Unit 4. Design of a (Yet Another) Simple Academic Computer

Computer Structure
E.T.S.I. Informática
Universidad de Sevilla

Jorge Juan-Chico <jjchico@dte.us.es> 2021-24

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Some images and other objects are used here according to their license. Follow the reference to the original work for details. Images not referenced are either original work of the author or are in the public domain.

Contents

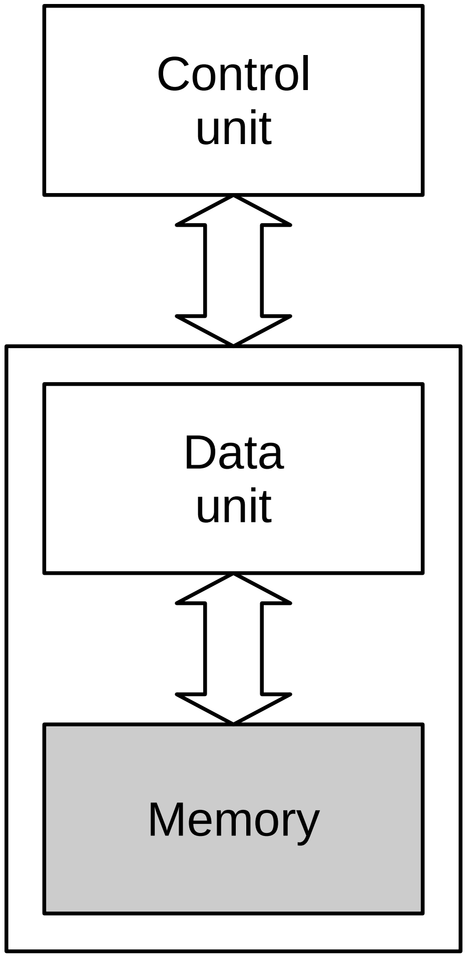
Computers: What, Why and How

- What is a computer?
  - A machine that can do tasks according to a “program”.
  - An algorithm can be expressed in a “program”.
  - They are backed up by strong theoretical results thanks to Alan Turing and other people.
    - Useful computers are Turing-complete.
- Why are computers so useful?
  - Many problems can be solved by executing an algorithm.
  - The program in modern computers can be easily changed.
  - The computer has become the universal machine.
- How are computers designed and built
  - You will see a simple example in this unit!

Computer architecture design basics

- Instruction Set Architecture
  - Programmers view
  - Links software and hardware (contract)
  - Finality of the computer
  - Human vs compiler friendly
  - Support high-level languages
- Microarchitecture
  - What elements you need to implement the ISA?
  - How to connect them?
- System design
  - Digital system designers wanted!

Stored program computers
Harvard vs Von Neumann architectures

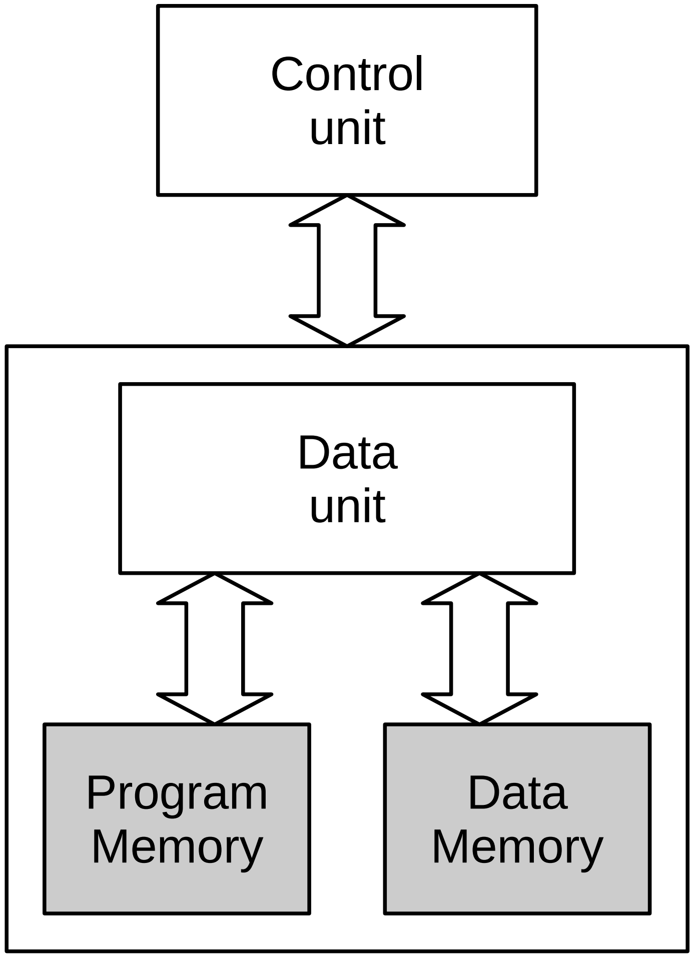


Von Neumann

- Simpler design.
- Von Neumann's bottleneck.
- Used in most standard computers.

Hardvard

- No Von Neumann bottleneck.
- Better program memory protection (can be read-only).
- Used in embedded systems and microcontrollers.



RISC vs CISC

- Complex Instruction Set Computer (CISC)
  - Powerful complex instructions that may take many cycles to execute.
  - Saves access to memory to fetch new instructions (good).
  - Difficult to execute various instructions in parallel (bad).
  - Complex system design and resource intensive (bad).
- Reduced Instruction Set Computer (RISC)
  - Simple instructions that may execute very quickly (one clock cycle most of the time).
  - Needs more instructions to perform the same task (bad).
  - More instructions consumes more memory (bad).
  - Easy to execute many instructions in parallel through instruction pipelining (good).
  - Simpler system design (good).

YASAC
Yet Another Simple Academic Computer

- Academic computer with some realistic characteristics.
- Simple enough to be understood by first-year students.
- Powerful enough to introduce all main computer fundamentals.
- Expandable: it is not a complete and full-featured design.
- Microcontroller-like functionality.
  - ISA inspired by Atmel's AVR family of microcontrollers (the core of the Arduino UNO board and others).
- Easy to model in Verilog.
- Easy to implement in FPGA chips.
- Harvard architecture.
- RISC.

The YASAC is a slightly re-designed version of the CS2010 and CS simple computers used in various courses of the Electronics Technology Department (Universidad de Sevilla), developed by professors and students of the Department along the years.

YASAC development overview

- Stage by stage development.
- Each stage has:
  - A specification: instructions and functionality to implement.
  - Design documentation: drawings, micro-operations, ASM charts, etc.
  - A Verilog model and test bench.
  - An FPGA implementation.
- Each stage builds on the previous one.

YASAC development overview. Stages

- Stage 1
  - Minimal computer. Only program memory. Basic I/O.
- Stage 2
  - Data memory and memory addressing. Memory-mapped I/O.
- Stage 3
  - Jumping and branching instructions.
- Stage 4
  - Logic and status register instructions
- Stage 5
  - Stack and subroutines.
- Stage 6
  - Interrupts.
- Stage 7
  - Writable program memory and boot loader.

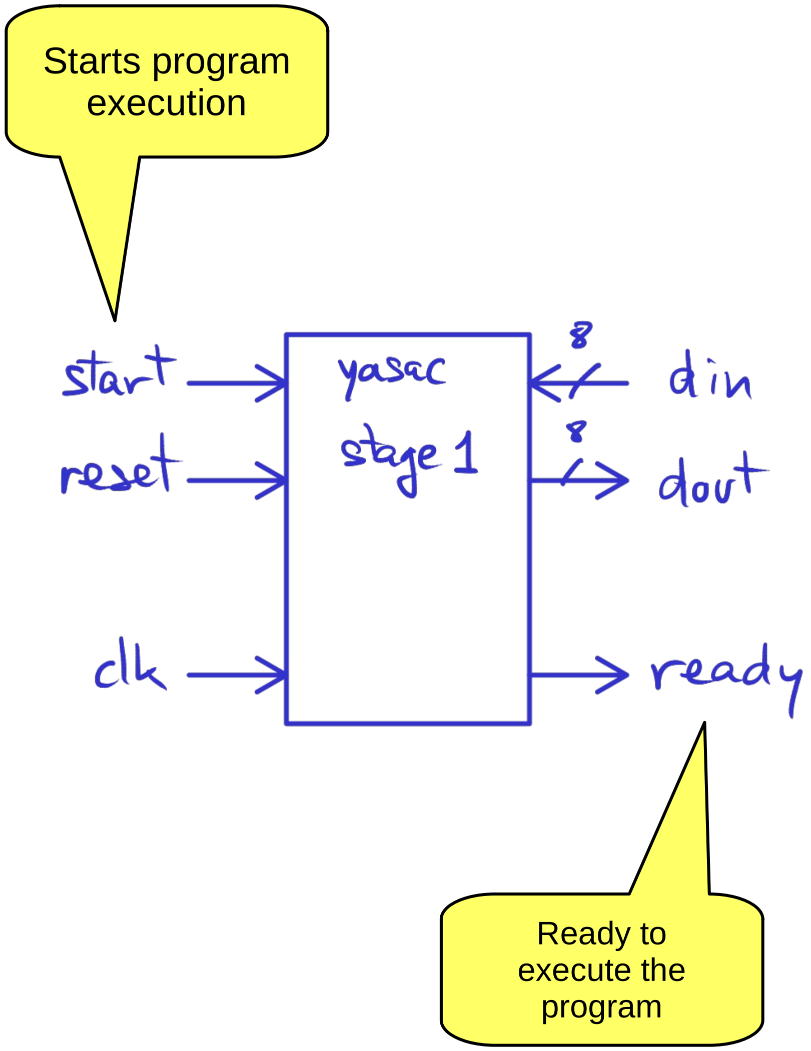
We will cover stages 1 to 5.

Improvements can be done at every stage (assignments).

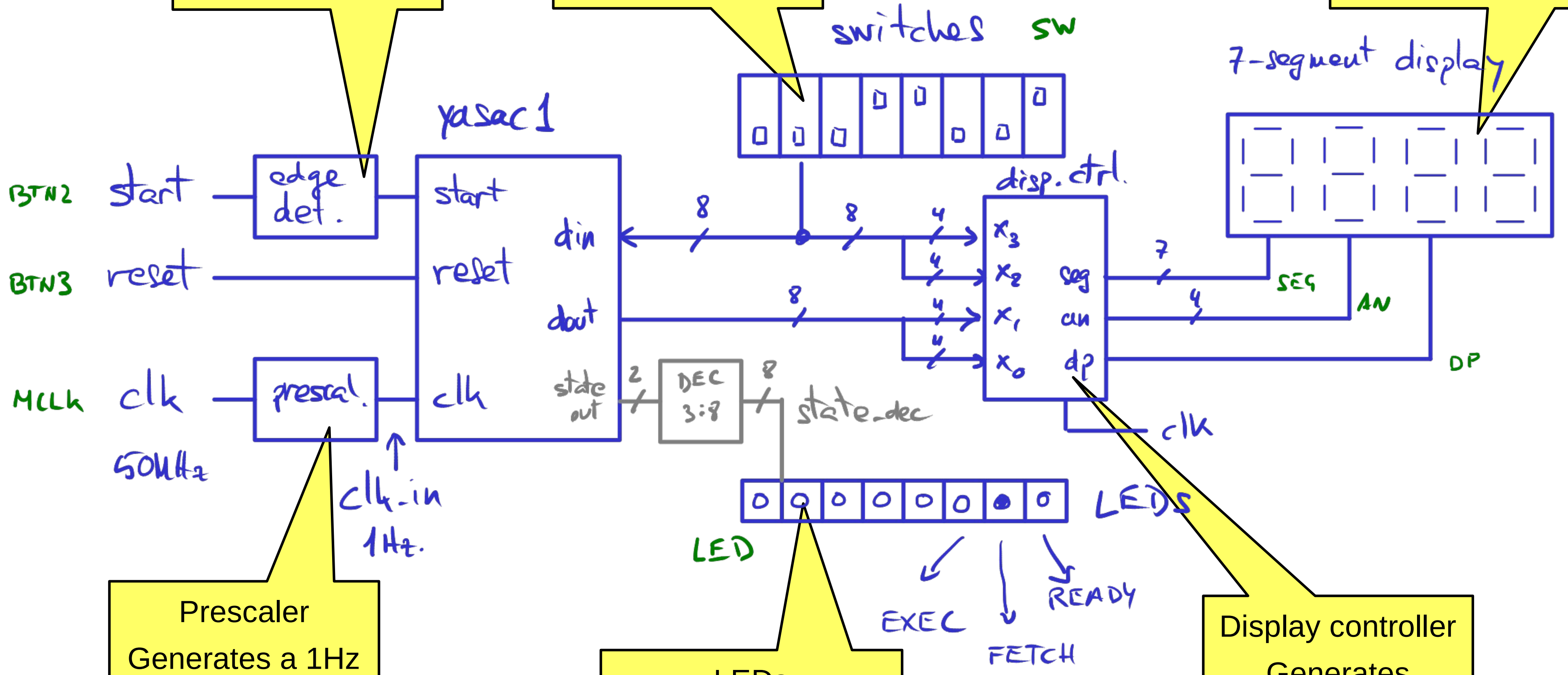
Every stage can be implemented in an FPGA board (labs).

YASAC Stage 1. General specification

- 8-bit data unit and registers
- 8 general purpose registers
  - R0 to R7
- One 8-bit input port and one 8-bit output port
  - din: input port (mapped to R7)
  - dout: output port (mapped to R6)
- Program memory: 256x16
  - ROM memory defined at design time.
- 16-bit instructions



YASAC Stage 1 Board implementation



Edge detector

Converts input into a single pulse

Switches

Provide input data

7 segment display

Displays input and output data

Prescaler

Generates a 1Hz clock (for testing)

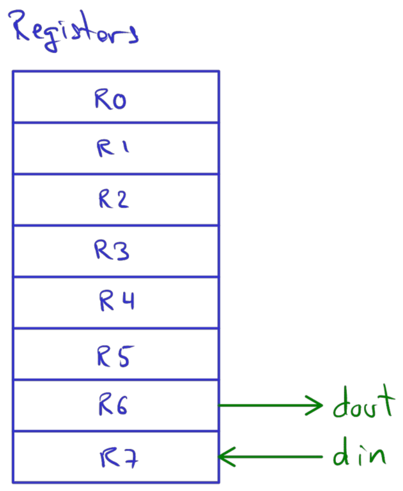
LEDs

Show current state (for testing)

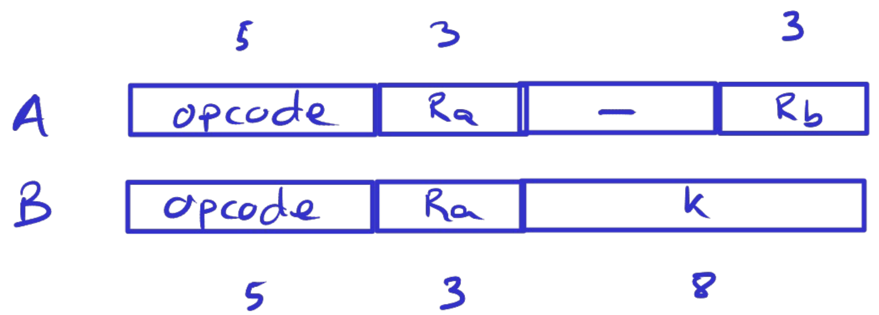
Display controller

Generates display signals

YASAC Stage 1
Instruction Set Architecture



Instruction format

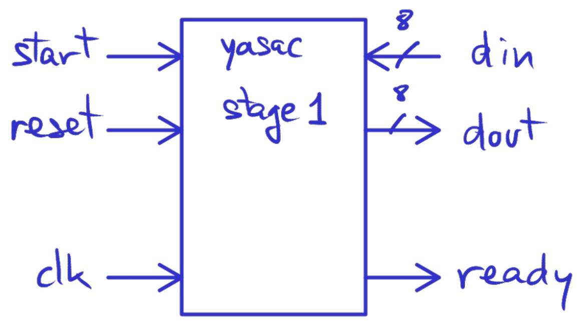


Mnemonic	code
LDI	00001 (1)
MOV	00010 (2)
ADD	00011 (3)
SUB	00100 (4)
STOP	00101 (5)

Instructions

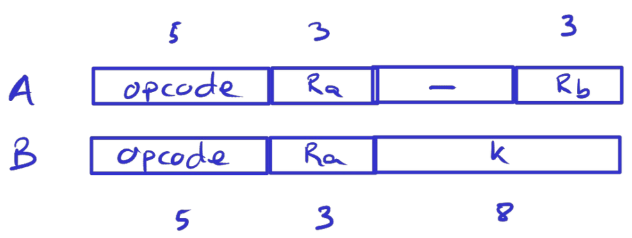
```
LDI Ra, k      ; Ra ← k
MOV Ra, Rb     ; Ra ← Rb
ADD Ra, Rb     ; Ra ← Ra + Rb
SUB Ra, Rb     ; Ra ← Ra - Rb
STOP
```

YASAC Stage 1 Sample program



Mnemonic	code
LDI	00001 (1)
MOV	00010 (2)
ADD	00011 (3)
SUB	00100 (4)
STOP	00101 (5)

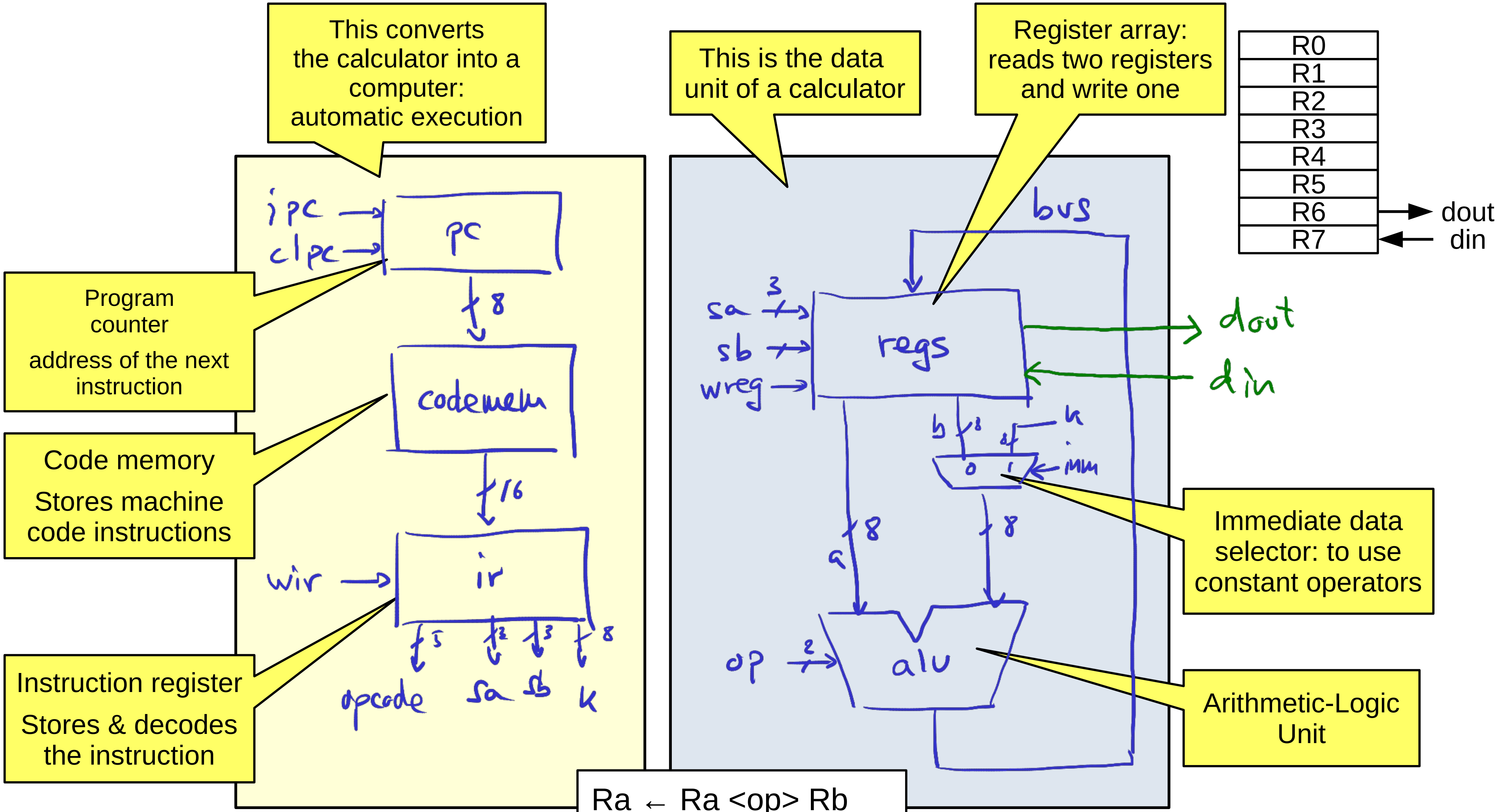
Instruction format



Assembly code	Machine code	RTL
MOV R1,R7	00010 001 00000111	R1 ← din
MOV R0,R1	00010 000 00000001	R0 ← R1
ADD R0,R1	00011 000 00000001	R0 ← R0 + R1
LDI R2,5	00001 010 00000101	R2 ← 5
SUB R0,R2	00100 000 00000010	R0 ← R0 – R2
MOV R6,R0	00010 110 00000000	R6 ← R0
STOP	00101 000 00000000	(none)

What is the value in dout at the end of the program if din=6?

YASAC Stage 1 Data unit



This converts the calculator into a computer: automatic execution

This is the data unit of a calculator

Register array: reads two registers and write one

Program counter

address of the next instruction

Code memory

Stores machine code instructions

Instruction register

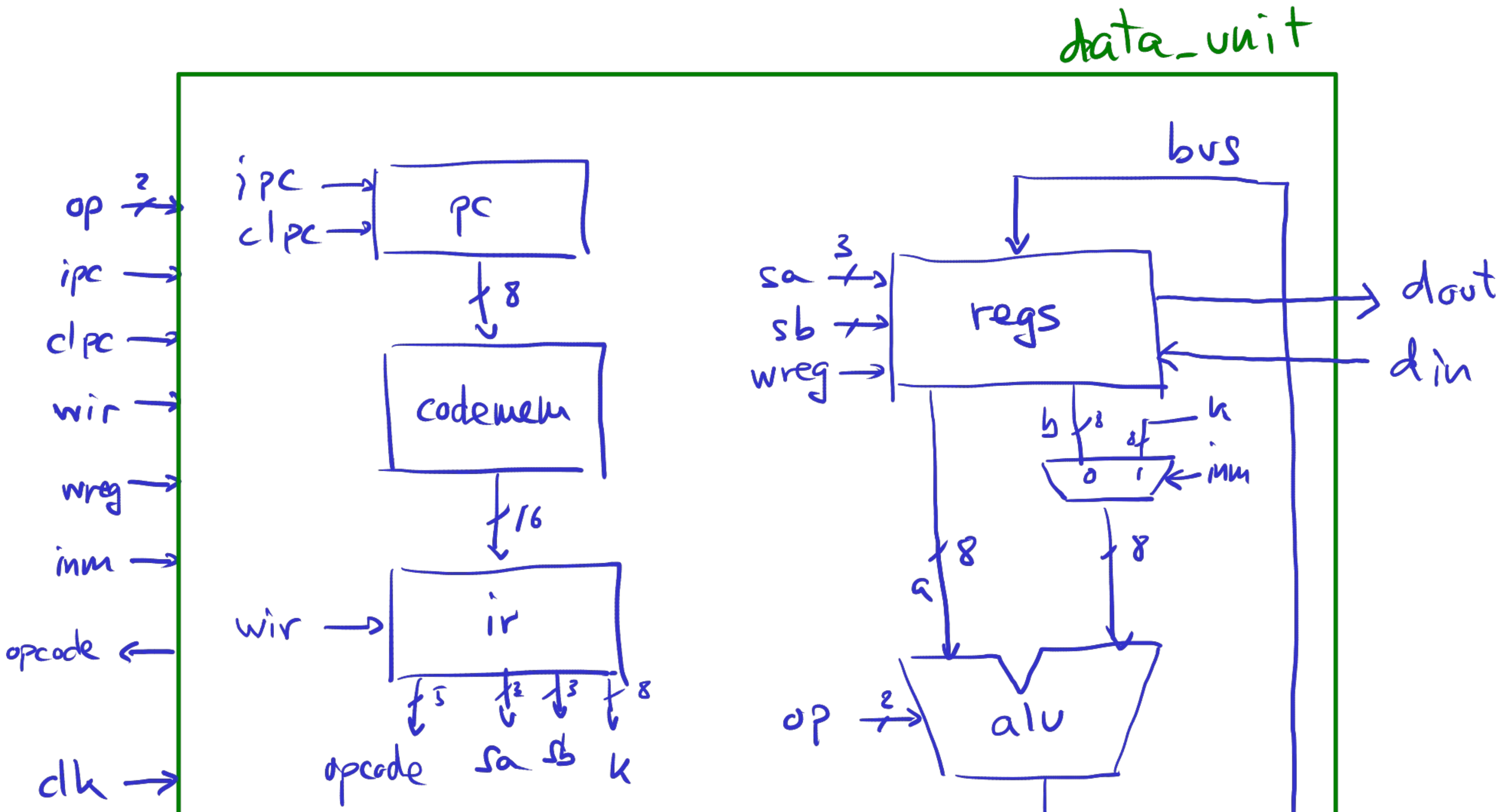
Stores & decodes the instruction

Immediate data selector: to use constant operators

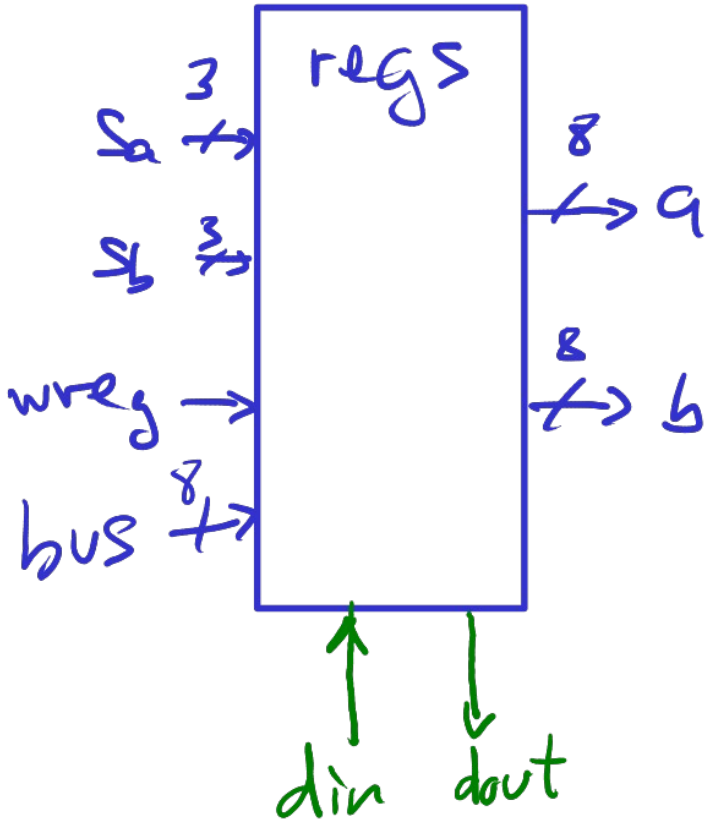
Arithmetic-Logic Unit

Ra ← Ra <op> Rb
inm: Ra ← Ra <op> k

YASAC Stage 1
Data unit



YASAC Stage 1
Register array



a = regs[sa]

b = regs[sb]

wreg : regs[sa] ← bus

$\overline{\text{wreg}}$: regs[7] ← din

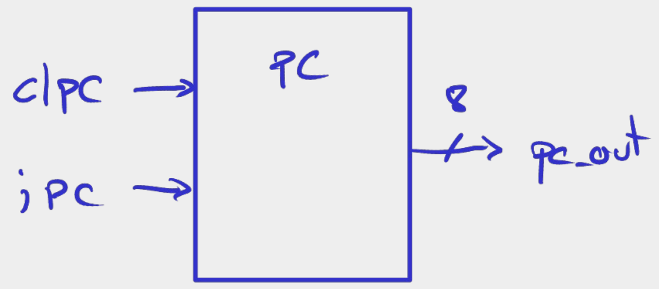
dout = regs[6]

Quick exercise

a) Design the register array using combinational and sequential subsystems.

b) Write a Verilog code fragment that models the register array.

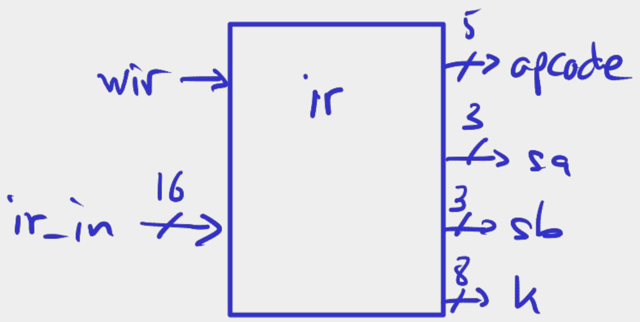
YASAC Stage 1
Program counter and Instruction register



clpc : $pc \leftarrow \emptyset$

ipc : $pc \leftarrow pc+1$

clpc	ipc	pc ←
0	0	pc (INH.)
1	—	∅
0	1	pc+1



wir : $ir \leftarrow ir\_in$

wir	ir ←
0	ir (INH)
1	ir_in

opcode = ir[15:11]

sa = ir[10:8]

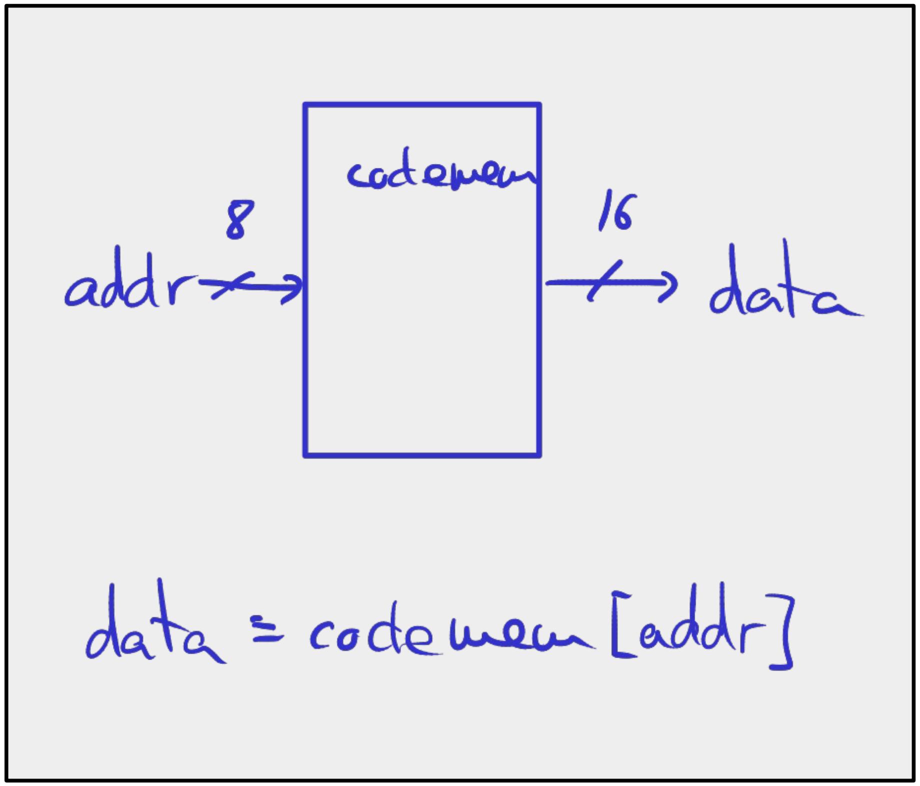
sb = ir[2:0]

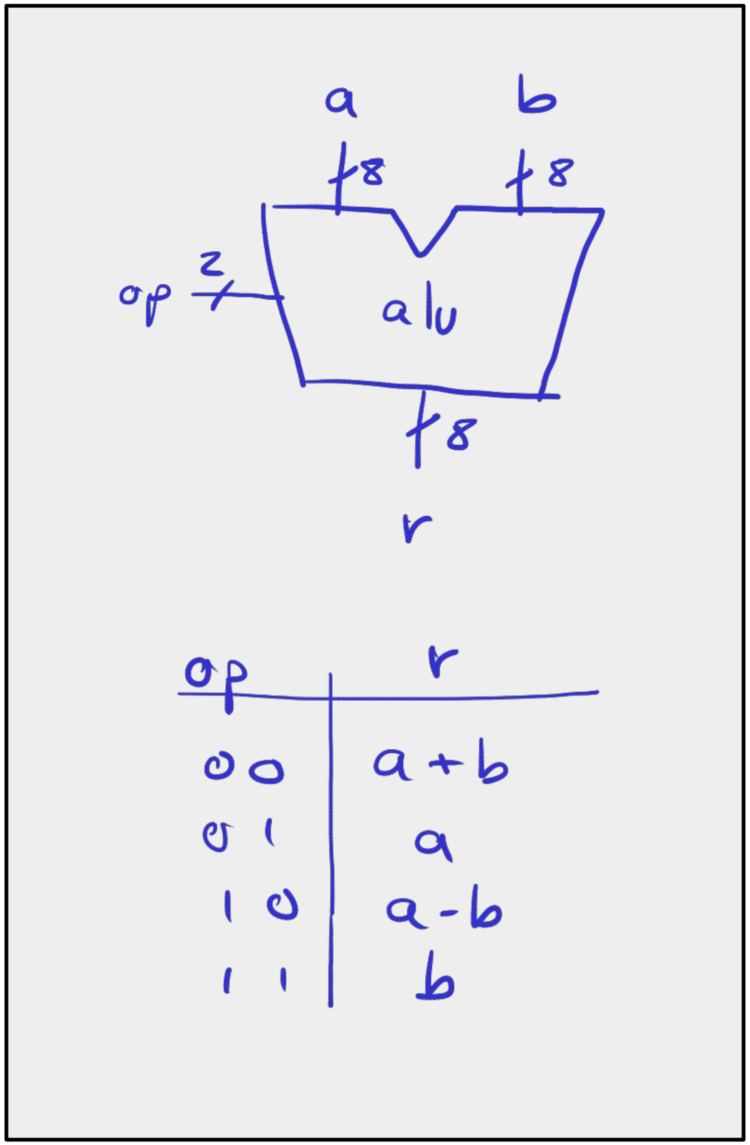
k = ir[7:0]

Quick exercise

Write a Verilog code fragment that models the registers.

YASAC Stage 1
Code memory and ALU



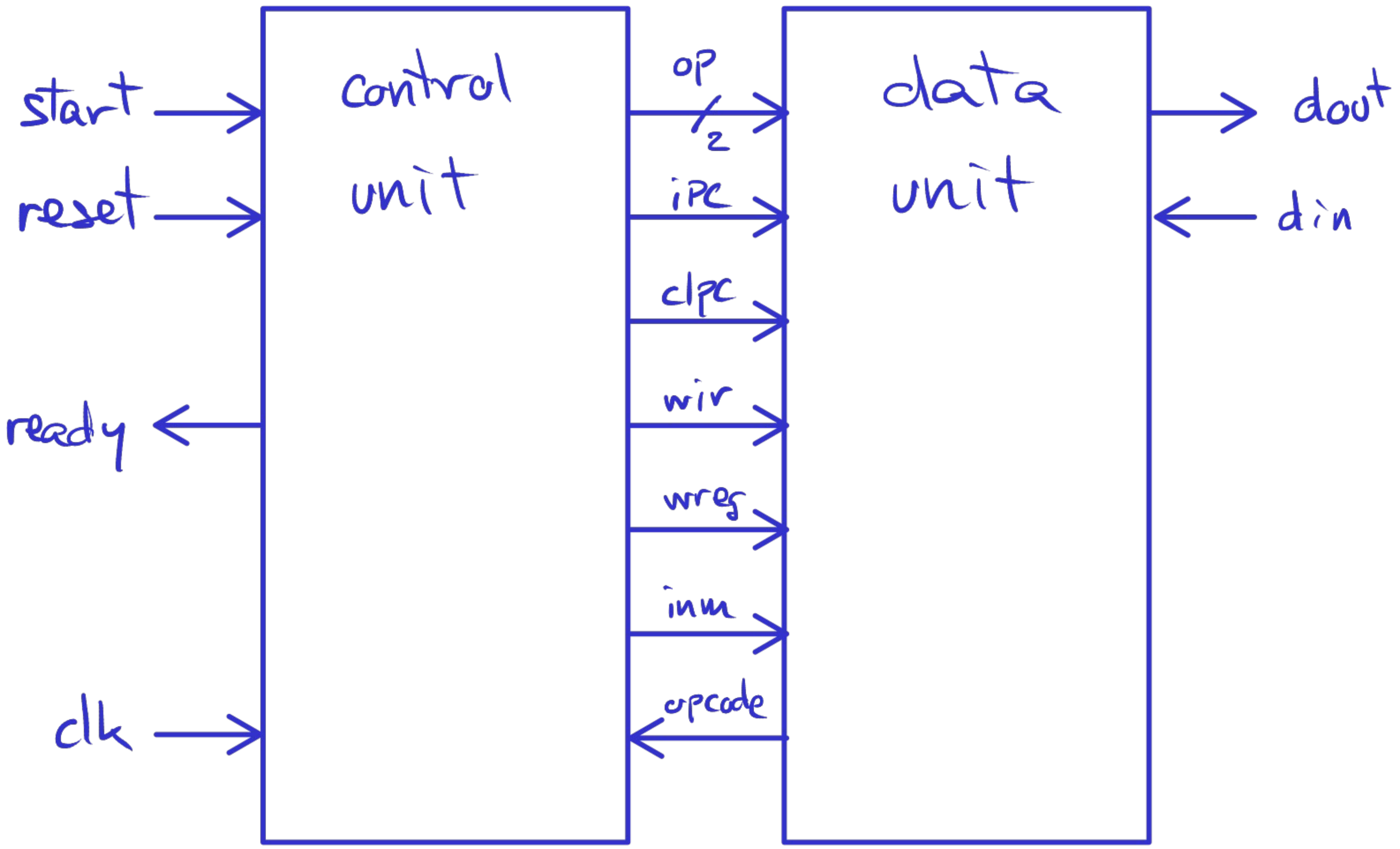


op	r
00	a+b
01	a
10	a-b
11	b

Quick exercise

Model the code memory in Verilog with a “case” statement. The memory has the contents of the sample program in a previous slide.

YASAC Stage 1
Control unit



YASAC Stage 1 Control unit. Initial analysis

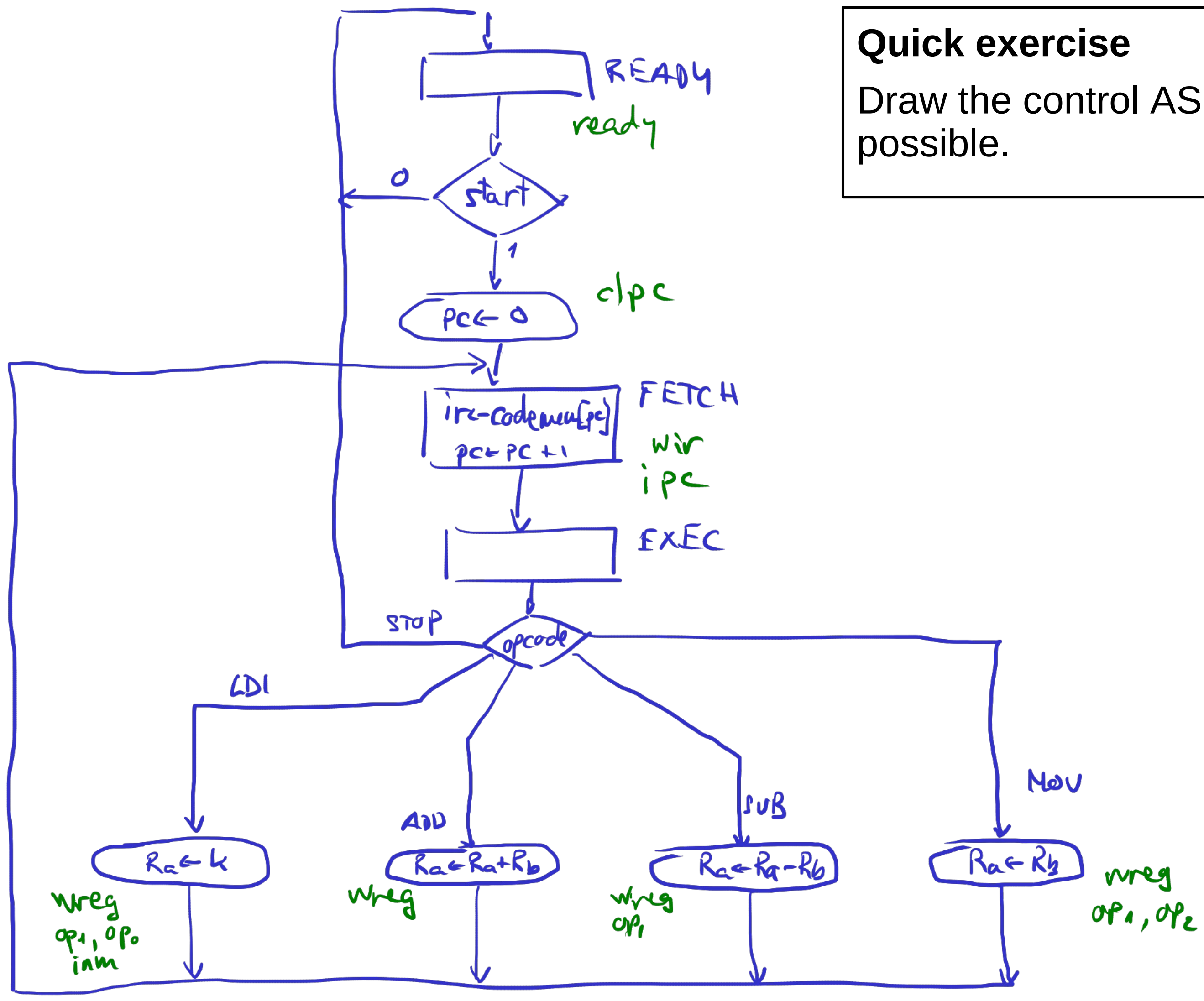
- The initial state waits for “start” activation. Then, the program counter is initialized:
  - pc ← 0 (clpc)
- For each instruction, the control unit must FETCH the instruction for memory and then EXECUTE the instruction by doing micro-operations on the data unit.
- FETCH: read the instruction code into the instruction register and increment the program counter:
  - ir ← code_mem(pc); pc ← pc + 1 (wir, ipc)
- EXECUTION
  - Depends on the “opcode”.
  - All the instructions in this stage can be executed in one clock cycle.
  - After executing and instruction, go back to FETCH the next one.

YASAC Stage 1
Control unit. Execution macro-operations

ADD :	$Ra \leftarrow Ra + Rb$	wreg ; op = 00
SUB :	$Ra \leftarrow Ra - Rb$	wreg ; op = 10
MOV :	$Ra \leftarrow Rb$	wreg ; op = 11
LDI :	$Ra \leftarrow k$	wreg ; inm ; op = 11

STOP : $\rightarrow$ Next state: READY

YASAC Stage 1
Control unit. ASM chart



Quick exercise

Draw the control ASM chart. Simplify things if possible.

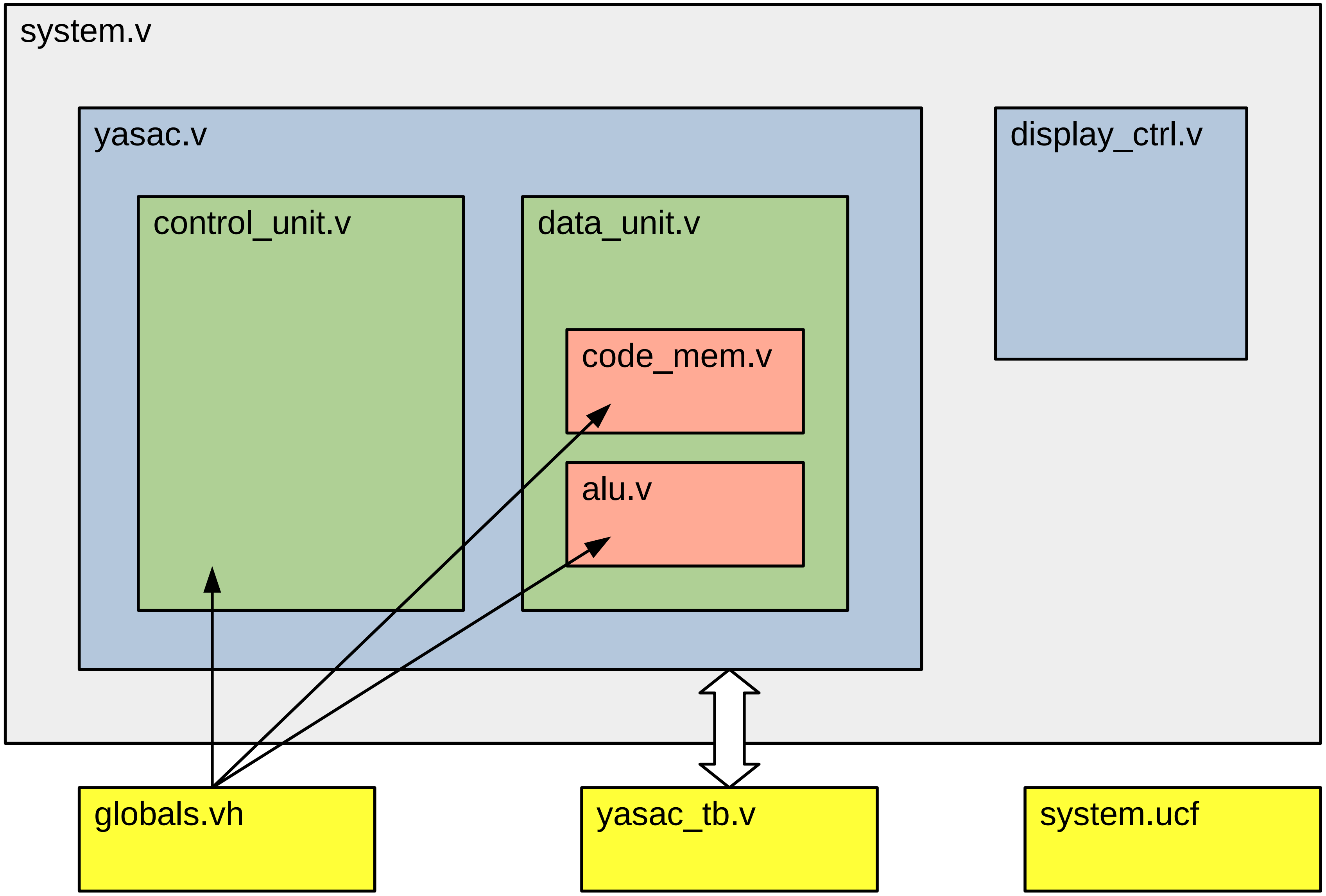
YASAC Stage 1
Control unit. States and control table

	READY	FETCH	EXEC
LDI Ra, k	ready start: clpc	wir, ipc	op=11, wreg, inm →FETCH
MOV Ra, Rb			op=11, wreg →FETCH
ADD Ra, Rb			op=00, wreg →FETCH
SUB Ra, Rb			op=10, wreg →FETCH
STOP			→READY

The table represents the control signals to activate at every execution step depending on the instruction (opcode) to execute.

It is a convenient way to organize the information about the control unit that simplifies HDL coding.

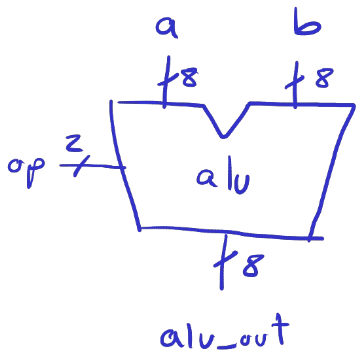
YASAC Stage 1
Verilog coding



YASAC Stage 1 Verilog coding globals.vh

```
//// Assembly operation codes
`define LDI     5'd1
`define MOV     5'd2
`define ADD     5'd3
`define SUB     5'd4
`define STOP    5'd5

//// Registers
`define R0      3'd0
`define R1      3'd1
`define R2      3'd2
`define R3      3'd3
`define R4      3'd4
`define R5      3'd5
`define R6      3'd6
`define R7      3'd7

//// ALU operation codes
`define ALU_ADD  2'd0
`define ALU_TRA  2'd1
`define ALU_SUB  2'd2
`define ALU_TRB  2'd3
```

YASAC Stage 1 Verilog coding alu.v



op	alu_out=
00	a+b
01	a
10	a-b
11	b

```
module alu (
    input wire [7:0] a,
    input wire [7:0] b,
    input wire [1:0] op,
    output reg [7:0] r
);
```

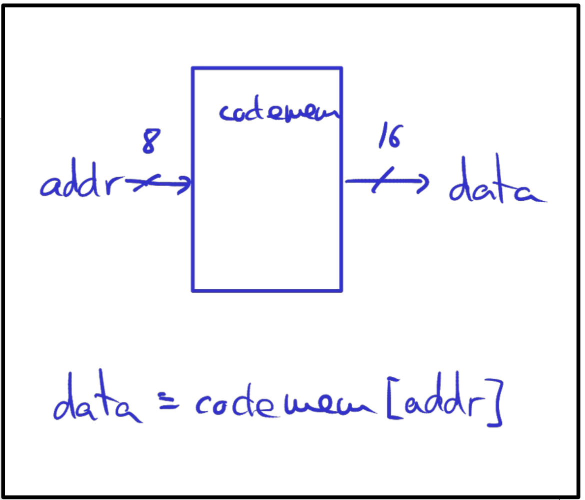
```
    always @* begin
        case(op)
        `ALU_ADD: begin
            r = a + b;
        end
        `ALU_SUB: begin
            r = a - b;
        end
        `ALU_TRA: begin
            r = a;
        end
        `ALU_TRB: begin
            r = b;
        end
        default:
            r = 'bx;
        endcase
    end
endmodule
```

YASAC Stage 1 Verilog coding code_mem.v

```
// Assembly op. codes
`define LDI     5'd1
`define MOV     5'd2
`define ADD     5'd3
`define SUB     5'd4
`define STOP    5'd5

// Registers
`define R0      3'd0
`define R1      3'd1
`define R2      3'd2
`define R3      3'd3
`define R4      3'd4
`define R5      3'd5
`define R6      3'd6
`define R7      3'd7

// ALU op. codes
`define ALU_ADD  2'd0
`define ALU_TRA  2'd1
`define ALU_SUB  2'd2
```

```
MOV R1,R7
MOV R0,R1
ADD R0,R1
LDI R2,5
SUB R0,R2
MOV R6,R0
STOP
```

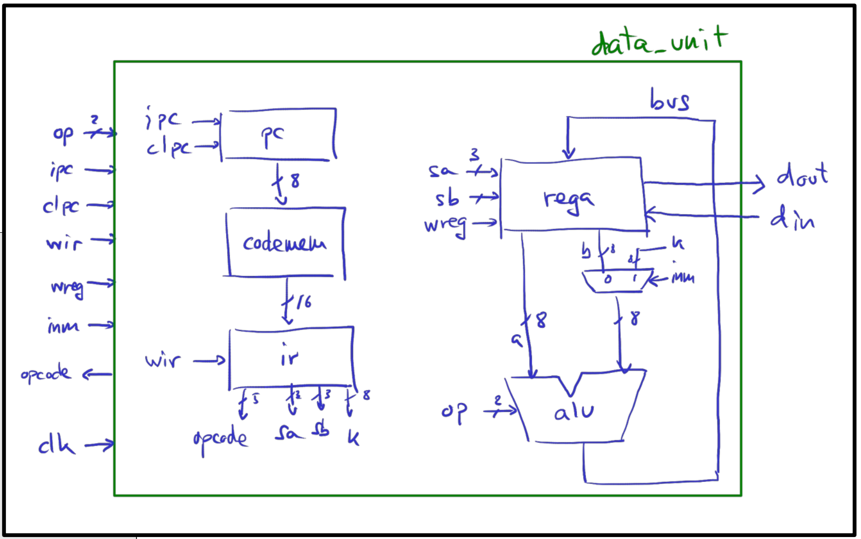
```
module code_mem (
    input wire [7:0] addr,
    output wire [15:0] data
);

    reg [15:0] code[0:255];
    integer i;

    assign data = code[addr];

    initial begin
        for (i=0; i<256; i=i+1)       // Initialization
            code[i] = 16'h0000;

        // Code memory contents (program)
        code['h0] = {`MOV, `R1, 5'd0, `R7};     // format A
        code['h1] = {`MOV, `R0, 5'd0, `R1};
        code['h2] = {`ADD, `R0, 5'd0, `R1};
        code['h3] = {`LDI, `R2, 8'h05};         // format B
        code['h4] = {`SUB, `R0, 5'd0, `R2};
        code['h5] = {`MOV, `R6, 5'd0, `R0};
        code['h6] = {`STOP, 11'd0};

    end
endmodule
```



YASAC Stage 1 Verilog coding data_unit.v

```verilog
module data_unit (
    input wire clk,
    input wire [1:0] op,
    input wire ipc,
    input wire clpc,
    input wire wir,
    input wire wreg,
    input wire inm,
    output wire [4:0] opcode,
    input wire [7:0] din,
    output wire [7:0] dout
);

    reg [7:0] pc;
    reg [15:0] ir;
    reg [7:0] regs [0:7];

    //// Internal signals

    wire [15:0] inst;
    wire [2:0] sa, sb;
    wire [7:0] k;
    wire [7:0] rega, regb;
    wire [7:0] alu_b;
    wire [7:0] bus;
```

```verilog
    //// PC register
    always @(posedge clk)
        if (clpc)
            pc <= 'b0;
        else if (ipc)
            pc <= pc + 1;

    //// IR register
    always @(posedge clk)
        if (wir)
            ir <= inst;
    assign opcode = ir[15:11];
    assign sa = ir[10:8];
    assign sb = ir[2:0];
    assign k = ir[7:0];

    //// Register array
    always @(posedge clk)
        if(wreg)
            regs[sa] <= bus;
        else
            regs[7] <= din;
    assign rega = regs[sa];
    assign regb = regs[sb];
    assign dout = regs[6];
```

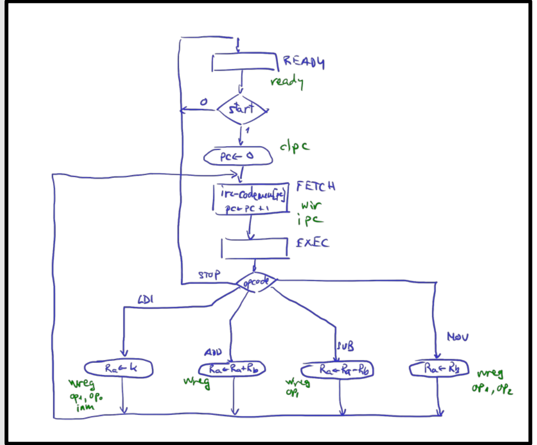


```verilog
    //// Code memory
    code_mem code_mem (
        .addr(pc),
        .data(inst)
    );

    //// ALU
    assign alu_b = inm ? k : regb;
    alu alu (
        .a(rega),
        .b(alu_b),
        .op(op),
        .r(bus)
    );
endmodule
```

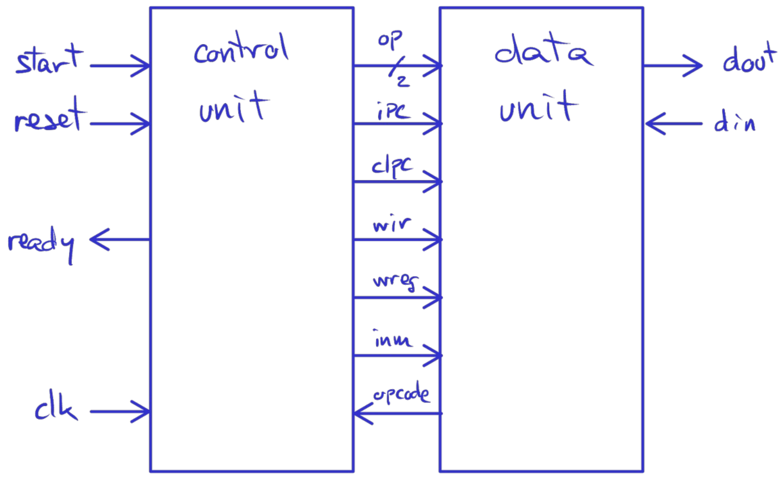
YASAC Stage 1 Verilog coding control_unit.v

```verilog
`include "globals.vh"

module control_unit (
    // External signals
    input wire clk,   // clock (rising edge)
    input wire reset, // reset (synchronous)
    input wire start, // start operation
    output reg ready, // ready output indicator

    // Data unit signals
    input wire [4:0] opcode,
    output reg [1:0] op, // ALU operation code
    output reg ipc,      // PC increment
    output reg clpc,     // PC clear
    output reg wir,      // write IR
    output reg wreg,     // write register array
    output reg inm,      // use inmediate value
    // FSM state output for testing
    output wire [1:0] state_out
);

    // Route state signal for testing
    assign state_out = state;

    // State definition
    localparam [1:0] READY = 0,
                     FETCH = 1,
                     EXEC  = 2;

    // State variables
    reg [1:0] state, next_state;

    // State change process
    always @(posedge clk)
        if (reset == 1'b1)
            state <= READY;
        else
            state <= next_state;
```

```verilog
    // Next state and output process
    always @* begin
        // Default output values
        ready = 1'b0; op = 'b0; ipc = 1'b0; clpc = 1'b0;
        wir = 1'b0; wreg = 1'b0; inm = 1'b0;
        next_state = 'bx;
        case (state)
        READY: begin
            ready = 1'b1;
            if (start) begin
                clpc = 1'b1;
                next_state = FETCH;
            end else begin
                next_state = READY;
            end
        end
        FETCH: begin
            wir = 1'b1;
            ipc = 1'b1;
            next_state = EXEC;
        end
        EXEC: begin
            next_state = FETCH; // except if STOP
            case(opcode)
            `LDI: begin
                op = `ALU_TRB;
                wreg = 1'b1; inm = 1'b1;
            end
            [...]
            default:        // including STOP
                next_state = READY;
            endcase
        end
        default: // Should not reach this point
            next_state = 'bx;
        endcase
    end
endmodule
```



YASAC Stage 1 Verilog coding yasac.v



```
module yasac (
    input wire clk,
    input wire reset,
    input wire start,
    output wire ready,
    input wire [7:0] din,
    output wire [7:0] dout,
    output wire [1:0] state_out

);

    // Internal signals
    wire [4:0] opcode;
    wire [1:0] op;
    wire ipc, clpc, wir, wreg, inm;
```

```
    // Control unit instance
    control_unit control_unit (
        .clk(clk),
        .reset(reset),
        .start(start),
        .ready(ready),
        .opcode(opcode),
        .op(op),
        .ipc(ipc),
        .clpc(clpc),
        .wir(wir),
        .wreg(wreg),
        .inm(inm),
        .state_out(state_out)
    );
    // Data unit instance
    data_unit data_unit (
        .clk(clk),
        .op(op),
        .ipc(ipc),
        .clpc(clpc),
        .wir(wir),
        .wreg(wreg),
        .inm(inm),
        .opcode(opcode),
        .din(din),
        .dout(dout)
);
endmodule
```

YASAC Stage 1 Verilog coding yasac_tb.v

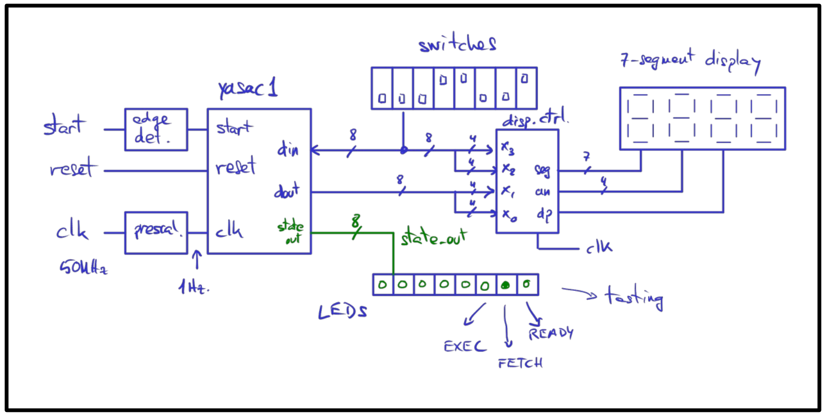
- Generate a clock signal.
- Reset the system.
- Activate execution (start).
- Wait for “ready” activation.
  - End if not activated in a long tin
- Print “din” and “dout”.
- “dout” not as expected?
  - Open a waveform viewer.

```
module test ();
    reg clk;            // clock (rising edge)
    reg reset;          // reset (synchronous, a
    reg start;          // start operation
    wire ready;         // ready output indicate
    reg [7:0] din;      // external data input
    wire [7:0] dout;    // external data output

    yasac uut (
        .clk(clk),      // clock (rising edge)
        .reset(reset),  // reset (synchronous, a
        .start(start),  // start operation
        .ready(ready),  // ready output indicate
        .din(din),      // external data input
        .dout(dout)     // external data output
    );

    // Clock generator (T=20ns, f=50MHz)
    always
        #10 clk = ~clk;
```

```
    initial begin
        // output generation
        $dumpfile("yasac_tb.vcd");
        $dumplimit(10000000);     // limit dump file to 10MB
        $dumpvars(0, test);
        // input signal initialization
        clk = 1'b0;
        reset = 1'b0;
        start = 1'b0;
        din = 8'd6;
        // global reset
        @(posedge clk) #1 reset = 1'b1;
        @(posedge clk) #1 reset = 1'b0;

        repeat(3) @(posedge clk) #1;

        // start program execution
        start = 1'b1;
        @(posedge clk) #1;
        start = 1'b0;
        // wait for "ready"
        wait(ready)
            $display("'ready' activation detected.");

        repeat(3) @(posedge clk) #1;
        $display("Normal simulation end.");

        // Print input and output ports (quick check results)
        $display("din: %h, dout: %h", din, dout);
        $finish;
    end

    // Force finish after 1000 clock cycles
    initial begin
        #(20*1000);
        $display("'ready' not detected. Abnormal simulation end.");
        $display("Check the design.");
    end
endmodule
```

YASAC Stage 1 Verilog coding system.v

```verilog
module system (
    // External signals
    input wire clk,          // clock (rising edge)
    input wire reset,        // reset (synchronous)
    input wire start,        // start operation
    output wire ready,       // ready output
    input wire [7:0] din,    // external data input
    output wire [7:0] dout,  // external data output
    output wire [0:6] seg,   // 7-segment output
    output wire [3:0] an,    // anode output
    output wire dp,          // decimal point output
    // FSM state output for testing
    output reg [7:0] state_dec
);
    // Clock divider to 1Hz
    reg [24:0] prescaler;
    reg clk_in;
    always @(posedge clk)
        if (prescaler == 25000000-1) begin
            clk_in = ~clk_in;
            prescaler = 'b0;
        end else begin
            prescaler = prescaler + 1;
        end

    // Edge detector for 'start'
    reg start0=0, start1=0;
    wire start_pulse;
    always @(posedge clk_in) begin
        start1 <= start0;
        start0 <= start;
    end
    assign start_pulse = start0 & ~start1;

    // State decoder (for testing)
    wire [1:0] state_out;
    always @* begin
        state_dec = 'b0;
        state_dec[state_out] = 1'b1;
    end
```



```verilog
    // Processor instance
    yasac yasac (
        .clk(clk_in),          // clock (rising edge)
        .reset(reset),         // reset (synchronous)
        .start(start_pulse),   // start operation
        //.ready(ready),       // ready output indicator
        .din(din),             // external data input
        .dout(dout),           // external data output
        // state output (for testing)
        .state_out(state_out)
    );

    // 7-segment controller instance
    display_ctrl #(
        .cdbits(18), .hex(1)
        ) display_ctrl (
        .ck(clk),              // system clock
        .x3(din[7:4]),         // display digits
        .x2(din[3:0]),
        .x1(dout[7:4]),
        .x0(dout[3:0]),
        .dp_in(4'b1011),       // decimal point vector
        .seg(seg),             // 7-segment output
        .an(an),               // anode output
        .dp(dp)                // decimal point output
    );
endmodule
```

YASAC Stage 2

- Stage 1 limitations
  - No data memory (storage limited to internal registers).
  - Limited input/output (only one input and one output port).
  - (Many more…)
- Why do we need a data memory?
  - 8 registers are not enough for most applications
  - Need more room to store data (lists, conversion tables, text, etc.)
- Why do we need more input/output ports
  - An useful computer, even a simple one, needs a few peripherals:
    - serial ports,
    - generic input/output ports -GPIO-,
    - keyboard,
    - display, etc.

Data memory

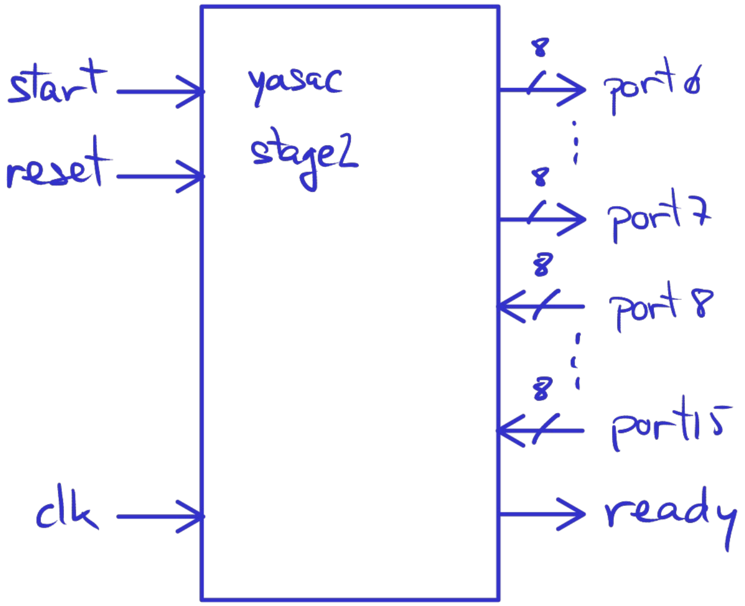
- Having a data memory implies new design decissions:
  - Need new instructions to transfer data to/from memory.
  - Q1: Can we operate with the memory data directly?
    - A1: Data processing architecture.
  - Q2: How do we transfer data to/from memory?
    - Addressing modes.
- Q1: Load/Store architecture
  - Processing instructions (ADD, SUB, etc.) only work on registers.
  - Data must be loaded into registers before processing and stored back in memory afterwards.
  - Simplifies the design of the computer.
  - Typical of RISC processors.
  - We will use this approach in the YASAC
  - Other processors can process data in the memory directly (with limitations). Eg. x86-XX.

Data memory
Addressing modes

- How data is accessed.
- Addressing modes in the YASAC
  - **Immediate mode**: data is within the instruction code (eg. LDI instruction).
  - **Register mode**: data is in a register (eg. MOV instruction).
  - **Direct memory mode**: data is in memory and the address is included the instruction code (new LDS instruction).
  - **Register indirect mode**: data is in memory and the memory address is in a register (new LD instruction).
- Other addressing modes (not in the YASAC)
  - **Displacement mode**: the memory address is obtained by adding a “small” displacement value (offset) to the value of a register.
  - **Indexed mode**: the memory address is obtained by adding the value of a register to the address included in the instruction.
  - **PC-relative addressing**: the memory address is obtained by adding a “small” displacement to the value of the Program Counter (PC).
  -

YASAC Stage 2. General specification

- 8-bit data unit and registers
- 8 general purpose registers
- Program memory: 256x16
- Data memory: 256x8
- 8 8-bit input ports and 8 8-bit output ports
  - port0 to port7: output ports mapped to memory addresses 240 to 247 (F0 to F7).
  - port8 to port15: input ports mapped to memory addresses 248 to 255 (F8 to FF).
- 16-bit instructions



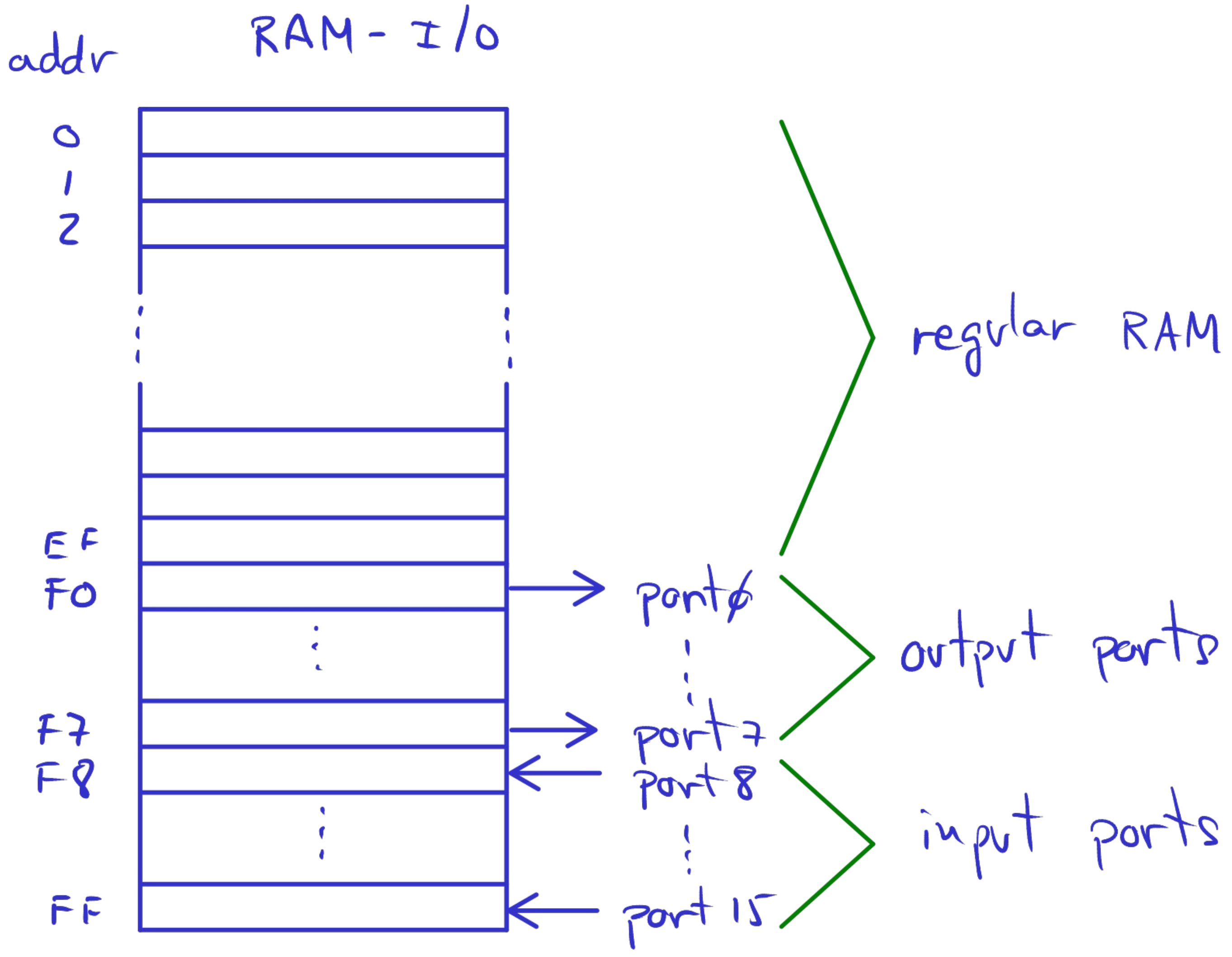
YASAC Stage 2
Needed changes

- Data memory module.
  - RAM memory.
- More input/output ports.
  - I/O ports will be mapped to memory addresses and will be implemented together with the memory module.
- Memory Address Register (MAR).
  - To hold the data memory address that is to be accessed.
- A multiplexed bus.
  - Now data can come from the ALU or the data memory.

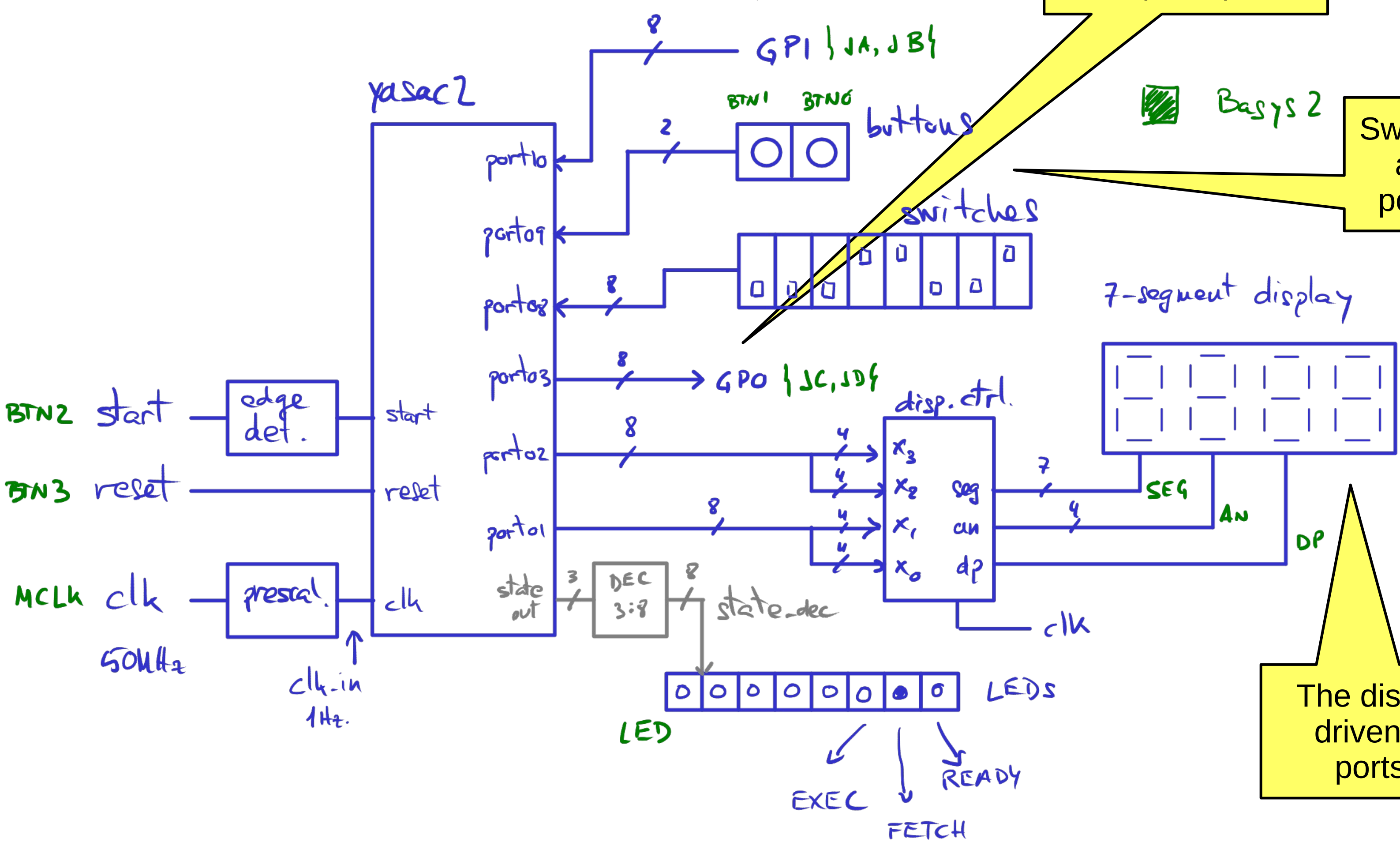
YASAC Stage 2
Memory-mapped I/O

- From the computer’s and programmer’s point of view, there is only data memory.
  - There are no special instructions to access input/output ports.
- Some ranges of the data memory address space are routed to input or output ports.
- A control circuit in the data memory module decides if it has to access memory or ports depending on the address.
- Memory-mapped input/output is common in real processors.

YASAC Stage 2
Memory-mapped I/O



YASAC Stage 2 Board implementation



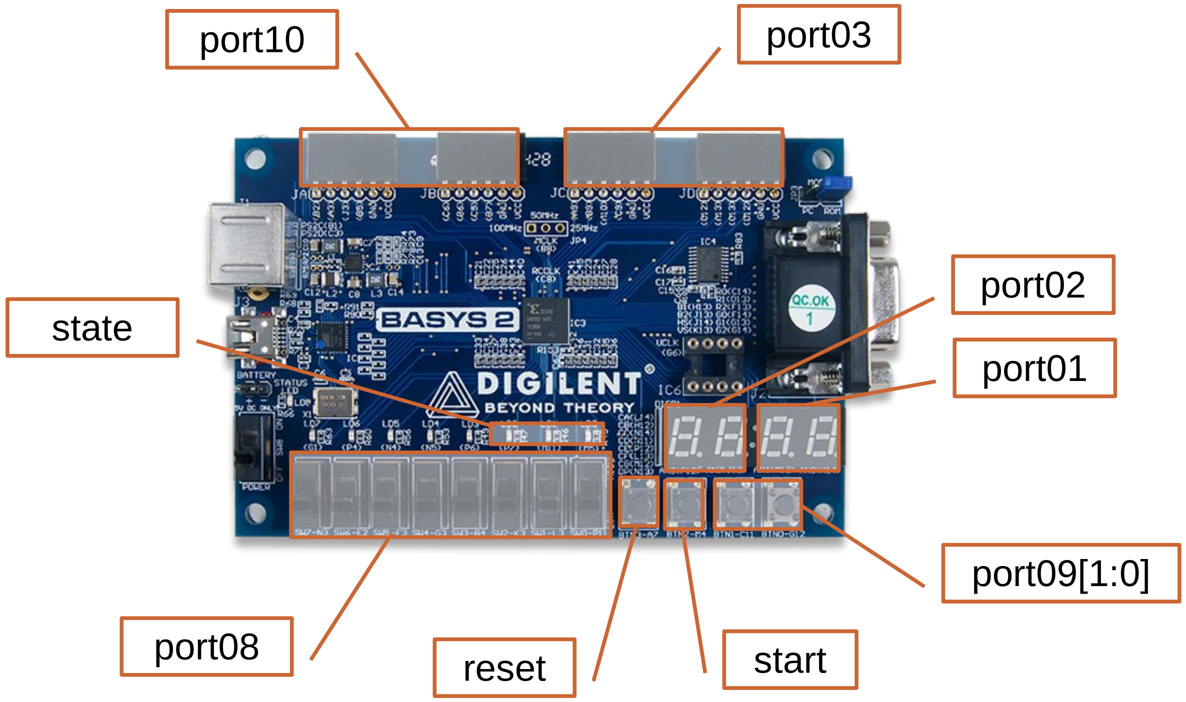
General Purpose digital Input (8 bits)

General Purpose digital Output (8 bits)

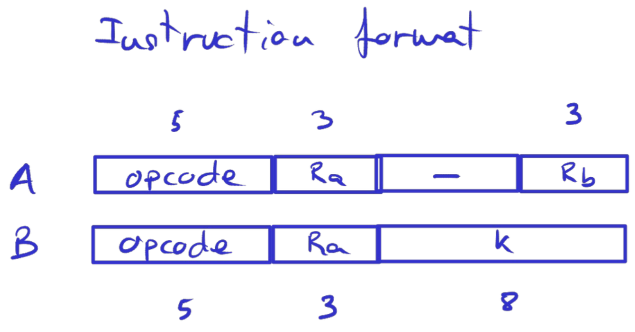
Switches and buttons are read at input ports 8 and 9 resp.

The display is now driven by output ports 1 and 2

YASAC Stage 2
Board implementation



YASAC Stage 2 Instruction Set



Op. code	Instruction	RTL
00001	LDI Ra, k	Ra ← k
00010	MOV Ra, Rb	Ra ← Rb
00011	ADD Ra, Rb	Ra ← Ra+Rb
00100	SUB Ra, Rb	Ra ← Ra-Rb
00101	STOP	–
00110	LD Ra, Rb	Ra ← datamem(Rb)
00111	ST Rb, Ra	datamem(Rb) ← Ra
01000	LDS Ra, k	Ra ← datamem(k)
01001	STS k, Ra	datamem(k) ← Ra

YASAC Stage 2
Sample program

Reads data from port8, adds to previous value at memory address 10h, saves the result to address 10h and outputs the result to output port1.

Assembly code

```
LDI R1,0x10   ; pointer to old value
LDI R2,0xF1   ; pointer to port1
LDS R3,0xF8   ; read port8
LD  R4,R1     ; read old value to R4
ADD R3,R4     ; add old and new value
STS 0x10,R3   ; store result in memory
ST  R2,R3     ; output result to port1
STOP
```

Quick exercise

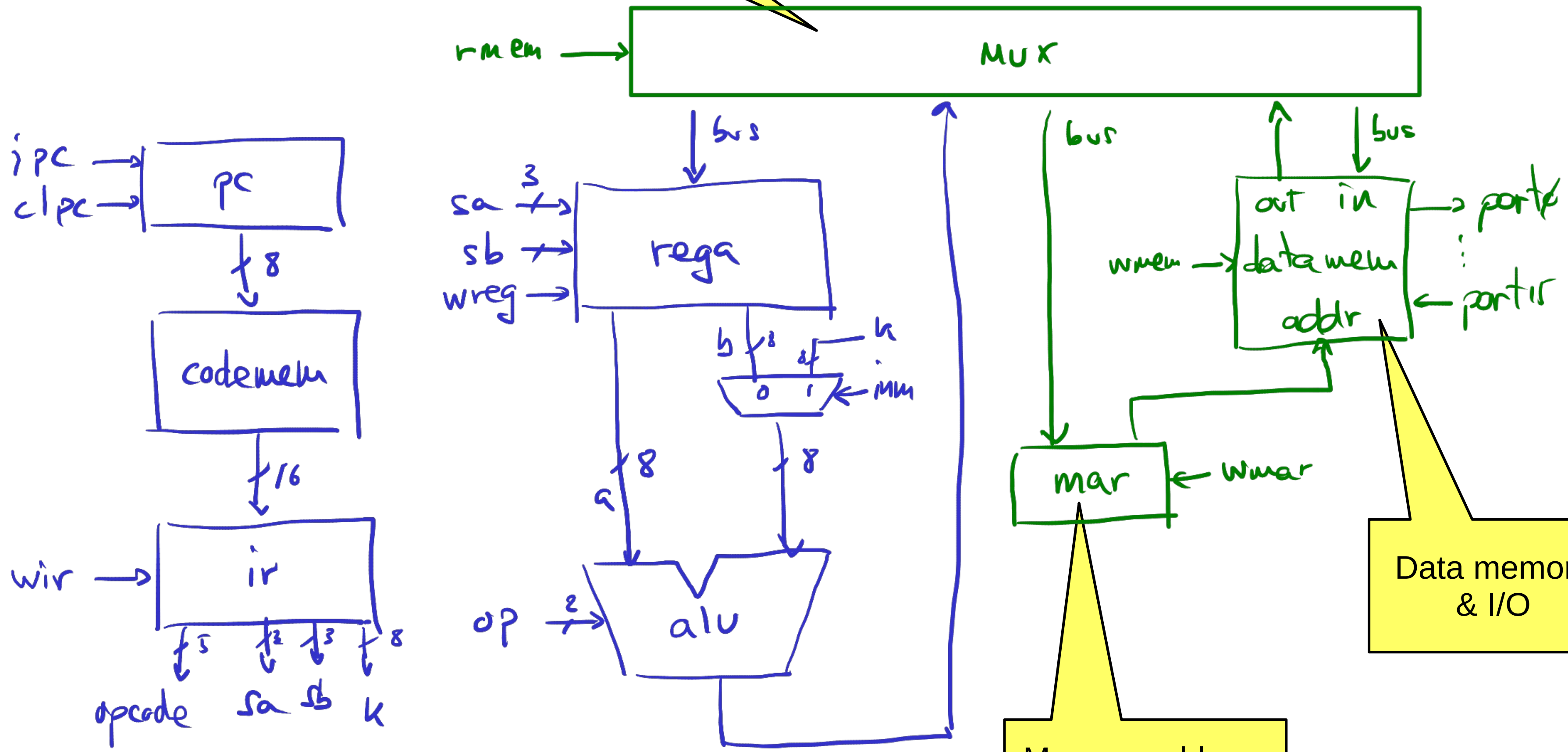
a) Obtain the machine code in binary and hexadecimal.

b) What is the output value at port01 if the initial value at memory address 0x10 is 25 and the input at port08 is 7?

c) What is the value at the output port if we execute the program again?

YASAC Stage 2 Data unit

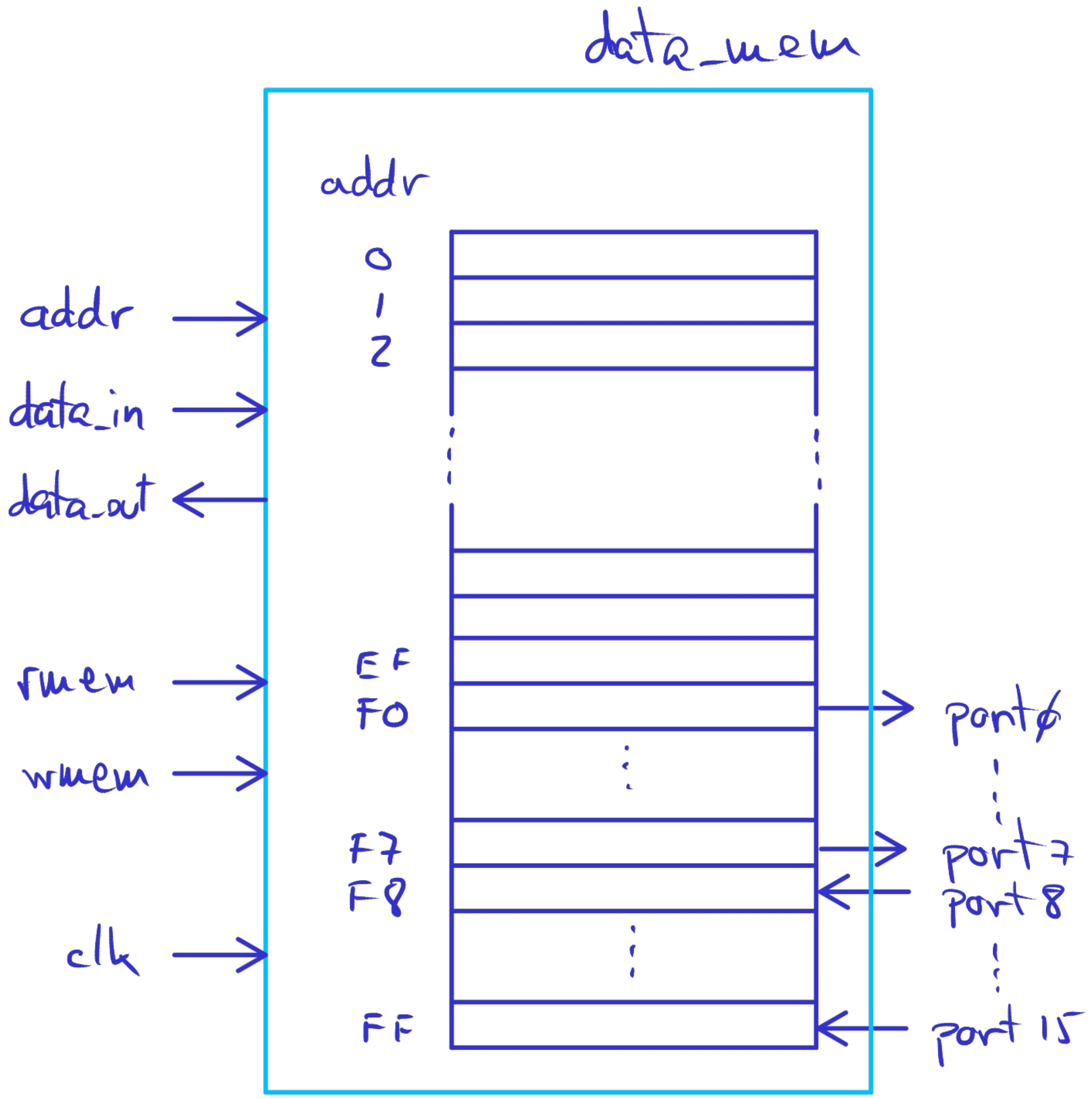
Multiplexed bus ALU output is selected by default



Data memory & I/O

Memory address register

YASAC Stage 2
Data memory and I/O



YASAC Stage 2 Control unit. Micro-operations

LD Ra, Rb

1)	mar ← rb	op=ALU_TRB , wmar
2)	ra ← data_mem [mar]	rmem, wreg

ST Rb, Ra

1)	mar ← rb	op=ALU_TRB , wmar
2)	data_mem [mar] ← ra	op= ALU_TRA , wmem

LDS Ra, k

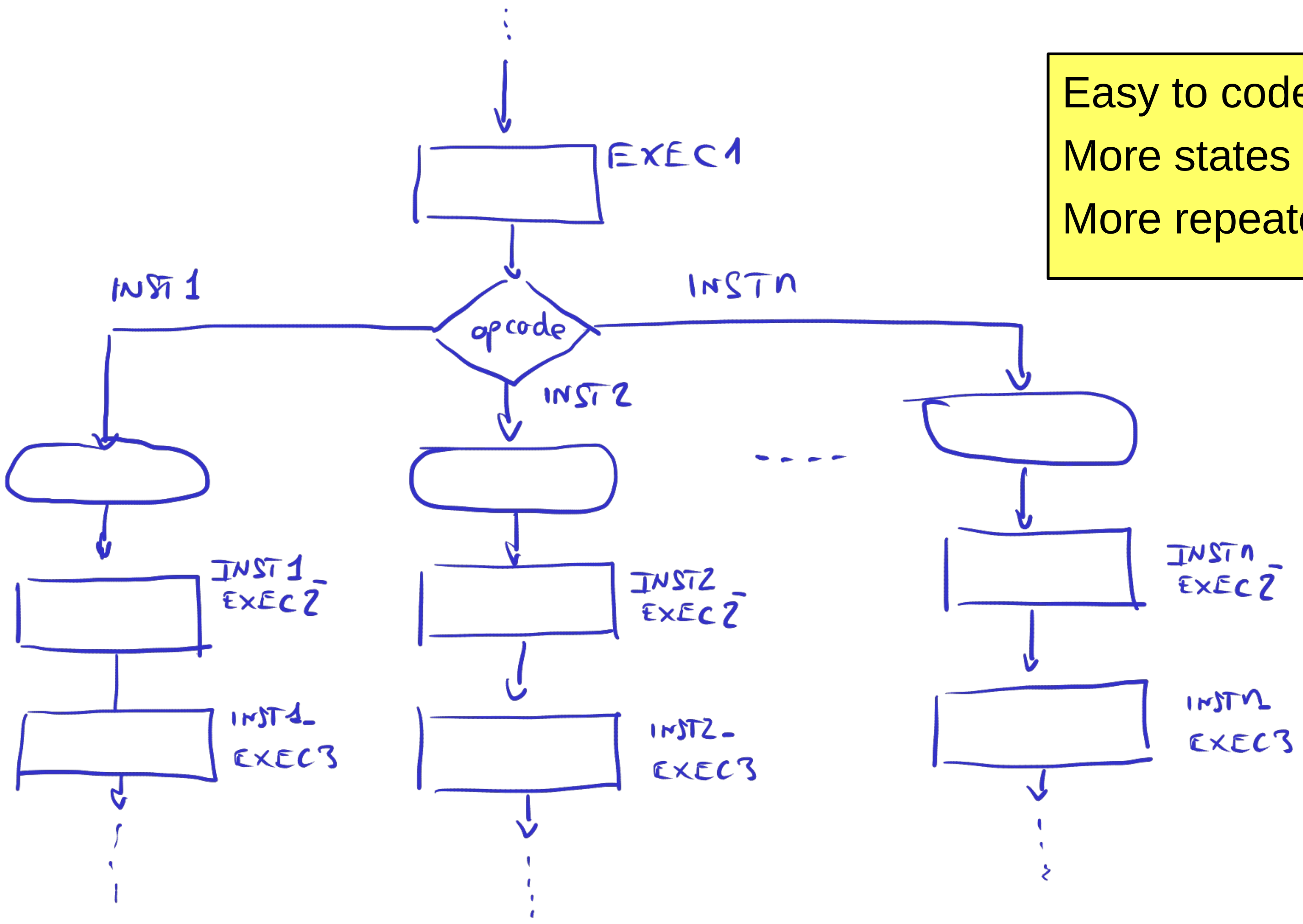
1)	mar ← k	op=ALU_TRB , inm , wmar
2)	ra ← data_mem [mar]	rmem, wreg

STS k, ra

1)	mar ← k	op=ALU_TRB , inm , wmar
2)	data_mem [mar] ← ra	op= ALU_TRA , wmem

YASAC Stage 2 control unit State definition strategy

Different states for different instructions



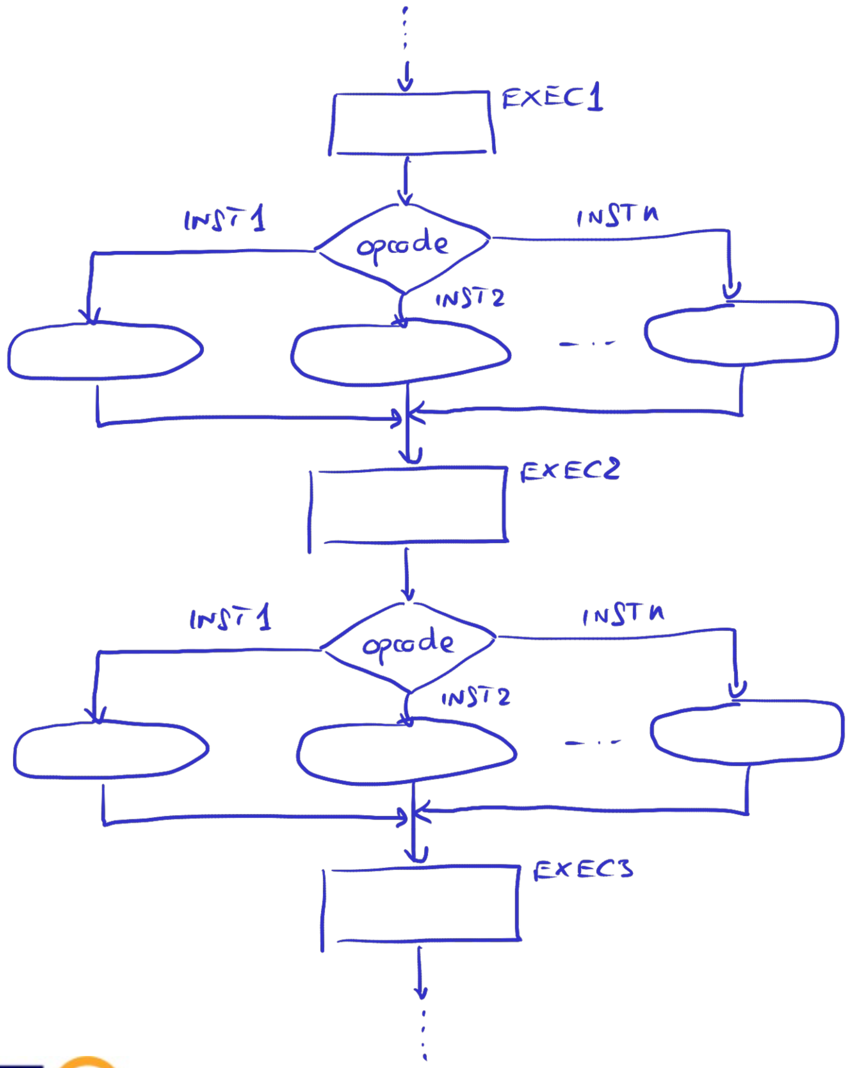
Easy to code in HDL.

More states than necessary.

More repeated code.

YASAC Stage 2 control unit State definition strategy

Same states for all instructions



Saves states.

Easier to combine similar instructions.

Easier to separate sequential and combinational processes.

More difficult to code?

We will use this one!

YASAC Stage 2 Control unit. States and control table

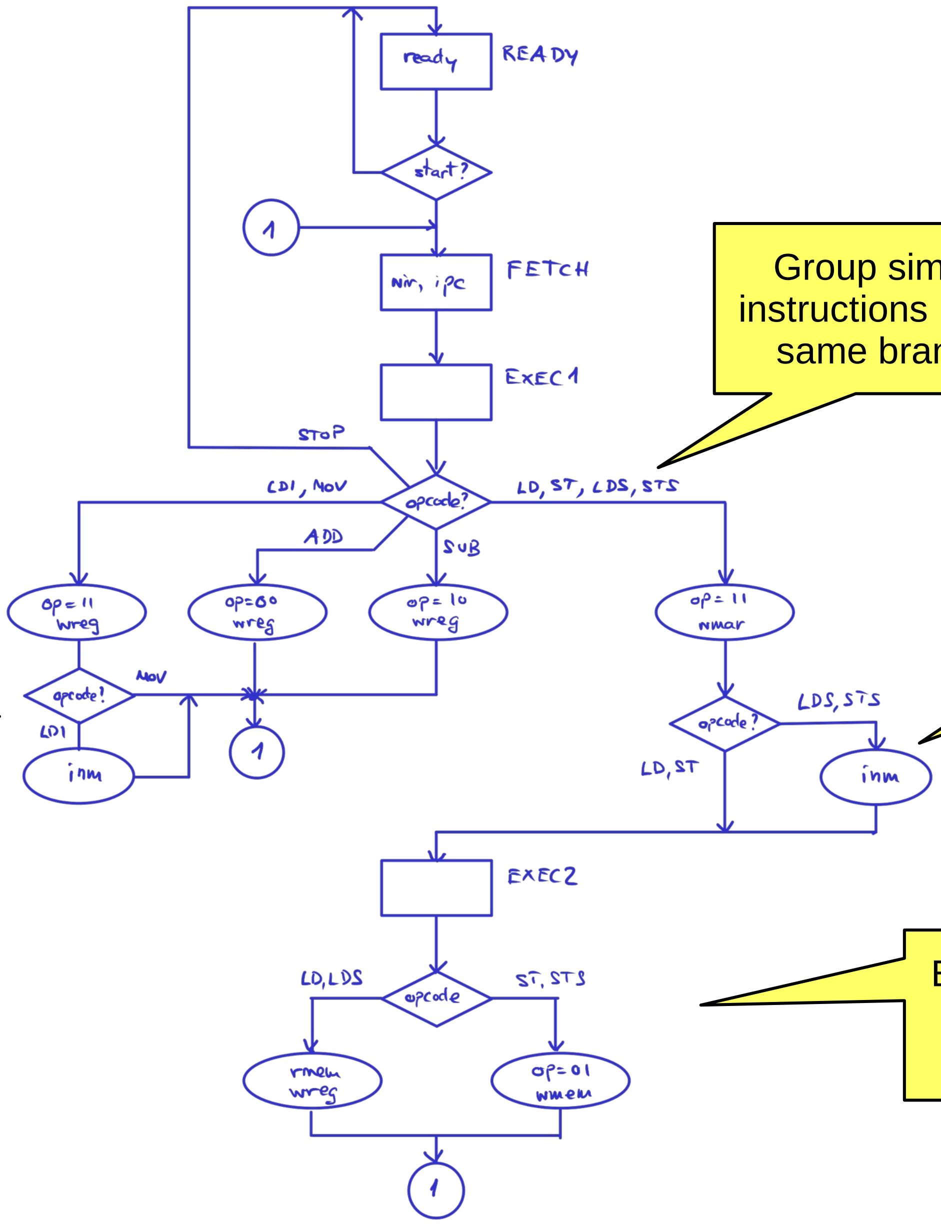
	READY	FETCH	EXEC1	EXEC2
LDI Ra, k	ready start: clpc	wir, ipc	op=11, wreg, inm →FETCH	
MOV Ra, Rb			op=11, wreg →FETCH	
ADD Ra, Rb			op=00, wreg →FETCH	
SUB Ra, Rb			op=10, wreg →FETCH	
STOP			→READY	
LD Ra, Rb			op=11, wmar	rmem, wreg →FETCH
ST Rb, Ra			op=11, wmar	op=01, wmem →FETCH
LDS Ra, k			op=11, wmar, inm	rmem, wreg →FETCH
STS k, Ra			op=11, wmar, inm	op=01, wmem →FETCH

The table represents the control signals to activate at every execution step depending on the instruction (opcode) to execute.

With this representation, the same states are used to execute all the instructions.

It is a convenient way to organize the information about the control unit that simplifies HDL coding.

YASAC Stage 2 Control unit. ASM control chart



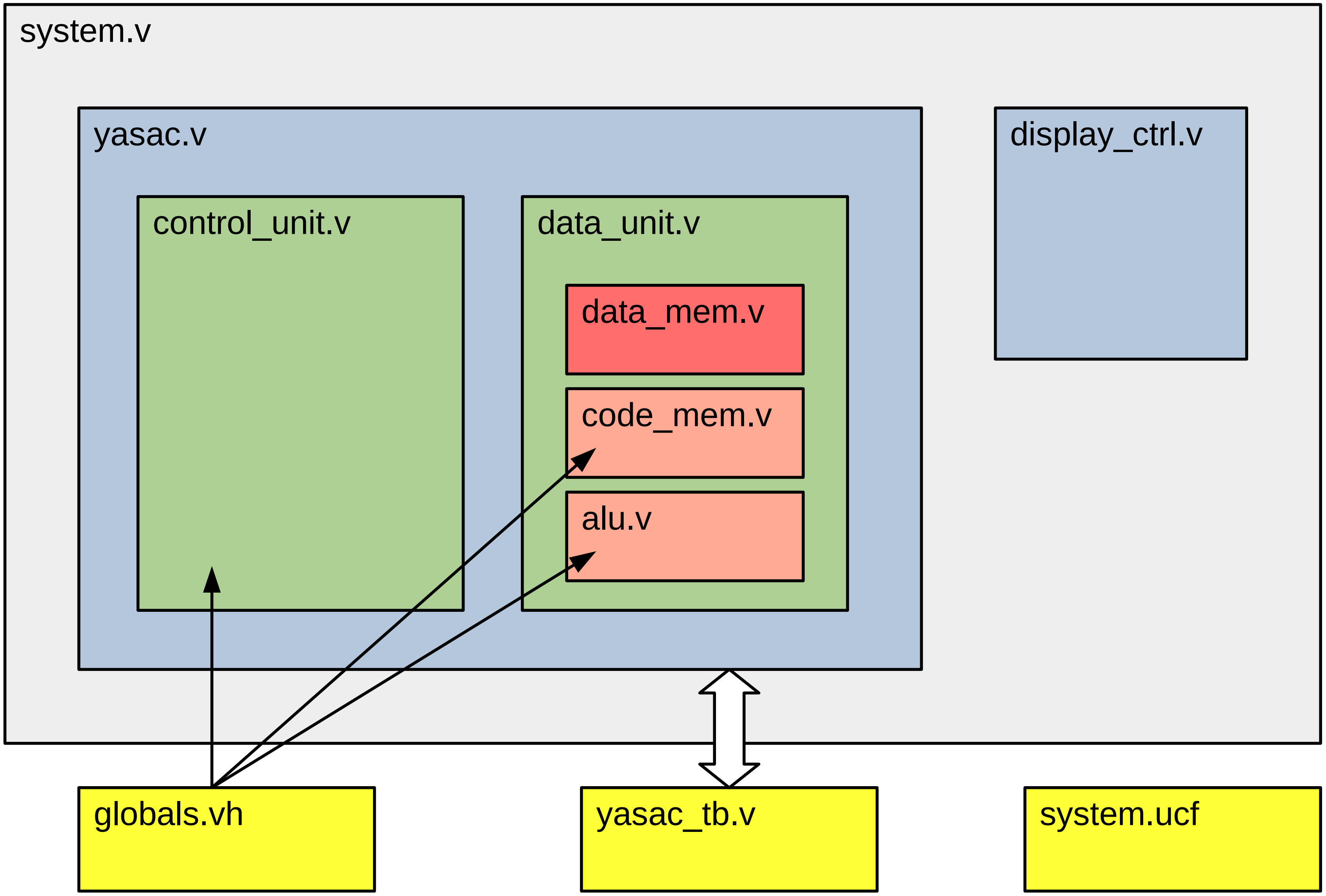
Group similar instructions in the same branch

LDI is like MOV but using an immediate value

'straight' instructions want a immediate value

EXEC2 is the same for LD/LDS and ST/STS

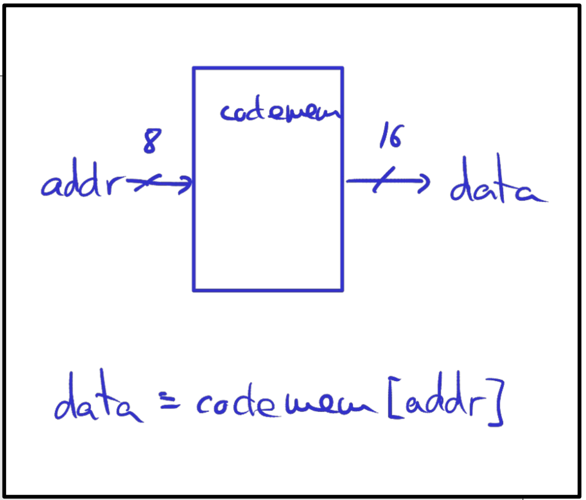
YASAC Stage 2 Verilog coding



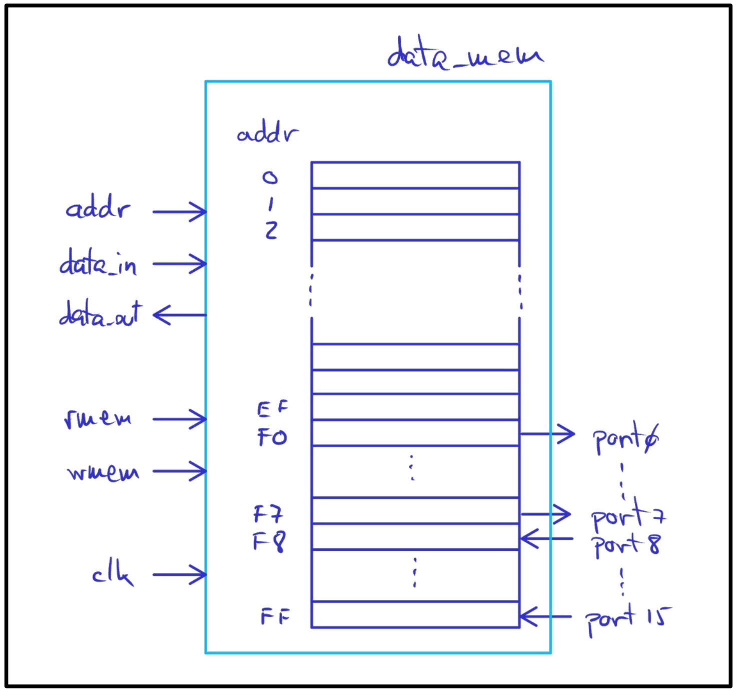
YASAC Stage 2 Verilog coding code_mem.v

```
`include "globals.vh"

module code_mem (
    input wire [7:0] addr,      // address port
    output wire [15:0] data     // data port
);

    reg [15:0] code[0:255];
    integer i;

    assign data = code[addr];

    initial begin
        // Code memory contents (program)
        code['h0] = {`LDI,  `R1, 8'h10};       // LDI R1,0x10
        code['h1] = {`LDI,  `R2, 8'hf1};       // LDI R2,0xf1
        code['h2] = {`LDS,  `R3, 8'hf8};       // LDS R3,0xf8 ; load from port08
        code['h3] = {`LD,   `R4, 5'd0, `R1};   // LD  R4,R1
        code['h4] = {`ADD,  `R3, 5'd0, `R4};   // ADD R3,R4
        code['h5] = {`STS,  `R3, 8'h10};       // STS 0x10,R3
        code['h6] = {`ST,   `R3, 5'd0, `R2};   // ST  R2,R3   ; store to port01
        code['h7] = {`STOP, 11'd0};            // STOP
    end
endmodule
```

Tests all the new instructions



YASAC Stage 2 Verilog coding data_mem.v



```
module data_mem (
  input wire clk,                // clock (rising edge)
  input wire wmem,               // write data memory
  input wire [7:0] addr,         // address
  input wire [7:0] data_in,      // input data
  output wire [7:0] data_out,    // output data
  output wire [7:0] port00, port01, port02, port03,    // output
                    port04, port05, port06, port07,    // ports
  input wire [7:0]  port08, port09, port10, port11,    // input
                    port12, port13, port14, port15     // ports
);
```

YASAC Stage 2 Verilog coding data_mem.v

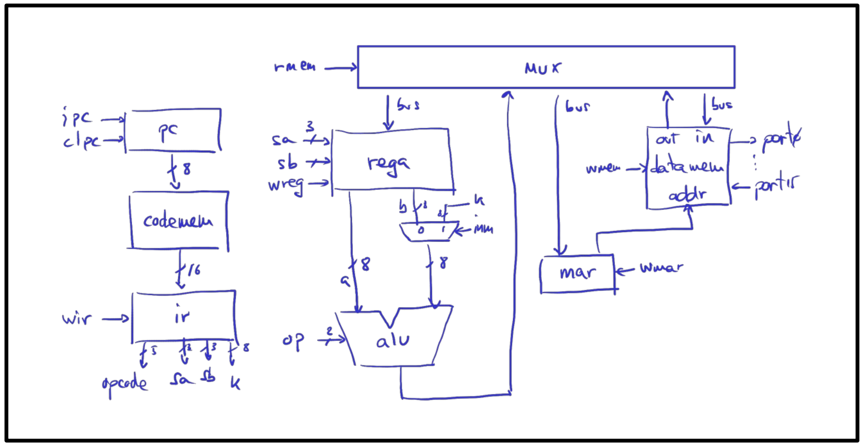
```
// RAM write
always @(posedge clk)
    // Write only RAM and output ports
    if (wmem && addr < 8'hf0)
        mem[addr] <= data_in;

// Output port write
always @(posedge clk)
    if (wmem && addr >= 8'hf0 && addr < 8'hf8)
        case(addr[3:0])
        4'h0:   port_reg00 <= data_in;
        [...]
        4'h7:   port_reg07 <= data_in;
        endcase

// Output port read
assign port00 = port_reg00;
[...]
assign port07 = port_reg07;

// Input port write (from external pins)
always @(posedge clk) begin
    port_reg08 <= port08;
    [...]
    port_reg15 <= port15;
end
```

```
// Asynchronous read
always @*
    case(addr[3:0])
    4'h0:   port_out = port_reg00;
    [...]
    4'h7:   port_out = port_reg07;
    4'h8:   port_out = port_reg08;
    [...]
    4'hf:   port_out = port_reg15;
    default: port_out = 'bx;
    endcase

// Data output generation
assign data_out = (addr < 8'hf0) ?
    mem[addr] : port_out;
```

Memory and ports reading

Memory and ports writing

YASAC Stage 2 Verilog coding data_unit.v



```
module data_unit (
    input wire clk,          // clock (rising edge)
    input wire [1:0] op,     // ALU operation code
    input wire ipc,          // PC increment
    input wire clpc,         // PC clear
    input wire wir,          // write IR
    input wire wreg,         // write register array
    input wire inm,          // use inmediate value
    output wire [4:0] opcode, // operation code
    input wire wmem,         // write data memory
    input wire rmem,         // read data memory
    input wire wmar,         // write MAR
    output wire [7:0] port00, port01, // output
                      port02, port03, // ports
                      port04, port05,
                      port06, port07,
    input wire [7:0]  port08, port09, // input
                      port10, port11, // ports
                      port12, port13,
                      port14, port15
);
```

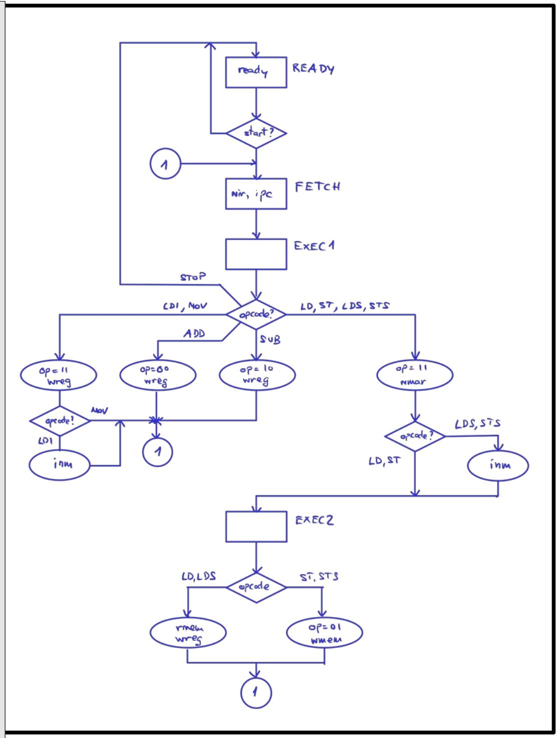
```
    [...]
    //// Memory address register

    always @(posedge clk)
        if (wmar)
            mar <= bus;

    //// Data memory

    data_mem data_mem (
        .clk(clk),
        .wmem(wmem),
        .addr(mar),
        .data_in(bus),
        .data_out(data_out),
        .port00(port00), .port01(port01),
        .port02(port02), .port03(port03),
        .port04(port04), .port05(port05),
        .port06(port06), .port07(port07),
        .port08(port08), .port09(port09),
        .port10(port10), .port11(port11),
        .port12(port12), .port13(port13),
        .port14(port14), .port15(port15)
    );
    [...]
```

YASAC Stage 2 Verilog coding control_unit.v

```
// Next state process
always @* begin
    // Default next state
    next_state = 'bx;
    case (state)
    READY:
        if (start)
            next_state = FETCH;
        else
            next_state = READY;
    FETCH:
        next_state = EXEC1;
    EXEC1:
        case (opcode)
        `LDI, `MOV, `ADD, `SUB:
            next_state = FETCH;
        `LD, `ST, `LDS, `STS:
            next_stat
        default: // 
            next_stat
        endcase
    EXEC2:
        next_state =
    default: // Shou
        next_state =
    endcase
end
```

```
// Output process
always @* begin

    // Default output
    ready = 1'b0; op =
    ipc = 1'b0; clpc =
    wir = 1'b0; wreg =
    inm = 1'b0; wmem =
    rmem = 1'b0; wmar

    case (state)

    READY: begin
        ready = 1'b1;
        if (start)
            clpc = 1'b
    end

    FETCH: begin
        wir = 1'b1;
        ipc = 1'b1;
    end
```

```
    EXEC1:
        case (opcode)
        `LDI: begin
            op = `ALU_TRB;
            wreg = 1'b1; inm = 1'b1;
        end
        `ADD: begin
            op = `ALU_ADD;
            wreg = 1'b1;
        end
        `SUB: begin
            op = `ALU_SUB;
            wreg = 1'b1;
        end
        `MOV: begin
            op = `ALU_TRB;
            wreg = 1'b1;
        end
        `LD, `ST, `LDS, `STS: begin
            op = `ALU_TRB;
            wmar = 1'b1;
            if (opcode==`LDS || opcode==`STS)
                inm = 1'b1;
        end
        endcase

    EXEC2:
        case (opcode)
        `LD, `LDS: begin
            rmem = 1'b1;
            wreg = 1'b1;
        end
        `ST, `STS: begin
            op = `ALU_TRA;
            wmem = 1'b1;
        end
        endcase
    endcase
end
```



Separate processes for next state and output calculation.

One thing at a time is easier!

YASAC Stage 2 Verilog coding yasac.v

```
module yasac (
    input wire clk,                 // clock (rising edge)
    input wire reset,               // reset (synchronous, active-high)
    input wire start,               // start operation
    output wire ready,              // ready output indicator
    output wire [7:0] port00, port01, port02, port03,   // output ports
                      port04, port05, port06, port07,
    input wire [7:0]  port08, port09, port10, port11,   // input ports
                      port12, port13, port14, port15,
    output wire [1:0] state_out     // FSM state output for testing
);

    // Internal signals
    wire [4:0] opcode;
    wire [1:0] op;
    wire ipc, clpc, wir, wre
```

```
    // Control unit instance
    control_unit control_unit (
        .clk(clk),                  // clock (rising edge)
        .reset(reset),              // reset (synchronous, active-low)
        .start(start),              // start operation
        .ready(ready),
        .opcode(opcode),
        .op(op),
        .ipc(ipc),
        .clpc(clpc),
        .wir(wir),
        .wreg(wreg),
        .inm(inm),
        .wmem(wmem),
        .rmem(rmem),
        .wmar(wmar),
        .state_out(state_out)
    );
```

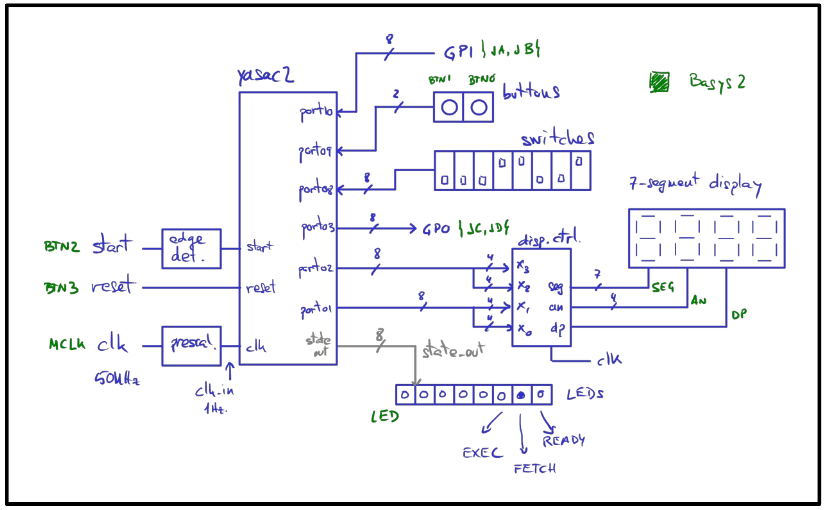
```
    // Data unit instance
    data_unit data_unit (
        .clk(clk),                  // clock (rising edge)
        .op(op),                    // ALU operation code
        .ipc(ipc),                  // PC increment
        .clpc(clpc),                // PC clear
        .wir(wir),                  // write IR
        .wreg(wreg),                // write register array
        .inm(inm),                  // use inmediate value
        .opcode(opcode),            // operation code of current ins
        .wmem(wmem),                // write data memory
        .rmem(rmem),                // read data memory
        .wmar(wmar),                // write memory address register
        .port00(port00), .port01(port01),   // output ports
        .port02(port02), .port03(port03),
        .port04(port04), .port05(port05),
        .port06(port06), .port07(port07),
        .port08(port08), .port09(port09),   // input ports
        .port10(port10), .port11(port11),
        .port12(port12), .port13(port13),
        .port14(port14), .port15(port15)
    );
endmodule
```

YASAC Stage 2 Verilog coding yasac_tb.v

```
module test ();

    reg clk;            // clock (rising edge)
    reg reset;          // reset (synchronous)
    reg start;          // start operation
    wire ready;         // ready output indicator
    // output ports
    wire [7:0] port00, port01, port02, port03,
               port04, port05, port06, port07;
    // imput ports
    reg [7:0]  port08, port09, port10, port11,
               port12, port13, port14, port15;


    yasac uut (
        .clk(clk),        // clock (rising edge)
        .reset(reset),    // reset (synchronous)
        .start(start),    // start operation
        .ready(ready),    // ready output indicator
        .port00(port00), .port01(port01),
        .port02(port02), .port03(port03),
        .port04(port04), .port05(port05),
        .port06(port06), .port07(port07),
        .port08(port08), .port09(port09),
        .port10(port10), .port11(port11),
        .port12(port12), .port13(port13),
        .port14(port14), .port15(port15)
    );

    // Clock generator (T=20ns, f=50MHz)
    always
        #10 clk = ~clk;
```

```
    initial begin
        // output generation
        $dumpfile("yasac_tb.vcd");
        $dumpvars(0, test);
        //$dumpvars(0, uut.data_unit.data_mem.mem['hf1]);

        // input signal initialization
        clk = 1'b0;
        reset = 1'b0;
        start = 1'b0;
        port08 = 8'd5;

        // global reset
        @(posedge clk) #1 reset = 1'b1;
        @(posedge clk) #1 reset = 1'b0;

        repeat(3) @(posedge clk) #1;

        // start program execution
        start = 1'b1;
        @(posedge clk) #1;
        start = 1'b0;
        // wait for "ready"
        wait(ready)
            $display("'ready' activation detected.");

        repeat(3) @(posedge clk) #1;
        $display("Normal simulation end.");

        // Check input and output ports
        $display("port08 (input): %h, port01 (output): %h",
                  port08, port01);

        // port01 (output) shoud be equal to port08 (input)
        if (port01 == port08) begin
            $display("Test bench result: PASS.");
            $finish;
        end else begin
            $display("Test bench result: FAIL.");
            $finish;
        end
    end
[...]
```

YASAC Stage 2 Verilog coding system.v



```
module system (
    // External signals
    input wire clk,                 // clock (rising edge)
    input wire reset,               // reset (synchronous, a
    input wire start,               // start operation
    output wire ready,              // ready output indicat
    //output wire [7:0] port00,     // 8xLED (disabled for 
    output wire [7:0] port03,       // generic digital outp
    input wire [7:0]  port08,       // 8xSwitches
    input wire [7:0]  port09,       // 4xButtons
    input wire [7:0]  port10,       // generic digital input
    output wire [0:6] seg,          // 7-segment output
    output wire [3:0] an,           // anode output
    output wire dp,                 // decimal point output
    output reg [7:0] state_dec      // FSM state output for
);
[...]
```

```
    // Processor instance
    yasac yasac (
        .clk(clk_in),
        .reset(reset),
        .start(start_pulse),
        //.ready(ready),        // disabled for debugging
        //.port00(port00),      // disabled for debugging
        .port01(port01),
        .port02(port02),
        .port03(port03),        // ports 4 to 7 not used
        .port08(port08),
        .port09(port09),
        .port10(port10),
        .port11(8'h00),         // not used
        .port12(8'h00),
        .port13(8'h00),
        .port14(8'h00),
        .port15(8'h00),
        .state_out(state_out) // state output
    );

    // 7-segment controller instance
    display_ctrl #(.cdbits(18), .hex(1)) display_ctrl (
        .ck(clk),               // system clock
        .x3(port02[7:4]),       // display digits
        .x2(port02[3:0]),
        .x1(port01[7:4]),
        .x0(port01[3:0]),
        .dp_in(4'b1011),        // decimal point vector
        .seg(seg),              // 7-segment output
        .an(an),                // anode output
        .dp(dp)                 // decimal point output
    );
```

YASAC Stage 3

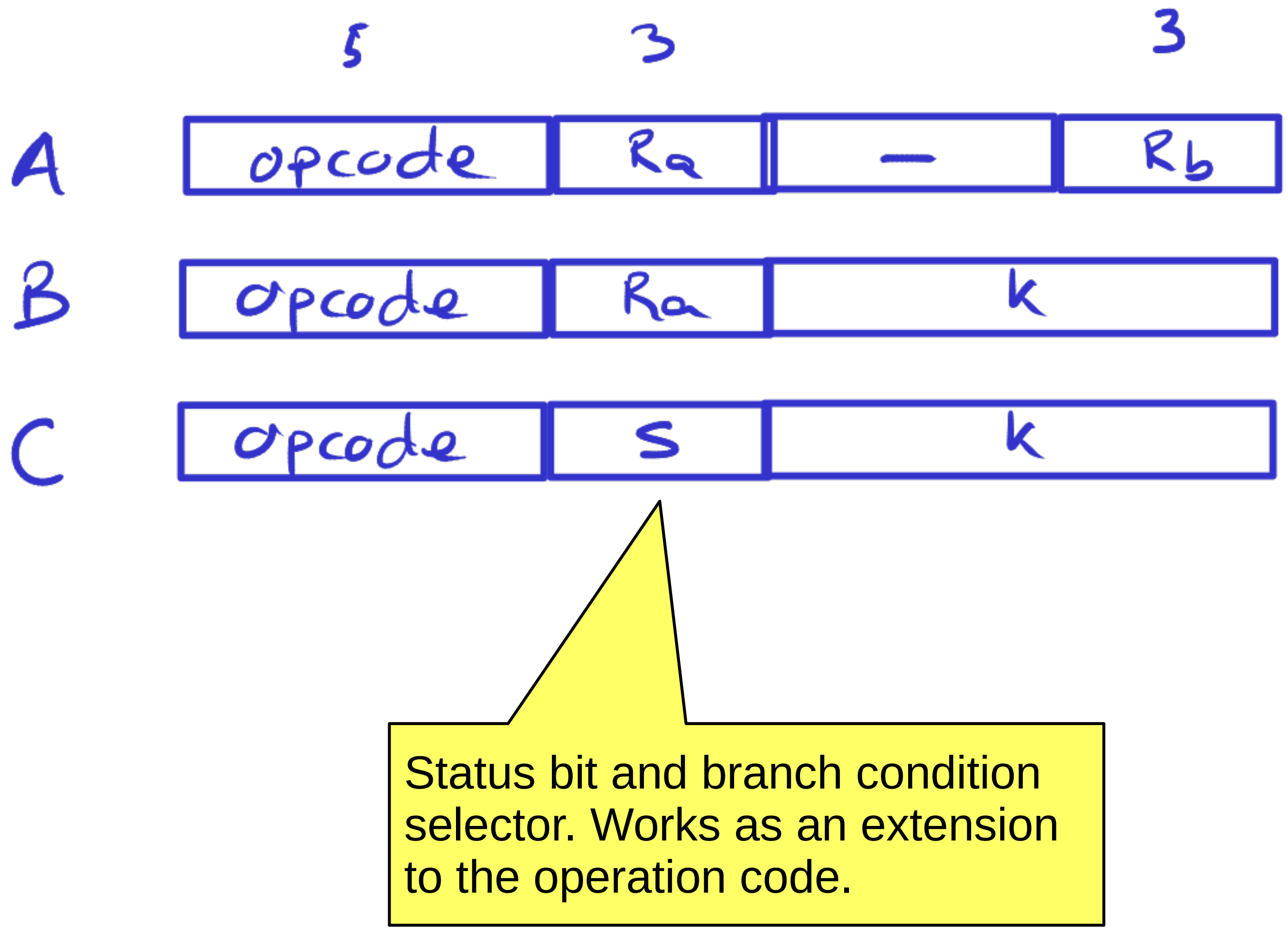
- Stage 2 limitations
  - YASAC Stage 2 programs cannot alter the sequence of instructions (branching). Cannot take decisions!
  - (A few more…)
- Why do we need (conditional) branching instructions?
  - Take decisions in our program.
  - Implement (any) algorithm.
  - Conditional branching makes a true computer.

With these modifications, YASAC will become a Turing-complete computer: it can implement any algorithm (except for memory limitations).

YASAC Stage 3
Needed changes

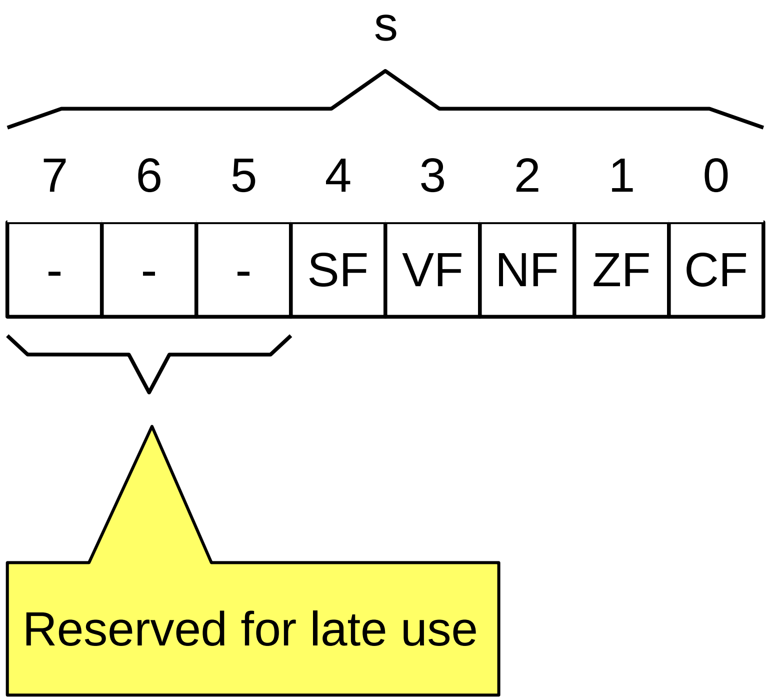
- New instruction format
  - To encode new branch instructions.
- Program counter:
  - Connect to the bus to allow PC writing (branch instructions).
- ALU:
  - Needs to generate the status output of the last performed operation.
    - CF: carry flag. Set when carry (add) or borrow (sub).
    - ZF: zero flag. Set when result is zero.
    - NF: negative flag. Set when result is negative (in Two’s complement).
    - VF: overflow flag. Set when overflow.
    - SF: sign flag (S=N^V). Set to 1 when A-B is negative, even if there is overflow.
- Status register:
  - New register to store ALU’s status output.
- Control unit:
  - Implement new instructions

YASAC Stage 3
Instruction format



Status bit and branch condition selector. Works as an extension to the operation code.

YASAC Stage 3 Status register



Status register flags

- CF: Carry Flag
- ZF: Zero Flag
- NF: Negative Flag
- VF: oVerflow Flag
- SF: Sign Flag

YASAC Stage 3
Branch instructions

- JMP k
  - **JuMP** to instruction at memory address k.
  - It is like a MOV instruction for the program counter.
- BRBS s, k
  - **BR**anch to instruction at memory address k if **B**it s in the status register is **S**et.
- BRBC s, k
  - **BR**anch to instruction at memory address k if **B**it s in the status register is **C**leared.

```
JMP  0xA2         ; Using a numeric address
JMP  LOOP         ; Using a label (translated by the assembler)

BRBS 0, NEXT      ; Branch if carry set
BRBS ZF, CONT     ; Branch if zero (using a constant ZF=1)

BRBC VF, NO_OV    ; Branch if no overflow
BRBC SF, PLUS     ; Branch if the sign of the result is positive
```

YASAC Stage 3 Instruction set

Op. code	Instruction	RTL	SVNZC
00001	LDI Ra, k	Ra ← k	-----
00010	MOV Ra, Rb	Ra ← Rb	-----
00011	ADD Ra, Rb	Ra ← Ra + Rb	*****
00100	SUB Ra, Rb	Ra ← Ra - Rb	*****
00101	STOP	–	-----
00110	LD Ra, Rb	Ra ← datamem(Rb)	-----
00111	ST Rb, Ra	datamem(Rb) ← Ra	-----
01000	LDS Ra, k	Ra ← datamem(k)	-----
01001	STS k, Ra	datamem(k) ← Ra	-----
01010	JMP k	PC ← k	-----
01011	BRBS s, k	sreg[s]: PC ← k	-----
01100	BRBC s, k	sreg[s]==0: PC ← k	-----

Now we need to specify how the status register is updated

YASAC Stage 3
Branch pseudo instructions

Instruction	Pseudo-instructions	Description
BRBS 0, k	BRCS k BRLO k	Branch if carry (Carry Set) Branch if A<B after unsigned A-B (LOwer)
BRBS 1, k	BRZS k BREQ k	Branch if the result is zero (Zero Set) Branch if A=B after A-B (EQual)
BRBS 2, k	BRMI k	Branch if the sign is Minus
BRBS 3, k	BRVS k	Branch if overflow (oVerflow Set)
BRBS 4, k	BRLT k	Branch A<B after signed A-B (Less Than)
BRBC 0, k	BRCC k BRSH k	Branch if no carry (Carry Cleared) Branch if A≥B after uns. A-B (Same or Higher)
BRBC 1, k	BRZC k BRNE k	Branch if the result is not zero (Zero Cleared) Branch if A≠B after A-B (Not Equal)
BRBC 2, k	BRPL k	Branch if the sign is PLus
BRBC 3, k	BRVC k	Branch if not overflow (oVerflow Cleared)
BRBC 4, k	BRGE k	Branch A≥B after signed A-B (Greater or Equal)

- Easier to remember.
- Automatically translated by the assembler.

YASAC Stage 3
Sample program

Assembly code

```
       LDI R0, 120
       LDS R1, 40
       STS 0xf1, R0
       STS 0xf2, R1
       ADD R0, R1
       STS 0xf1, R0
       BRVS OK
       STOP
OK:    JMP CONT
       STOP
CONT:  LDI R3, 0xff
       STS 0xf2, R3
       STOP
```

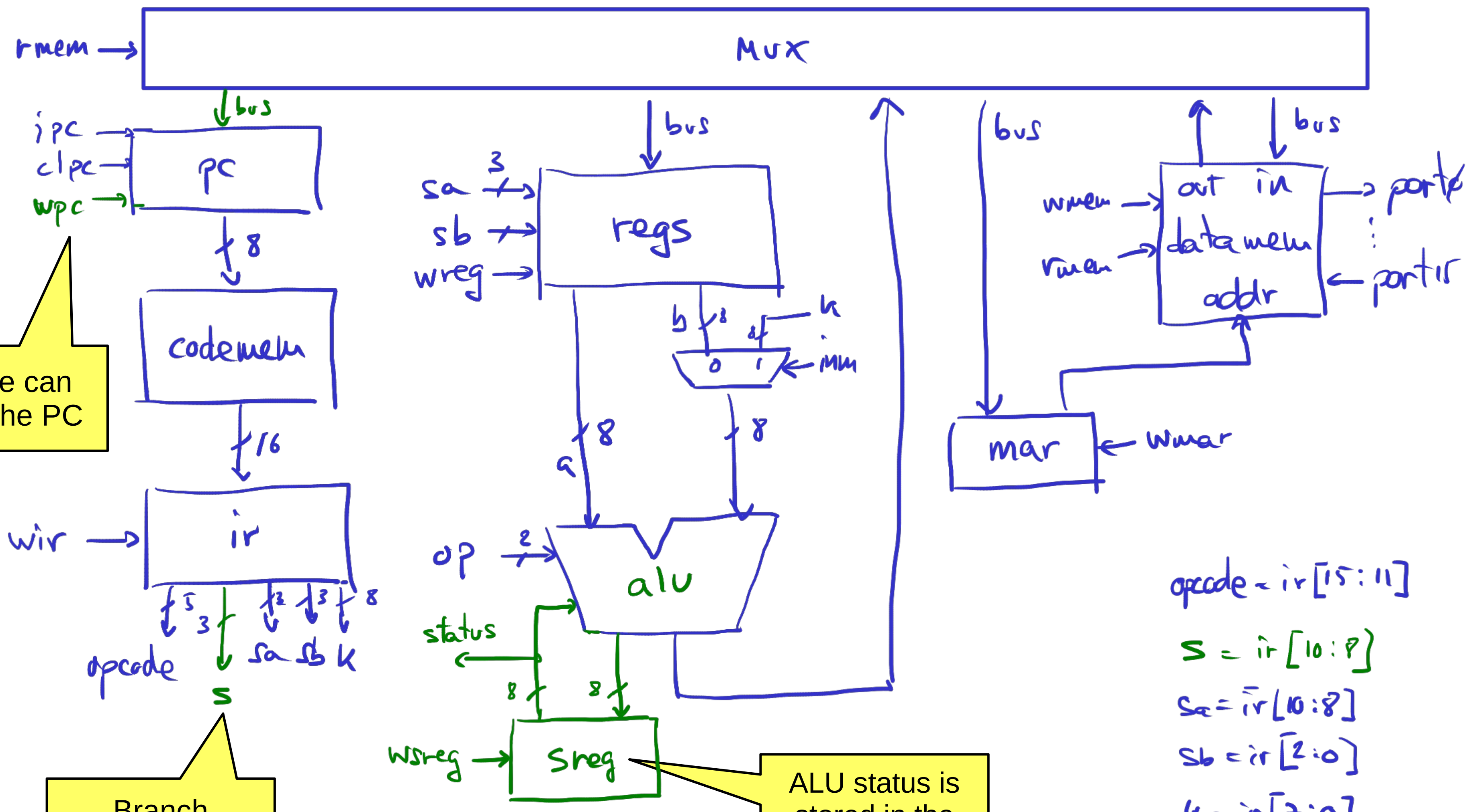
Quick exercise

a) What is the output at ports 01 and 02 if the program runs correctly?

b) What is the real instruction for “BRVS OK”?

Change these to try other numbers and instructions

YASAC Stage 3
Data unit



No we can write the PC

Branch condition code

ALU status is stored in the status register

opcode = ir[15:11]

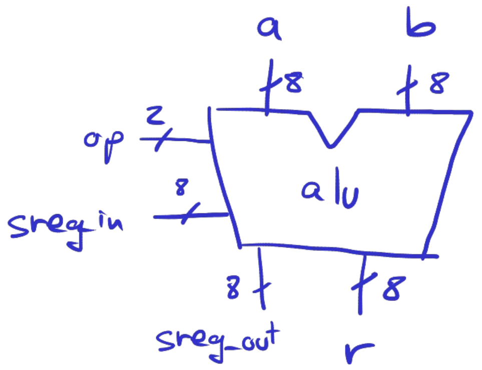
s = ir[10:8]

sa = ir[10:8]

sb = ir[2:0]

k = ir[7:0]

YASAC Stage 3 Updated ALU



op sym.	op	alu_out	sreg_out S V N Z C
ALU_ADD	0 0	a+b	* * * * *
ALU_TRA	0 1	a	- - - - -
ALU_SUB	1 0	a-b	* * * * *
ALU_TRB	1 1	b	- - - - -

sreg_in = {-, -, -, S_i, V_i, N_i, Z_i, C_i}

sreg_out = {-, -, -, S_o, V_o, N_o, Z_o, C_o}

ADD

$C_o = a_7 b_7 + b_7 \overline{r_7} + a_7 \overline{r_7}$

$Z_o = \overline{OR(r)}$

$N_o = r(7)$

$V_o = a_7 b_7 \overline{r_7} + \overline{a_7}\,\overline{b_7} r_7$

$S_o = V_o \oplus N_o$

SUB

$C_o = \overline{a_7} b_7 + b_7 r_7 + \overline{a_7} r_7$

$Z_o = \overline{OR(r)}$

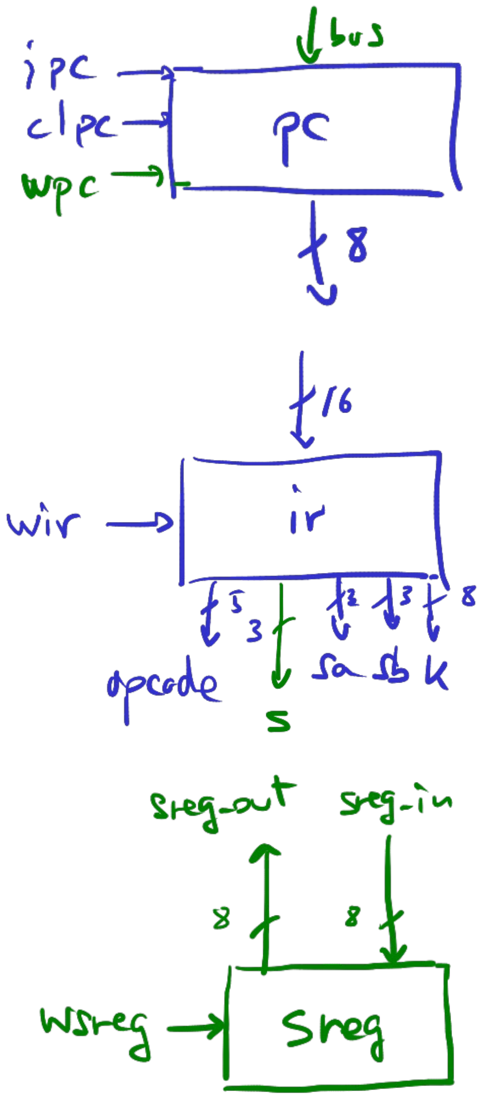
$N_o = r(7)$

$V_o = a_7 \overline{b_7}\,\overline{r_7} + \overline{a_7} b_7 r_7$

$S_o = V_o \oplus N_o$

$$S_o = \begin{cases} N_o & \text{if } V_o = 0, \\ \overline{N_o} & \text{if } V_o = 1 \end{cases}$$

YASAC Stage 3 Updated/new data unit blocks



ipc : pc ← (pc + 1) mod 256

clpc : pc ← 0

wpc : pc ← bus

s = ir [10:8]

wsreg : sreg ← sreg_in

YASAC Stage 3
Control unit. Micro-operations

Updated arithmetic operations

ADD Ra, Rb	1)	Ra ← Ra + Rb ; sreg ← sreg_out	op=ALU_ADD, wreg, wsreg
SUB Ra, Rb	1)	Ra ← Ra − Rb ; sreg ← sreg_out	op=ALU_SUB, wreg, wsreg

New jump instructions

JMP k	1)	pc ← k	inm, op=ALU_TRB, wpc
BRBS s,k	1)	status[s] : pc ← k	status[s] : (inm, op=ALU_TRB, wpc)
BRBC s,k	1)	$\overline{status[s]}$: pc ← k	$\overline{status[s]}$: (inm, op=ALU_TRB, wpc)

All new instructions are executed in a single clock cycle.

YASAC Stage 3
Control unit. States and control table

	READY	FETCH	EXEC1	EXEC2
LDI Ra, k	ready start: clpc	wir, ipc	op=11, wreg, inm →FETCH	
MOV Ra, Rb			op=11, wreg, →FETCH	
ADD Ra, Rb			op=00, wreg, wsreg →FETCH	
SUB Ra, Rb			op=10, wreg, wsreg →FETCH	
STOP			→READY	
LD Ra, Rb			op=11, wmar	rmem, wreg →FETCH
ST Rb, Ra			op=11, wmar	op=01, wmem →FETCH
LDS Ra, k			op=11, wmar, inm	rmem, wreg →FETCH
STS k, Ra			op=11, wmar, inm	op=01, wmem →FETCH
JMP k			inm, op=11, wpc	
BRBS s, k			status[s]: inm, op=11, wpc →FETCH	
BRBC s, k			~status[s]: inm, op=11, wpc →FETCH	

With this table we can update the Verilog code (do not need to draw an ASM chart).

YASAC Stage 3
Verilog coding. Try it yourself!

- Update the data unit modules that have changed:
  - ALU (alu.v)
  - PC, IR (data_unit.v)
- Design the new elements
  - Status register (data_unit.v)
- Update the control unit with the new instructions
- Update the input/output interface of the control and data units and update its interconnection (yasac.v).
- Write a simple program that uses branch instructions.
- Simulate:
  - Resolve syntax and compiler problems.
  - Check result.
  - If not correct, debug with Gtkwave.

YASAC Stage 4

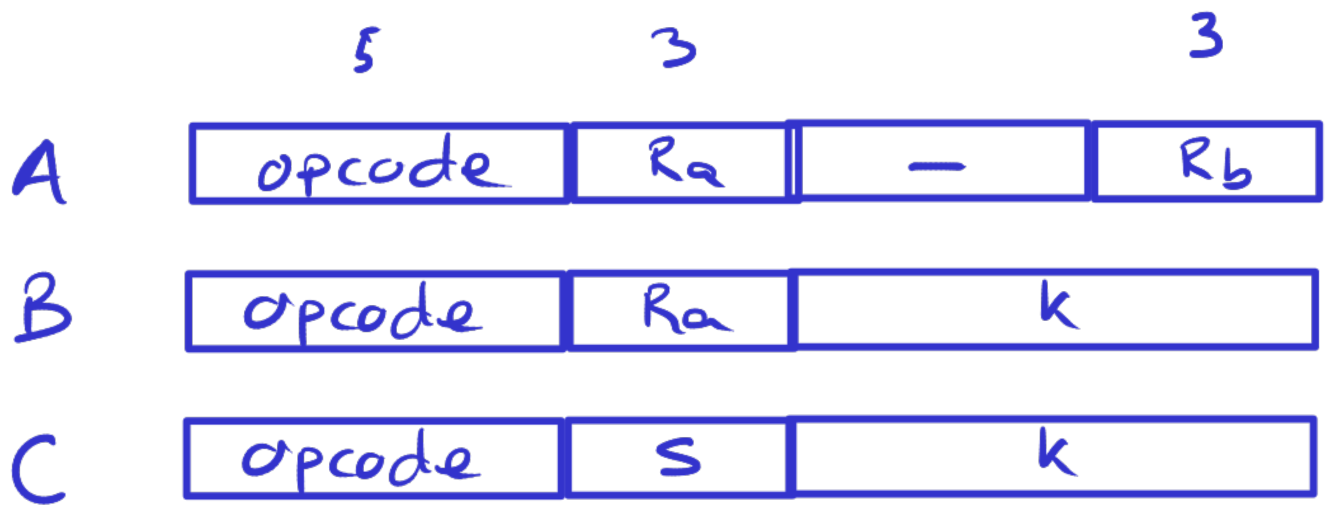
- Stage 3 limitations
  - The instruction set is poor: no logic or bit manipulation instructions.
  - (A few more…)
- Why logic instructions?
  - Bit manipulation (through masks).
  - Very useful to read/write individual input/output bits.
- Why shifting instructions?
  - Bit manipulation: bit counting, parity, etc.
  - Arithmetic: multiply/divide by 2.
  - Serial communications.
- What about updating the status register?
  - Useful to clean previous states.

With these modifications, the YASAC ISA is fairly complete and makes writing assembly programs much easier.

YASAC Stage 4
Overall needed changes

- ALU:
  - Implement logic operations.
  - Implement shift operations.
- Status register:
  - New inputs to allow register updates.
- Control unit:
  - Implement new instructions.

YASAC Stage 4
Instruction format



Now also used as status register bit selector.

YASAC Stage 4 Instruction set

Op. code	Instruction	RTL	SVNZC
00001	LDI Ra, k	Ra ← k	-----
00010	MOV Ra, Rb	Ra ← Rb	-----
00011	ADD Ra, Rb	Ra ← Ra + Rb	*****
00100	SUB Ra, Rb	Ra ← Ra - Rb	*****
00101	STOP	–	-----
00110	LD Ra, Rb	Ra ← datamem(Rb)	-----
00111	ST Rb, Ra	datamem(Rb) ← Ra	-----
01000	LDS Ra, k	Ra ← datamem(k)	-----
01001	STS k, Ra	datamem(k) ← Ra	-----
01010	JMP k	PC ← k	-----
01011	BRBS s, k	status[s]: PC ← k	-----
01100	BRBC s, k	~status[s]: PC ← k	-----
01101	AND Ra, Rb	Ra ← Ra & Rb	*****
01110	OR Ra, Rb	Ra ← Ra | Rb	*****
01111	EOR Ra, Rb	Ra ← Ra ^ Rb	*****
10000	ROR Ra	Ra ← SHR(Ra,C)	*****
10001	ROL Ra	Ra ← SHL(Ra,C)	*****
10010	BCLR s	sreg[s] ← 0	*****
10011	BSET s	sreg[s] ← 1	*****

YASAC Stage 4
BCLR and BSET pseudo instructions

Instruction	Pseudo-instructions	Description
BCLR 0	CLC	CLear Carry bit
BCLR 1	CLZ	CLear Zero bit
BCLR 2	CLN	CLear Negative bit
BCLR 3	CLV	CLear oVerflow bit
BCLR 4	CLS	CLear Sign bit
BSET 0	SEC	SEt Carry bit
BSET 1	SEZ	SEt Zero bit
BSET 2	SEN	SEt Negative bit
BSET 3	SEV	SEt oVerflow bit
BSET 4	SES	SEt Sign bit

YASAC Stage 4
Sample program

```
; input:  port08 = 0x5a (01011010)
; output: port01 = 0xa2
;         port02 = 0x8a
;
    ldi r0, 0         ; r0 = 0
    sub r0, r0        ; C=N=V=S=0, Z=1
    lds r0, 0xf8      ; 0 01011010 (0 5a)
    rol r0            ; 0 10110100 (0 b4)
    rol r0            ; 1 01101000 (1 68)
    rol r0            ; 0 11010001 (0 c1)
    rol r0            ; 1 10100010 (1 a2)
    sts 0xf1, r0      ; port01 = a2
    clc               ; 0 10100010 (0 a2)
    ror r0            ; 0 01010001 (0 51)
    clc               ; 0 01010001 (0 51)
    ror r0            ; 1 00101000 (1 28)
    clc               ; 0 00101000 (0 28)
    ror r0            ; 0 00010100 (0 14)
    sec               ; 1 00010100 (1 14)
    ror r0            ; 0 10001010 (0 8a)
    sts 0xf2, r0      ; port02 = 8a
    stop              ;)
```

C, R0

Quick exercise

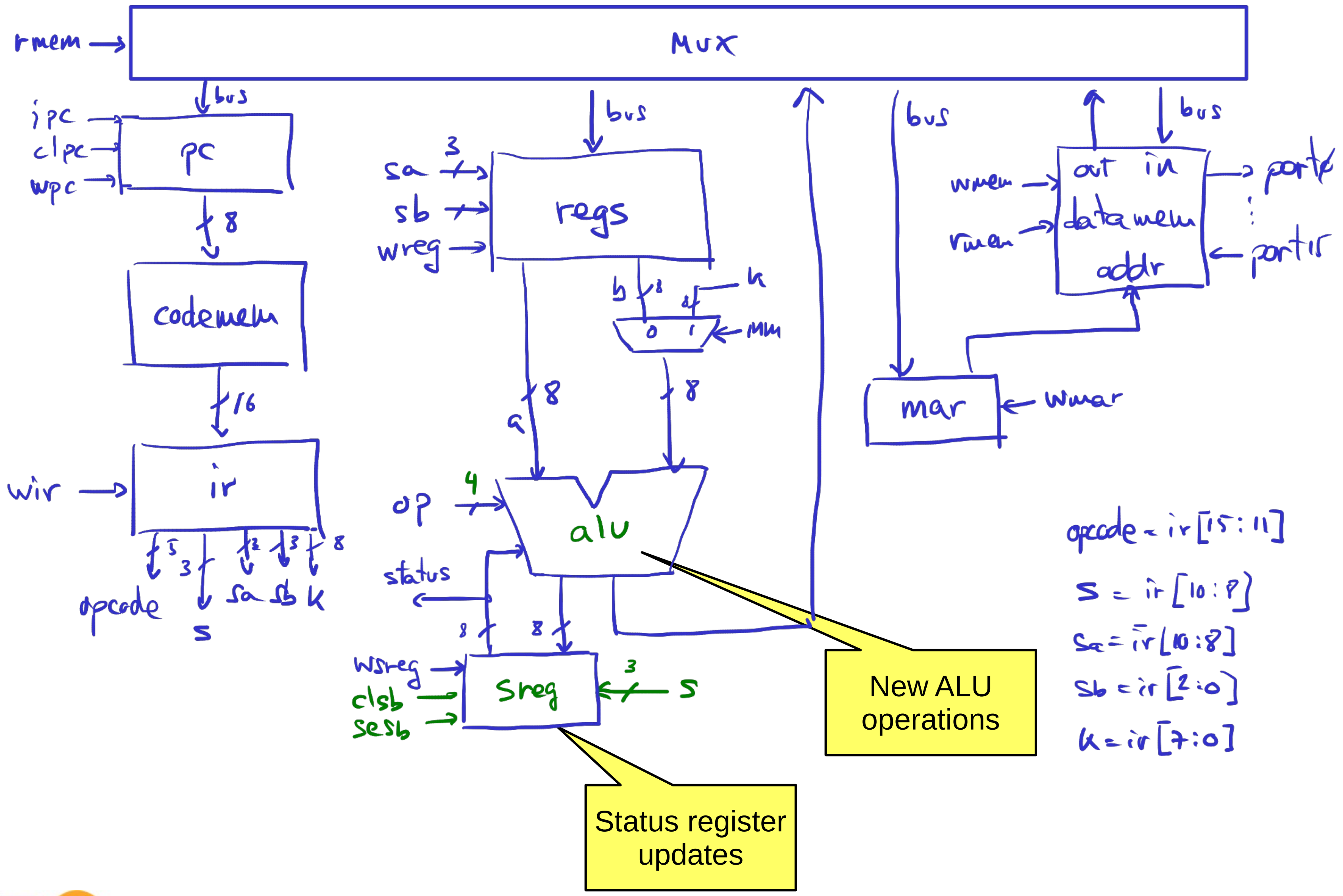
a) What is the output at ports 01 and 02 if port08 = 11010011 initially?

b) What are the real instruction for “CLC” and “SEC”?

We do not have simple shift instructions but:

- CLC + ROR → SHR
- CLC + ROL → SHL

YASAC Stage 4
Data unit



New ALU operations

Status register updates

opcode = ir[15:11]

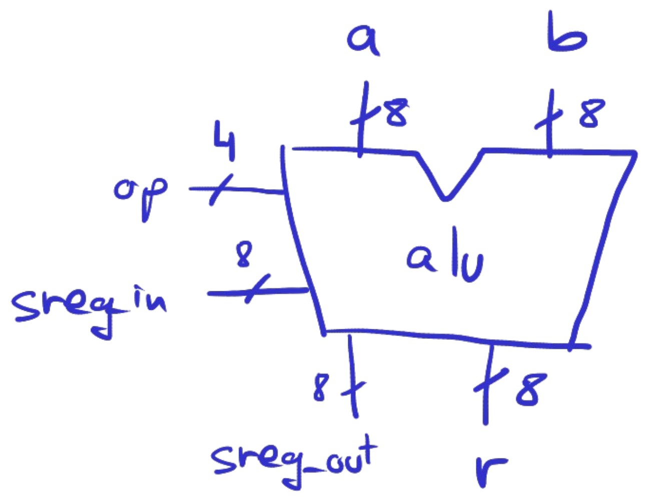
s = ir[10:8]

sa = ir[10:8]

sb = ir[2:0]

k = ir[7:0]

YASAC Stage 4 Updated ALU



		sreg				
op sym.	op	r	S	V	N	Z	C
ALU_ADD	0000	$a+b$	*	*	*	*	*
ALU_TRA	0001	a	-	-	-	-	-
ALU_SUB	0010	$a-b$	*	*	*	*	*
ALU_TRB	0011	b	-	-	-	-	-
ALU_NEG	0100	$-a$	*	*	*	*	*
ALU_AND	0101	AND(a,b)	*	*	*	*	-
ALU_OR	0110	OR(a,b)	*	*	*	*	-
ALU_EOR	0111	EOR(a,b)	*	*	*	*	-
ALU_ROR	1000	$SHR(a, c_{in})$	*	*	*	*	*
ALU_ROL	1001	$SHL(a, c_{in})$	*	*	*	*	*

NEG

$C_0 = \overline{OR(a)}$

$Z_0 = \overline{OR(r)}$

$N_0 = r(7)$

$V_0 = a_7 r_7 + \bar{a}_7 \bar{r}_7$

$S_0 = V_0 \oplus N_0$

AND, OR, EOR

$C_0 = C_i$

$Z_0 = \overline{OR(r)}$

$N_0 = r(7)$

$V_0 = 0$

$S_0 = V_0 \oplus N_0$

ROR

$C_0 = a_0$

$Z = \overline{OR(r)}$

$N = r(7)$

$V_0 = C_{in} \oplus a_0$

$S_0 = V_0 \oplus N_0$

ROL

$C_0 = a_7$

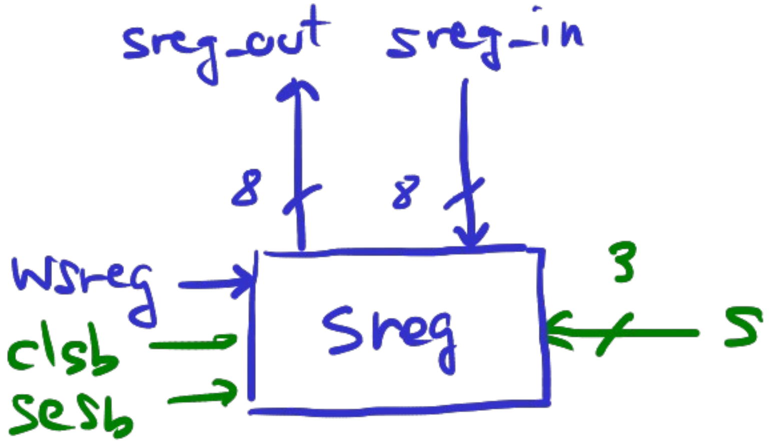
$Z = \overline{OR(r)}$

$N = r(7)$

$V_0 = a_7 \oplus a_6$

$S_0 = V_0 \oplus N_0$

YASAC Stage 4
Updated/new data unit blocks



wsreg : sreg ← sreg_in

clsb : sreg[s] ← 0

sesb : sreg[s] ← 1

YASAC Stage 4
Control unit. Micro-operations

Logic instructions

AND Ra,Rb 1) Ra $\leftarrow$ Ra & Rb ; sreg $\leftarrow$ sreg_out (op: ALU_AND, wreg, wsreg)

OR Ra, Rb 1) Ra $\leftarrow$ Ra | Rb ; sreg $\leftarrow$ sreg_out (op: ALU_OR, wreg, wsreg)

EOR Ra, Rb 1) Ra $\leftarrow$ Ra ^ Rb ; sreg $\leftarrow$ sreg_out (op: ALU_EOR, wreg, wsreg)

Shifting (rotating) instructions

ROR Ra 1) Ra $\leftarrow$ SHR(Ra, c) ; sreg $\leftarrow$ sreg_out (op: ALU_ROR, wreg, wsreg)

ROL Ra 1) Ra $\leftarrow$ SHL(Ra, c) ; sreg $\leftarrow$ sreg_out (op: ALU_ROL, wreg, wsreg)

Status update instructions

BCLR s 1) sreg[s] $\leftarrow$ 0 clsb

BSET s 1) sreg[s] $\leftarrow$ 1 sesb

All new instructions are executed in a single clock cycle.

YASAC Stage 4
Control unit. States and control table

	READY	FETCH	EXEC1	EXEC2
LDI Ra, k	ready start: clpc	wir, ipc	op=0011, wreg, inm →FETCH	
MOV Ra, Rb			op=0011, wreg, →FETCH	
ADD Ra, Rb			op=0000, wreg, wsreg →FETCH	
SUB Ra, Rb			op=0010, wreg, wsreg →FETCH	
STOP			→READY	
LD Ra, Rb			op=0011, wmar	rmem, wreg →FETCH
ST Rb, Ra			op=0011, wmar	op=0001, wmem →FETCH
LDS Ra, k			op=0011, wmar, inm	rmem, wreg →FETCH
STS k, Ra			op=0011, wmar, inm	op=0001, wmem →FETCH
JMP k			inm, op=0011, wpc	
BRBS s, k			status[s]: inm, op=0011, wpc →FETCH	
BRBC s, k			~status[s]: inm, op=0011, wpc →FETCH	

Old instructions only update the width of the ALU’s operation code.

YASAC Stage 4 Control unit. States and control table

	READY	FETCH	EXEC1
AND Ra, Rb	ready start: clpc	wir, ipc	op=0101, wreg, wsreg →FETCH
OR Ra, Rb			op=0110, wreg, wsreg →FETCH
EOR Ra, Rb			op=0111, wreg, wsreg →FETCH
ROR Ra			op=1000, wreg, wsreg →FETCH
ROL Ra			op=1001, wreg, wsreg →FETCH
BCLR s			clsb →FETCH
BSET s			sesb →FETCH

With this table we can update the Verilog code (do not need to draw an ASM chart).

YASAC Stage 4
Verilog coding. Try it yourself!

- Update the data unit modules that have changed:
  - ALU (alu.v)
  - Status register
- Update the control unit with the new instructions
- Update the input/output interface of the control and data units and update its interconnection (yasac.v).
- Write a simple program that uses the new instructions.
  - Or use the example in a previous slide.
- Simulate:
  - Resolve syntax and compiler problems.
  - Check result.
  - If not correct, debug with Gtkwave.

YASAC Stage 5

- Stage 4 limitations
  - No subroutine support and no stack.
  - (A few more…)
- Why subroutines?
  - Repetitive tasks.
  - Code re-usability.
  - Recursion.

With these modifications, the YASAC ISA is ready to implement complex programs in assembly, like programs to support higher-level languages: parsers, assemblers, compilers, etc.

- Why a stack?
  - It is a convenient way to store data needed to implement subroutines.
  - It is also useful to save register contents to memory and recall it later.
  - Greatly simplifies the job of programs that implement higher-level languages (compilers):
    - Parameter passing.
    - Local data (to a function or method).

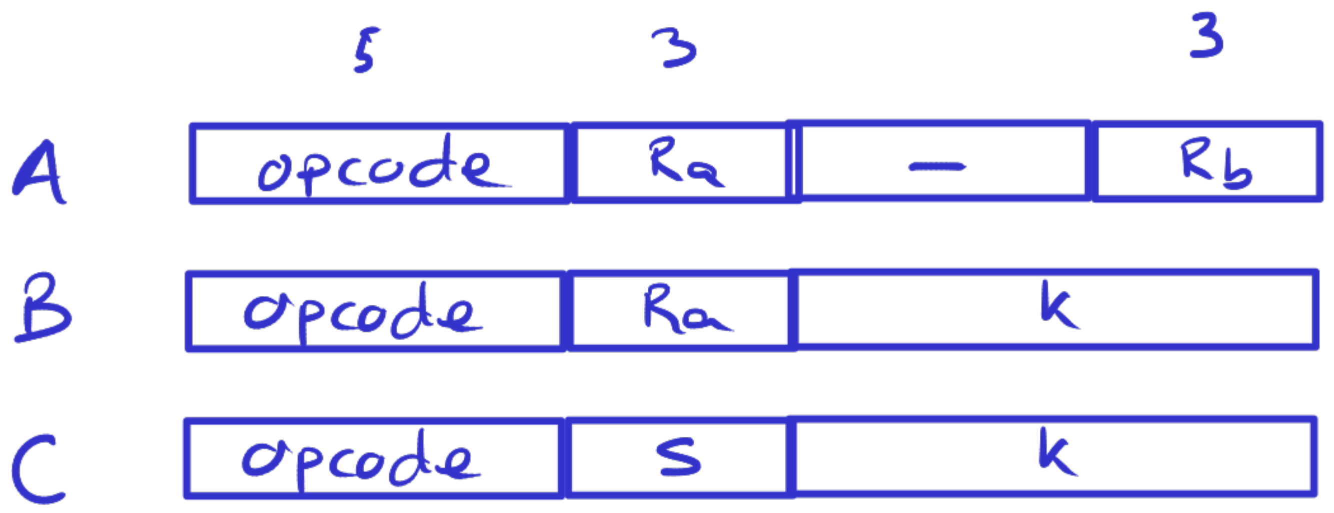
Stacks

- A type of Last-In, First-Out memory (LIFO).
- Two operations:
  - PUSH: stores data in the stack.
  - POP: retrieves the last data pushed to the stack, not previously popped.
- Implementation in computers
  - Use RAM memory.
  - Start at a stack's base memory address.
  - Use a stack pointer register to store the current stack's top address.
  - Make the stack to grow downwards to maximize memory usage.
  - Be careful not to overrun your program's code or date by growing the stack too much.
    - Do not worry, modern processors have mechanisms to handle it.

YASAC Stage 5
Needed changes

- Stack pointer register.
- Program counter read operation.
  - Needed to save it onto the stack when CALL.
- Control unit:
  - Implement new instructions.

YASAC Stage 5
Instruction format (same as stage 4)



YASAC Stage 5 Instruction set

Op. code	Instruction	RTL	SVNZC
00001	LDI Ra, k	Ra ← k	-----
00010	MOV Ra, Rb	Ra ← Rb	-----
00011	ADD Ra, Rb	Ra ← Ra + Rb	*****
00100	SUB Ra, Rb	Ra ← Ra - Rb	*****
00101	STOP	–	-----
00110	LD Ra, Rb	Ra ← datamem(Rb)	-----
00111	ST Rb, Ra	datamem(Rb) ← Ra	-----
01000	LDS Ra, k	Ra ← datamem(k)	-----
01001	STS k, Ra	datamem(k) ← Ra	-----
01010	JMP k	PC ← k	-----
01011	BRBS s, k	status[s]: PC ← k	-----
01100	BRBC s, k	~status[s]: PC ← k	-----
01101	AND Ra, Rb	Ra ← Ra & Rb	*****
01110	OR Ra, Rb	Ra ← Ra | Rb	*****
01111	EOR Ra, Rb	Ra ← Ra ^ Rb	*****
10000	ROR Ra	Ra ← SHR(Ra,C)	*****
10001	ROL Ra	Ra ← SHL(Ra,C)	*****
10010	BCLR s	sreg[s] ← 0	*****
10011	BSET s	sreg[s] ← 1	*****
10100	PUSH Ra	datamem[sp] ← Ra; sp ← sp - 1	-----
10101	POP Ra	Ra ← datamem[sp+1]; sp ← sp + 1	-----
10110	CALL k	datamem[sp] ← pc; pc ← k; sp ← sp - 1	-----
10111	RET	pc ← datamem[sp+1]; sp ← sp + 1	-----

YASAC Stage 5
Sample program

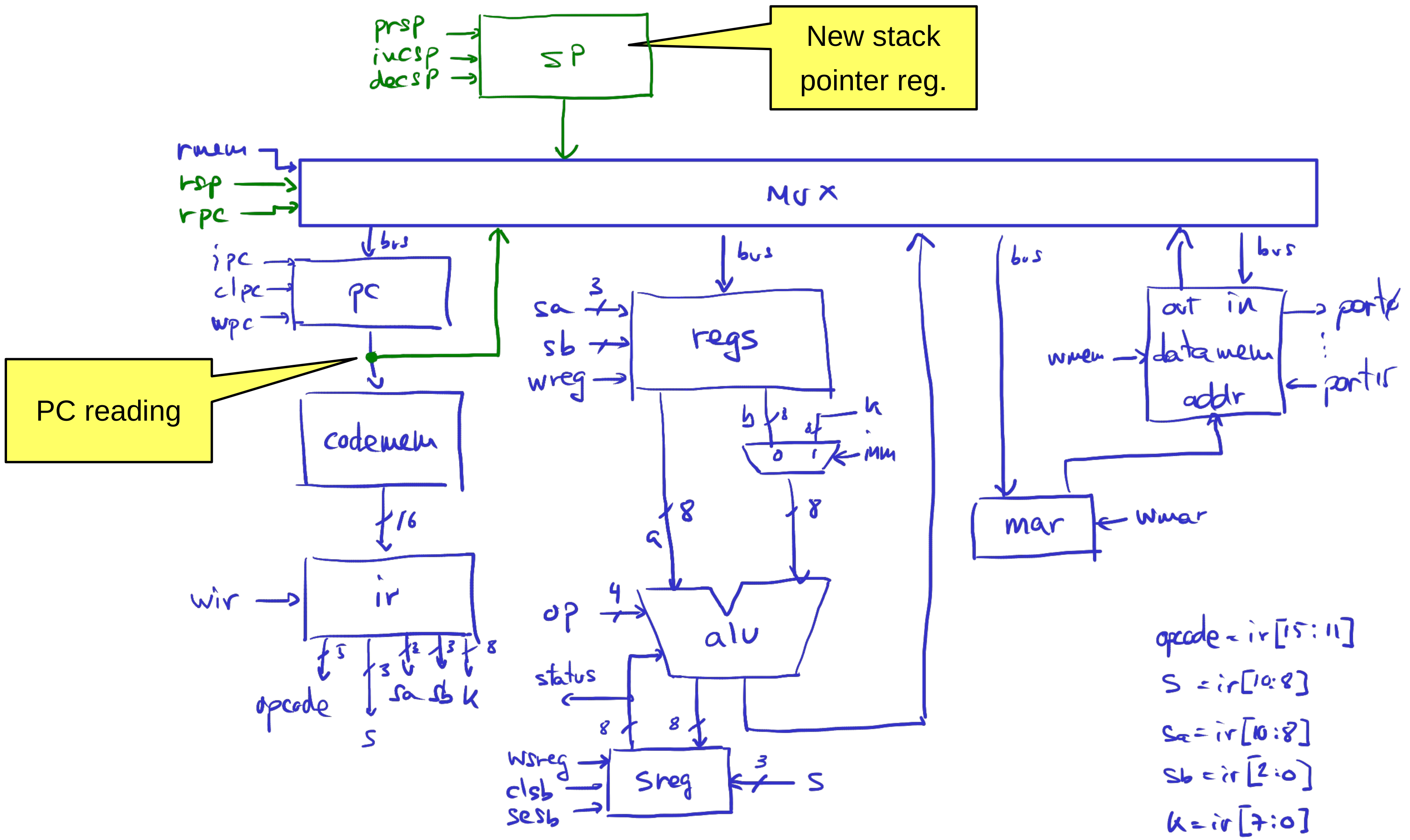
```
; input:  port08 = 0x5a (01011010)
; output: port01 = ?
;         port02 = ?
;
        lds r0, 0xf8    ; r0 = port08
        ldi r1, 0xff
        push r0
        pop r1
        call send
        stop
        stop
        stop
        stop
send:   sts 0xf1, r1
        sts 0xf2, r0
        ret
        stop
```

Quick exercise

a) What is the expected output at port01 and port02?

b) How many instruction in the sample code are never executed?

YASAC Stage 5 Data unit



opcode = ir[15:11]

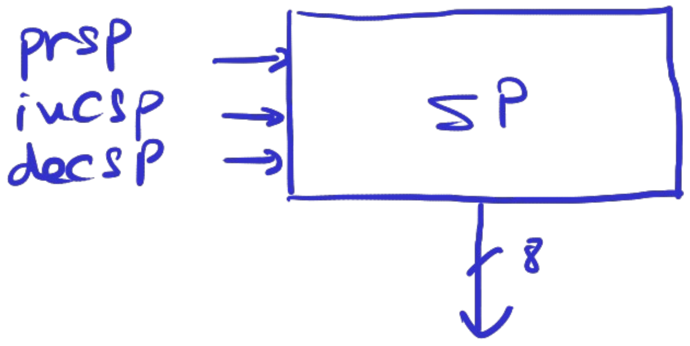
s = ir[10:8]

sa = ir[10:8]

sb = ir[2:0]

k = ir[7:0]

YASAC Stage 5
New stack pointer register



prsp : sp ← STACK_BASE

decsp : sp ← sp - 1

incsp : sp ← sp+1

STACK_BASE = 0xEF

YASAC Stage 5
Control unit. Micro-operations

PUSH Ra $\quad$ data_mem(sp) ← Ra ; sp ← sp - 1

1)	mar ← sp ; sp ← sp - 1	rsp , wmar, decsp
2)	data_mem (mar) ← Ra	op = ALU_TRA , wmem

POP Ra $\quad$ Ra ← data_mem (sp+1) ; sp ← sp + 1

1)	sp ← sp + 1	incsp
2)	mar ← sp	rsp , wmar
3)	Ra = data_mem (mar)	rmem , wreg

Contents

YASAC Stage 5
Control unit. Micro-operations

CALL k	data_mem (sp) ← pc ; pc ← k ; sp ← sp − 1	
1)	mar ← sp ; sp ← sp − 1	rsp, wmar, decsp
2)	data_mem (mar) ← pc	rpc, wmem
3)	pc ← k	inm, op = ALU_TRB, wpc

RET	pc ← data_mem (sp + 1) ; sp ← sp + 1	
1)	sp ← sp + 1	incsp
2)	mar ← sp	rsp, wmar
3)	pc ← data_mem (mar)	rmem, wpc

YASAC Stage 5
Control unit. States and control table

Instruction	EXEC1	EXEC2
LDI Ra, k	op=0011, wreg, inm	
MOV Ra, Rb	op=0011, wreg	
ADD Ra, Rb	op=0000, wreg, wsreg	
SUB Ra, Rb	op=0010, wreg, wsreg	
STOP		
LD Ra, Rb	op=0011, wmar	rmem, wreg
ST Rb, Ra	op=0011, wmar	op=0001, wmem
LDS Ra, k	op=0011, wmar, inm	rmem, wreg
STS k, Ra	op=0011, wmar, inm	op=0001, wmem
JMP k	inm, op=0011, wpc	
BRBS s, k	status[s]: inm, op=0011, wpc	
BRBC s, k	~status[s]: inm, op=0011, wpc	

NOTE:
- READY and FETCH states have been omitted.
- All instructions go back to FETCH except STOP that returns to READY.

YASAC Stage 5
Control unit. States and control table

	EXEC1	EXEC2	EXEC3
AND Ra, Rb	op=0101, wreg, wsreg		
OR Ra, Rb	op=0110, wreg, wsreg		
EOR Ra, Rb	op=0111, wreg, wsreg		
ROR Ra	op=1000, wreg, wsreg		
ROL Ra	op=1001, wreg, wsreg		
BCLR s	clsb		
BSET s	sesb		
PUSH Ra	rsp, wmar, decsp	op=ALU_TRA, wmem	
POP Ra	incsp	rsp, wmar	rmem, wreg
CALL k	rsp, wmar, decsp	rpc, wmem	inm, op=ALU_TRB, wpc
RET	incsp	rsp, wmar	rmem, wpc

NOTE:
- EXEC1 is the same for (PUSH, CALL) and (POP, RET).
- EXEC2 is the same for (POP, RET).

YASAC Stage 5
Verilog coding. Try it yourself!

- Update the data unit:
  - Add stack pointer register.
  - Add PC out.
- Update the control unit with the new instructions.
- Update the input/output interface of the control and data units and update its interconnection (yasac.v).
- Write a simple program that uses the new instructions.
  - Or use the example in a previous slide.
- Simulate:
  - Resolve syntax and compiler problems.
  - Check result.
  - If not correct, debug with Gtkwave.

What’s next?
Stage 6. Interrupts

- Interrupt lines are activated by external devices.
- The computer executes an interrupt service routine every time an external interrupt is activated.
  - Service routines are very similar to subroutines.
- Interrupts may be disabled.
- Greatly simplifies input/output programming:
  - No need to poll input ports in a loop, the external device will activate an interrupt when new data is available.
- Greatly simplifies common tasks:
  - A timer can activate an interrupt at regular intervals to update the time, check a keyboard, update output ports, etc.
- The basis for multitasking:
  - Periodic interrupts can be used to execute a task scheduler that will switch from one task to another.

What's next?
Stage 7. Bootloading

- Currently, the program is fixed an can only be changed by re-programming the FPGA chip.
- Bootloading: ability of the computer to load programs from the outside world.
- Two strategies:
  - Programming interface:
    - The control unit has a "programming" mode that reads data form the outside world and writes to the program memory.
    - Needs additional hardware, an external programmer device and external programming software.
  - Software bootloader and serial port:
    - The initial program in the computer will load a new program from the outside world by using some peripheral (e.g. a serial port).
    - Needs a hardware peripheral (serial port controller), instructions to load and store data from/to program memory (LDPM, STPM) and external programming software.

What’s next? System improvements

- Pulse-Width Modulation (PWM) output:
  - Light control, motor and servo control, etc.
- Serial communications: RS-232, SPI, I2C
  - Many peripherals use these: sensors, LCD screens, keyboards, etc.
- Timer (specially if we implement interrupts)
  - Measure time, do things at regular intervals, etc.

What's next? Toolchain

- Toolchain
  - Software tool used for software or hardware development.
- Basic toolchain:
  - Assembler: converts assembly code into machine code.
  - Programmer/uploader: uploads machine code to the computer using a programing interface or bootloader.
- Other typical toolchain programs:
  - C language compiler: entry point to high-level programming.
  - Linker: combines machine code fragments, library functions, etc.
  - Debugger: executes programs in a controlled way in order to find errors (bugs).
  - Object file (machine code) inspection: disassembly, etc.

We'll see more on this in the next unit with a real computer.