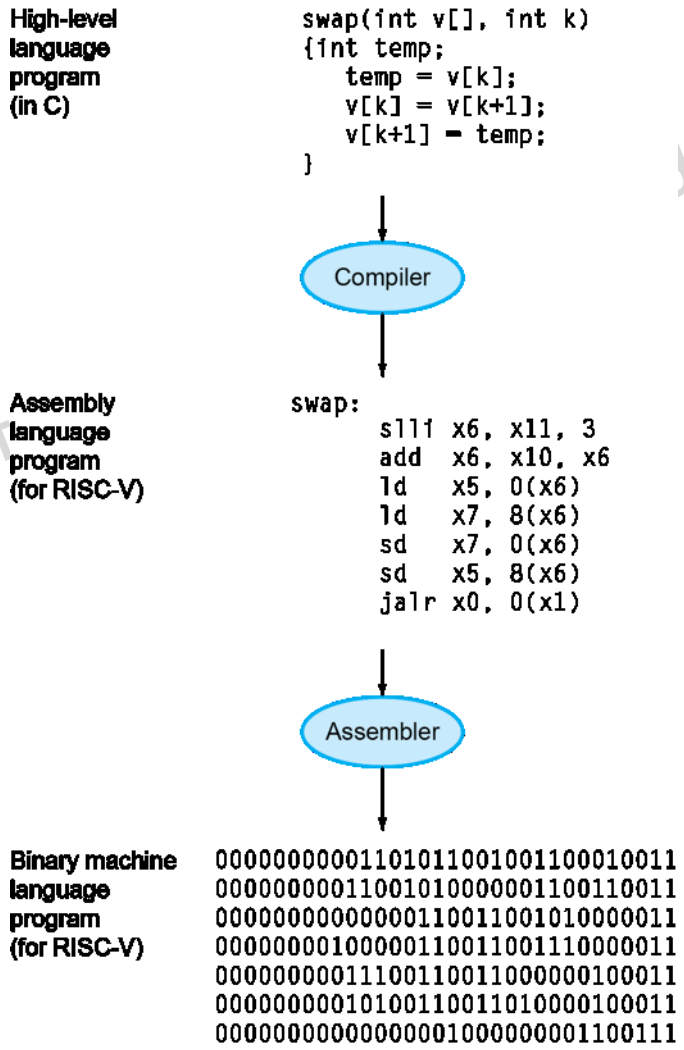
- System software
  - Compiler: translates HLL code to machine code
  - Operating System: service code
    - Handling input/output
    - Managing memory and storage
    - Scheduling tasks & sharing resources

- Hardware
  - Processor, memory, I/O controllers



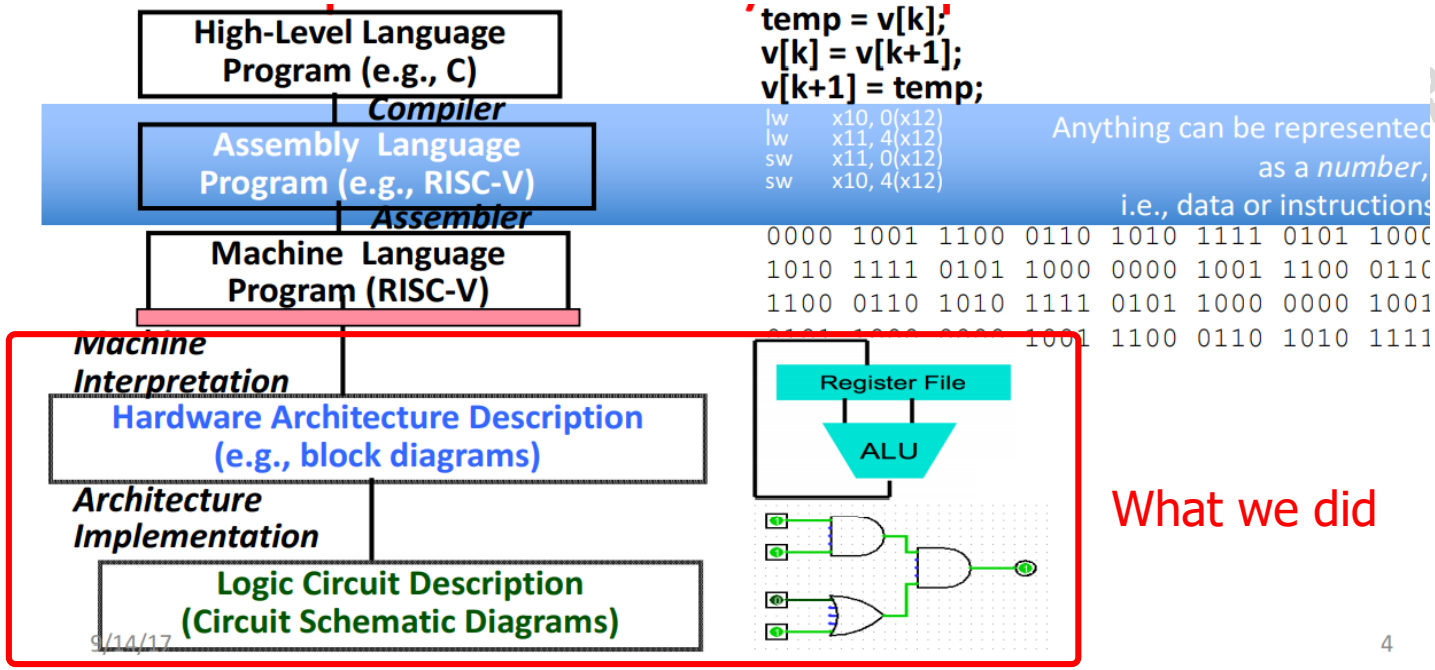
Levels of Program Code

- 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



General purpose processor

- Level of representation
  - Compiler: HLL → Assembly language program
  - Assembler: Assembly code → Machine code
  - Machine code is executed by hardware modules



What we did

```
riscv_memory #(.FIRMWARE("mem.hex"))
u_riscv_memory
(
  .clk_i(clk_i),
  .reset_i(reset_i),
  .iaddr_i(iaddr),
  .irdata_o(irdata),
  .ird_i(ird),
  .daddr_i(daddr),
  .dwdata_i(dwdata),
  .drdata_o(drdata),
  .dsize_i(dsize),
  .drd_i(drd),
  .dwr_i(dwr)
);

riscv_core_sim
u_riscv_core_sim
(
  .clk_i(clk_i),
  .reset_i(reset_i),
  .lock_o(lock),
  .iaddr_o(iaddr),
  .irdata_i(irdata),
  .ird_o(ird),
  .daddr_o(daddr),
  .dwdata_o(dwdata),
  .drdata_i(drdata),
  .dsize_o(dsize),
  .drd_o(drd),
  .dwr_o(dwr)
);
```

RISC-V code generation

- Venus: A RISC-V Simulator
- **https://www.kvakil.me/venus/**

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

C code

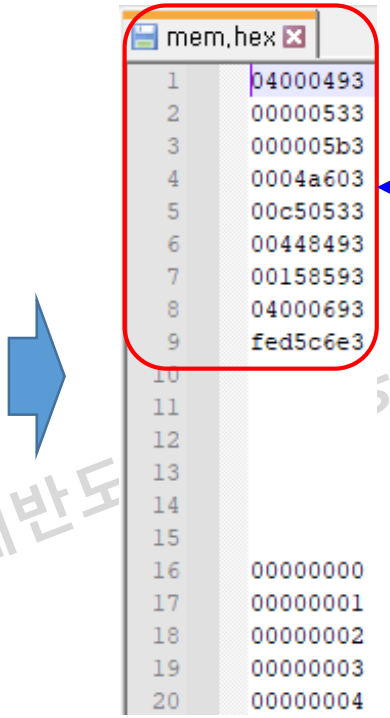
```
addi x9, x0, 64 # x9=&A[0]
add x10, x0, x0 # sum=0
add x11, x0, x0 # i=0
Loop:
  lw x12, 0(x9) # x12=A[i]
  add x10,x10,x12 # sum+=
  addi x9,x9,4  # &A[i++]
  addi x11,x11,1 # i++
  addi x13,x0,64 # x13=64
  blt x11,x13,Loop
```

Assembly code

RISC-V code generation

- Manually generate the RISC-V codes from Assembly codes

```
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
```



Machine Code	Basic Code	Original Code
0x04000493	addi x9 x0 64	addi x9, x0, 64
0x00000533	add x10 x0 x0	add x10, x0, x0 # sum=0
0x000005b3	add x11 x0 x0	add x11, x0, x0 # i=0
0x0004a603	lw x12 0(x9)	lw x12, 0(x9) # x12=A[i]
0x00c50533	add x10 x10 x12	add x10,x10,x12 # sum+=
0x00448493	addi x9 x9 4	addi x9,x9,4 # &A[i++]
0x00158593	addi x11 x11 1	addi x11,x11,1 # i++
0x04000693	addi x13 x0 64	addi x13,x0,64 # x13=64
0xfed5c6e3	blt x11 x13 -20	blt x11,x13,Loop

Compiler

- RISCV-GNU-Toolchain
  - An automatic tool to generate a machine code from a C program

Submodule	Contents
riscv-fesvr	RISC-V Frontend Server
riscv-isa-sim	Functional ISA simulator (“Spike”)
riscv-qemu	Higher-performance ISA simulator
riscv-gnu-toolchain	binutils, gcc, newlib, glibc, Linux UAPI headers
riscv-llvm	LLVM, riscv-clang submodule
riscv-pk	RISC-V Proxy Kernel
(riscv-linux)	Linux/RISC-V kernel port
riscv-tests	ISA assembly tests, benchmark suite

All listed submodules are hosted under the riscv GitHub organization: https://github.com/riscv

riscv-tools —Installation

- Build riscv-gnu-toolchain (riscv*-*-elf / newlib target), riscv-fesvr, riscv-isa-sim, and riscv-pk: (pre-installed in VM)

```
$ git clone https://github.com/riscv/riscv-tools
$ cd riscv-tools
$ git submodule update --init --recursive
$ export RISCV=<installation path>
$ export PATH=${PATH}:${RISCV}/bin
$ ./build.sh
```

riscv-tools utilities

```
$ ls ${RISCV}/bin
elf2hex
fesvr-eth
fesvr-rs232
fesvr-zedboard
riscv64-unknown-elf-addr2line
riscv64-unknown-elf-ar
riscv64-unknown-elf-as
riscv64-unknown-elf-c++
riscv64-unknown-elf-c++filt
riscv64-unknown-elf-cpp
riscv64-unknown-elf-elfedit
riscv64-unknown-elf-g++
riscv64-unknown-elf-gcc
riscv64-unknown-elf-gcc-4.9.2
riscv64-unknown-elf-gcc-ar
riscv64-unknown-elf-gcc-nm
riscv64-unknown-elf-gcc-ranlib
riscv64-unknown-elf-gcov
riscv64-unknown-elf-gprof
riscv64-unknown-elf-ld
riscv64-unknown-elf-ld.bfd
riscv64-unknown-elf-nm
riscv64-unknown-elf-objcopy
riscv64-unknown-elf-objdump
riscv64-unknown-elf-ranlib
riscv64-unknown-elf-readelf
riscv64-unknown-elf-size
riscv64-unknown-elf-strings
riscv64-unknown-elf-strip
spike
spike-dasm
termios-xspike
xspike
```

There are many utilities for compiler

Visualize the object files

Example

- Write a simple program in C
  - Calculate a sum of all elements in an array

```c
#include <stdio.h>
int main(){
    int i = 0;
    int sum = 0;
        int A[64];

    for(i = 0; i<64; i++){
       sum += A[i];
    }

    return sum;
}
```

  - Compile the program with RISC-V GNU toolchain
    - ../toolchain/riscv-unknown-elf-gcc/bin/riscv64-unknown-elf-gcc -o array_sum array_sum.c
    - Compile and build “array_sum.c” and output is a binary file named “array_sum”

Example

- Inspect the output binary file
  - ../toolchain/riscv-unknown-elf-gcc/bin/riscv64-unknown-elf-objdump -d array_sum

```c
#include <stdio.h>
int main( ){
    int i = 0;
    int sum = 0;
        int A[64];

    for( i = 0; i<64; i++){
       sum += A[i];
    }

    return sum;
}
```

```
00000000000101a4 <main>:
   101a4:       712d                    addi    sp,sp,-288
   101a6:       ee22                    sd      s0,280(sp)
   101a8:       1200                    addi    s0,sp,288
   101aa:       fe042623                sw      zero,-20(s0)
   101ae:       fe042423                sw      zero,-24(s0)
   101b2:       fe042623                sw      zero,-20(s0)
   101b6:       a01d                    j       101dc <main+0x38>
   101b8:       fec42783                lw      a5,-20(s0)
   101bc:       078a                    slli    a5,a5,0x2
   101be:       ff040713                addi    a4,s0,-16
   101c2:       97ba                    add     a5,a5,a4
   101c4:       ef87a783                lw      a5,-264(a5) # fffffffffffffef8
   101c8:       fe842703                lw      a4,-24(s0)
   101cc:       9fb9                    addw    a5,a5,a4
   101ce:       fef42423                sw      a5,-24(s0)
   101d2:       fec42783                lw      a5,-20(s0)
   101d6:       2785                    addiw   a5,a5,1
   101d8:       fef42623                sw      a5,-20(s0)
   101dc:       fec42783                lw      a5,-20(s0)
   101e0:       0007871b                sext.w  a4,a5
   101e4:       03f00793                li      a5,63
   101e8:       fce7d8e3                ble     a4,a5,101b8 <main+0x14>
   101ec:       fe842783                lw      a5,-24(s0)
   101f0:       853e                    mv      a0,a5
   101f2:       6472                    ld      s0,280(sp)
   101f4:       6115                    addi    sp,sp,288
   101f6:       8082                    ret
```

Example

- Main function

Base address

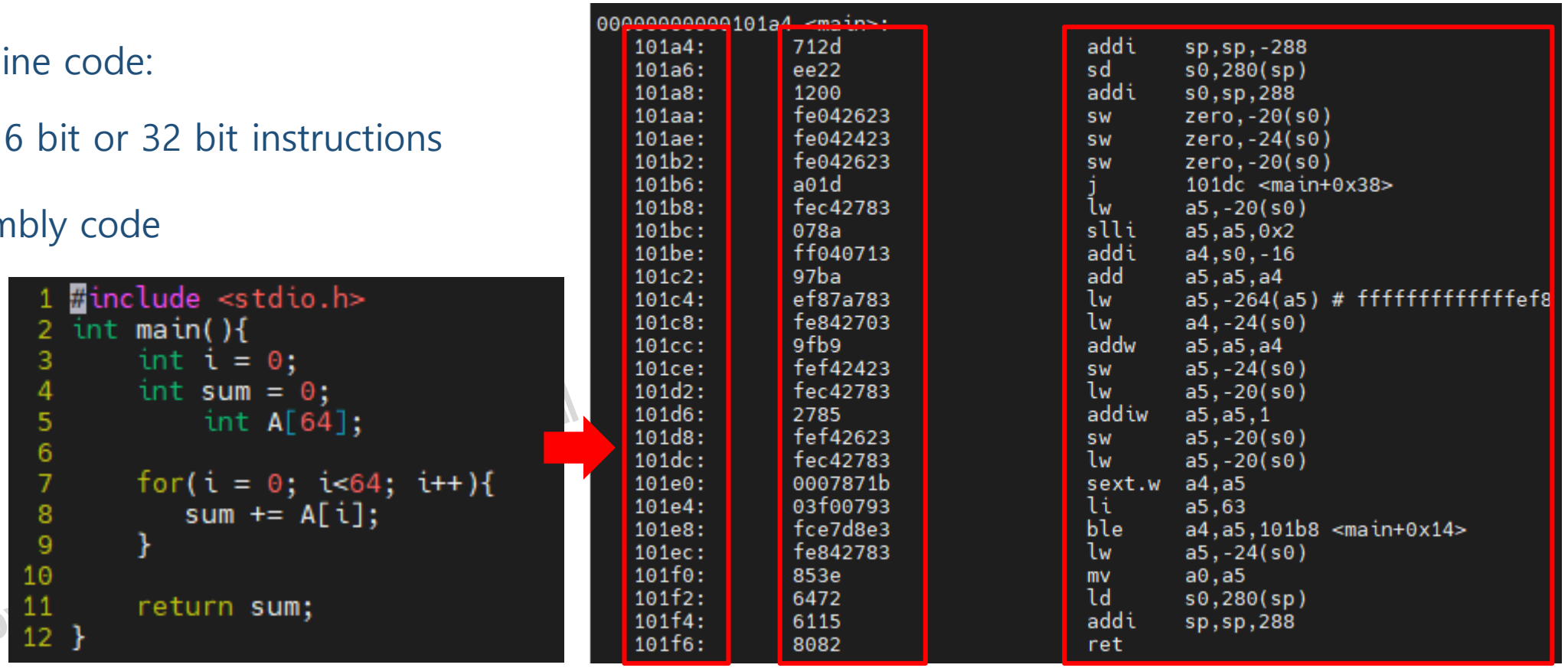
```
1  #include <stdio.h>
2  int main( ){
3      int i = 0;
4      int sum = 0;
5          int A[64];
6
7      for( i = 0; i<64; i++){
8          sum += A[i];
9      }
10
11     return sum;
12 }
```

```
00000000000101a4 <main>:
   101a4:       712d                    addi    sp,sp,-288
   101a6:       ee22                    sd      s0,280(sp)
   101a8:       1200                    addi    s0,sp,288
   101aa:       fe042623                sw      zero,-20(s0)
   101ae:       fe042423                sw      zero,-24(s0)
   101b2:       fe042623                sw      zero,-20(s0)
   101b6:       a01d                    j       101dc <main+0x38>
   101b8:       fec42783                lw      a5,-20(s0)
   101bc:       078a                    slli    a5,a5,0x2
   101be:       ff040713                addi    a4,s0,-16
   101c2:       97ba                    add     a5,a5,a4
   101c4:       ef87a783                lw      a5,-264(a5) # fffffffffffffef8
   101c8:       fe842703                lw      a4,-24(s0)
   101cc:       9fb9                    addw    a5,a5,a4
   101ce:       fef42423                sw      a5,-24(s0)
   101d2:       fec42783                lw      a5,-20(s0)
   101d6:       2785                    addiw   a5,a5,1
   101d8:       fef42623                sw      a5,-20(s0)
   101dc:       fec42783                lw      a5,-20(s0)
   101e0:       0007871b                sext.w  a4,a5
   101e4:       03f00793                li      a5,63
   101e8:       fce7d8e3                ble     a4,a5,101b8 <main+0x14>
   101ec:       fe842783                lw      a5,-24(s0)
   101f0:       853e                    mv      a0,a5
   101f2:       6472                    ld      s0,280(sp)
   101f4:       6115                    addi    sp,sp,288
   101f6:       8082                    ret
```

Example

- Memory address: Program counter
- Machine code:
  - 16 bit or 32 bit instructions
- Assembly code

```c
#include <stdio.h>
int main(){
    int i = 0;
    int sum = 0;
        int A[64];

    for(i = 0; i<64; i++){
       sum += A[i];
    }

    return sum;
}
```



Memory address Machine Code Assembly Code

Example

- Initialization

```c
#include <stdio.h>
int main(){
    int i = 0;
    int sum = 0;
        int A[64];

    for(i = 0; i<64; i++){
       sum += A[i];
    }

    return sum;
}
```

```
00000000000101a4 <main>:
   101a4:       712d                    addi    sp,sp,-288
   101a6:       ee22                    sd      s0,280(sp)
   101a8:       1200                    addi    s0,sp,288
   101aa:       fe042623                sw      zero,-20(s0)
   101ae:       fe042423                sw      zero,-24(s0)
   101b2:       fe042623                sw      zero,-20(s0)
   101b6:       a01d                    j       101dc <main+0x38>
   101b8:       fec42783                lw      a5,-20(s0)
   101bc:       078a                    slli    a5,a5,0x2
   101be:       ff040713                addi    a4,s0,-16
   101c2:       97ba                    add     a5,a5,a4
   101c4:       ef87a783                lw      a5,-264(a5) # ffffffffffffef8
   101c8:       fe842703                lw      a4,-24(s0)
   101cc:       9fb9                    addw    a5,a5,a4
   101ce:       fef42423                sw      a5,-24(s0)
   101d2:       fec42783                lw      a5,-20(s0)
   101d6:       2785                    addiw   a5,a5,1
   101d8:       fef42623                sw      a5,-20(s0)
   101dc:       fec42783                lw      a5,-20(s0)
   101e0:       0007871b                sext.w  a4,a5
   101e4:       03f00793                li      a5,63
   101e8:       fce7d8e3                ble     a4,a5,101b8 <main+0x14>
   101ec:       fe842783                lw      a5,-24(s0)
   101f0:       853e                    mv      a0,a5
   101f2:       6472                    ld      s0,280(sp)
   101f4:       6115                    addi    sp,sp,288
   101f6:       8082                    ret
```

Memory address Machine Code Assembly Code

Example

- Sum accumulation

a5: sum

a4: A[i]

addw a5,a5,a4

```c
#include <stdio.h>
int main(){
    int i = 0;
    int sum = 0;
        int A[64];

    for(i = 0; i<64; i++){
        sum += A[i];
    }

    return sum;
}
```

```
00000000000101a4 <main>:
   101a4:       712d                addi    sp,sp,-288
   101a6:       ee22                sd      s0,280(sp)
   101a8:       1200                addi    s0,sp,288
   101aa:       fe042623            sw      zero,-20(s0)
   101ae:       fe042423            sw      zero,-24(s0)
   101b2:       fe042623            sw      zero,-20(s0)
   101b6:       a01d                j       101dc <main+0x38>
   101b8:       fec42783            lw      a5,-20(s0)
   101bc:       078a                slli    a5,a5,0x2
   101be:       ff040713            addi    a4,s0,-16
   101c2:       97ba                add     a5,a5,a4
   101c4:       ef87a783            lw      a5,-264(a5) # fffffffffffffef8
   101c8:       fe842703            lw      a4,-24(s0)
   101cc:       9fb9                addw    a5,a5,a4
   101ce:       fef42423            sw      a5,-24(s0)
   101d2:       fec42783            lw      a5,-20(s0)
   101d6:       2785                addiw   a5,a5,1
   101d8:       fef42623            sw      a5,-20(s0)
   101dc:       fec42783            lw      a5,-20(s0)
   101e0:       0007871b            sext.w  a4,a5
   101e4:       03f00793            li      a5,63
   101e8:       fce7d8e3            ble     a4,a5,101b8 <main+0x14>
   101ec:       fe842783            lw      a5,-24(s0)
   101f0:       853e                mv      a0,a5
   101f2:       6472                ld      s0,280(sp)
   101f4:       6115                addi    sp,sp,288
   101f6:       8082                ret
```

Memory address Machine Code Assembly Code

Example

- Conditional branch

a4: loop index

a5: max iter.

BLE a4, a5, 101b8

```c
#include <stdio.h>
int main(){
    int i = 0;
    int sum = 0;
        int A[64];

    for(i = 0; i<64; i++){
        sum += A[i];
    }

    return sum;
}
```

```
00000000000101a4 <main>:
   101a4:       712d                addi    sp,sp,-288
   101a6:       ee22                sd      s0,280(sp)
   101a8:       1200                addi    s0,sp,288
   101aa:       fe042623            sw      zero,-20(s0)
   101ae:       fe042423            sw      zero,-24(s0)
   101b2:       fe042623            sw      zero,-20(s0)
   101b6:       a01d                j       101dc <main+0x38>
   101b8:       fec42783            lw      a5,-20(s0)
   101bc:       078a                slli    a5,a5,0x2
   101be:       ff040713            addi    a4,s0,-16
   101c2:       97ba                add     a5,a5,a4
   101c4:       ef87a783            lw      a5,-264(a5) # fffffffffffffef8
   101c8:       fe842703            lw      a4,-24(s0)
   101cc:       9fb9                addw    a5,a5,a4
   101ce:       fef42423            sw      a5,-24(s0)
   101d2:       fec42783            lw      a5,-20(s0)
   101d6:       2785                addiw   a5,a5,1
   101d8:       fef42623            sw      a5,-20(s0)
   101dc:       fec42783            lw      a5,-20(s0)
   101e0:       0007871b            sext.w  a4,a5
   101e4:       03f00793            li      a5,63
   101e8:       fce7d8e3            ble     a4,a5,101b8 <main+0x14>
   101ec:       fe842783            lw      a5,-24(s0)
   101f0:       853e                mv      a0,a5
   101f2:       6472                ld      s0,280(sp)
   101f4:       6115                addi    sp,sp,288
   101f6:       8082                ret
```

Memory address Machine Code Assembly Code

Road map

Review

Load, store instructions

RISC-V core integration

RISC-V GNU Compiler

Optimization

Lab 3: Optimization

- Lab 3:
  - Execute ***different machine codes*** on the RISC-V core
  - Do a simulation
  - Show the output result

Different Assembly codes

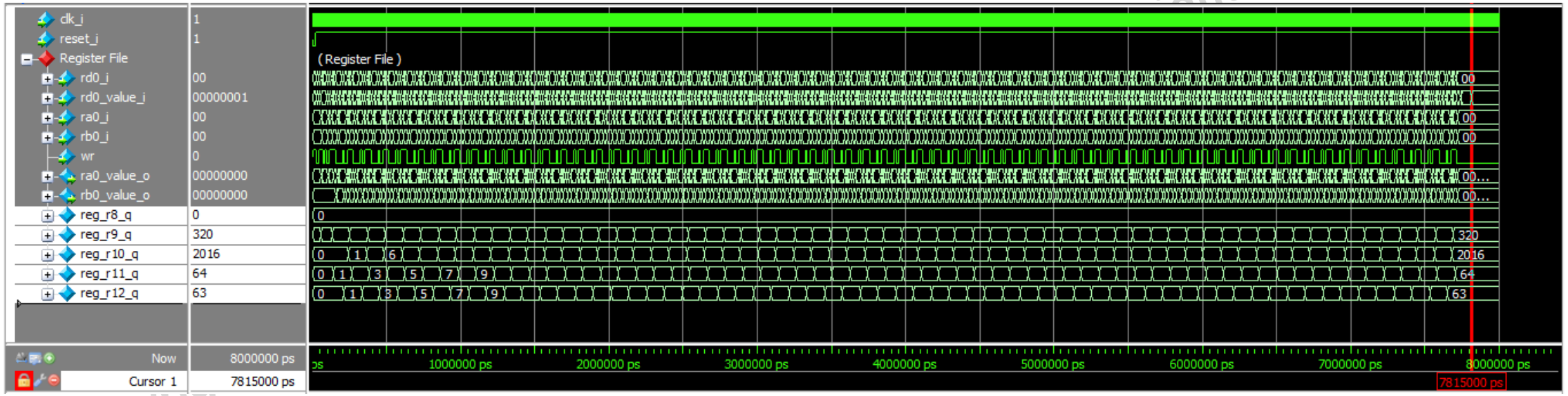
- RISC-V core runs different programs
- Three assemble versions of a C program
  - Baseline
  - OPT1: reorder instructions 5 and 6.
  - OPT2: reorder instructions 5, 6, and 7.

C code	Assembly Code		
	Baseline	Optimized 1 (Opt1)	Optimized 2 (Opt2)

int A[64];
int sum = 0;
for (int i=0; i<64; i++)
  sum += A[i];
	addi x9, x0, 64
	add x10, x0, x0
	add x11, x0, x0
	Loop:
	  lw x12, 0(x9)
	  add x10, x10, x12
	  addi x9, x9, 4
	  addi x11, x11, 1
	  addi x13, x0, 64
	  blt x11, x13, Loop
		addi x9, x0, 64
		add x10, x0, x0
		add x11, x0, x0
		Loop:
		  lw x12, 0(x9)
		  addi x9, x9, 4
		  add x10, x10, x12
		  addi x11, x11, 1
		  addi x13, x0, 64
		  blt x11, x13, Loop
			addi x9, x0, 64
			add x10, x0, x0
			add x11, x0, x0
			Loop:
			  lw x12, 0(x9)
			  addi x9, x9, 4
			  addi x11, x11, 1
			  add x10, x10, x12
			  addi x13, x0, 64
			  blt x11, x13, Loop

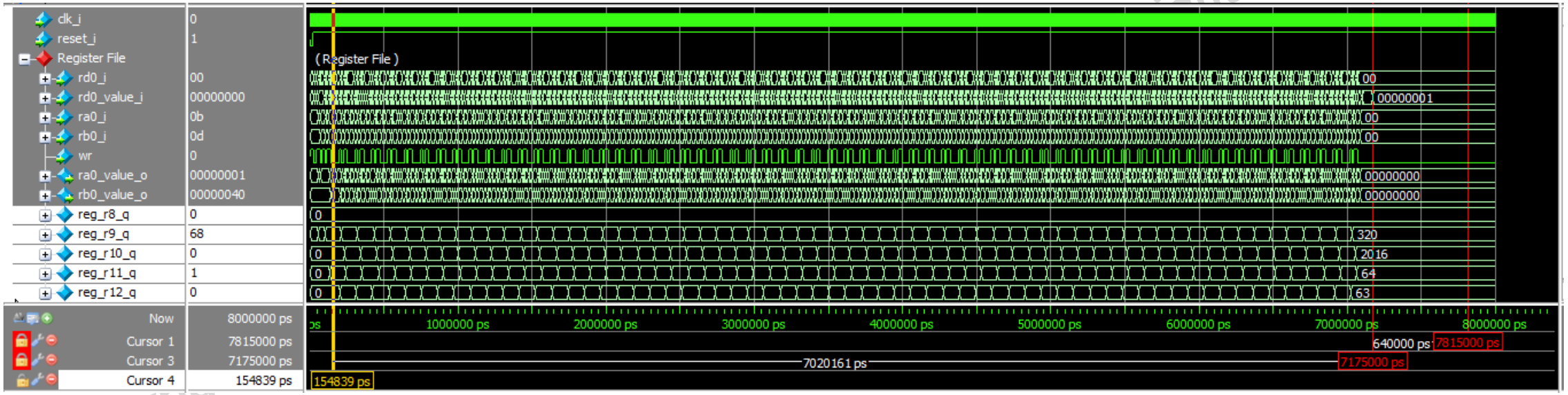
Waveform: Baseline

- Check the execution time
  - 7,815 ns



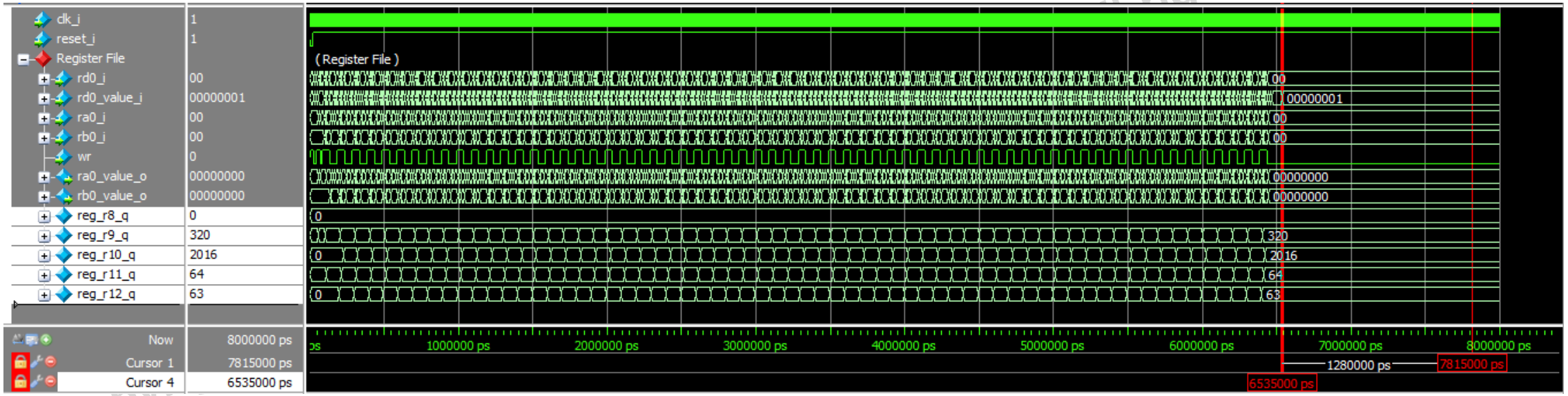
Waveform: Optimized code 1

- Check the execution time
  - 7,175 ns
  - OPT1 vs Baseline: Reduce by 640ns



Waveform: Optimized code 2

- Check execution time
  - 6,535 ns
  - OPT2 vs Baseline: Reduce by 1280ns



To do …

- Create memory files (mem_opt1.hex and mem_opt2.hex).
- Modify a test bench with new memory files (riscv_core_tb.v)
- Do a simulation
- Captures the waveform.
- Compare the running times of three versions: *baseline, opt1, and opt2.*
- Explain the experimental results.