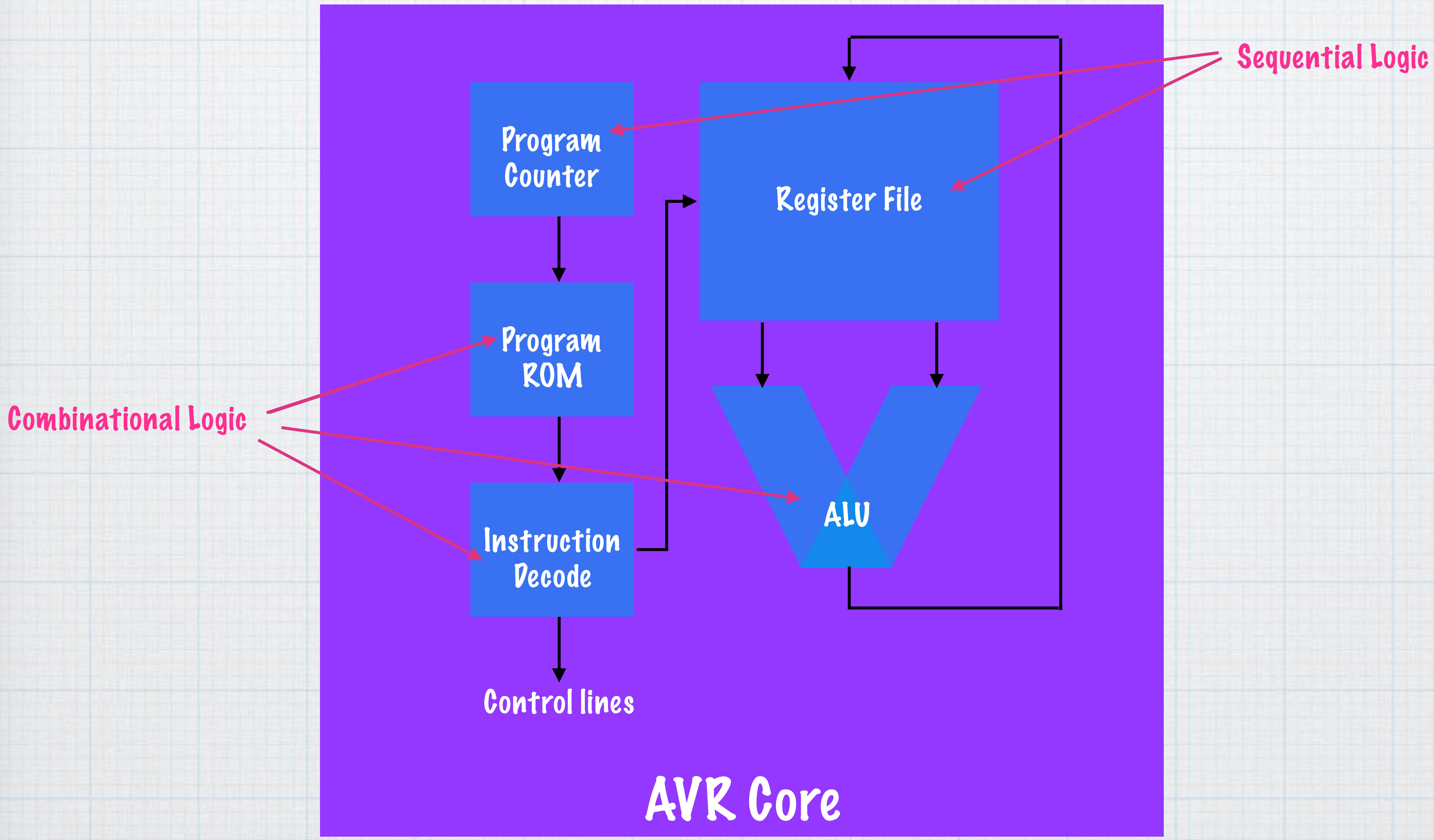
MHV FPGA Workshop

Part 2

Workshop Goals

- To learn about
  - FPGA's
  - Verilog
  - Digital design
- Simplified AVR processor as challenge

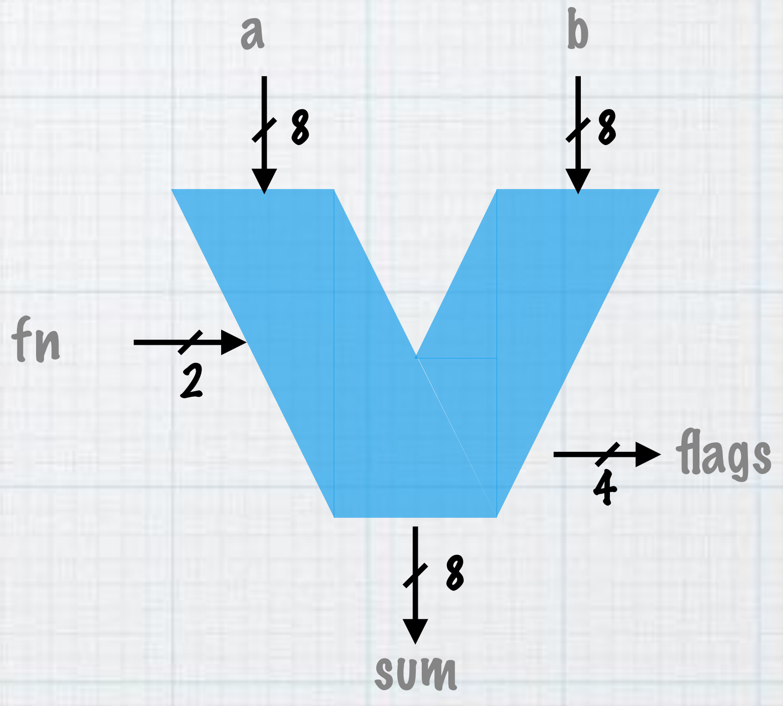
Roadmap


Sequential Logic
Program Counter
Register File
Program ROM
Combinational Logic
ALU
Instruction Decode
Control lines
AVR Core

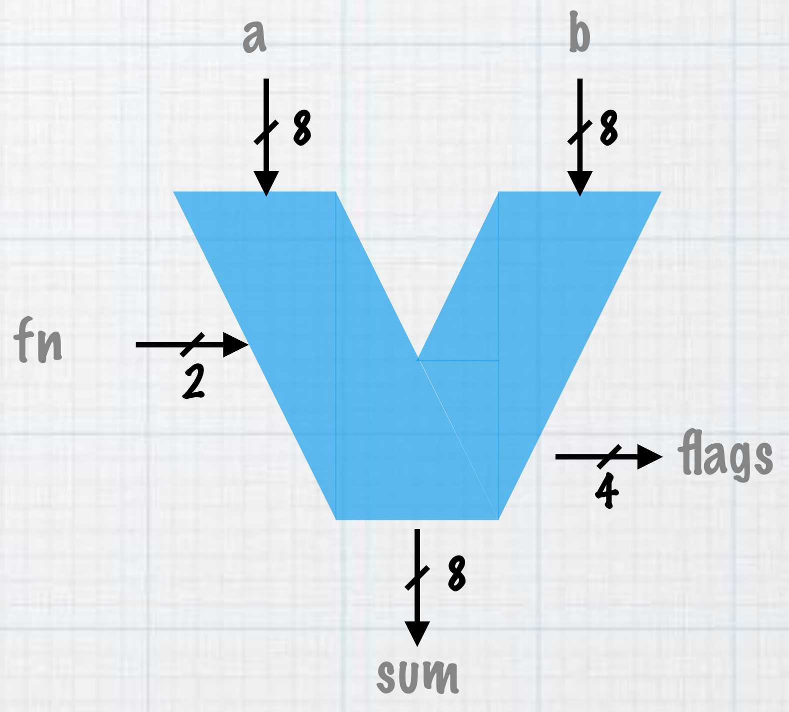
Simplified ALU

* Combinational logic

fn	name	sum
2'b00	AND	a & b
2'b01	OR	a | b
2'b10	XOR	a ^ b
2'b11	PASSB	b

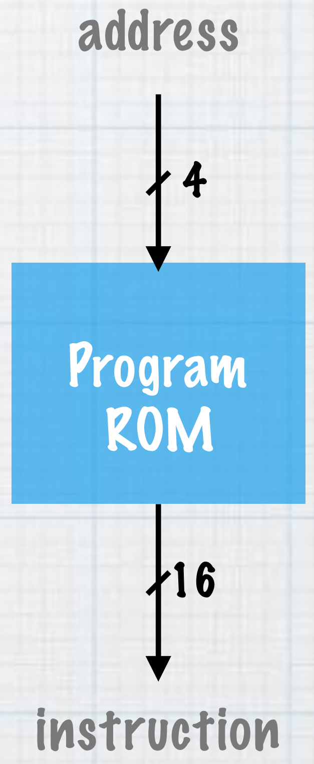


```
module logic(
    input        [1:0]   fn,
    input        [7:0]   a,
    input        [7:0]   b,
    output reg   [7:0]   sum,
    output               s, v, n, z
    );

    always @*
        case (fn)
            `ALUFN_AND:  sum = a & b;
            `ALUFN_OR:   sum = a | b;
            `ALUFN_XOR:  sum = a ^ b;
            default:     sum = b;
        endcase

    // Flag logic, …

endmodule
```



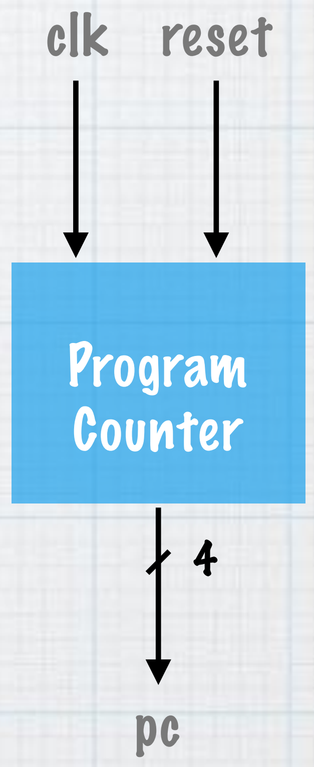
Program ROM

```
module prog1(
    input          [3:0]   a,      // address
    output reg     [15:0]  dout    // instruction
    );

    always @*
        case (a)
            4'h0:    dout = 16'hea05;   // ldi
            4'h1:    dout = 16'he01f;   // ldi
            4'h2:    dout = 16'h2301;   // and
            default: dout = 16'h0000;   // nop
        endcase

endmodule
```



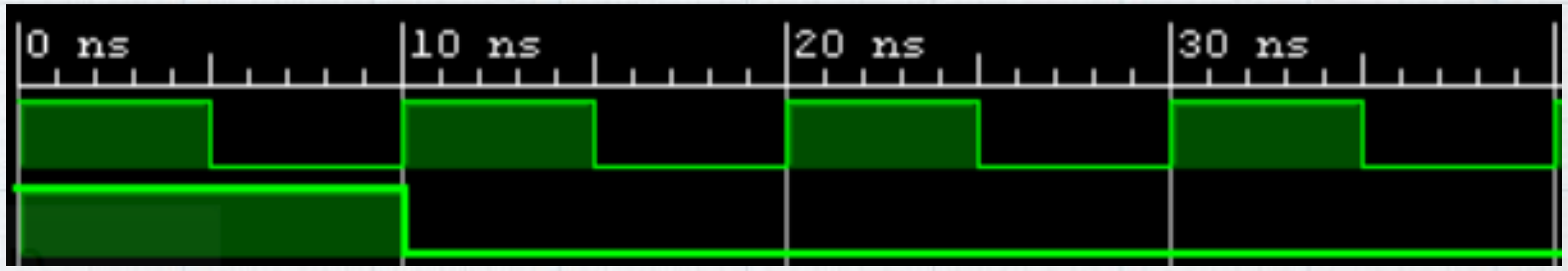
Program Counter

```
module pc(
    input                 clk,
    input                 reset,
    output reg   [3:0]    pc      // address
    );

    always @(posedge clk)
        if (reset)
            pc <= 0;
        else
            pc <= pc + 1;

endmodule
```



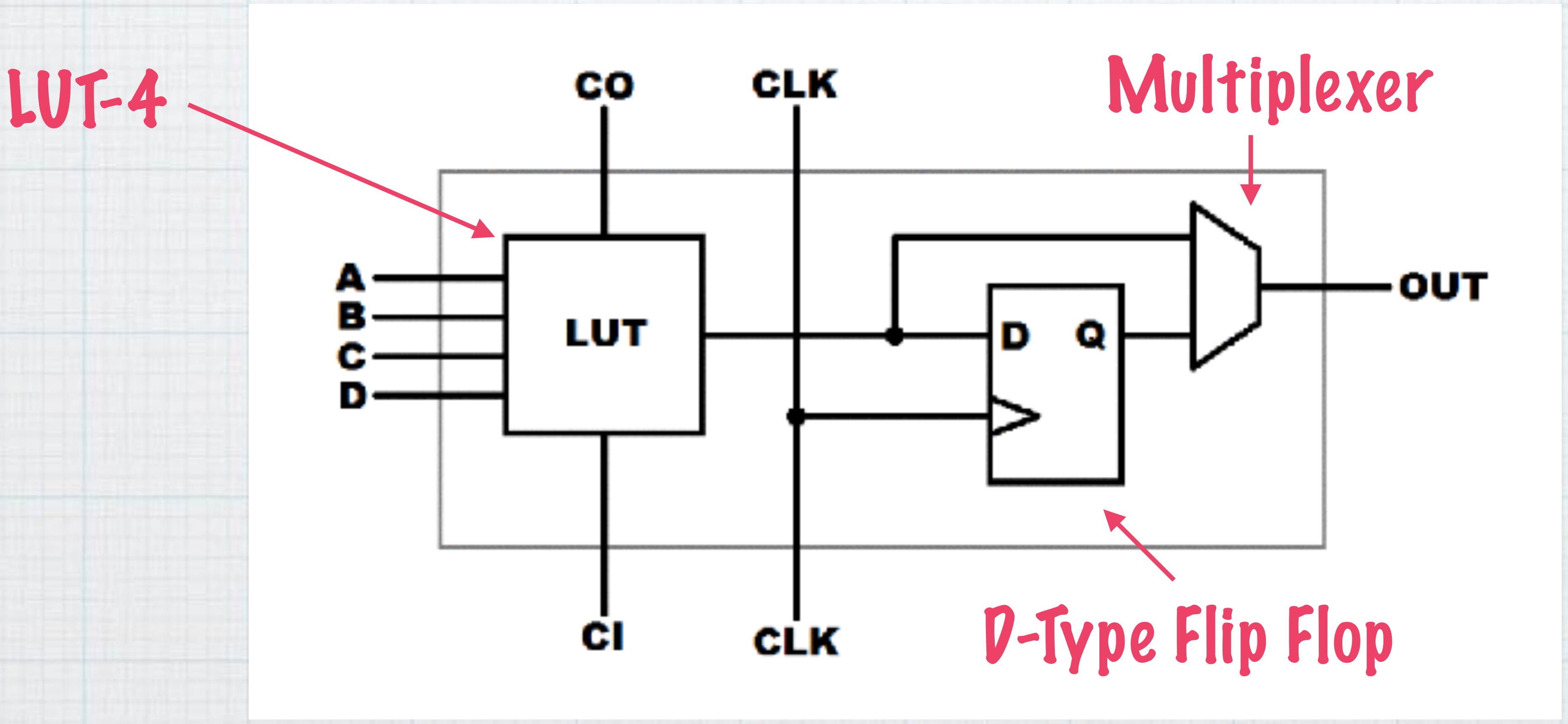
Sequential Logic

* Finite State Machines (FSM)
* Key signals
  * Clock
  * Reset



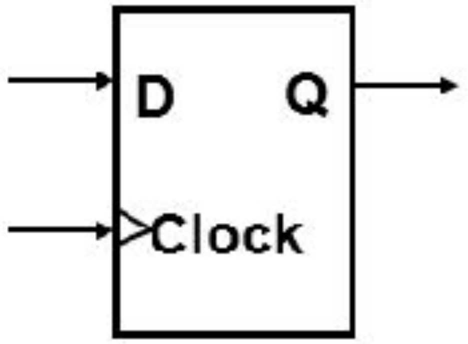
FPGA Logic Cell

* Combines LUT with Flip-Flop



D flip-flop

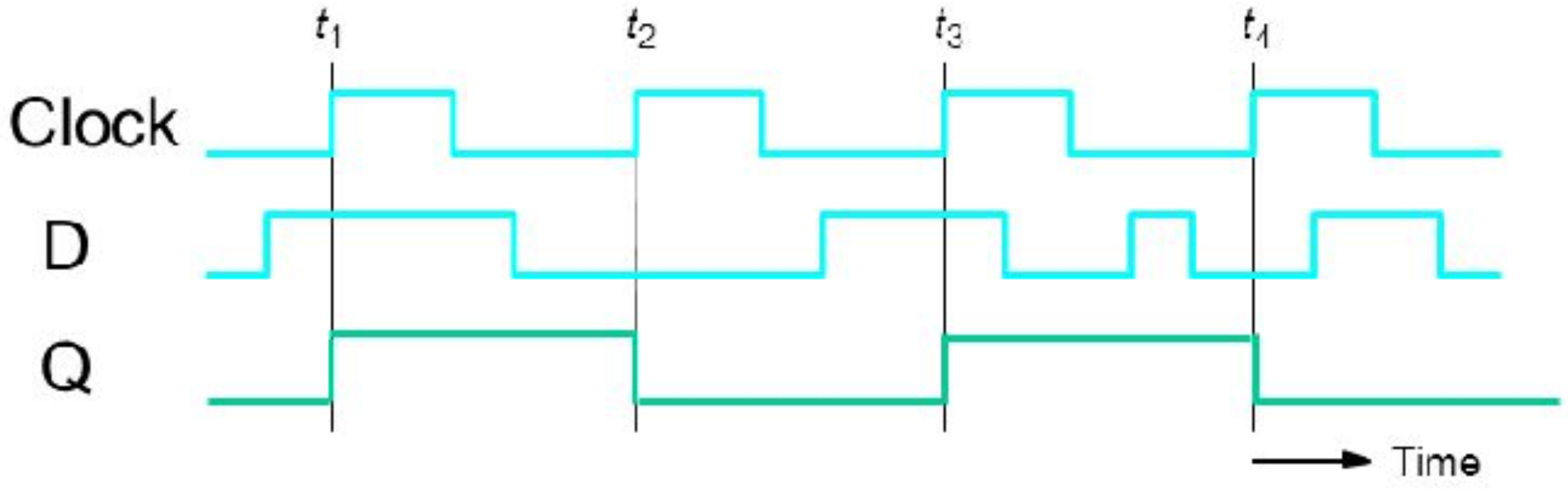
Graphical symbol



Truth table

Clk	D	$Q(t+1)$
↑	0	0
↑	1	1
0	–	$Q(t)$
1	–	$Q(t)$

Timing diagram



Sequential Verilog

* Edge sensitivity - **posedge, negedge**
* Non-blocking assignment, <=

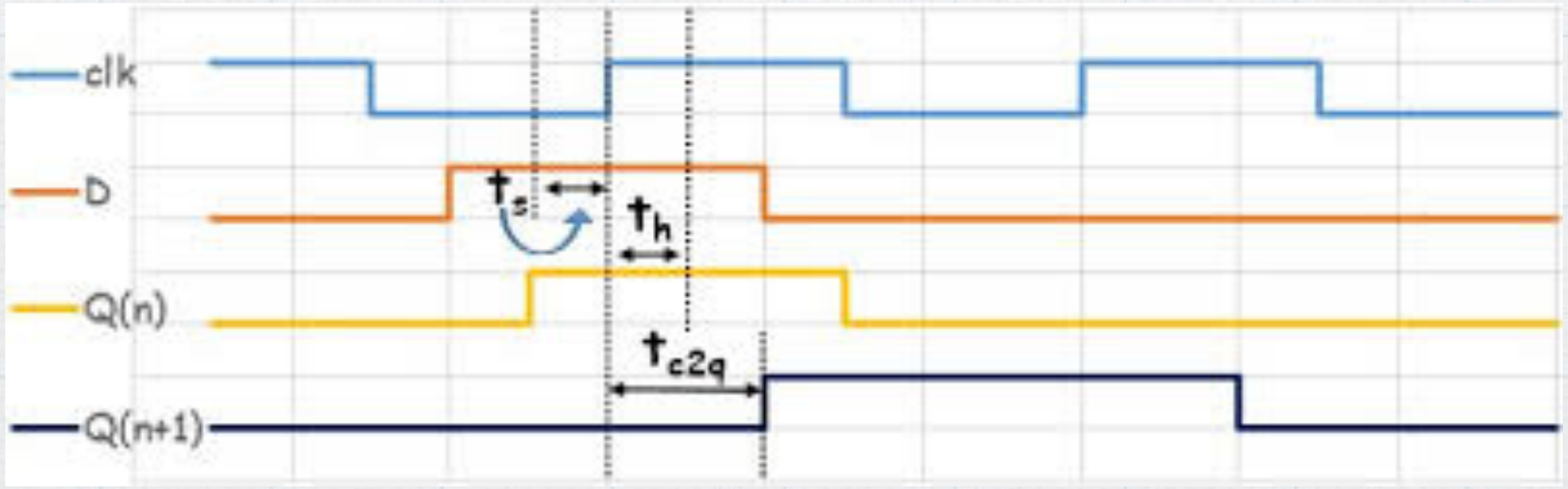
```
always @(posedge clk)
    q <= d;
```

Combinational - always use =

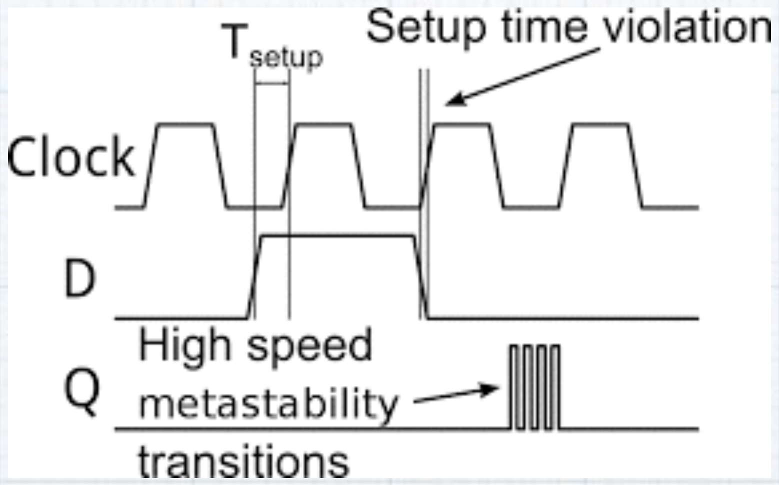
Sequential - always use <=

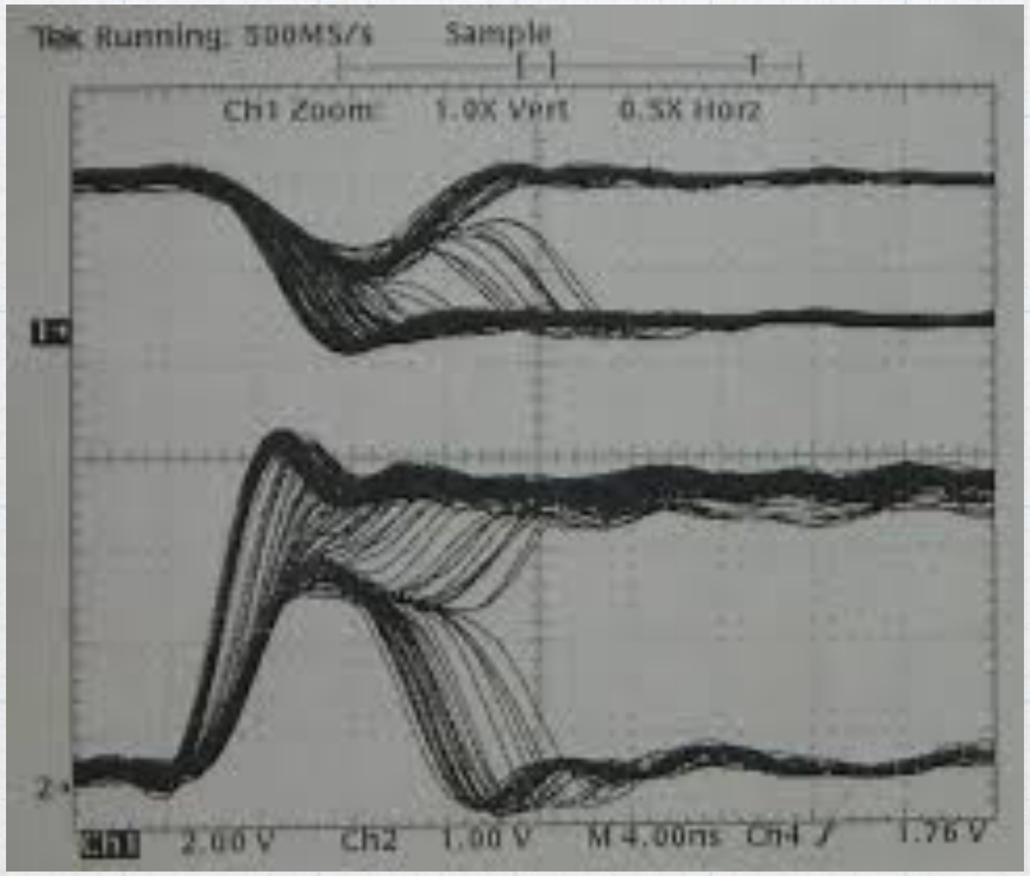
Flip Flop Timing

- Setup Time - t_{SU}
- Hold Time - t_h
- Propagation Delay - t_{PLH}, t_{PHL}



Metastability



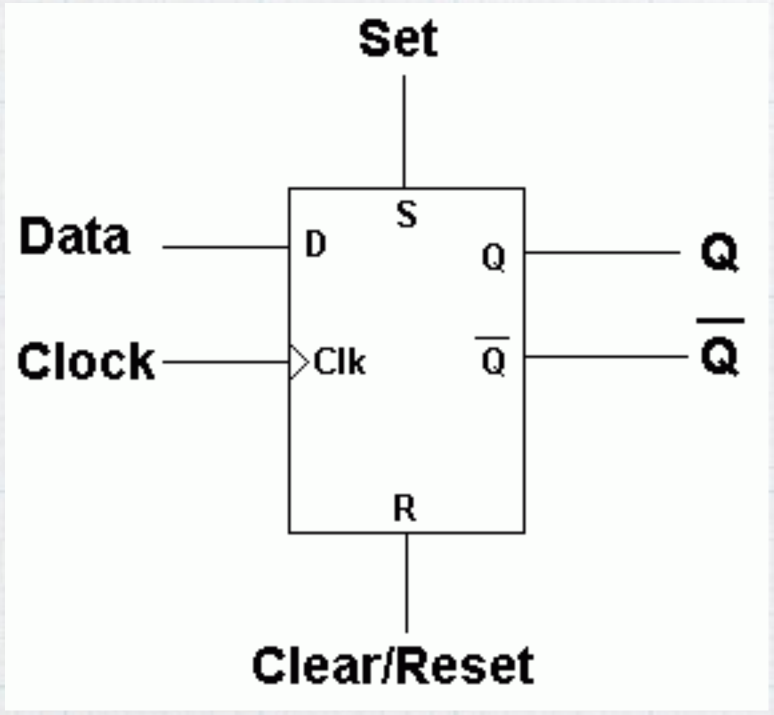


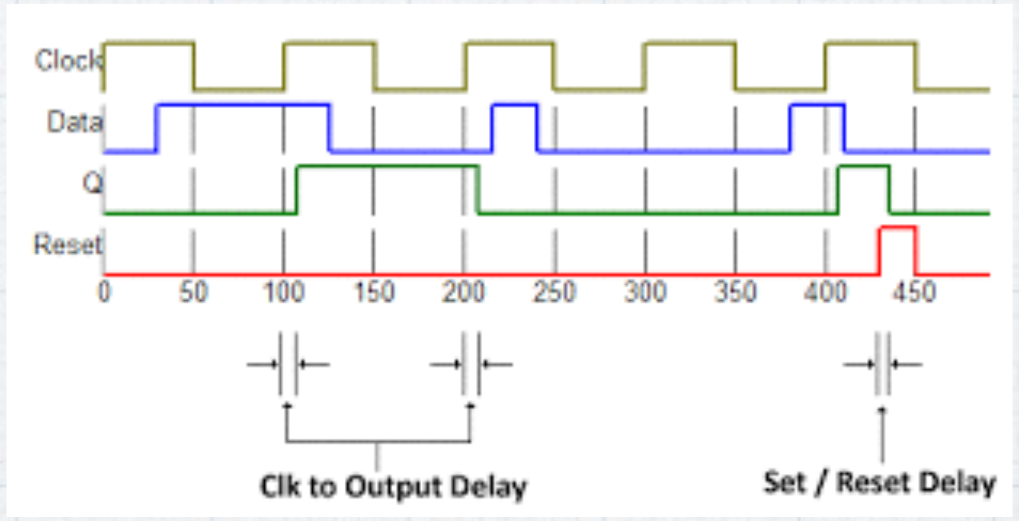
Lab 06 - FF Introduction

* Review Vivado simulation tools
* Verilog sequential testbench

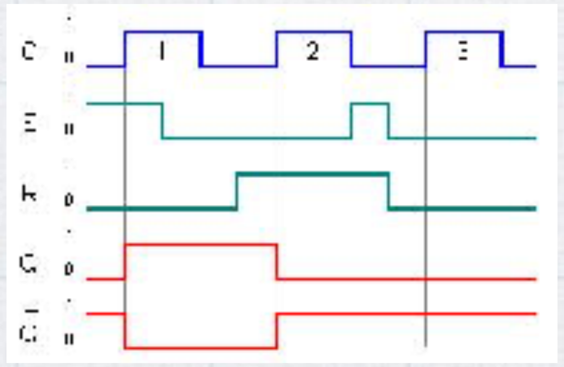
Set and Reset

* Asynchronous or Synchronous





Asynchronous



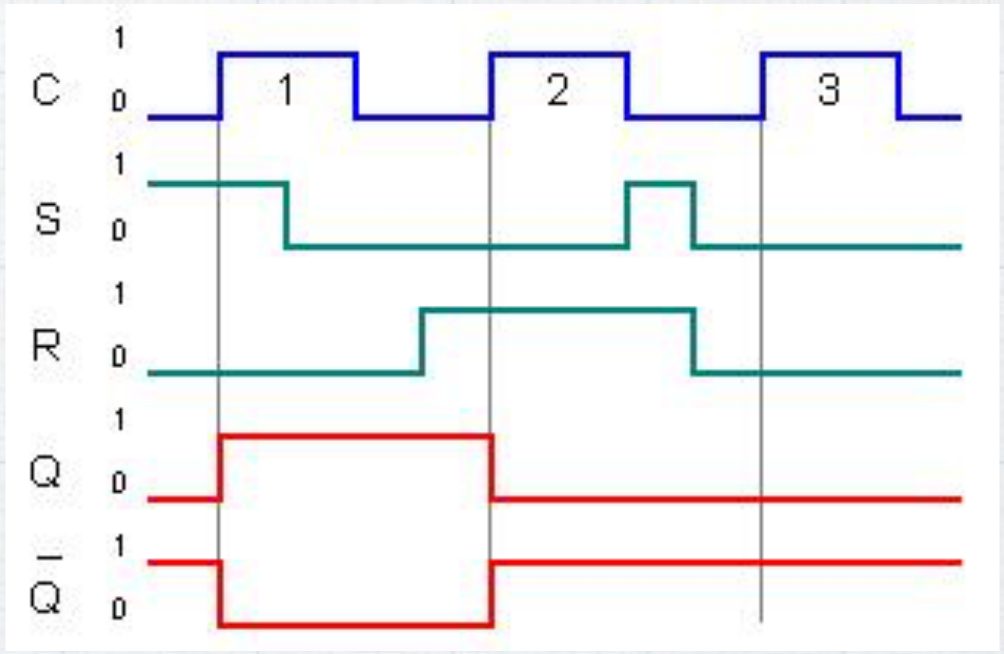
Synchronous

Synchronous Reset/Set

* Overrides D value

```
always @(posedge clk)
    if (reset)
        q <= 0;
    else
        q <= d;
```

```
always @(posedge clk)
    if (set)
        q <= 1;
    else
        q <= d;
```

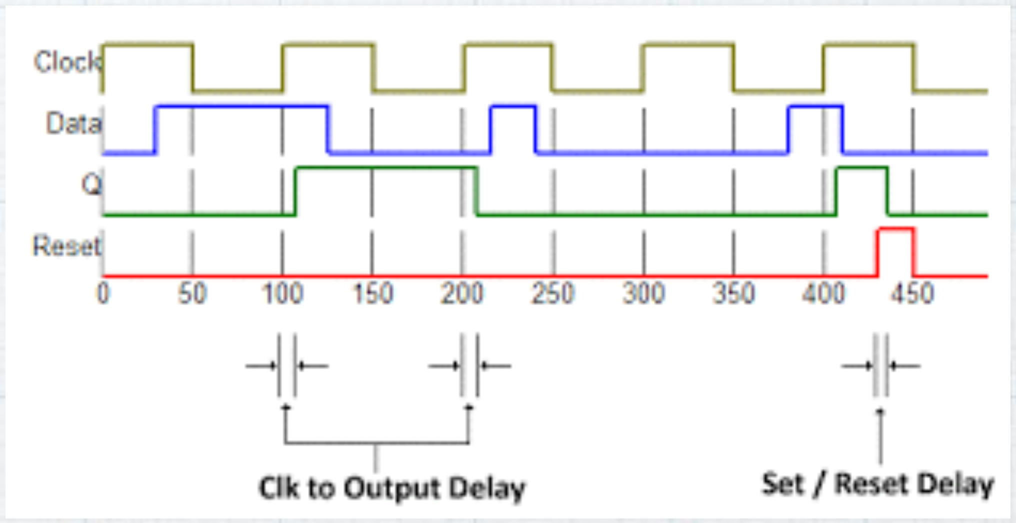


Asynchronous Set/Reset

* Terminology change - Preset/Clear

```
always @(posedge clk,
         posedge clear)
    if (clear)
        q <= 0;
    else
```

```
always @(posedge clk,
         posedge preset)
    if (preset)
        q <= 1;
    else
```



FPGA Clocking

* LOTs of synchronous elements
  * Flip flops
  * DSP's
  * Block RAMs
  * IO SerDes
* All need to be synchronised

Clocking Issues

* Skew
  * Same edge at different times
* Jitter
  * Random variation

7 Series Clocking

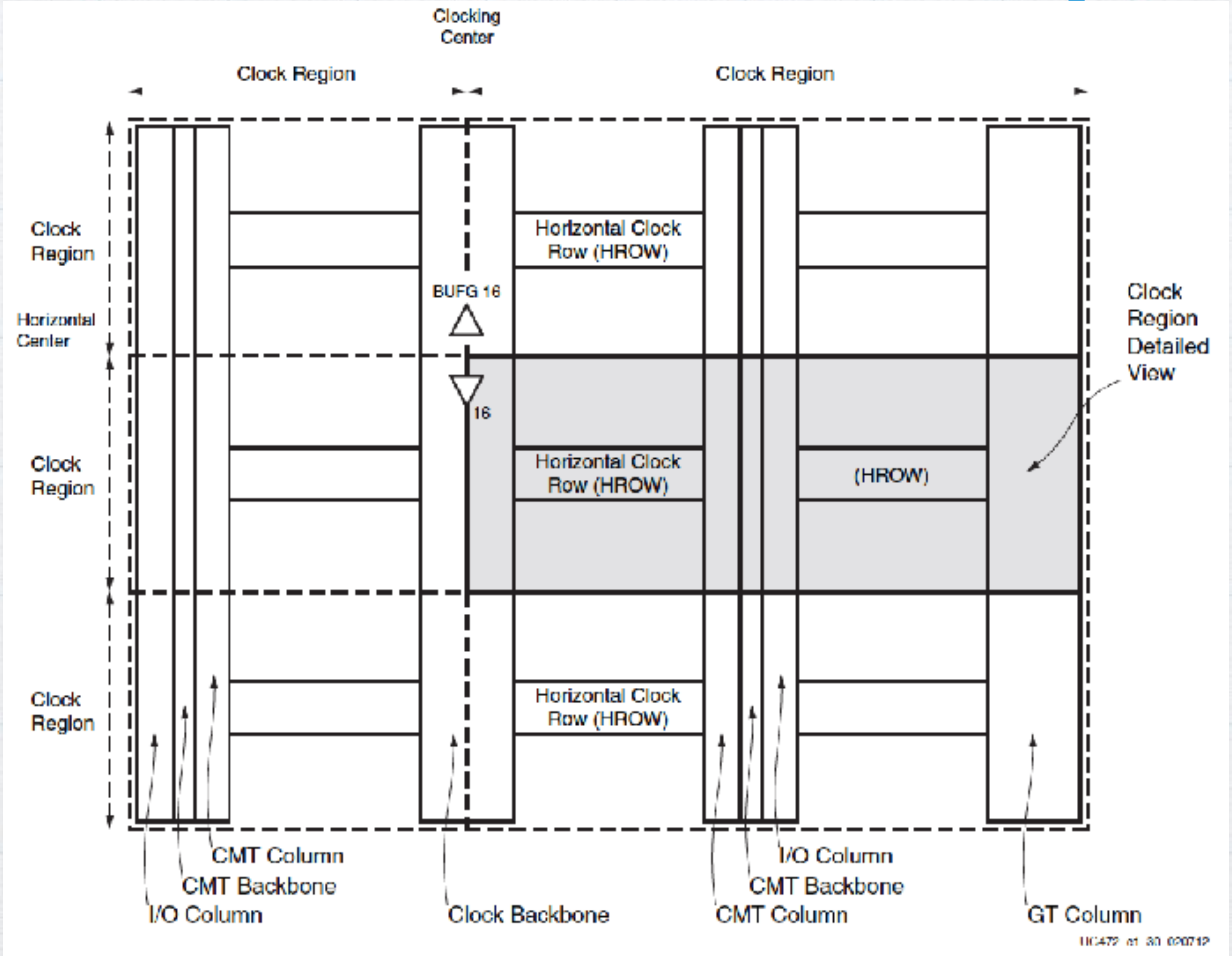


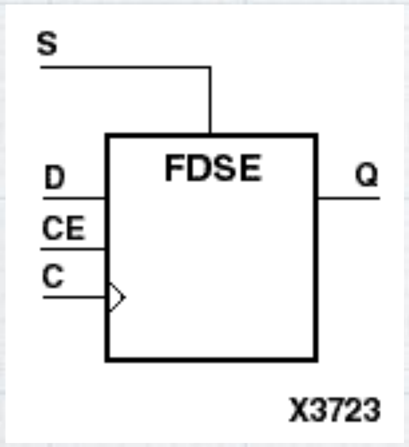
Figure 1-1: **7 Series FPGA High-Level Clock Architecture View**

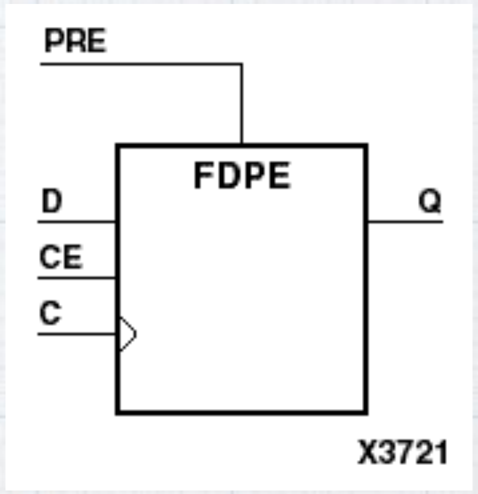
Clock Design Constraints

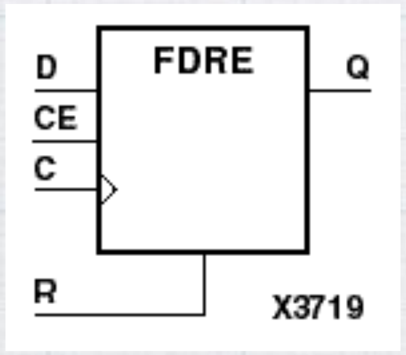
* 'Clock tree'
  * Limited # of clocks (BUFG)
* Power saving
  * CMOS power consumption

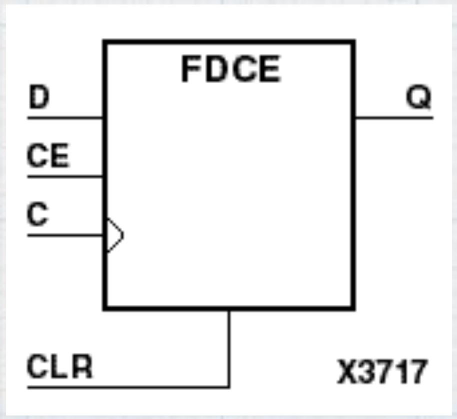
Clock Enable

* All 7 series FF primitives have a clock enable (CE) input





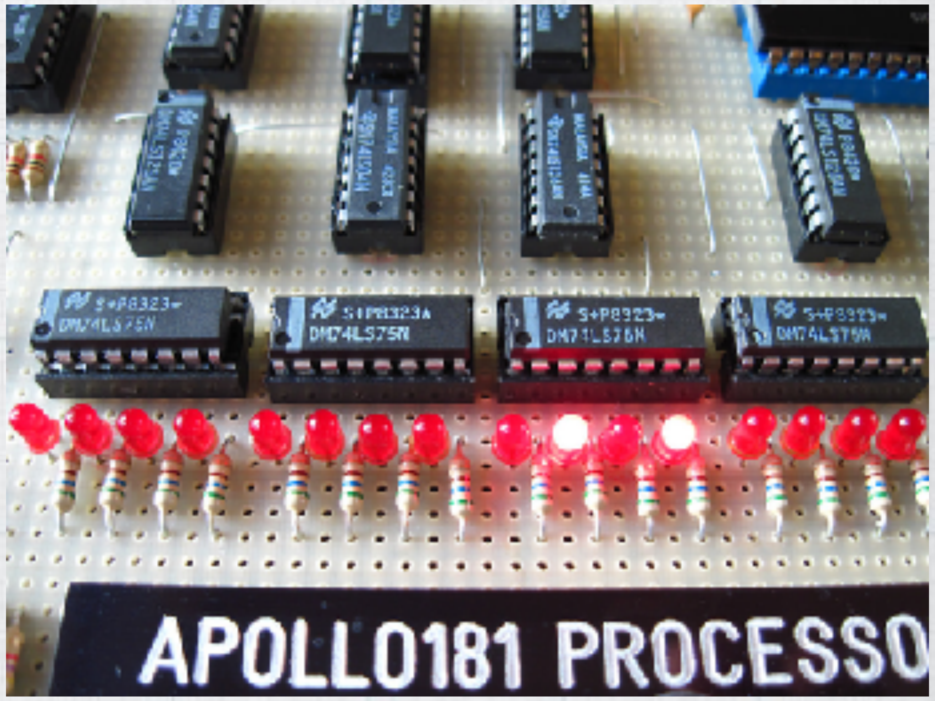




Instantiation

* 'Creates' a module instance
    * 'wires' a component into the circuit
    * Same as Object Oriented program

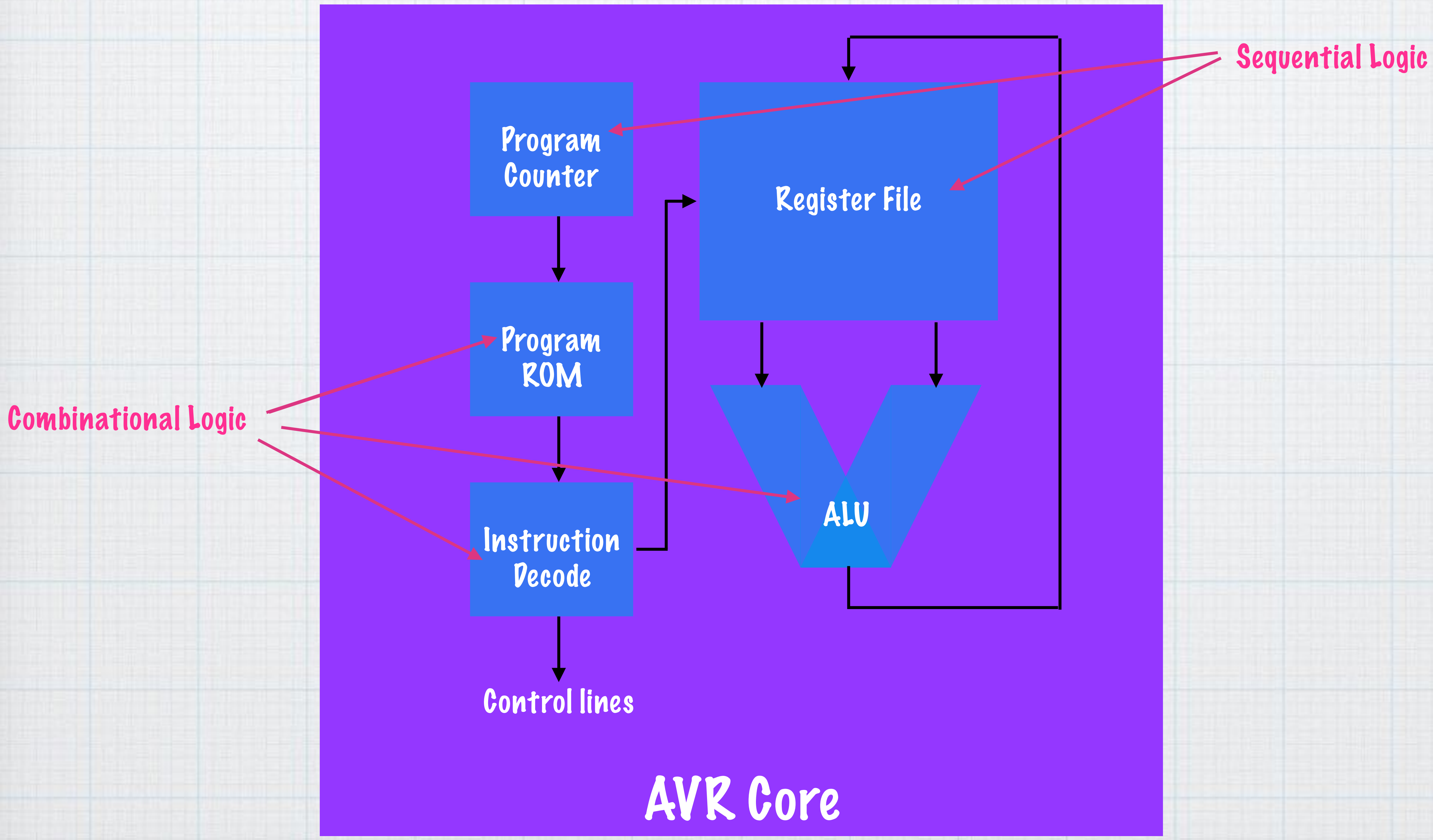
```
logic u1 (
    .fn(funk),
    .a(a), .b(b), .sum(out1),
    .s(s), .v(v), .n(n), .z(z)
    );
```



Lab 07 - FF Controls

* Synchronous reset
* Asynchronous clear
* Clock enable

Roadmap



Sequential Logic
Program Counter
Register File
Program ROM
Combinational Logic
ALU
Instruction Decode
Control lines
AVR Core

AVR Register File

- Memory locations within processor core
  - Calculation "ground zero"
- 32 x 8 bit registers
  - Names are R_0 to R_{31}
- Two register banks
  - R_0 - R_{15}
  - R_{16} - R_{31}

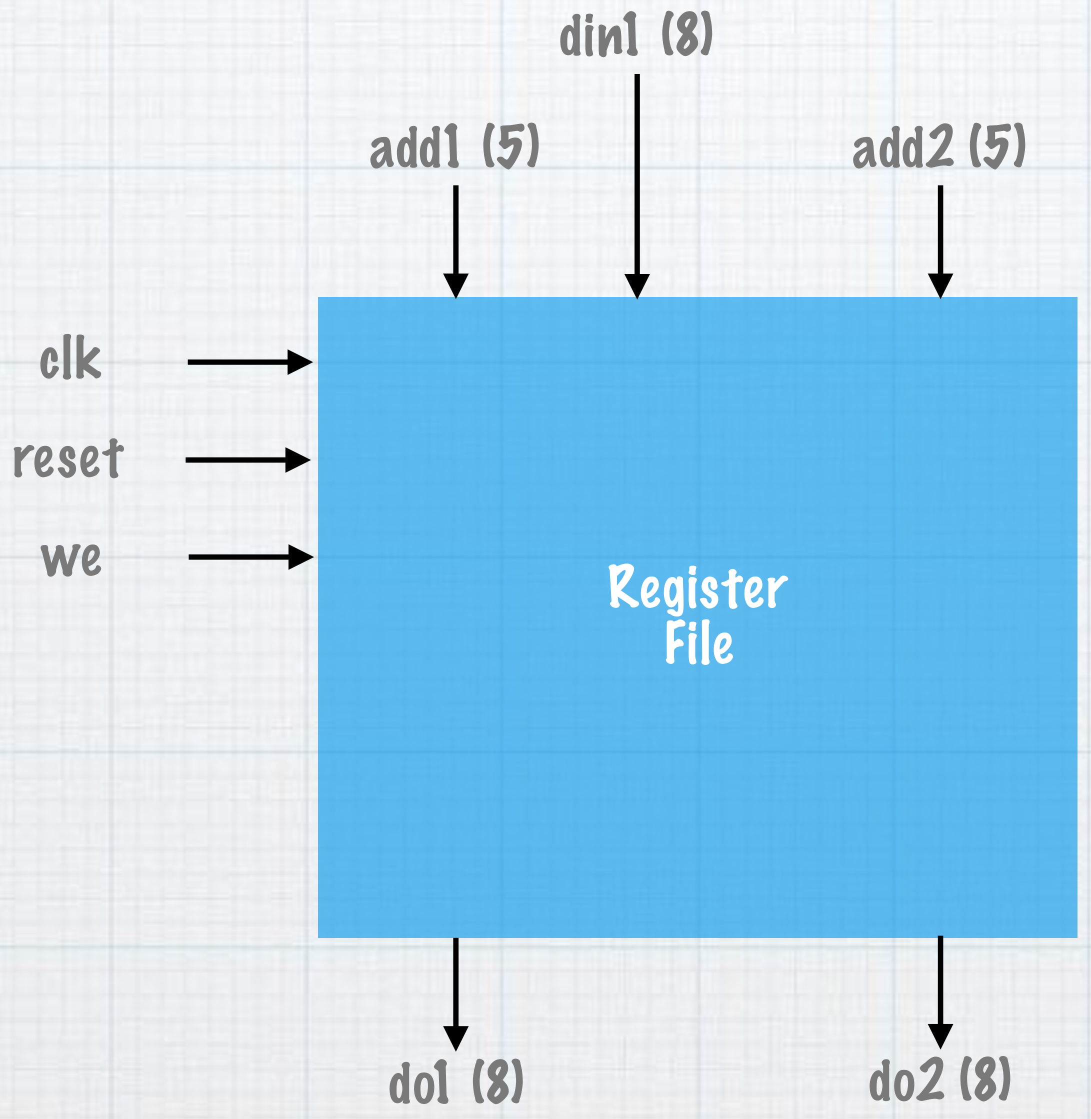
AVR Register File

Binary	[15:8]	[7:0]	
00000		R_0	
		...	
01111		R_{15}	
10000		R_{16}	
		...	
11001		R_{25}	
	R_{27}	R_{26}	X
	R_{29}	R_{28}	Y
11111	R_{31}	R_{30}	Z

AVR Register File

* Dual ported
    * Can read two registers in parallel
        * Two address inputs
        * Two data outputs
* Can write one register as well
    * WE (write enable) + data input

AVR Register File



Reset is synchronous

Reads are asynchronous

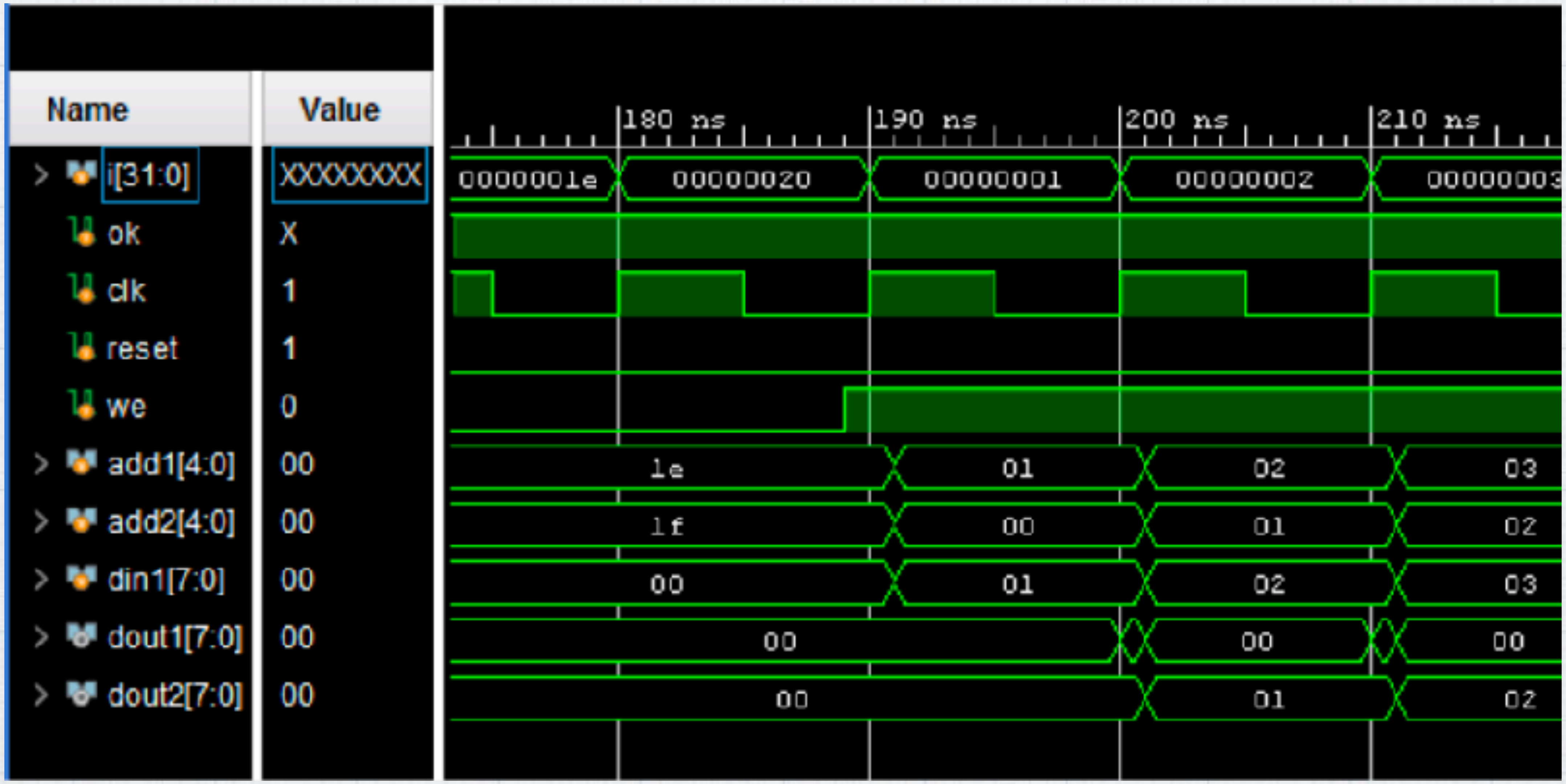
Write is synchronous

```
module register_file(
    input               clk,
    input               reset,
    input               we,
    input         [4:0] add1,
    input         [4:0] add2,
    input         [7:0] din1,
    output        [7:0] do1,
    output        [7:0] do2
    );

    reg [7:0] regfile [31:0];

    assign dout1 = regfile[add1];
    assign dout2 = regfile[add2];

    always @(posedge clk)
        if (reset) begin
            regfile[0] <= 0;  regfile[1] <= 0;
            regfile[2] <= 0;  regfile[3] <= 0;
            …
            regfile[30] <= 0; regfile[31] <= 0;
        end
        else if  (we)
            regfile[add1] <= din1;

endmodule
```

Lab 08 - Register File

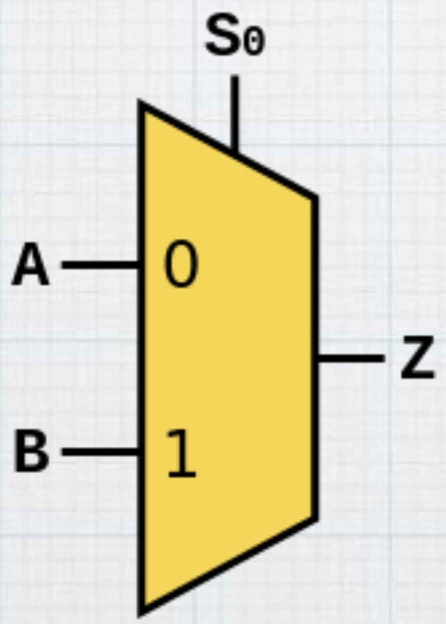
* Register file test bench walkthrough

Lab 08 Output



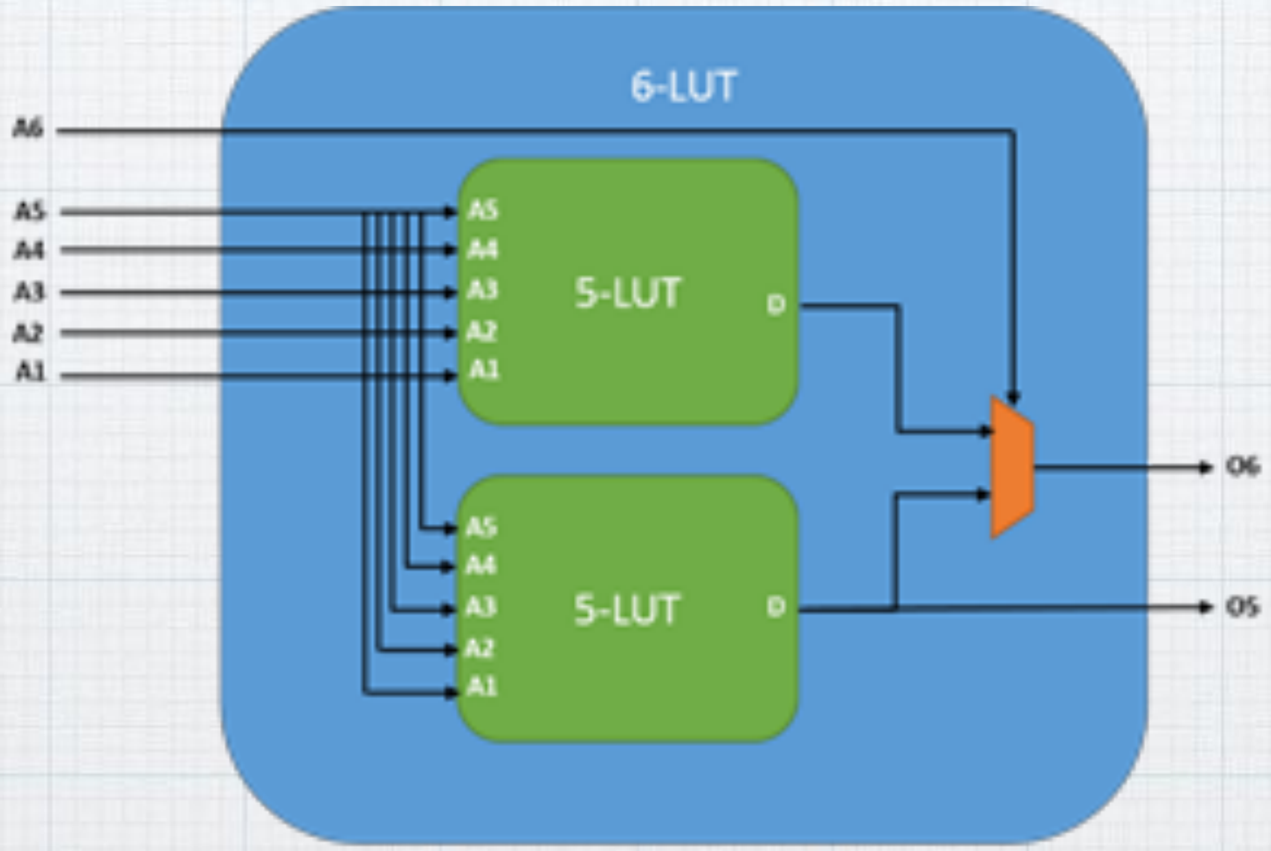
Multiplexer

- Basic logic component
- Selects one of many inputs - 2, 4, 8 etc.



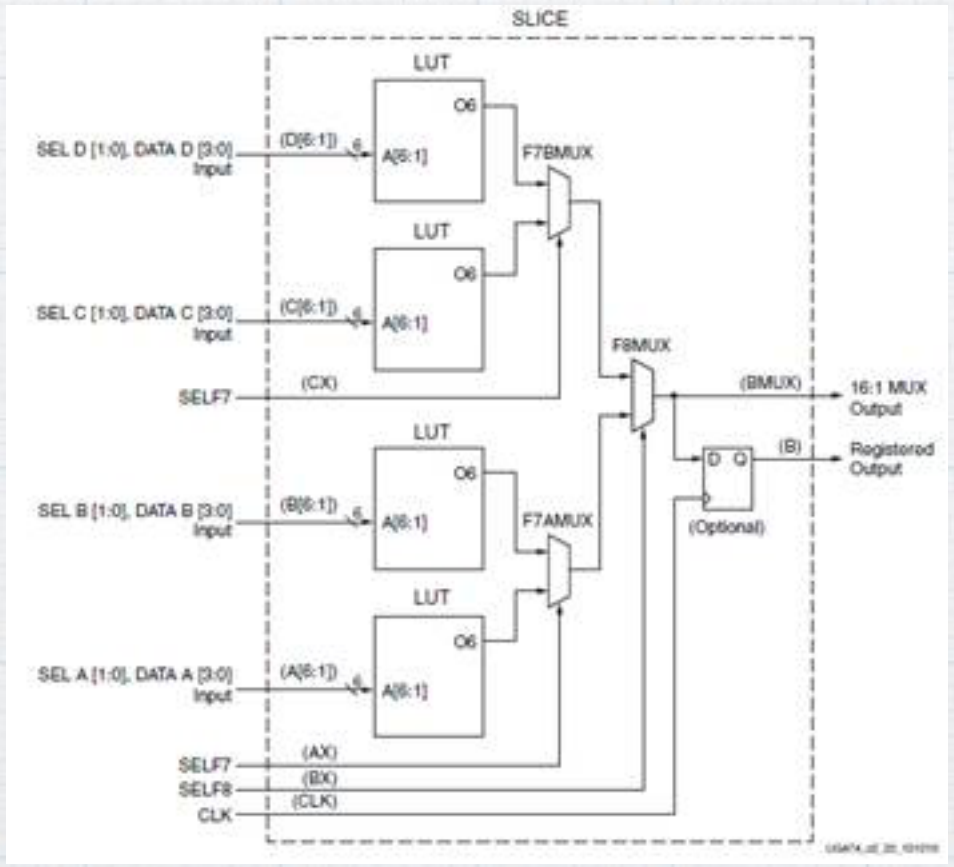
S_0	Z
0	A
1	B

LUT-6 Implementation

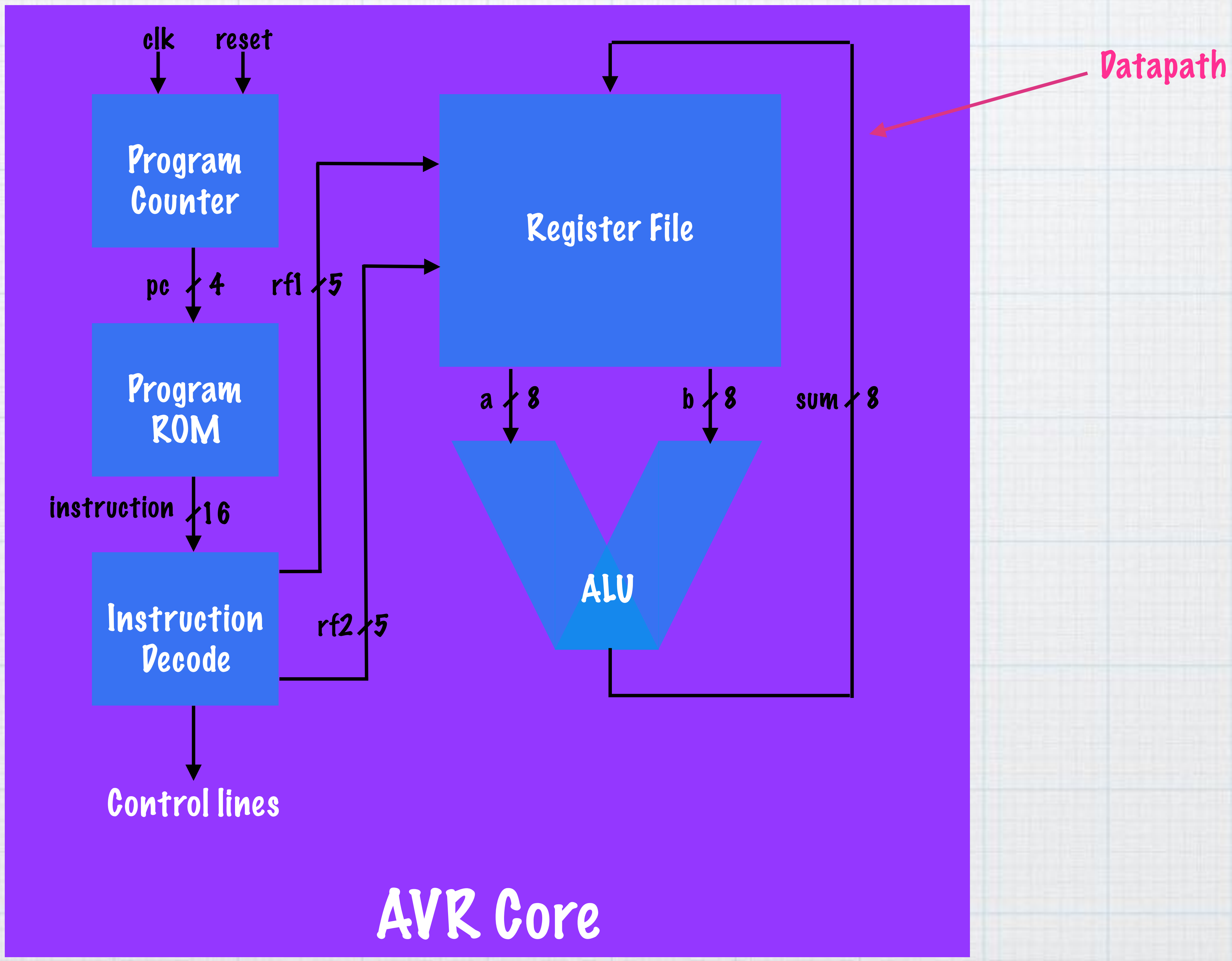


Wide Multiplexers

- 7 series uses LUT-6's as muxes
- Wider muxes require cascades



Roadmap


clk
reset
Program Counter
pc 4
Program ROM
instruction 16
Instruction Decode
Control lines
rf1 5
rf2 5
Register File
a 8
b 8
sum 8
ALU
Datapath
AVR Core

Lab 09 - AVR Core

* Assemble components
* Wire them together

General ISA's

- Instruction Set Architecture
- Instructions are binary values
  - Fields encoded within
  - Multiple instruction formats
- AVR is a RISC processor
  - 16/32 bit instructions

RISC

* Reduced Instruction Set Computer
* External memory to register(s)
    * LOAD
* Register to register calculations
* Write register value to memory
    * STORE

AVR Documentation

* Instruction Set Manual
* Instruction Encoding
* Excel Spreadsheet

Program ROM

```
module prog1(
    input         [3:0]   a,      // address
    output reg    [15:0]  dout    // instruction
    );

    always @*
        case (a)
            4'h0:    dout = 16'hea05;  // ldi  r16,0xa5
            4'h1:    dout = 16'he01f;  // ldi  r17,0x0f
            4'h2:    dout = 16'h2301;  // and  r16,r17
            default: dout = 16'h0000;  // nop
        endcase

endmodule
```

AVR Getting Started

- Sample ROM has 4 instructions
- 3 different formats
  - NOP
  - Direct Register (2 of 32 registers)
  - Immediate

AVR General Format

Opcode						Arguments									
0	0	0	0	0	0	X	X	X	X	X	X	X	X	X	X

- Exceptions
    - Immediate instructions

No Operation - nop

0				0				0				0			
Opcode															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

* No-Operation
    * Do nothing
* Realistically...
    * Change nothing

Direct Register - 2 of 32

Opcode						Rr	Rd					Rr			
0	0	op	op	op	op	r_4	d_4	d_3	d_2	d_1	d_0	r_3	r_2	r_1	r_0

- Rd = Rd <op> Rr
- Rd and Rr into ALU (a, b)
- <op> determines calculation, e.g. AND, OR
- Output written to Rd

Dir. Register Ops - 2/32

<op>		<op>	
0000	16 bit move	1000	and/tst
0001	cpc	1001	eor/clr
0010	sbc	1010	or
0011	add/lsl	1011	mov
0100	cpse	*	
0101	cp	*	
0110	sub	*	
0111	adc/rol	*	

name/alias - Rr same as Rd

* - not a register direct instruction (cpi)

Logical AND - and

2				3				0				1			
Operation						Rr	Rd					Rr			
0	0	1	0	0	0	1	1	0	0	0	0	0	0	0	1

* Operation = 001000 = RD (2/32) + and
* Rr = 10001 = R_{17}
* Rd = 10000 = R_{16}

Immediate Instructions

Opcode				K				Rd				K			
op	op	op	op	K_7	K_6	K_5	K_4	d_3	d_2	d_1	d_0	K_3	K_2	K_1	K_0

* K is a signed 8 bit constant
* Rd missing a bit
    * Prefix Rd with a 1 e.g. 0000 = R_{16}
    * Immediate instructions target R_{16}-R_{31}
* Output written to Rd

Immediate Instructions

<op>	
0011 *	cpi
0100	sbci
0101	subi
0110	ori
0111	andi
1110	ldi

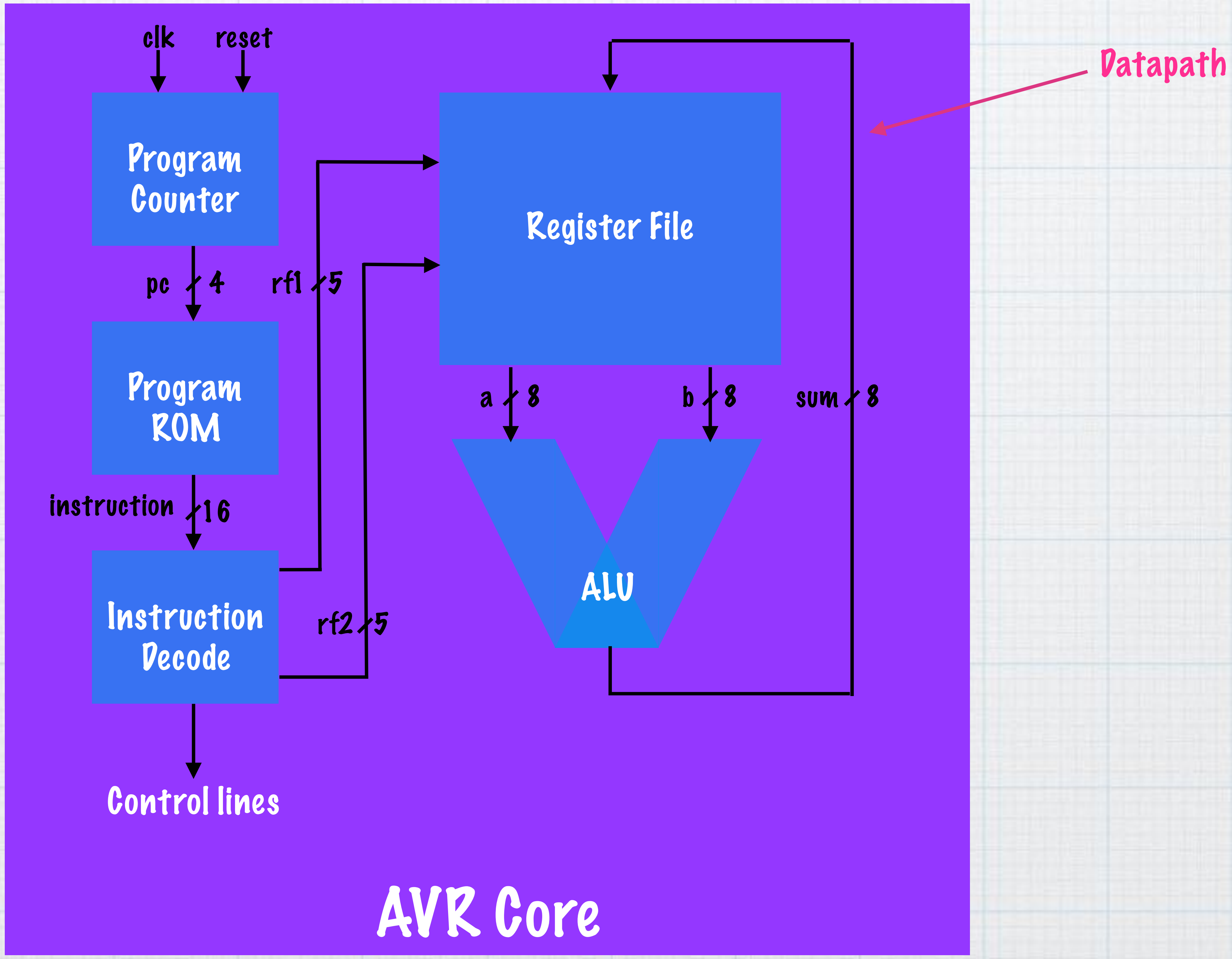
* - direct register overlap

Load Immediate - ldi

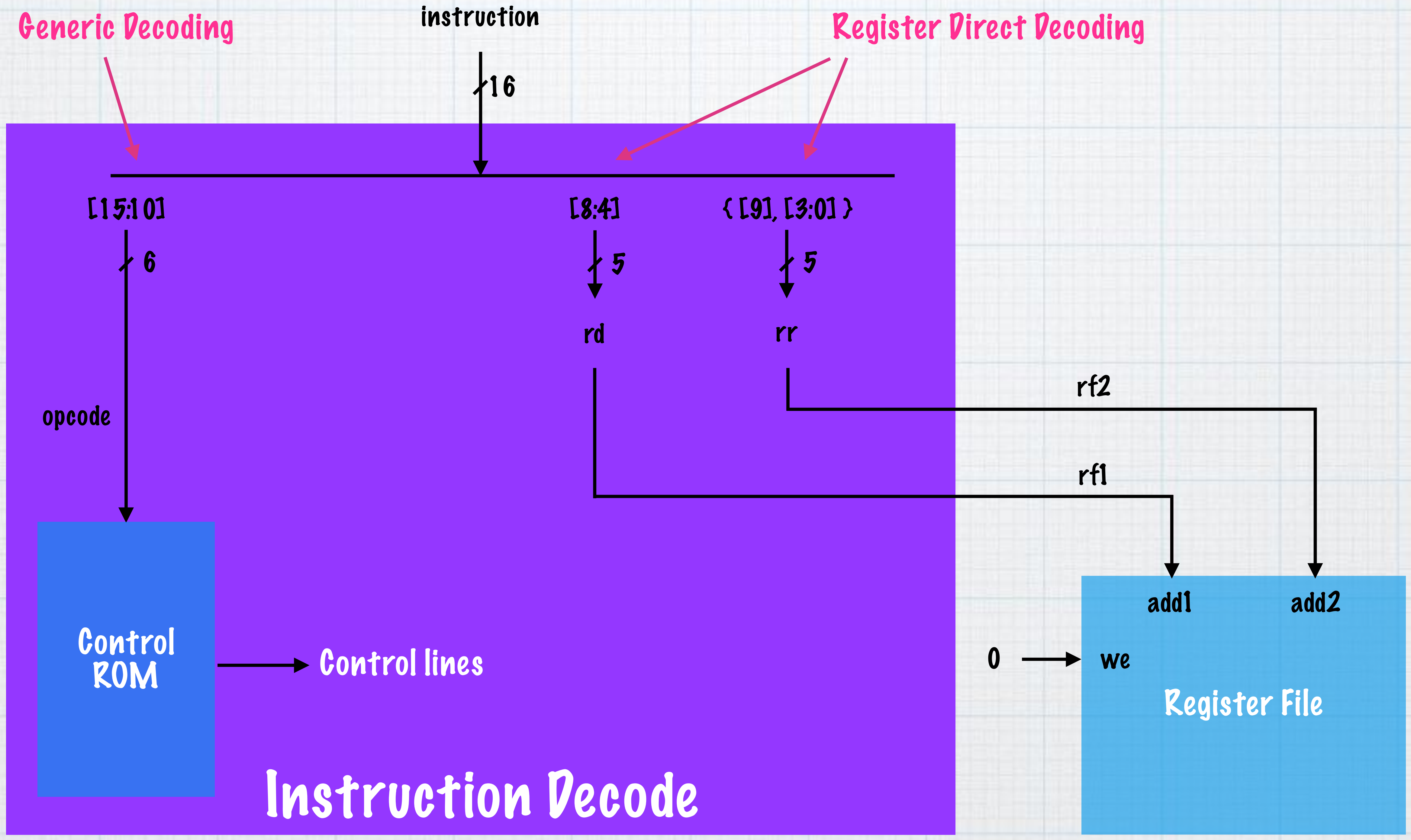
e				a				0				5			
Operation				K				Rd				K			
1	1	1	0	1	0	1	0	0	0	0	0	0	1	0	1

- Operation = 1110 = ldi
- K = 1010 , 0101 = 10100101 = 0xa5
- Rd = 0000 = 1 , 0000 = R_{16}

Roadmap


clk
reset
Program Counter
pc 4
rf1 5
Program ROM
instruction 16
Instruction Decode
rf2 5
Control lines
Register File
a 8
b 8
sum 8
ALU
Datapath
AVR Core

Instruction Decode



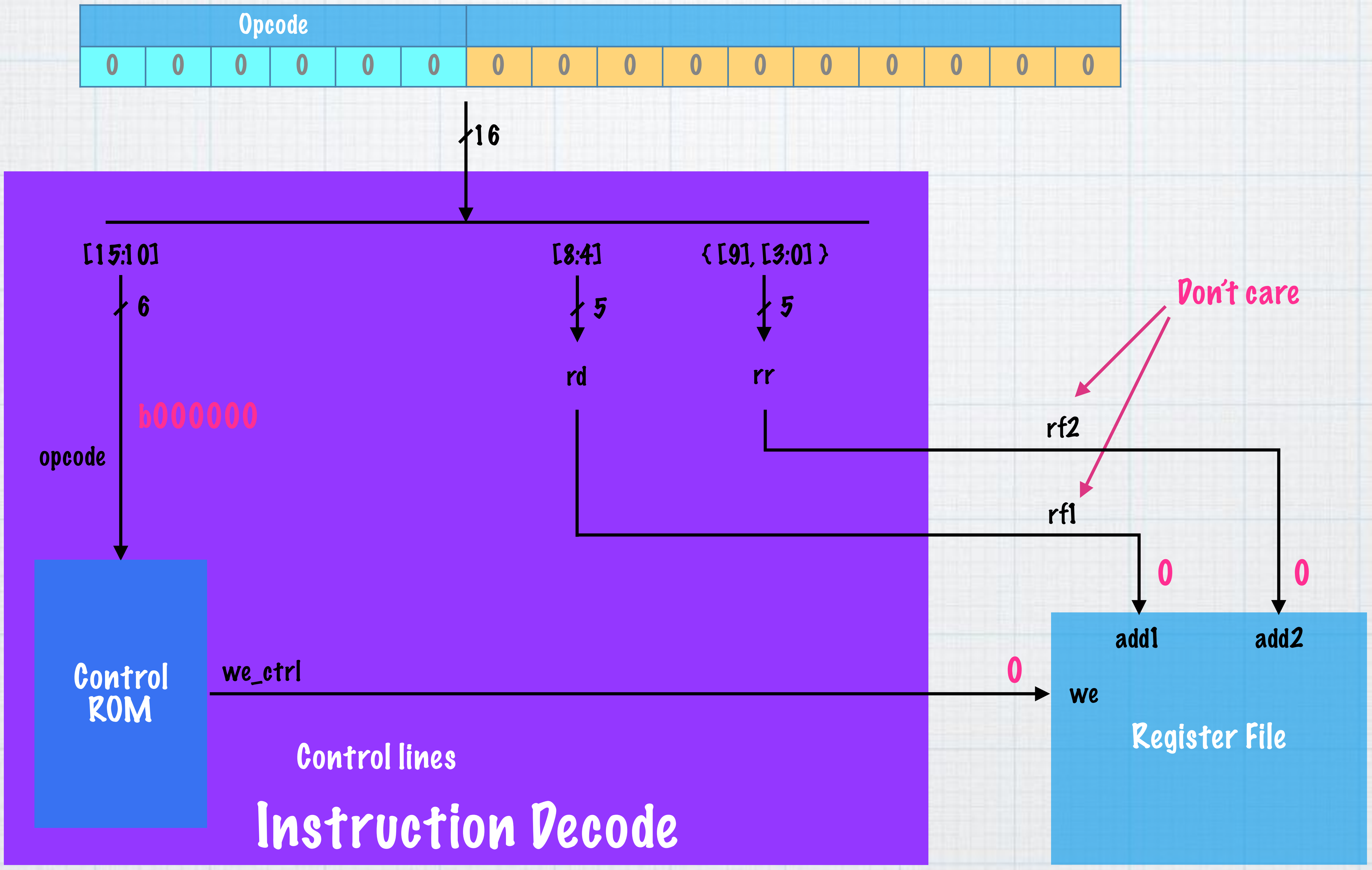
```
module control_rom(
    input  [5:0]  opcode,        // 6 bits from instruction
    output        we_ctrl,       // register file write enable
    output [1:0]  alu_ctrl,      // ALU operation code
    output        rdmux_ctrl,    // Rd calculation DDDDD/1DDDD
    output        bmux_ctrl      // ALU port B mux
    );

    reg [0:4] data;

    always @*
        casez(opcode)
            //                   we,   aluop,        rdmux, bmux
            6'b1110??: data = { 1'b1, `ALUFN_PASSB, 1'b1,  1'b1 }; // ldi
            6'b001000: data = { 1'b1, `ALUFN_AND,   1'b0,  1'b0 }; // and
            default:   data = { 1'b0, `ALUFN_PASSB, 1'b0,  1'b0 }; // nop
        endcase

    assign we_ctrl    = data[0];
    assign alu_ctrl   = data[1:2];
    assign rdmux_ctrl = data[3];
    assign bmux_ctrl  = data[4];

endmodule
```

Control ROM

Opcode	Encoding	we	aluop	rf1	b
nop	000000	0	X	X	X
ldi	1110??	1	PASSB	1DDDD	K
and	001000	1	AND	DDDDD	rf2out

X - Don't care

Instruction Decode - nop

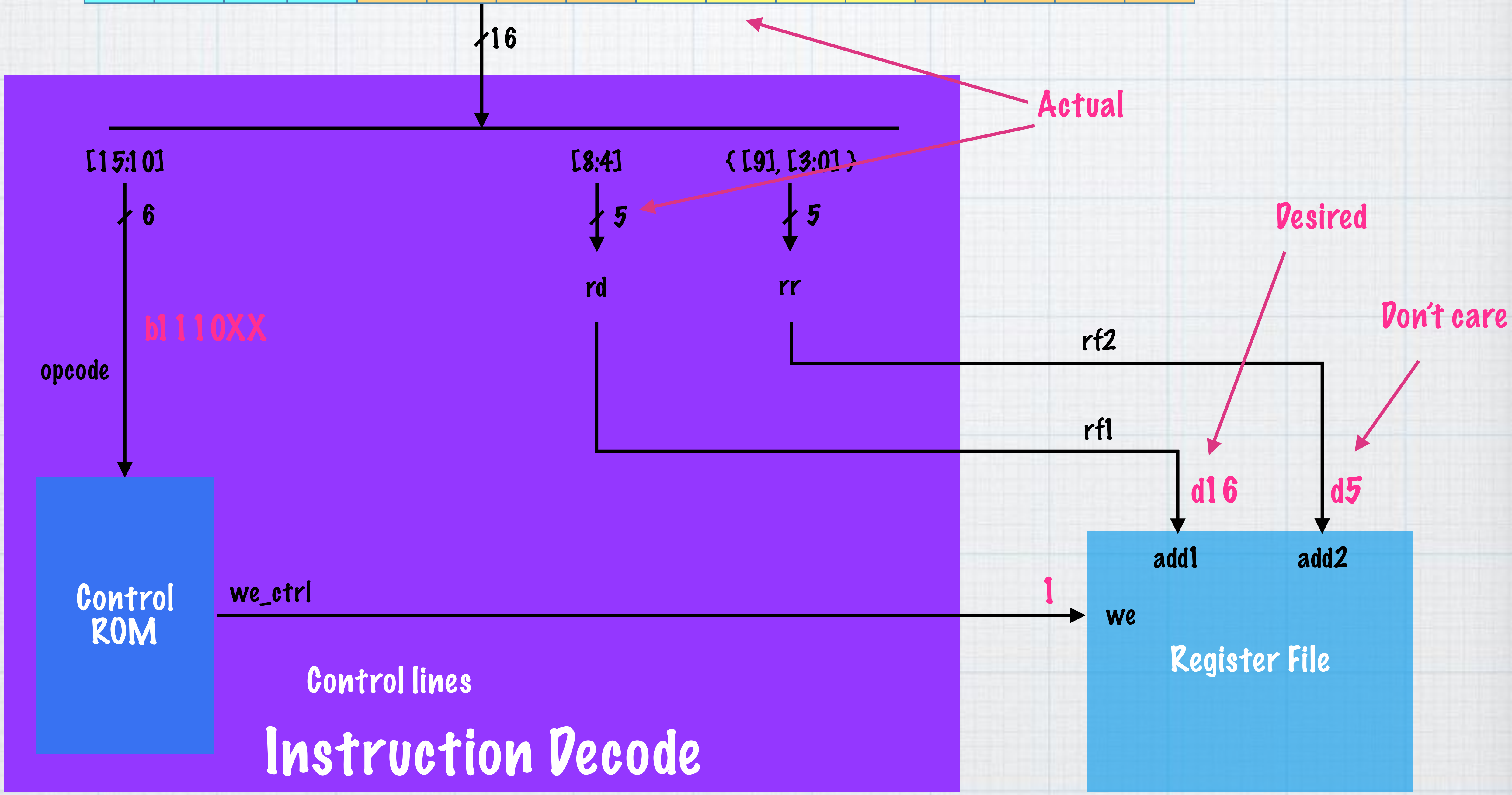


Lab 10 - Part 1

* Add the control ROM
* Wire up register file write enable (WE)

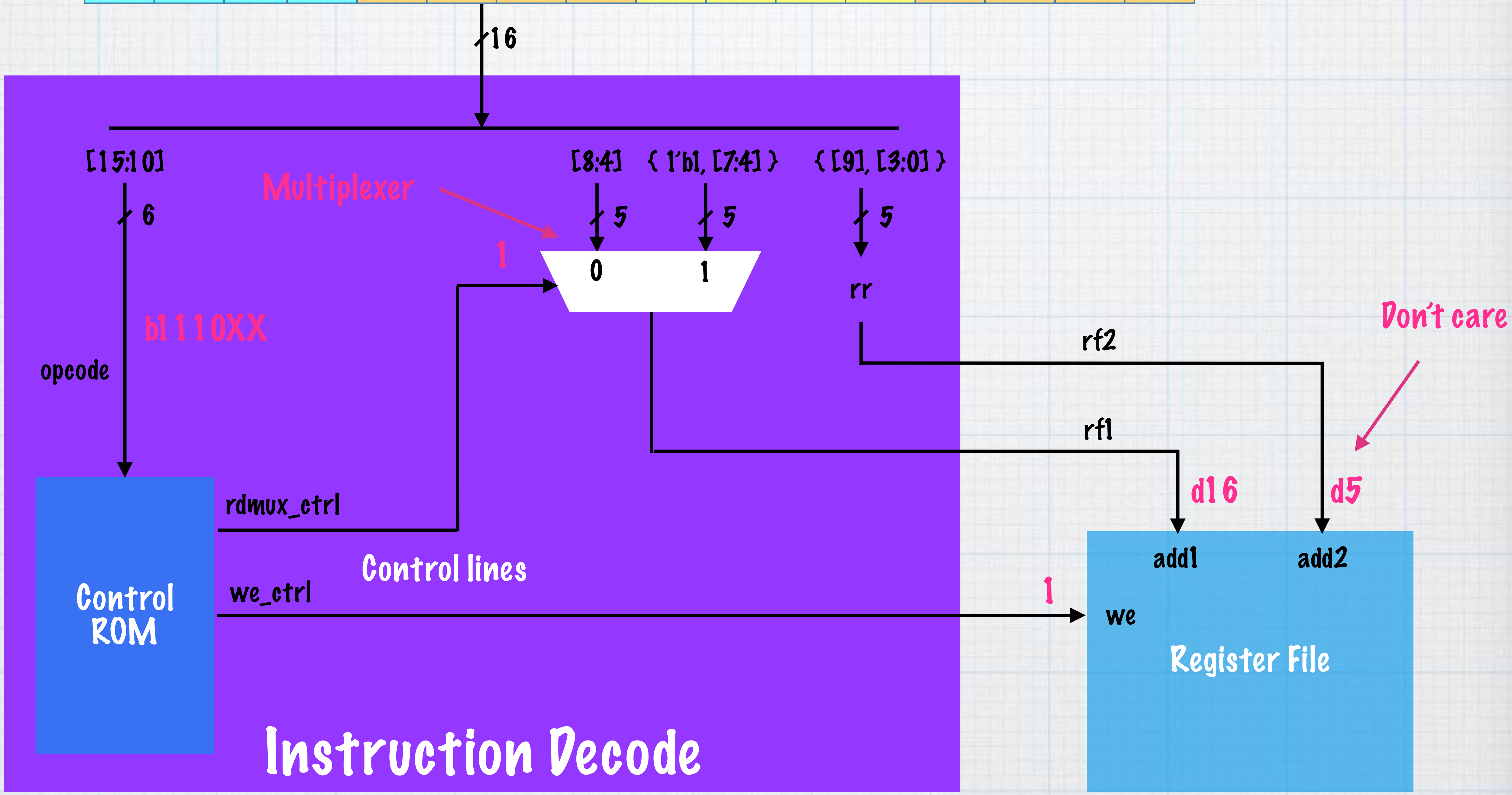
Instruction Decode - ldi

Opcode				K				Rd				K			
1	1	1	0	1	0	1	0	0	0	0	0	0	1	0	1



Instruction Decode - ldi

Opcode				K				Rd				K			
1	1	1	0	1	0	1	0	0	0	0	0	0	1	0	1



Verilog Multiplexers

* Two ways
  * Conditional operator
    * Wires
  * Procedural if-else
    * Reg (variables)

Conditional Operator

* Same as C and Java
* Condition can be value or boolean expression

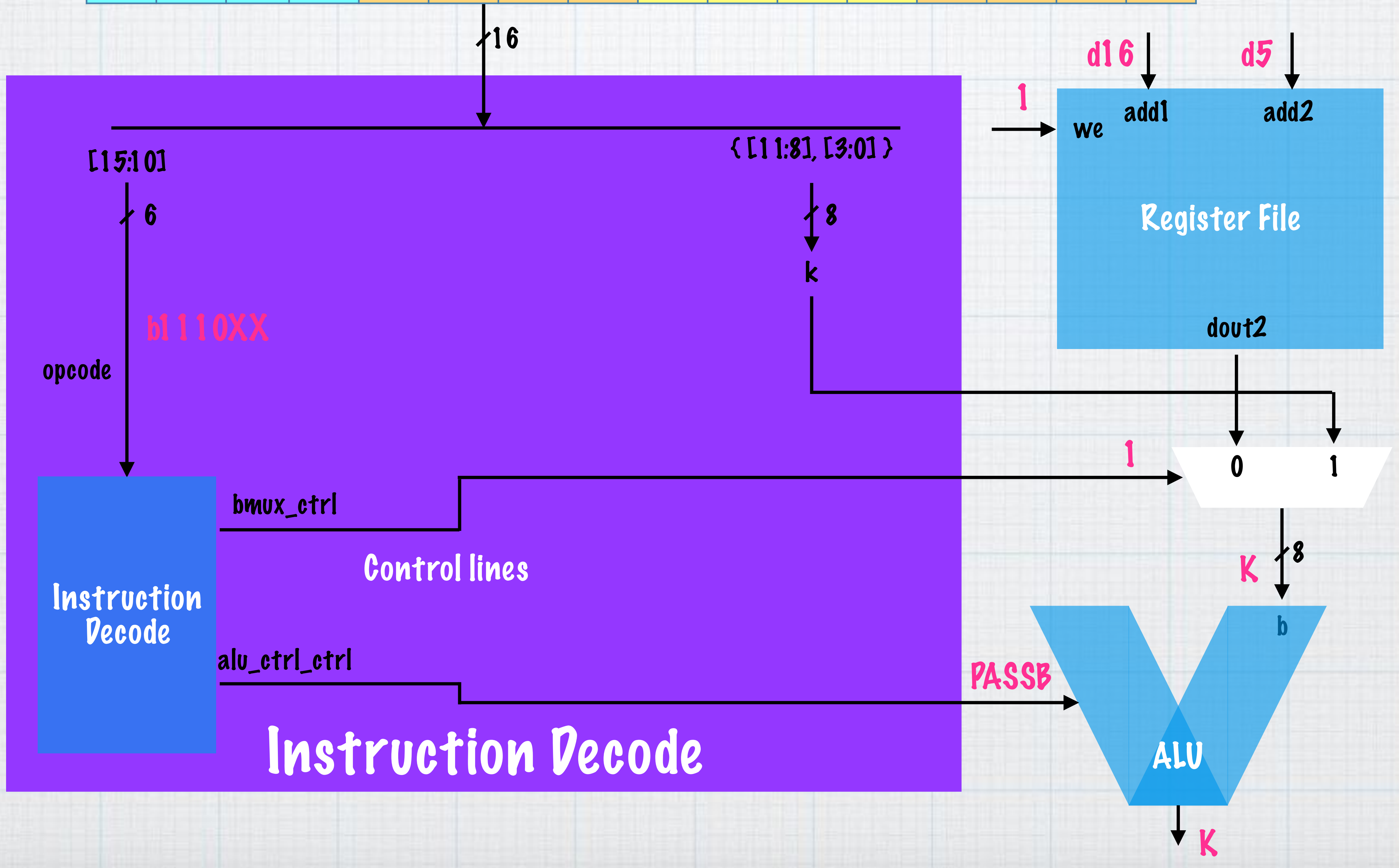
```
wire         rf1;
wire         immediate;                         true                   false
                                                  ↙                      ↓
assign rf1 = immediate ? {1'b1, instruction[7:4] } : rd;
                 ↑
             condition
```

Lab 10 - Part 2

- Alter the Rd datapath
- Handle alternatives
  - Direct Register - DDDDD
  - Immediate - 1DDDD

Instruction Decode - ldi

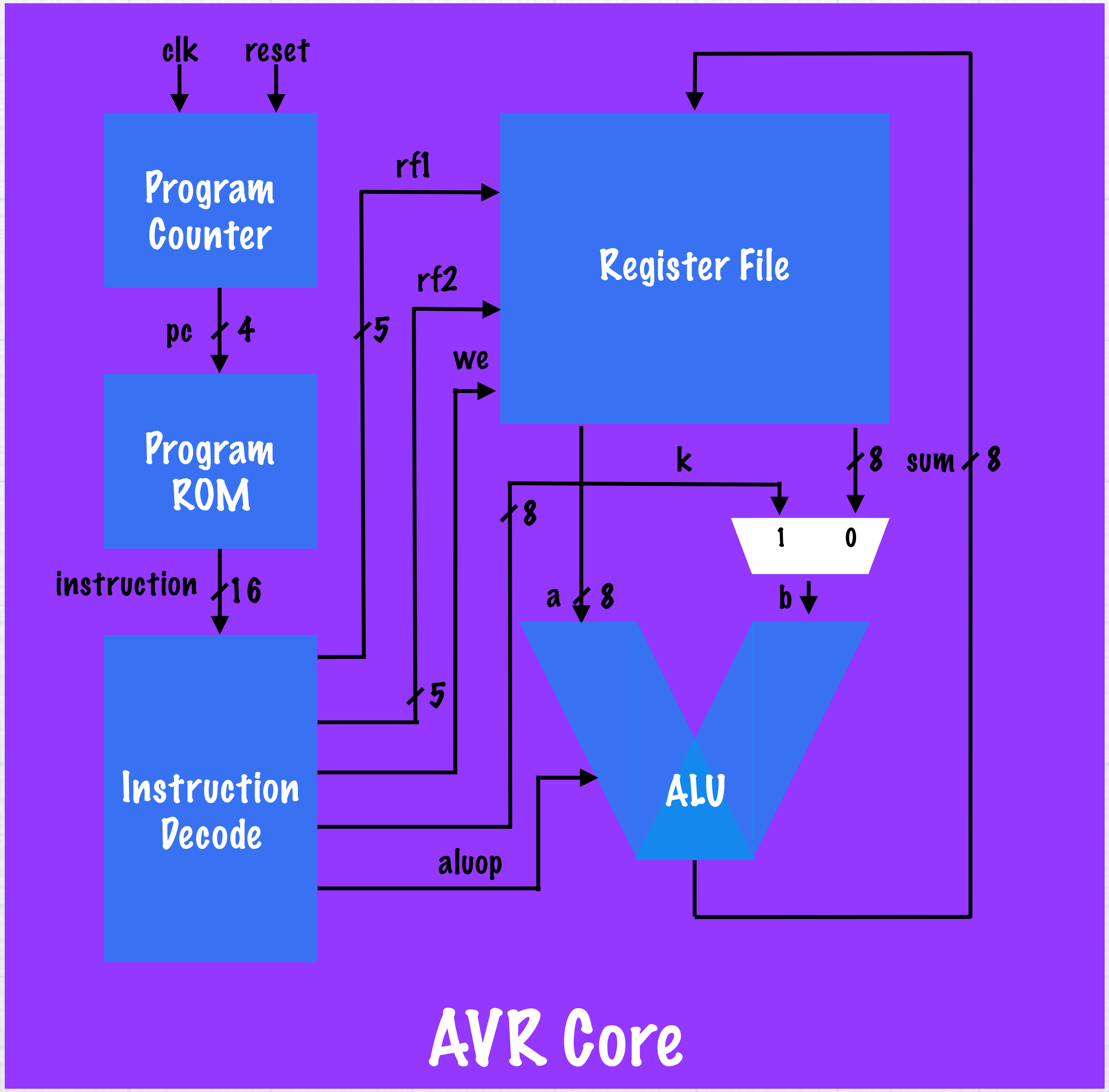
Opcode				K				Rd				K			
1	1	1	0	1	0	1	0	0	0	0	0	0	1	0	1



Lab 10 - Part 3

* Alter the datapath into ALU port B
* Multiplex
  * rf2out
  * K
* Connect alu_ctrl (remove hardcoding)

Roadmap


clk
reset
Program Counter
pc
4
Program ROM
instruction
16
Instruction Decode
rf1
5
rf2
5
we
Register File
k
8
8
sum
8
1
0
a
8
b
ALU
aluop
AVR Core