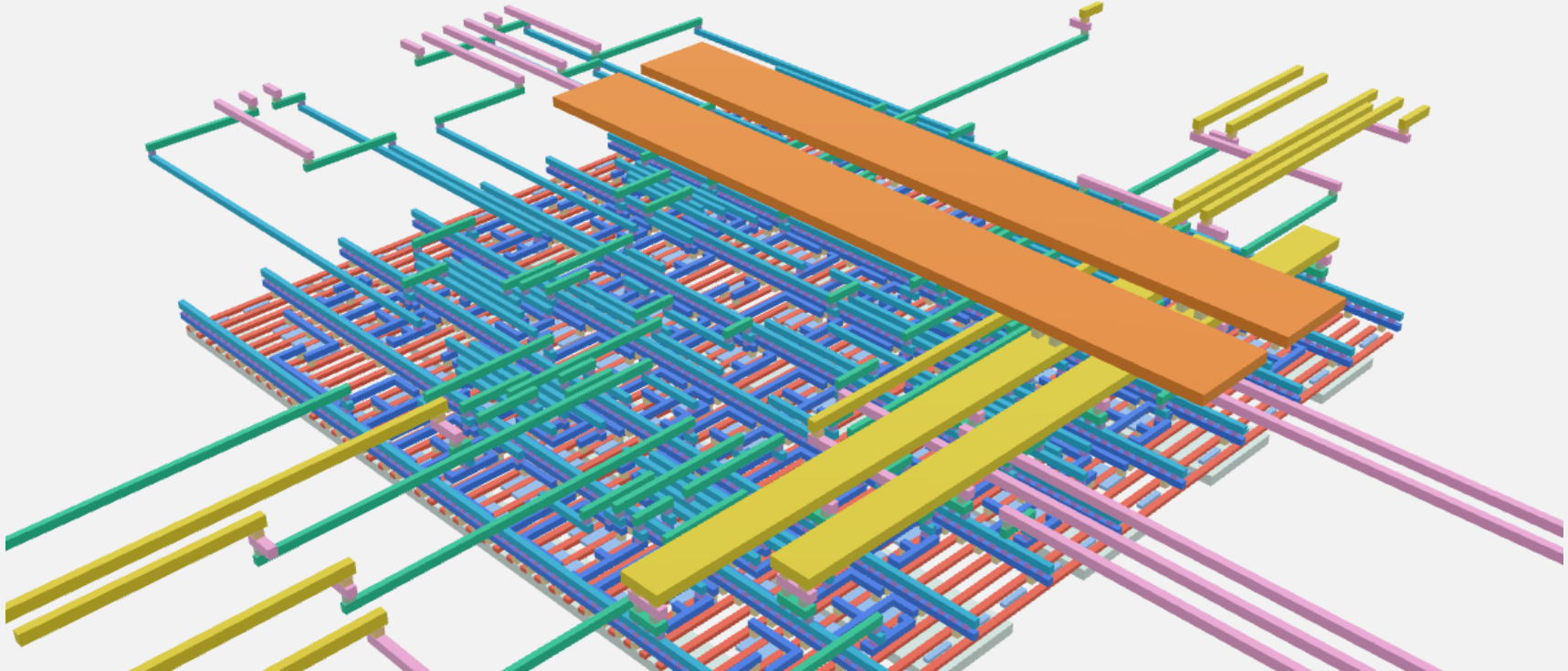
5.2
CSE140 – Digital Design

2026 Summer Session 1

Abarajithan G



Daily reminder:

- Open FeedbackFruits
  Canvas → CSE140 → Assignments → Slides 5.2

- Attendance is mandatory

- In class quiz correctness – Out of all quizzes, we will choose the best 50%

- Optimization presentations – don't put too much effort in preparing it
  - Keep it simple and to the point – no introduction, conclusion…etc.
  - Have: idea (why it works), implementation (snippets of your code), results (% improvement)
  - Diagrams – draw on paper & take photo / draw on ipad

CPU

- CPU in 40 lines of SystemVerilog
- Single-cycle RISC CPU
- Custom assembly language
- Only 7 opcodes
- Can write interesting programs:
  Sum to N, Fibonacci, Image Processing

```
// Initialize N, constant one, and the sum.
imem.mem[0] = {8'h02,        4'h2, LOAD};  // r2 (counter) = N = 10
imem.mem[1] = {8'h01,        4'h1, LOAD};  // r1 (one)     = mem[1] = 1
imem.mem[2] = {8'h00,        4'h0, LOAD};  // r0 (sum)     = mem[0] = 0

// Add each counter value and count down to zero.
imem.mem[3] = {4'h2,  4'h0,  4'h0, ADD};   // r0 (sum) += r2 (counter)
imem.mem[4] = {4'h1,  4'h2,  4'h2, SUB};   // r2 (counter) -= r1 (one)
imem.mem[5] = {8'h03,        4'h2, JNZ};   // repeat while r2 (count)!=0

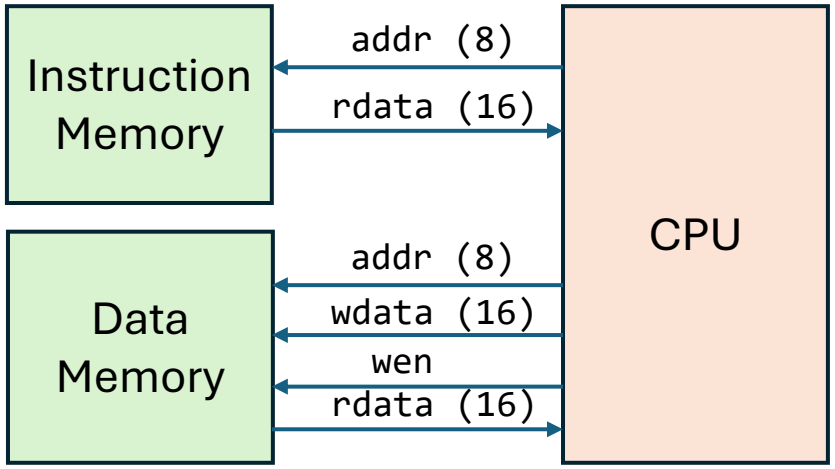
imem.mem[6] = {8'h04,        4'h0, STORE}; // mem[4] = r0 (sum)
```

```
module cpu (
  input  logic        clk, reset,
  output logic [7 :0] imem_addr,  dmem_addr,
  input  logic [15:0] imem_rdata, dmem_rdata,
  output logic [15:0] dmem_wdata,
  output logic        dmem_wen
  );
  logic [7:0] pc, addr;
  enum logic [3:0]{LOAD,STORE,MOVE,ADD,SUB,MUL,JNZ} opcode;
  logic [ 3:0] i_rs1, i_rs2, i_rd, i_reg;
  logic [15:0] regs [16];
  logic [15:0] reg_1, reg_2;
  always_comb begin
    imem_addr  = pc;
    {addr        , i_reg, opcode} = imem_rdata;
    {i_rs2, i_rs1, i_rd , opcode} = imem_rdata;

    dmem_addr  = addr;
    dmem_wdata = regs[i_reg];
    dmem_wen   = opcode == STORE;

    reg_1      = regs[i_rs1];
    reg_2      = regs[i_rs2];
  end
  always_ff @(posedge clk)
    if (reset) begin
      pc   <= '0;
      for (int i = 0; i < 16; i++) regs[i] <= '0;
    end else begin
      pc   <= pc + 1'b1;
      case (opcode)
        LOAD: regs[i_reg] <= dmem_rdata;
        MOVE: regs[i_rd ] <= reg_1;
        ADD : regs[i_rd ] <= reg_1 + reg_2;
        SUB : regs[i_rd ] <= reg_1 - reg_2;
        MUL : regs[i_rd ] <= reg_1 * reg_2;
        JNZ : if (regs[i_reg] != '0) pc <= addr;
      endcase end
endmodule
```

Memory



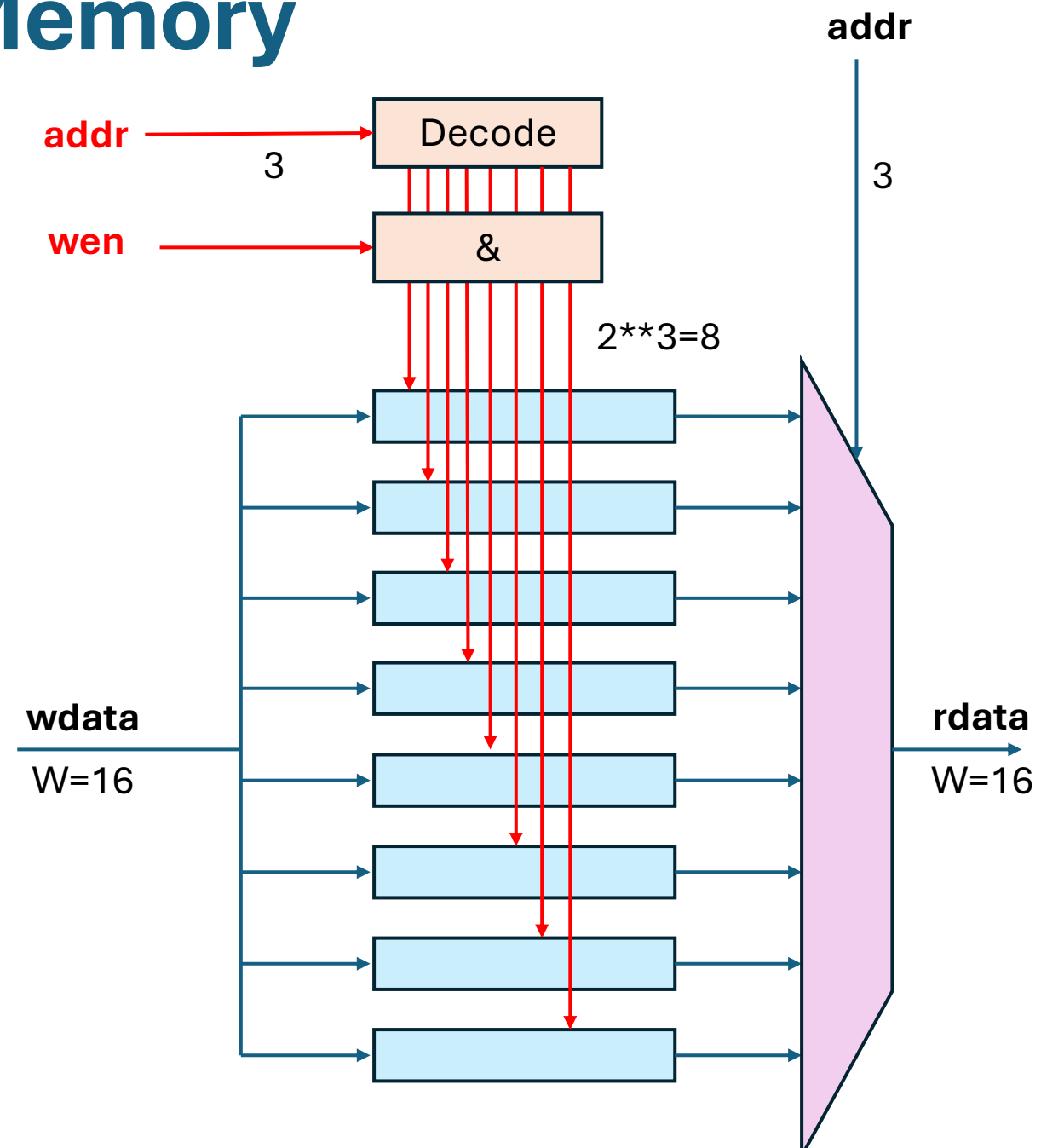
- Address:
  - We are keeping the address bus of memory (and CPU) fixed with – 8 bits: 256 rows maximum
  - We'll see why
  - But we want to parametrize the number of rows in memory, to allow less than 256 rows
  - Number of actual rows = 2**ADDR_W
- Activity:
  - Draw the circuit for the code
  - Latency (clock cycles) for read, write?
  - Interval (clock cycles) for read, write?

```
module memory #(
  parameter int ADDR_W = 3
  // depth = 2**ADDR_W rows
)(
  input  logic         clk,
  input  logic [7:0]   addr,
  input  logic [15:0]  wdata,
  input  logic         wen,
  output logic [15:0]  rdata
);
  logic [15:0] mem [2**ADDR_W];
  logic [ADDR_W-1:0] addr_short;
  always_comb addr_short = addr[ADDR_W-1:0];

  always_ff @(posedge clk)
    if (wen) mem[addr_short] <= wdata;

  always_comb rdata = mem[addr_short];
endmodule
```

Memory




```
module memory #(
  parameter int ADDR_W = 3
  // depth = 2**ADDR_W rows
)(
  input  logic        clk,
  input  logic [7:0]  addr,
  input  logic [15:0] wdata,
  input  logic        wen,
  output logic [15:0] rdata
);
  logic [15:0] mem [2**ADDR_W];
  logic [ADDR_W-1:0] addr_short;
  always_comb addr_short = addr[ADDR_W-1:0];

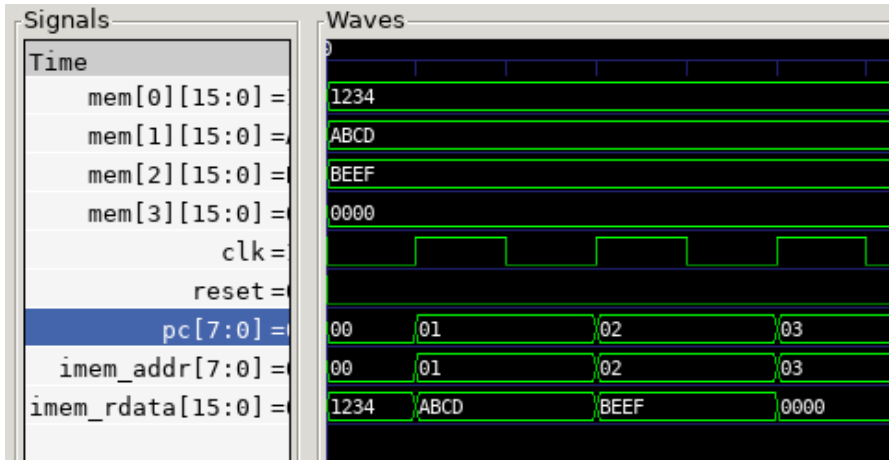
  always_ff @(posedge clk)
    if (wen) mem[addr_short] <= wdata;

  always_comb rdata = mem[addr_short];
endmodule
```

- Read latency = 0
- Write latency = 1
- Read throughput = 1 every cycle
- Write throughput = 1 every cycle

1. Load Instruction

- A counter = (imem_addr) counts every cycle
- imem_rdata (our instruction) arrives in the same clock. Read latency = 0
- This is how a program is executed line by line
- One line (in assembly) in one clock cycle
- Single cycle CPU




```
module cpu_1_load_instruction (
  input  logic        clk, reset,
  output logic [7 :0] imem_addr, // program counter
  input  logic [15:0] imem_rdata // our instruction!
);
  logic [7:0] pc;
  always_comb imem_addr = pc;
  always_ff @(posedge clk)
    if (reset) pc   <= '0;
    else       pc   <= pc + 1'b1;
endmodule
```

The CPU is a simple counter (for now)

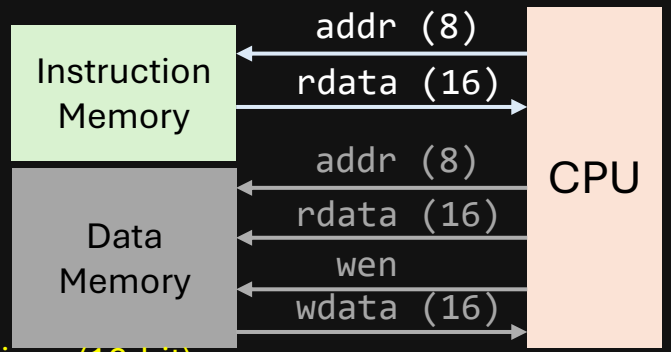
```
`timescale 1ns/1ps
module tb_cpu_1_load_instruction;
  logic clk = 0, reset = 0;
  logic [7:0] imem_addr, dmem_addr;
  logic [15:0] imem_rdata, dmem_rdata, dmem_wdata;
  logic dmem_wen;
  memory imem(clk, imem_addr,          '0,     1'b0, imem_rdata);
  memory dmem(clk, dmem_addr, dmem_wdata, dmem_wen, dmem_rdata);
  cpu_1_load_instruction dut(.*);

  typedef enum logic[3:0]{LOAD,STORE,MOVE,ADD,SUB,MUL,JNZ} op_t;

  initial forever #1 clk = ~clk;
  initial begin
    $dumpfile(`FST_PATH); $dumpvars;
    // Read three example instructions.
    imem.mem[0] = 16'h1234; // ins0
    imem.mem[1] = 16'hABCD; // ins1
    imem.mem[2] = 16'hBEEF; // ins2
    repeat (20) @(posedge clk);
    $finish;
  end
endmodule
```

only imem is connected and used now
dmem is used in step 2

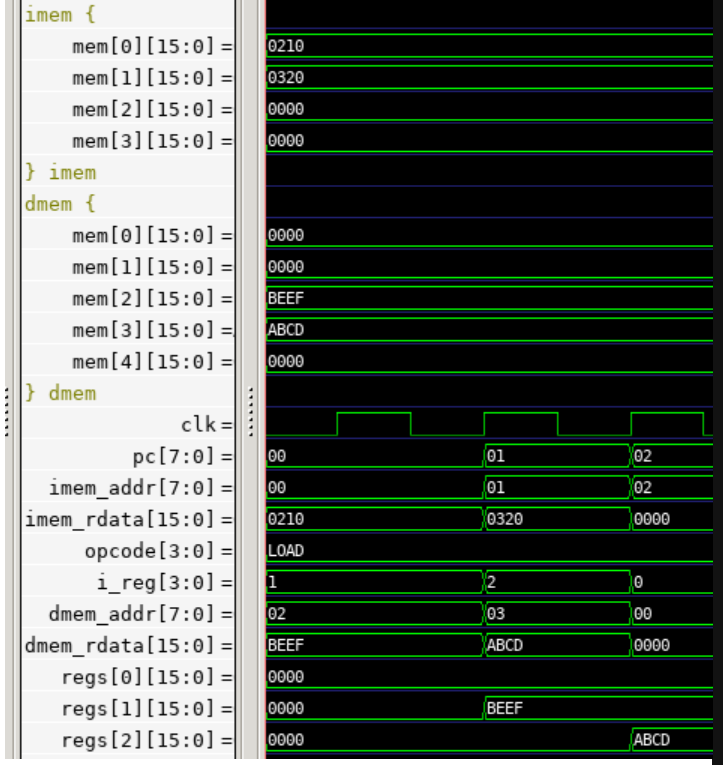
Opcodes – types of instructions (to be used later)



Program is a sequence of instructions (16-bit)
Loaded into imem before starting the CPU

2. Load Data

- Load (copy) a word (16 bits) from data mem into register file
- Register file with 16 registers
- Data memory interface
- Instruction (16 bits) is “decoded” into
  - **opcode** (4) : 0 = LOAD
  - **i_reg** (4) : which of the 16 registers to load into
  - **addr** (8) : Addr of the data memory to load from
  - 16’h**0030** : **LOAD(0)** regs[**3**] ← mem[8’h**00**]
- What is the latency of load?



Instructions	Format	4 Bits [15:12]	4 Bits [11:8]	4 Bits [7:4]	4 Bits [3:0]
LOAD, STORE, JNZ	Address	addr		i_reg	opcode

```
dmem.mem[2] = 16'hBEEF;
dmem.mem[3] = 16'hABCD;
// Load memory locations 2 and 3 into registers 1 and 2.
imem.mem[0] = {8'h02, 4'h1, LOAD}; // R1 = *(2)
imem.mem[1] = {8'h03, 4'h2, LOAD}; // R2 = *(3)
```

Inputs to the program, initial state of data mem

Program

```
module cpu_2_load_data_into_registers (
  input  logic        clk, reset,
  output logic [7:0]  imem_addr,
  input  logic [15:0] imem_rdata,
  output logic [7:0]  dmem_addr,  // --- new
  input  logic [15:0] dmem_rdata // --- new
);
  logic [7:0] pc, addr;
  enum logic [3:0] {LOAD} opcode; // --- new
  logic [ 3:0] i_reg; // --- new
  logic [15:0] regs [16]; // --- new

  always_comb begin
    imem_addr         = pc;
    {addr, i_reg, opcode} = imem_rdata; // --- new
    dmem_addr = addr; // --- new
  end
  always_ff @(posedge clk) begin
    if (reset) begin
      pc    <= '0;
      for (int i=0; i< 16; i++) regs[i] <= '0;//-new
    end else begin
      pc    <= pc + 1'b1;
      case (opcode) // --- new
        LOAD: regs[i_reg] <= dmem_rdata; // --- new
      endcase
    end
  end
endmodule
```

Register file

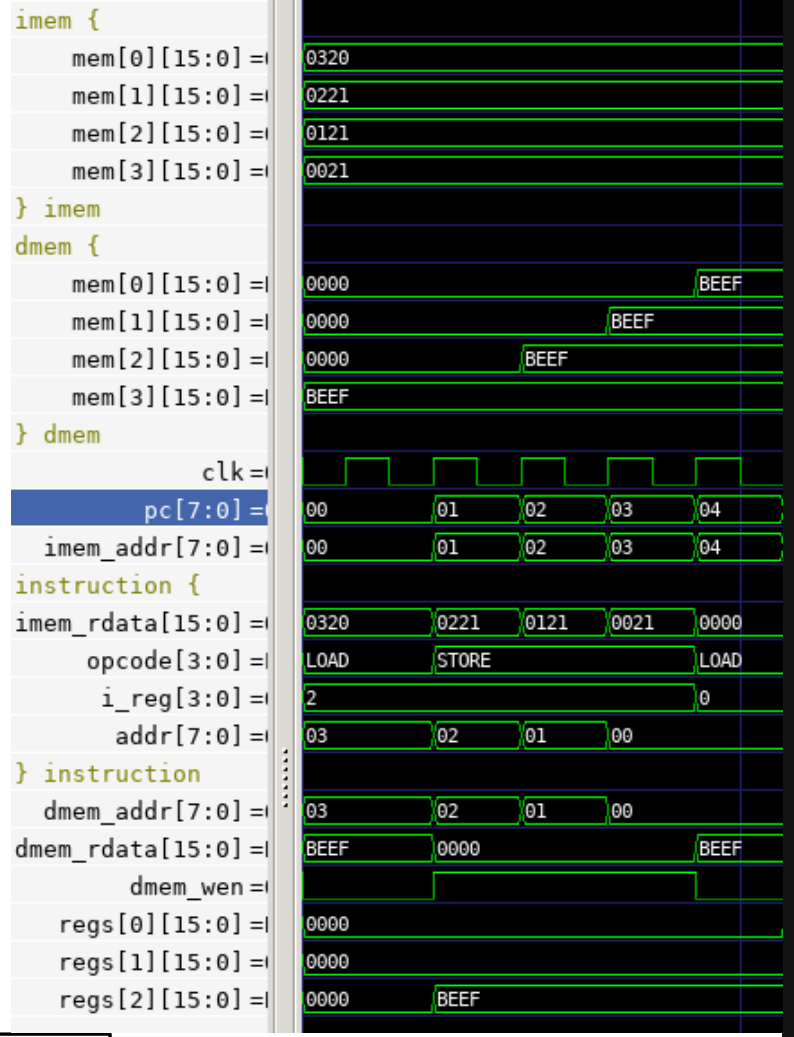
Decoding an instruction

Initialize regs to zero

Loading into register file

3. Store Data

- Opcodes: **LOAD=0, STORE=1**
- **STORE: dmem[addr] ← regs[i_reg]**
- dmem_wen = (opcode==STORE)
- Reading our instructions:
  - **0320- load(0):regs[2] ← mem[03]**
  - **0221-store(1):mem[02] ← regs[2]**
- Activity:
  Draw the datapath from registers to dmem



Instructions	Format	4 Bits [15:12]	4 Bits [11:8]	4 Bits [7:4]	4 Bits [3:0]
LOAD, STORE, JNZ	Address	addr		i_reg	opcode

```
dmem.mem[3] = 16'hBEEF;
// Load 0xBEEF from addr 3 into r2, then store r2 at addresses 2, 1 and 0.
imem.mem[0] = {8'h03, 4'h2, LOAD };  // R2 = *(3)
imem.mem[1] = {8'h02, 4'h2, STORE};  // *(2) = R2
imem.mem[2] = {8'h01, 4'h2, STORE};  // *(1) = R2
imem.mem[3] = {8'h00, 4'h2, STORE};  // *(0) = R2
```

Initial data

Program

```
module cpu_3_store_data (
  input  logic        clk, reset,
  output logic [7:0]  imem_addr,
  input  logic [15:0] imem_rdata,
  output logic [7:0]  dmem_addr,
  input  logic [15:0] dmem_rdata,
  output logic [15:0] dmem_wdata, // --- new
  output logic        dmem_wen    // --- new
);
  logic [7:0] pc, addr;
  enum logic [3:0] {LOAD, STORE} opcode;
  logic  [3:0] i_reg;
  logic [15:0] regs [16];

  always_comb begin
    imem_addr             = pc;
    {addr, i_reg, opcode} = imem_rdata;
    dmem_addr  = addr;
    dmem_wdata = regs[i_reg]; // --- new
    dmem_wen   = opcode == STORE; // --- new
  end
  always_ff @(posedge clk) begin
    if (reset) begin
      pc   <= '0;
      for (int i=0; i<16; i++) regs[i]<='0;
    end else begin
      pc   <= pc + 1'b1;
      case (opcode)
        LOAD: regs[i_reg] <= dmem_rdata;
        default: ;
      endcase end end
endmodule
```

Dmem write port

Enum vals auto assigned

Directly connected

4. Move, ALU

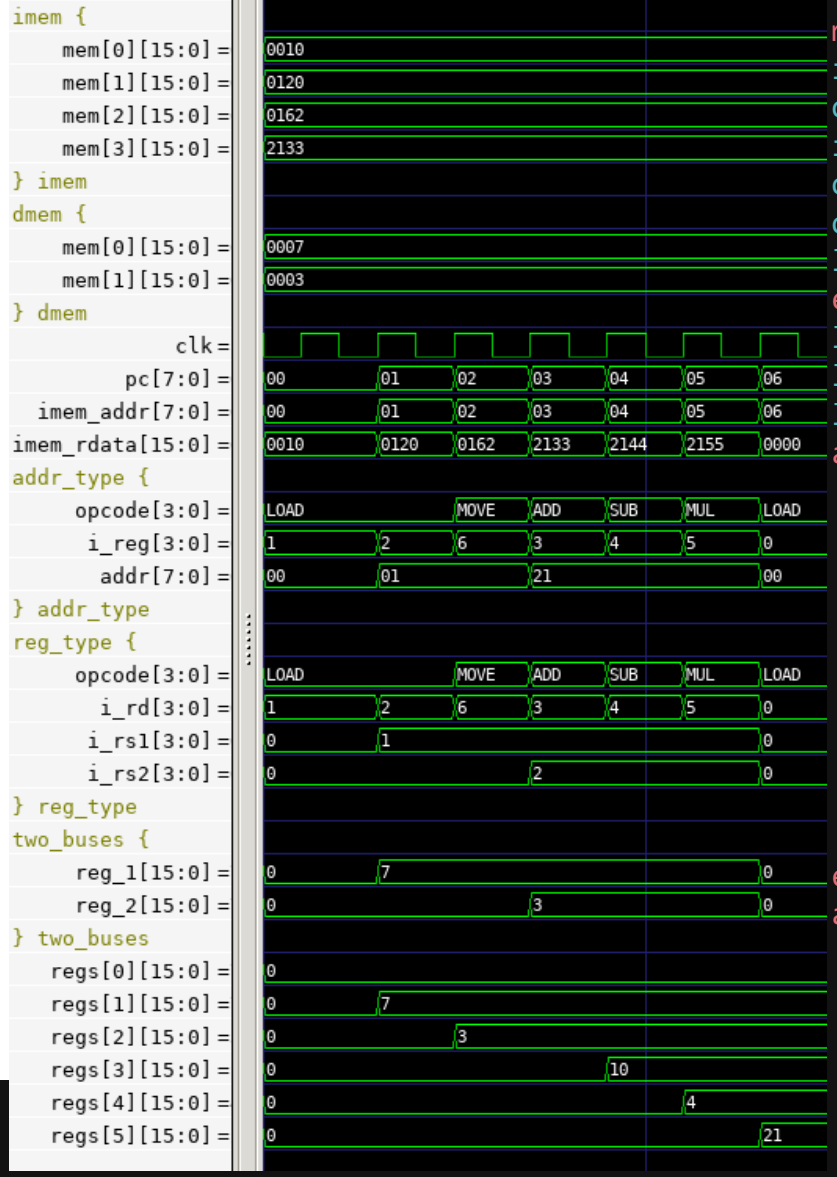
field	[3:0]	[3:0]	[3:0]	[3:0]
a_type	addr		i_reg	opcode
r_type	i_rs2	i_rs1	i_rd	opcode

- i_rs1 – index of source reg 1
- i_rs2 – index of source reg 2
- i_rd – index of destination register

```
MOVE(2):r[rd]←r[i_rs1]
ADD (3):r[rd]←r[i_rs1]+r[i_rs2]
SUB (4):r[rd]←r[i_rs1]-r[i_rs2]
MUL (5):r[rd]←r[i_rs1]*r[i_rs2]
```

- Activity: draw datapath from registers to registers, via ALU

```
dmem.mem[0] = 16'd7;
dmem.mem[1] = 16'd3;
// Load 7,3; copy r1; calculate sum, difference, and product.
imem.mem[0] = {8'h00,        4'h1, LOAD}; // R1 = *(0) = 7
imem.mem[1] = {8'h01,        4'h2, LOAD}; // R2 = *(1) = 3
imem.mem[2] = {4'h0,  4'h1,  4'h6, MOVE}; // R6 = R1
imem.mem[3] = {4'h2,  4'h1,  4'h3, ADD }; // R3 = R1 + R2
imem.mem[4] = {4'h2,  4'h1,  4'h4, SUB }; // R4 = R1 - R2
imem.mem[5] = {4'h2,  4'h1,  4'h5, MUL }; // R5 = R1 * R2
```




```
module cpu_4_move_alu (
input  logic         clk, reset,
output logic [7 :0] imem_addr,  dmem_addr,
input  logic [15:0] imem_rdata, dmem_rdata,
output logic [15:0] dmem_wdata,
output logic         dmem_wen);
logic [7:0] pc, addr;
enum logic [3:0]{LOAD,STORE,MOVE,ADD,SUB,MUL}opcode;
logic [ 3:0] i_rs1, i_rs2, i_rd, i_reg;
logic [15:0] regs [16];
logic [15:0] reg_1, reg_2; // --- new
always_comb begin
  imem_addr = pc;
  {addr         , i_reg, opcode} = imem_rdata;
  {i_rs2, i_rs1, i_rd , opcode} = imem_rdata;//--new

  dmem_addr  = addr;
  dmem_wdata = regs[i_reg];
  dmem_wen   = opcode == STORE;

  reg_1      = regs[i_rs1]; // --- new
  reg_2      = regs[i_rs2]; // --- new
end
always_ff @(posedge clk) begin
  if (reset) begin
    pc   <= '0;
    for (int i = 0; i < 16; i++) regs[i] <= '0;
  end else begin
    pc   <= pc + 1'b1;
    case (opcode)
      LOAD: regs[i_reg] <= dmem_rdata;
      MOVE: regs[i_rd ] <= reg_1;           // --- new
      ADD : regs[i_rd ] <= reg_1 + reg_2;   // --- new
      SUB : regs[i_rd ] <= reg_1 - reg_2;   // --- new
      MUL : regs[i_rd ] <= reg_1 * reg_2;   // --- new
    endcase end end
endmodule
```

Decode the 16-bit instruction in two ways

Two buses – select two registers

ALU to Reg

5. Jump

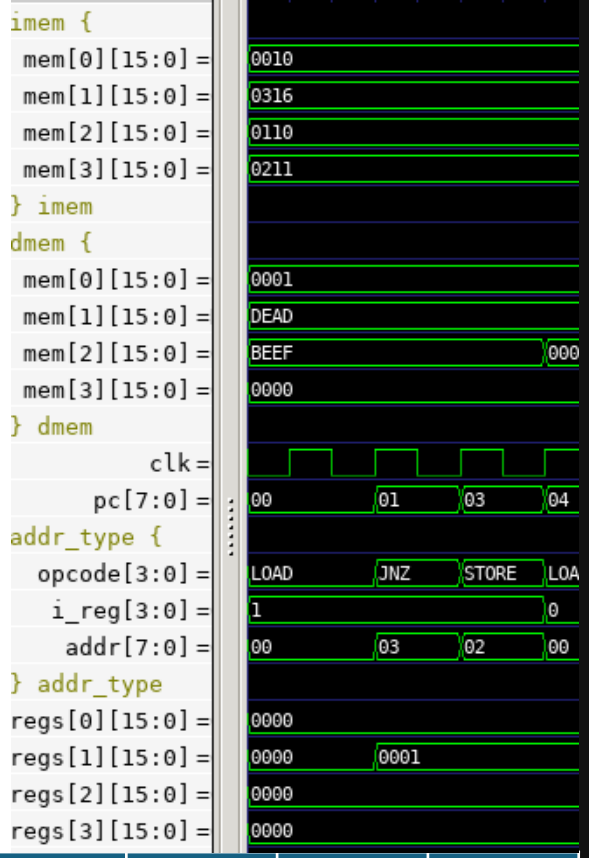
- Jump is used for branching if/else, for loops
- E.g.

```
R1 = *(0);
if (R1==0) R1 = *(1);
*(2) = R1;
```

- Can be written as

```
R1 = *(0);
if (R1!=0) goto 3;
R1 = *(1);
*(2) = R1;
```

- Other kinds of jumps: if zero, if equal...



field	[3:0]	[3:0]	[3:0]	[3:0]
a_type	addr		i_reg	opcode
r_type	i_rs2	i_rs1	i_rd	opcode

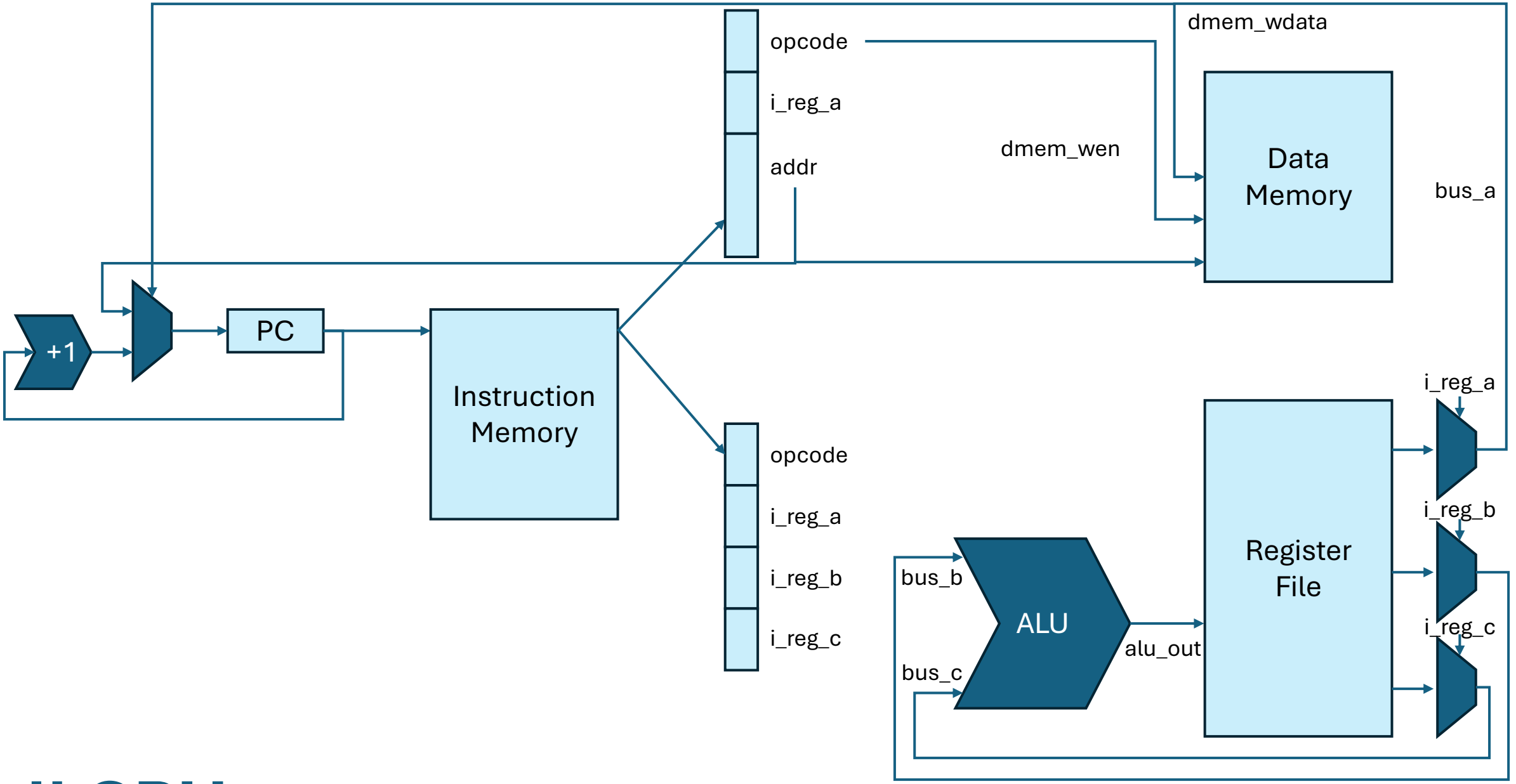
```
dmem.mem[0] = 16'd1;
dmem.mem[1] = 16'hDEAD;
dmem.mem[2] = 16'hBEEF;
// Put 1 in r1, then jump from PC 1 to imem address 3.
imem.mem[0] = {8'h00, 4'h1, LOAD }; // R1=*(0)
imem.mem[1] = {8'h03, 4'h1, JNZ  }; // if (R1!=0) goto [3]
imem.mem[2] = {8'h01, 4'h1, LOAD }; // R1=*(1)=DEAD (skipped)
imem.mem[3] = {8'h02, 4'h1, STORE}; // *(2) = R1 = 1
```

```
module cpu_5_jump (
  input  logic         clk, reset,
  output logic [7 :0] imem_addr,  dmem_addr,
  input  logic [15:0] imem_rdata, dmem_rdata,
  output logic [15:0] dmem_wdata,
  output logic         dmem_wen
  );
  logic [7:0] pc, addr;
  enum logic [3:0] {LOAD, STORE, MOVE, ADD, SUB, MUL, JNZ} opcode;
  logic [ 3:0] i_rs1, i_rs2, i_rd, i_reg;
  logic [15:0] regs [16];
  logic [15:0] reg_1, reg_2;
  always_comb begin
    imem_addr  = pc;
    {addr        , i_reg, opcode} = imem_rdata;
    {i_rs2, i_rs1, i_rd , opcode} = imem_rdata;

    dmem_addr  = addr;
    dmem_wdata = regs[i_reg];
    dmem_wen   = opcode == STORE;

    reg_1      = regs[i_rs1];
    reg_2      = regs[i_rs2];
  end
  always_ff @(posedge clk) begin
    if (reset) begin
      pc   <= '0;
      for (int i = 0; i < 16; i++) regs[i] <= '0;
    end else begin
      pc   <= pc + 1'b1;
      case (opcode)
        LOAD: regs[i_reg] <= dmem_rdata;
        MOVE: regs[i_rd ] <= reg_1;
        ADD : regs[i_rd ] <= reg_1 + reg_2;
        SUB : regs[i_rd ] <= reg_1 - reg_2;
        MUL : regs[i_rd ] <= reg_1 * reg_2;
        JNZ : if (regs[i_reg] != '0) pc <= addr; //---new
      endcase end end // last assignment wins
endmodule
```

Last assignment wins


opcode
i_reg_a
addr
dmem_wdata
dmem_wen
Data Memory
bus_a
+1
PC
Instruction Memory
opcode
i_reg_a
i_reg_b
i_reg_c
i_reg_a
i_reg_b
i_reg_c
bus_b
bus_c
ALU
alu_out
Register File

Full CPU

CPU: Sum 0 to N

Sum the numbers 0 to 10

```
uint16_t mem[256];
mem[0] = 0;     // sum seed
mem[1] = 1;     // the constant one
mem[2] = 10;    // N
uint16_t sum = 0;
for (uint16_t i=mem[2]; i!=0; i-=mem[1])
    sum += i;
mem[4] = sum;   //55
```

```
0: R2_COUNT = *(2);
1: R1_ONE   = *(1);
2: R0_SUM   = *(0);
3: R0_SUM   = R0_SUM + R2_COUNT;
4: R2_COUNT = R2_COUNT - R1_ONE;
5: if (R2_COUNT!=0) goto 3;
6: *(4) = R0_SUM;
```

```
// Initialize N, constant one, and the sum.
imem.mem[0] = {8'h02,       4'h2, LOAD};  // r2 (counter) = N = 10
imem.mem[1] = {8'h01,       4'h1, LOAD};  // r1 (one)     = mem[1] = 1
imem.mem[2] = {8'h00,       4'h0, LOAD};  // r0 (sum)     = mem[0] = 0

// Add each counter value and count down to zero.
imem.mem[3] = {4'h2,  4'h0,  4'h0, ADD};   // r0 (sum) += r2 (counter)
imem.mem[4] = {4'h1,  4'h2,  4'h2, SUB};   // r2 (counter) -= r1 (one)
imem.mem[5] = {8'h03,       4'h2, JNZ};   // repeat while r2 (count)!=0

imem.mem[6] = {8'h04,       4'h0, STORE}; // mem[4] = r0 (sum)
```

```
module cpu (
  input  logic        clk, reset,
  output logic [7 :0] imem_addr,  dmem_addr,
  input  logic [15:0] imem_rdata, dmem_rdata,
  output logic [15:0] dmem_wdata,
  output logic        dmem_wen
  );
  logic [7:0] pc, addr;
  enum logic [3:0] {LOAD,STORE,MOVE,ADD,SUB,MUL,JNZ}opcode;
  logic [ 3:0] i_rs1, i_rs2, i_rd, i_reg;
  logic [15:0] regs [16];
  logic [15:0] reg_1, reg_2;
  always_comb begin
    imem_addr  = pc;
    {addr       , i_reg, opcode} = imem_rdata;
    {i_rs2, i_rs1, i_rd , opcode} = imem_rdata;

    dmem_addr  = addr;
    dmem_wdata = regs[i_reg];
    dmem_wen   = opcode == STORE;

    reg_1      = regs[i_rs1];
    reg_2      = regs[i_rs2];
  end
  always_ff @(posedge clk)
    if (reset) begin
      pc   <= '0;
      for (int i = 0; i < 16; i++) regs[i] <= '0;
    end else begin
      pc   <= pc + 1'b1;
      case (opcode)
        LOAD: regs[i_reg] <= dmem_rdata;
        MOVE: regs[i_rd ] <= reg_1;
        ADD : regs[i_rd ] <= reg_1 + reg_2;
        SUB : regs[i_rd ] <= reg_1 - reg_2;
        MUL : regs[i_rd ] <= reg_1 * reg_2;
        JNZ : if (regs[i_reg] != '0) pc <= addr;
      endcase end
endmodule
```

CPU: Sum 0 to N

Sum the numbers 0 to 10

```
uint16_t mem[256];
mem[0] = 0;     // sum seed
mem[1] = 1;     // the constant one
mem[2] = 10;    // N
uint16_t sum = 0;
for (uint16_t i=mem[2]; i!=0; i-=mem[1])
    sum += i;
mem[4] = sum;  //55
```

```
0: R2_COUNT = *(2);
1: R1_ONE   = *(1);
2: R0_SUM   = *(0);
3: R0_SUM   = R0_SUM + R2_COUNT;
4: R2_COUNT = R2_COUNT - R1_ONE;
5: if (R2_COUNT!=0) goto 3;
6: *(4) = R0_SUM;
```

```
// Initialize N, constant one, and the sum.
imem.mem[0] = {8'h02,       4'h2, LOAD};  // r2 (counter) = N = 10
imem.mem[1] = {8'h01,       4'h1, LOAD};  // r1 (one)     = mem[1] = 1
imem.mem[2] = {8'h00,       4'h0, LOAD};  // r0 (sum)     = mem[0] = 0

// Add each counter value and count down to zero.
imem.mem[3] = {4'h2,  4'h0,  4'h0, ADD};   // r0 (sum) += r2 (counter)
imem.mem[4] = {4'h1,  4'h2,  4'h2, SUB};   // r2 (counter) -= r1 (one)
imem.mem[5] = {8'h03,       4'h2, JNZ};   // repeat while r2 (count)!=0

imem.mem[6] = {8'h04,       4'h0, STORE}; // mem[4] = r0 (sum)
```

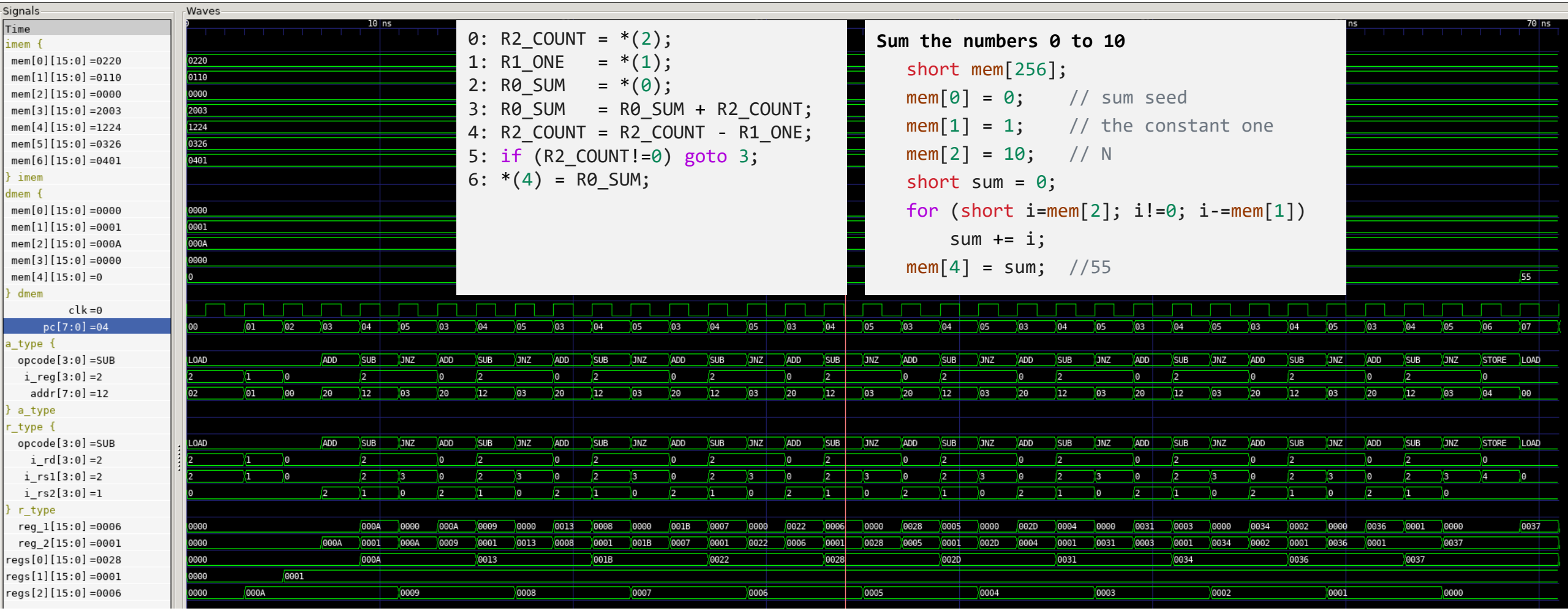
```
module cpu (
  input  logic        clk, reset,
  output logic [7 :0] pc,          addr,
  input  logic [15:0] instruction, read_data,
  output logic [15:0] write_data,
  output logic        dmem_wen
);
  enum logic [3:0] {NOP,LOAD,STORE,MOVE,ADD,SUB,MUL,JNZ} opcode;
  logic [ 3:0] i_reg_a, i_bus_b, i_bus_c;
  logic [15:0][15:0] regs;
  logic [15:0] bus_a, bus_b, bus_c, alu_out;
  logic reg_wen, jump;
  always_comb begin
    {addr,             i_reg_a, opcode} = instruction;
    {i_bus_c, i_bus_b, i_reg_a, opcode} = instruction;

    bus_a      = regs[i_reg_a];
    bus_b      = regs[i_bus_b];
    bus_c      = regs[i_bus_c];

    write_data = bus_a;
    dmem_wen   = opcode == STORE;
    jump       = (opcode == JNZ) && (bus_a != '0);

    alu_out    = '0;
    reg_wen    = 1'b1;
    case (opcode)
      LOAD   : alu_out = read_data;
      MOVE   : alu_out = bus_b;
      ADD    : alu_out = bus_b + bus_c;
      SUB    : alu_out = bus_b - bus_c;
      MUL    : alu_out = bus_b * bus_c;
      default: reg_wen = 1'b0;
    endcase
  always_ff @(posedge clk)
    if (reset) {pc, regs} <= '0;
    else begin
      pc <= jump ? addr : pc + 1;
      if (reg_wen) regs[i_reg_a] <= alu_out;
    end
endmodule
```

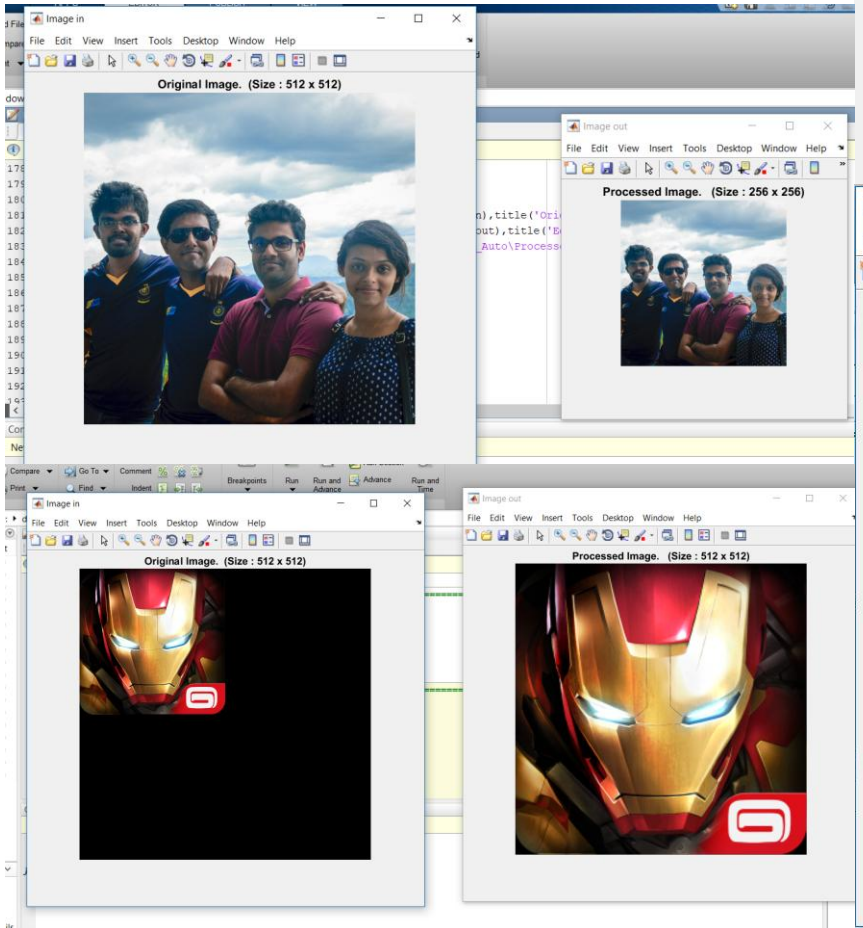
Waveform: Sum to N

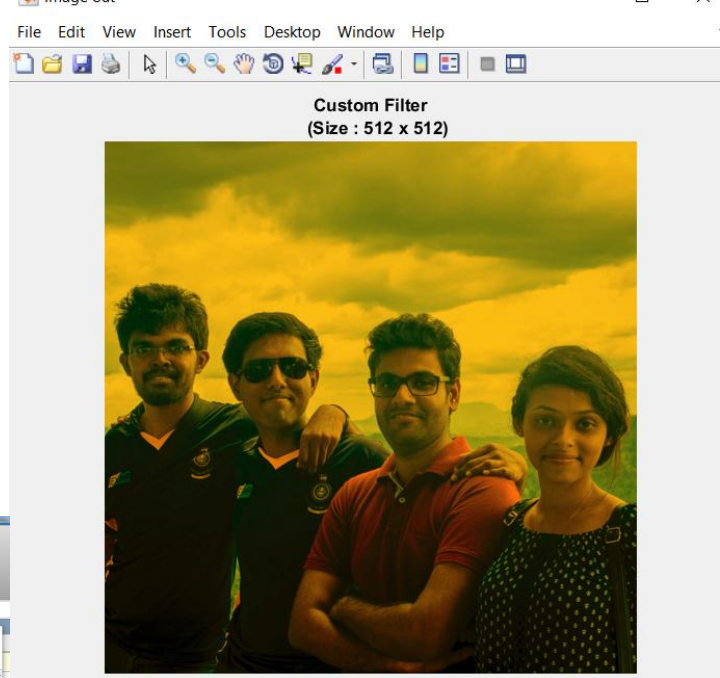


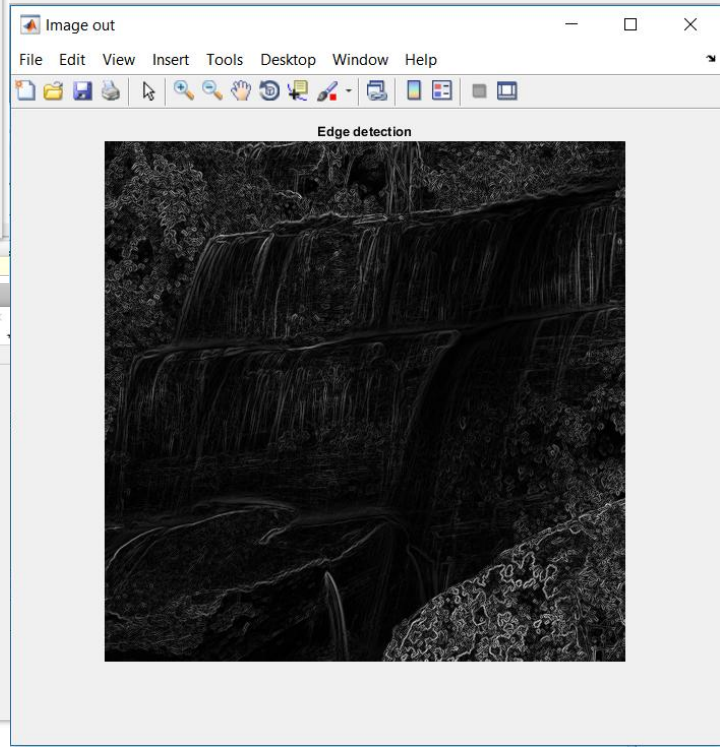
My First Design (2018)

- Task: downsample 512x512 image into 256x256
- Send data PC-FPGA-PC via UART

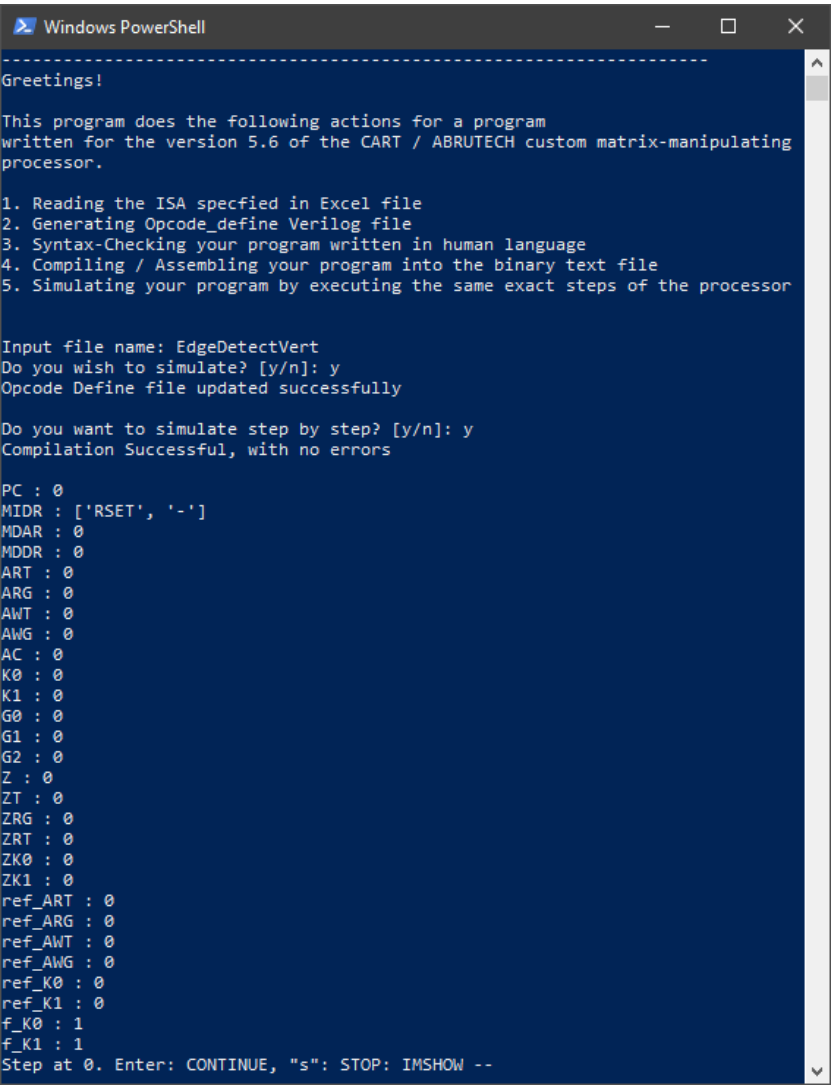
```
# Vertical Edge Detection
# 35 lines(bytes)
RSET
    [all]
        $row    LOAD: FROM_MAT
                COPY
                    [MDDR -> G0]
                COPY
                    [MDDR -> G0]
                INCR
                    [ARG]
                LOAD : FROM_MAT
                COPY
                    [MDDR -> AC, G0]
                SUBT : G2
                STAC : TO_MAT
                INCR
                    [AWG, ARG]
                $pixel  LOAD: FROM_MAT
                        COPY
                            [MDDR -> AC, G0]
                        SUBT : G2
                        STAC : TO_MAT
                        INCR
                            [AWG, ARG]
                JUMP: NZ_ARG
                    [pixel]
                COPY
                    [G0 -> AC]
                SUBT : G2
                STAC : TO_MAT
                INCR
                    [AWG, AWT, ART]
        JUMP: NZ_ART
            [row]
DECR
    [AWG,AWT]
```

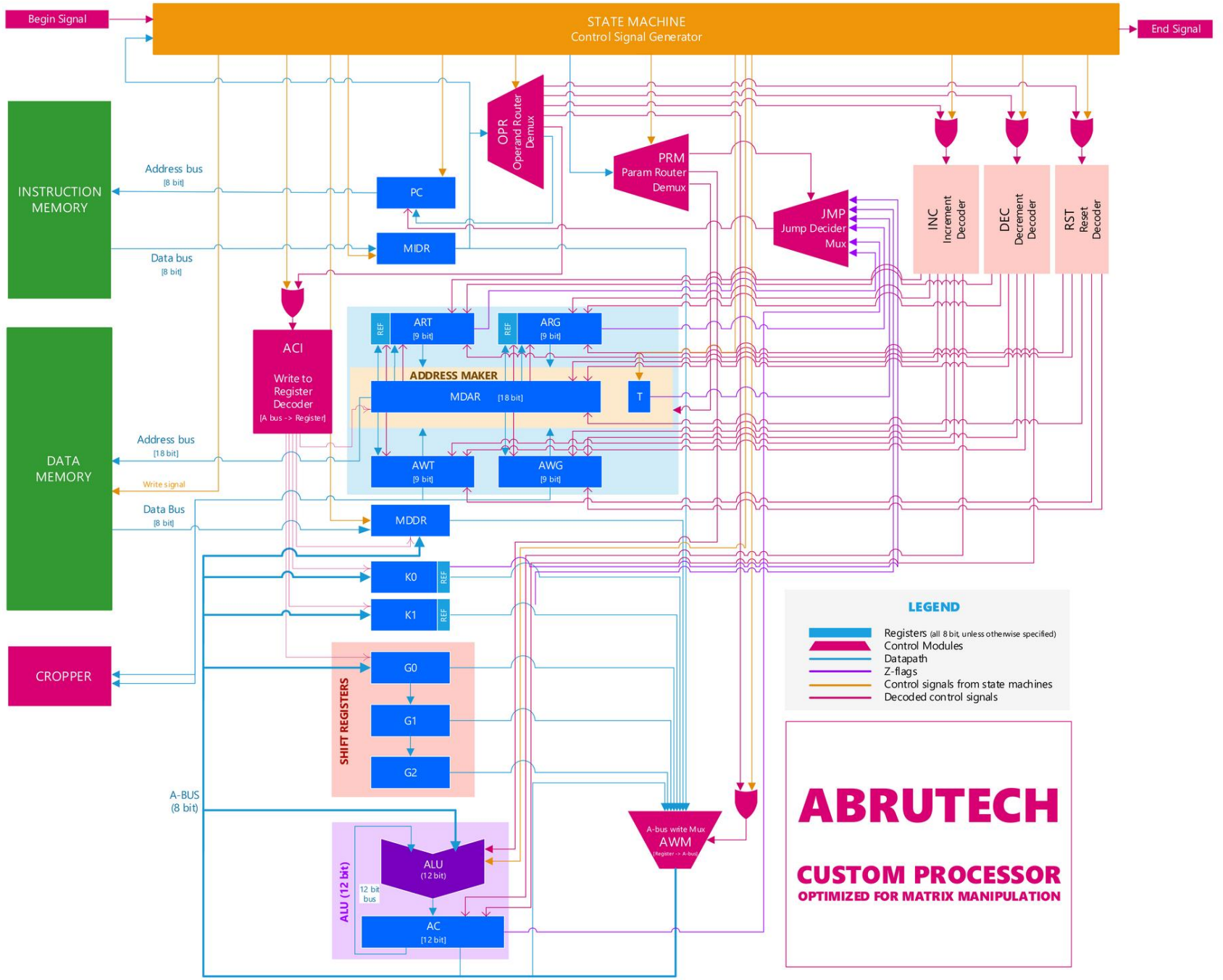






My First Design (2018)

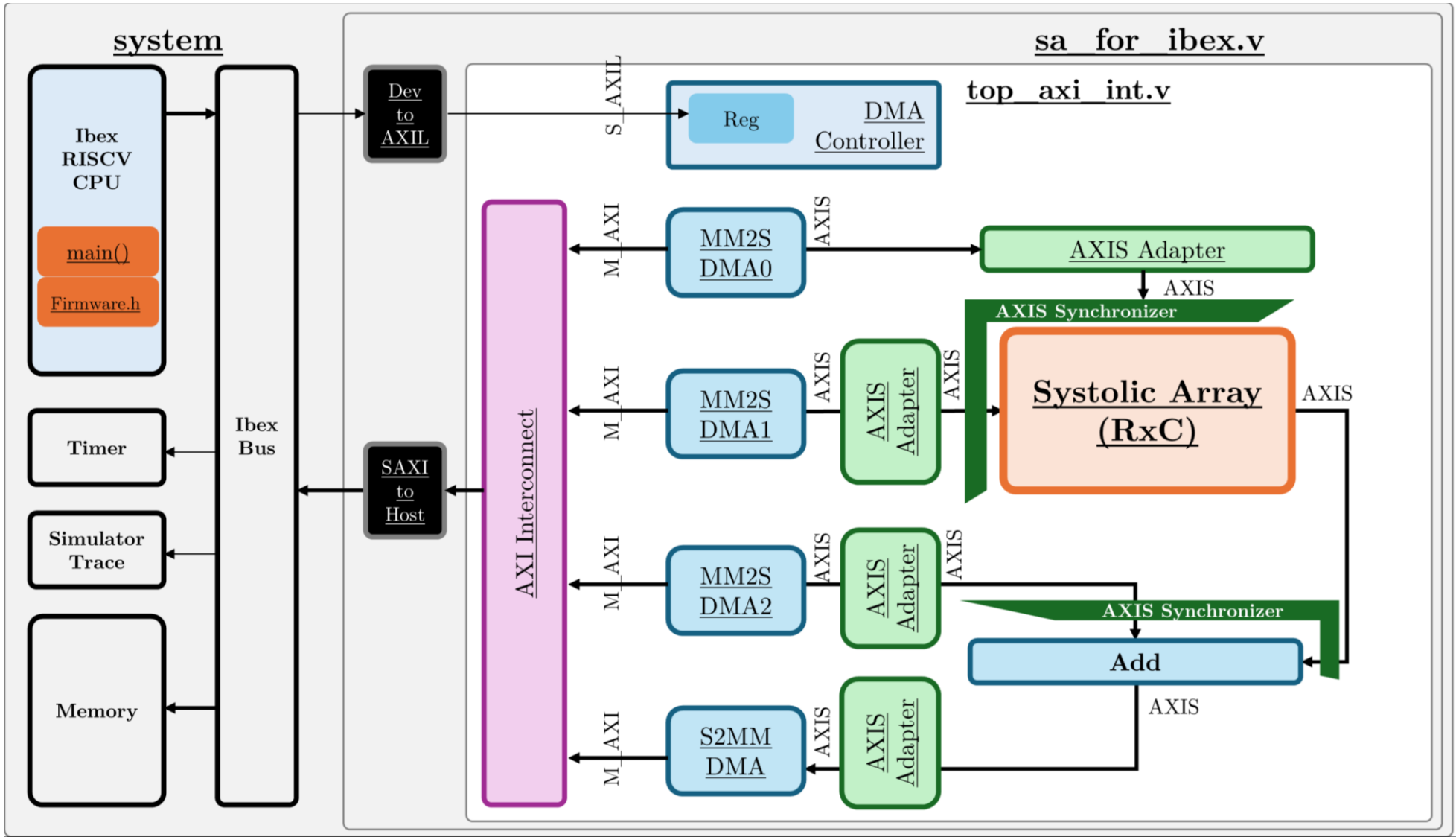




Your next steps?

You can use our System-on-chip infrastructure:

github.com/abarajithan11/axis-systolic-array

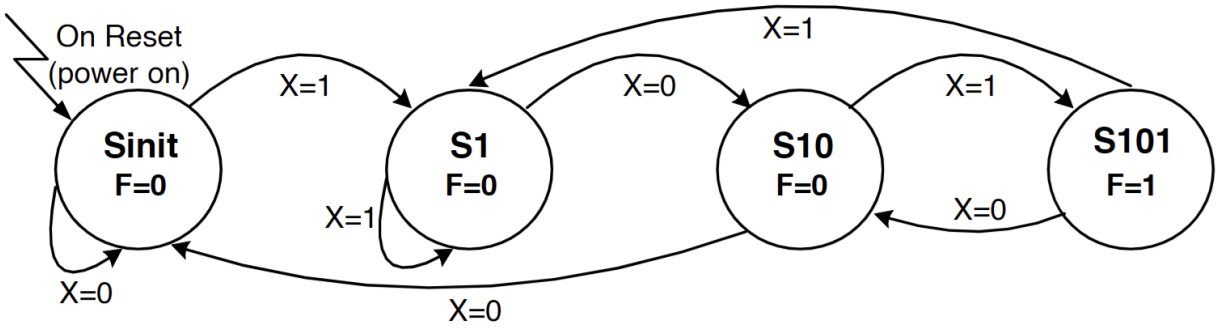


Projects

1. **Innovate FPGA Contest:** innovatefpga.com
2. AMD Design contest: openhw.eu/
3. DvCon: dvcontaiwan.org/call-for-design-verification-contest-2024/
4. **Green Machine** – Silver winning project by Pahan, Chathuni & Co [Link]
5. **Aba's projects** - aba-blog.xyz/tag/tech-projects
6. Projects for beginners: [thread]
7. Intermediate project ideas [thread][thread]
8. HDLbits: Problem set to learn Verilog: hdlbits.01xz.net/wiki/Problem_sets

Pattern Detector

"101" Sequence Detector should output F=1 when sequence 101 is found in consecutive order



State Diagram for "101" Sequence Detector

https://ee.usc.edu/~redekopp/ee209/slides/EE209Spiral1-7.pdf

```
module detector_101 (
  input  logic clk, rstn, x,
  output logic f
);

  typedef enum logic [1:0]
        {SINIT, S1, S10, S101} state_t;
  state_t state, next;

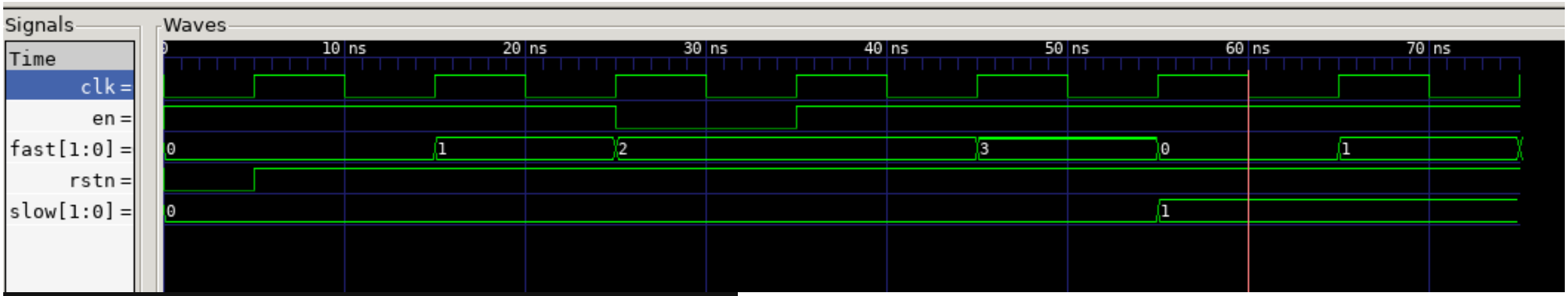
  always_ff @(posedge clk or negedge rstn)
    if (!rstn) state <= SINIT;
    else       state <= next;

  always_comb begin
    case (state)
      SINIT: next = x ? S1   : SINIT;
      S1:    next = x ? S1   : S10;
      S10:   next = x ? S101 : SINIT;
      S101:  next = x ? S1   : S10;
      default: next = SINIT;
    endcase
  end

  always_ff f = (state == S101);

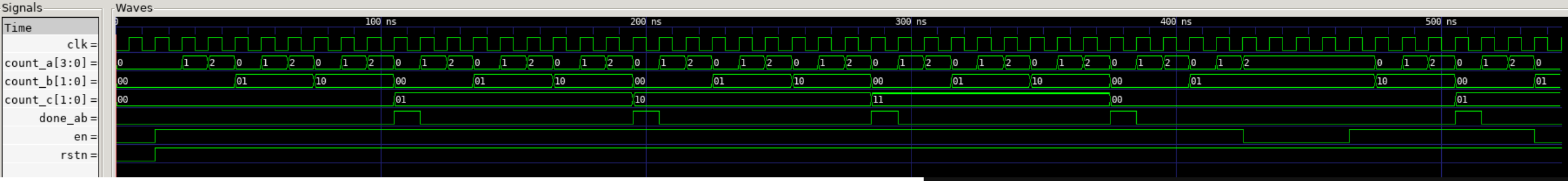
endmodule
```

Nested counter - Waveform




```
module nested_counter (
  input  logic clk, rstn, en,
  output logic [1:0] fast, slow);

  always_ff @(posedge clk)
    if (!rstn)
      {slow, fast} <= '0;
    else if (en) begin
      fast <= fast + 1'b1;
      if (fast == 2'd3)
        slow <= slow + 1'b1;
    end
endmodule
```




```
module triple_loop (
  input  logic clk, rstn, en,
  output logic [3:0] count_a,
  output logic [1:0] count_b,
  output logic [1:0] count_c,
  output logic       done_ab
);
  always_ff @(posedge clk) begin
    if (!rstn) {count_a, count_b, count_c, done_ab} <= '0;
    else begin
      done_ab <= 1'b0;
      if (en) begin
        if (count_a == 4'd2) begin
          count_a <= '0;
          if (count_b == 2'd2) begin
            count_b <= '0;
            count_c <= count_c + 2'd1;
            done_ab <= 1'b1;
          end else begin
            count_b <= count_b + 2'd1;
          end
        end else begin
          count_a <= count_a + 4'd1;
        end
      end
    end
  end
endmodule
```

Nested Counter - Waveform

Final Feedback

1. Have you filled the evaluation form?
2. How confident do you feel in SystemVerilog, in writing small designs?
3. Are you interested in a career in Digital Design?