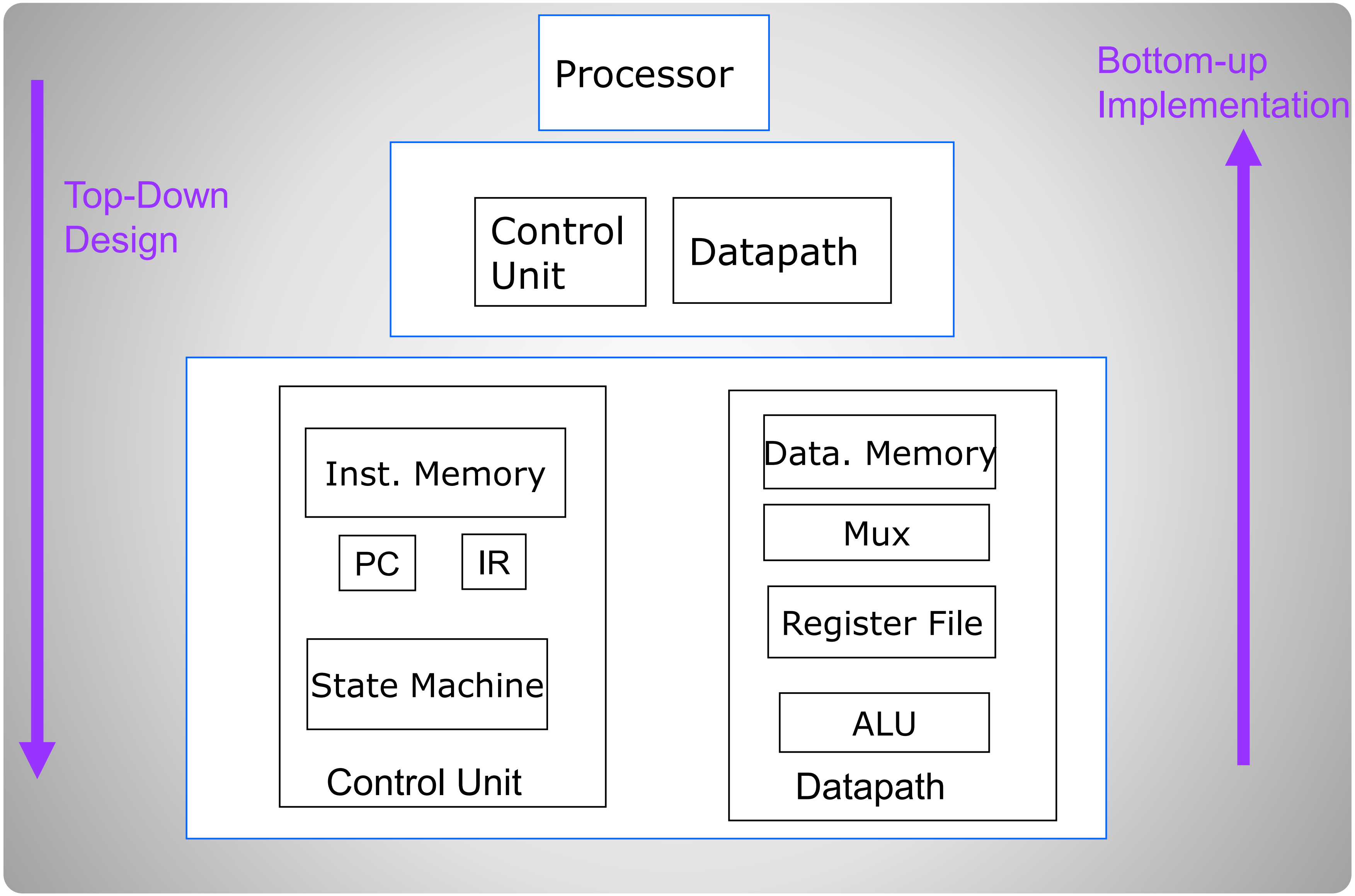
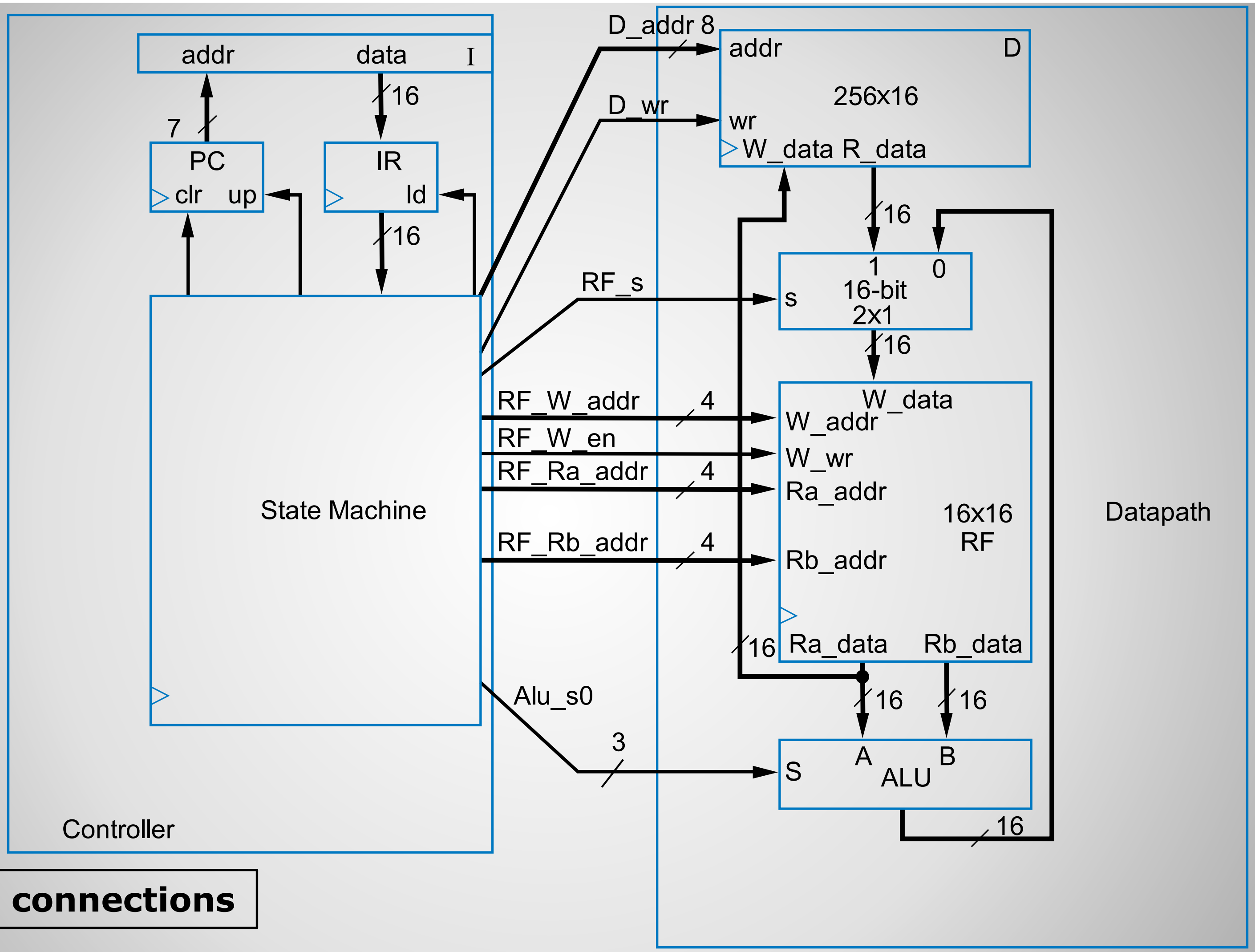
Programmable Processors

A Brief Summary

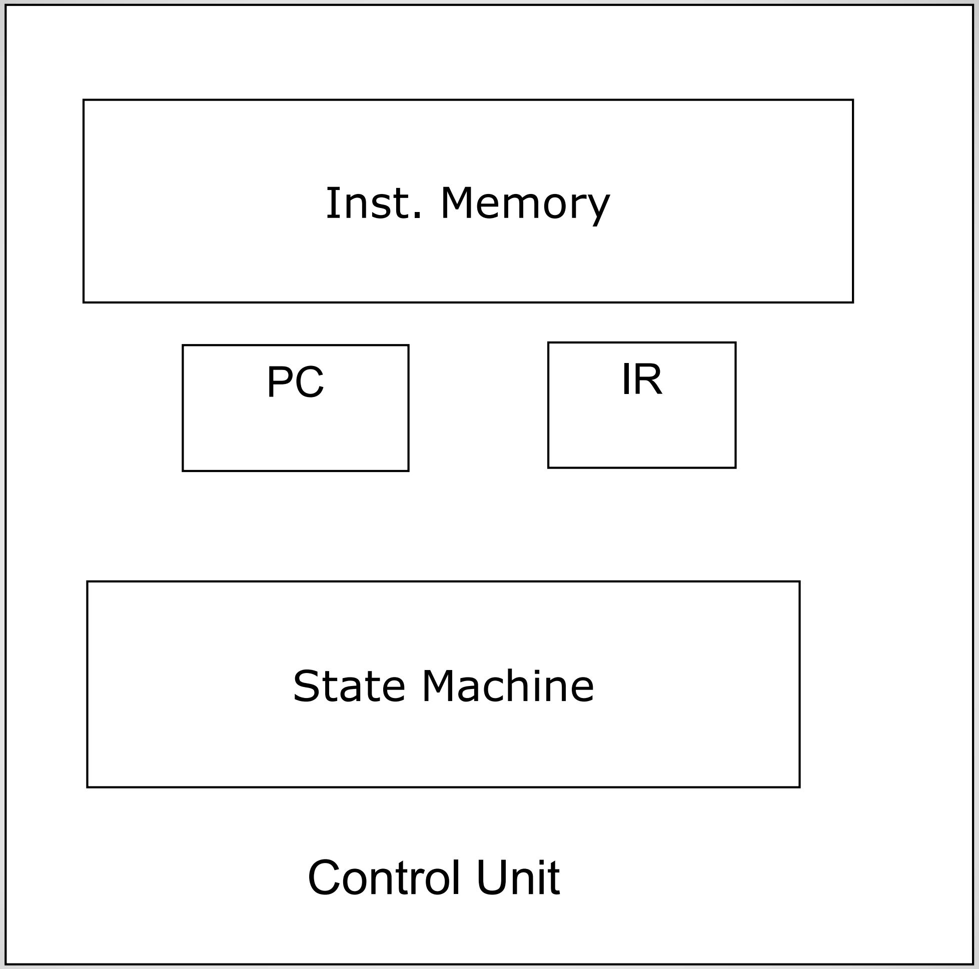


Pic of Processor Modules



Detailed connections

You need to check this pic often when connecting multi-modules



Along the Control Unit side …

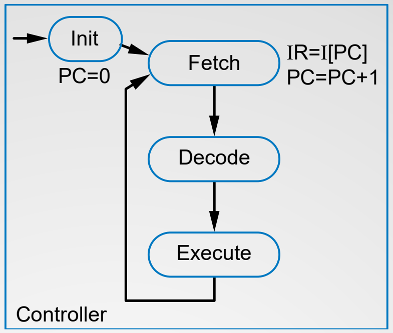
- As discussed during the lecture
  - Under Quartus, create a new .mif file, name it say, `A.mif`. Later manually type into each location a decimal number converted from your 16-bits instructions.
  - Use Altera Library and build a 1-port ROM module, say, `InstMemory.v`. Make sure during the configuration procedure you have associated `InstMemory.v` with `A.mif`.

Instruction Memory Module

- PC is purely a counter
  - Required input signals: Clock, Clr, Up
  - Required output signal: address to access instruction memory

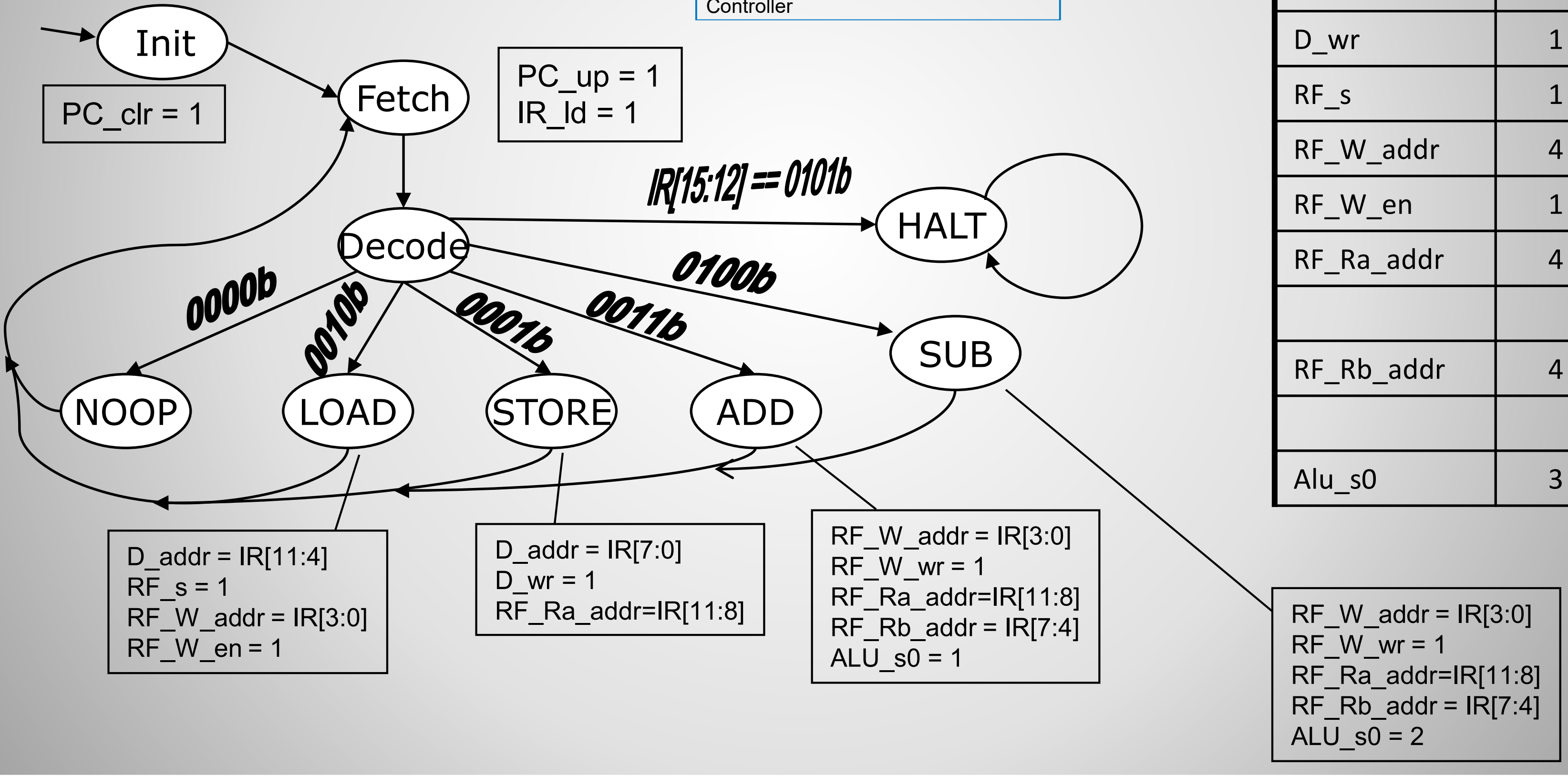
- IR is purely a group of flip-flops which will latch out the input signal
  - Required input signals: Clock, ld, instruction from instruction memory
  - Required output signal: instruction to the finite state machine

PC and IR Modules

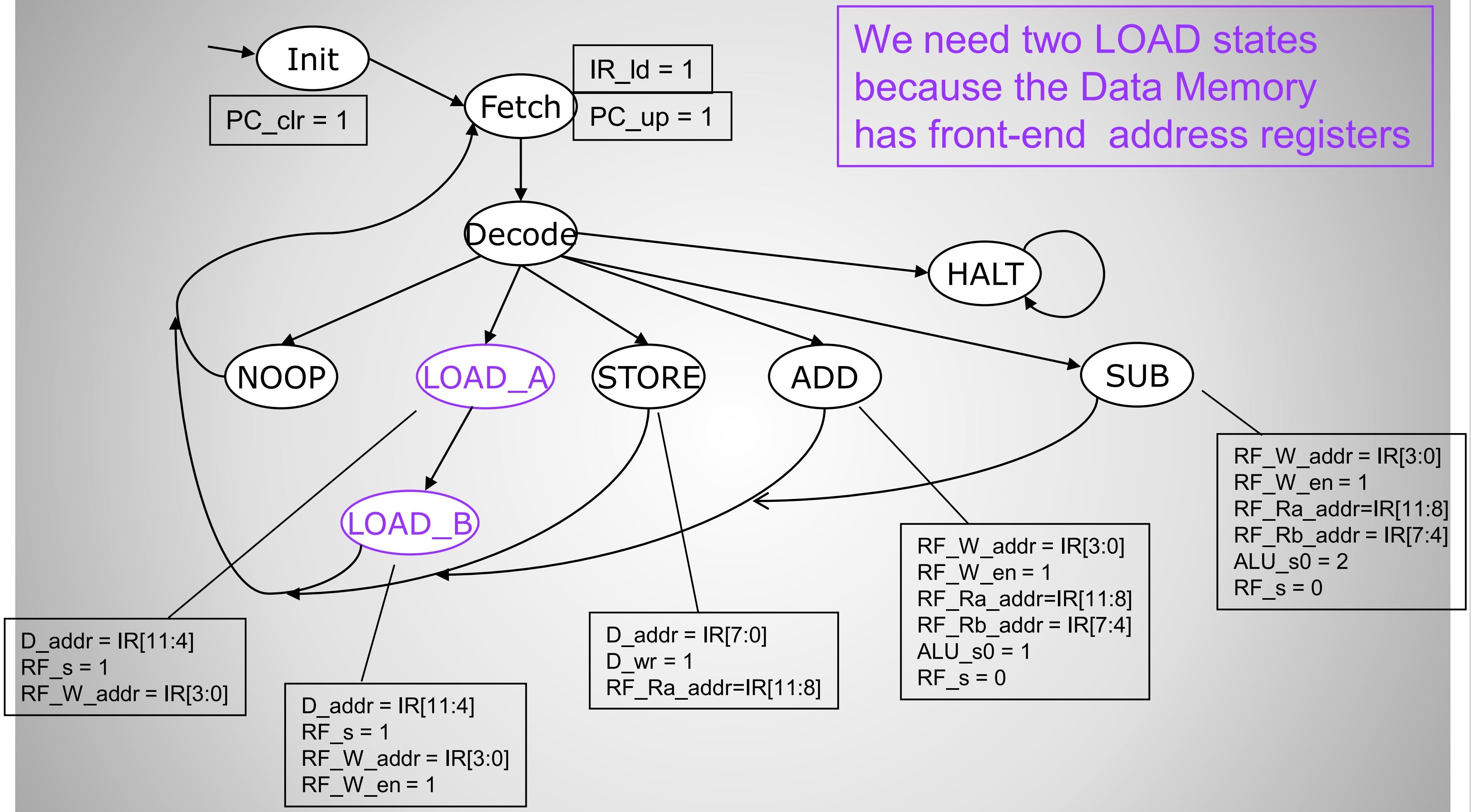


Control Lines

Control	Size
PC_clr	1
PC_up	1
IR_ld	1
D_addr	8
D_wr	1
RF_s	1
RF_W_addr	4
RF_W_en	1
RF_Ra_addr	4
RF_Rb_addr	4
Alu_s0	3



Original State Machine



We need two LOAD states because the Data Memory has front-end address registers

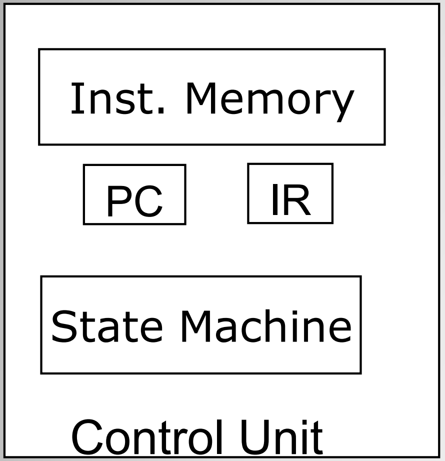
Controller FSM (stick with this design)

State Machine Output Definitions

Variable	Meaning
PC_clr	Program counter (PC) clear command
IR_ld	Instruction load command
PC_up	PC increment command
D_addr	Data memory address (8 bits)
D_wr	Data memory write enable
RF_s	Mux select line
RF_Ra_addr	Register file A-side read address (4 bits)
RF_Rb_addr	Register file B-side read address (4 bits)
RF_W_en	Register file write enable
RF_W_Addr	Register file write address (4 bits)
ALU_s0	ALU function select (3 bits)

State Machine State Outputs

State	Non-Zero Outputs
Init	PC_clr = 1
Fetch	IR_ld = 1, PC_up =1
Decode	Check the opcode
LoadA	D_addr = IR[11:4], RF_s = 1, RF_W_addr = IR[3:0]
LoadB	D_addr = IR[11:4], RF_s = 1, RF_W_addr = IR[3:0], RF_W_en = 1
Store	D_addr = IR[7:0], D_wr = 1, RF_Ra_addr = IR[11:8]
Add, Sub	RF_Ra_addr = IR[11:8], RF_Rb_addr = IR[7:4], RF_W_addr = IR[3:0], RF_W_en = 1, ALU_s0 = alu function (1 for Add or 2 for Sub), RF_s = 0
Halt	



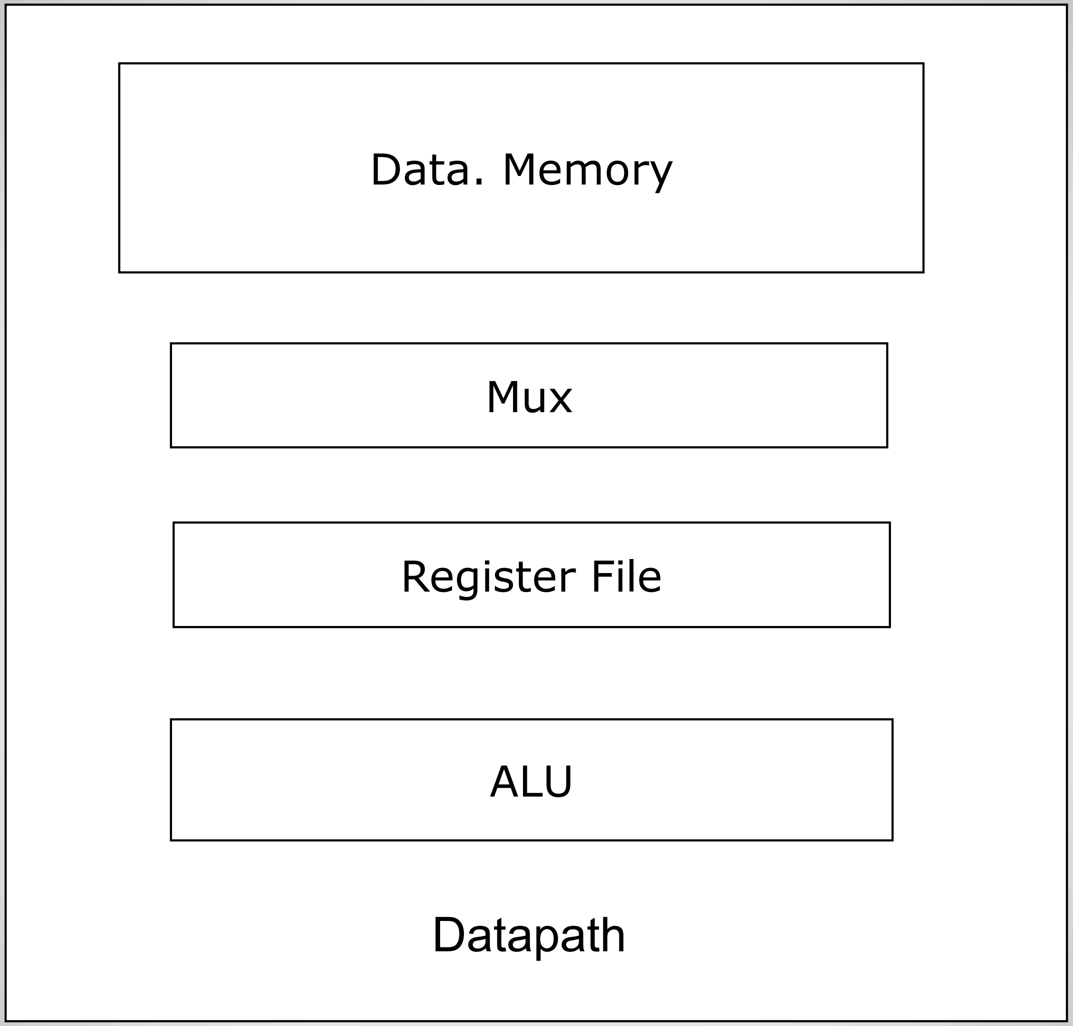
Control Unit module should instantiate the following submodules:

```
PC.sv
IR.sv
InstMemory.v
FSM.sv
```

Required input signals: Clock, Reset (Notice here the Reset signal is required for FSM)

Required output signals: PC_Out, IR_Out, OutState, NextState, D_Addr, D_Wr,RF_s, RF_W_en, RF_Ra_Addr, RF_Rb_Addr, RF_W_Addr, ALU_s0

Build Control Unit Module and Test it!



Along the Datapath side

- As discussed during the lecture
  - Under Quartus, create a new .mif file, name it say, `D.mif`. Later manually type into specific locations numbers according to the given requirements.
  - Use Altera Library and build a 1-port RAM module, say, `DataMemory.v`. Make sure during the configuration procedure you have associated `DataMemory.v` with `D.mif`.

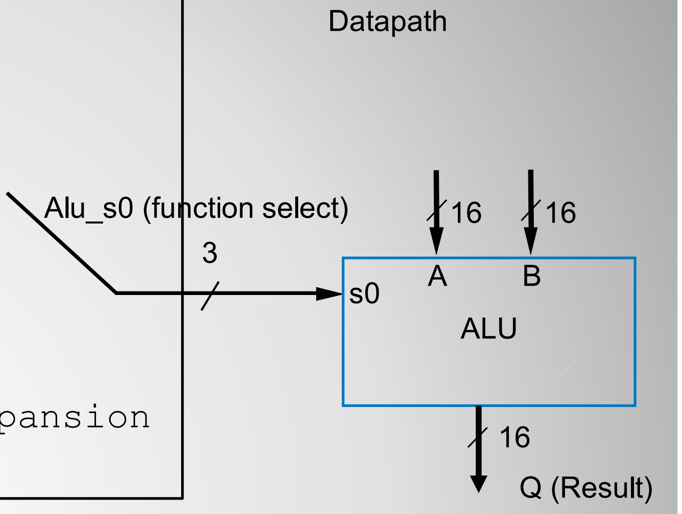
Data Memory Module

```
// This is a Verilog description for an 8 x 16 register file
module regfile8x16a
  (input clk,                   // system clock
   input write,                 // write enable
   input [2:0] wrAddr,          // write address
   input [15:0] wrData,         // write data
   input [2:0] rdAddrA,         // A-side read address
   output [15:0] rdDataA,       // A-side read data
   input [2:0] rdAddrB,         // B-side read address
   output [15:0] rdDataB );     // B-sdie read data

   logic [15:0] regfile [0:7];  // the registers

   // read the registers
   assign rdDataA = regfile[rdAddrA];
   assign rdDataB = regfile[rdAddrB];

   always @(posedge clk) begin
      if (write) regfile[wrAddr] <= wrData;
   end
endmodule
```

Expand on this idea for Project B

Register File (for reference)

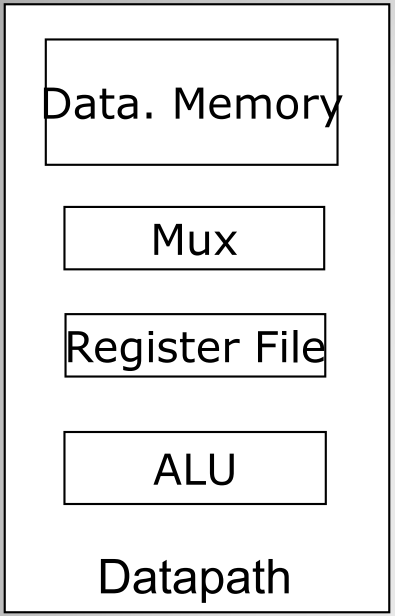
```
// This ALU has eight functions:
//   if s == 0 the output is 0
//   if s == 1 the output is A + B
//   if s == 2 the output is A - B
//   if s == 3 the output is A (pass-through)
//   if s == 4 the output is A ^ B
//   if s == 5 the output is A | B
//   if s == 6 the output is A & B
//   if s == 7 the output is A + 1;
// if additional functions added for future expansion
// you need to expand the selecting signal too
```



```
module ALU( A, B, Sel, Q );
  input [2:0] Sel;      // function select
  input [15:0] A, B;    // input data

  output [15:0] Q; // ALU output (result)
…
```

Expand on this idea for Project Extra points

ALU Idea (for reference)



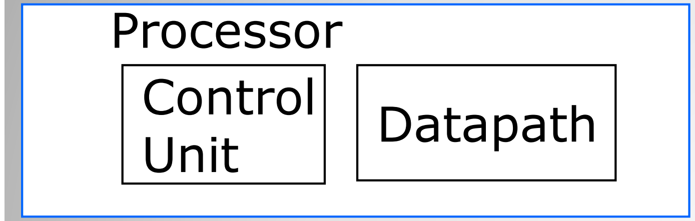
Datapath module should instantiate the following submodules:

```
DataMemory.v
Mux.sv
Register.sv
ALU.sv
```

Required input signals: Clock, D_Addr, D_Wr, RF_s, RF_W_Addr, RF_W_en, RF_Ra_Addr, RF_Rb_Addr, ALU_s0

Required output signals: ALU_inA, ALU_inB, ALU_out

Build Datapath Module and Test it!



Processor module should instantiate the following submodules:

```
    ControlUnit.sv
    DataPath.sv
```

Required input signals: Clk, Reset

Required output signals: IR_Out, PC_Out, State, NextState, ALU_A, ALU_B, ALU_Out

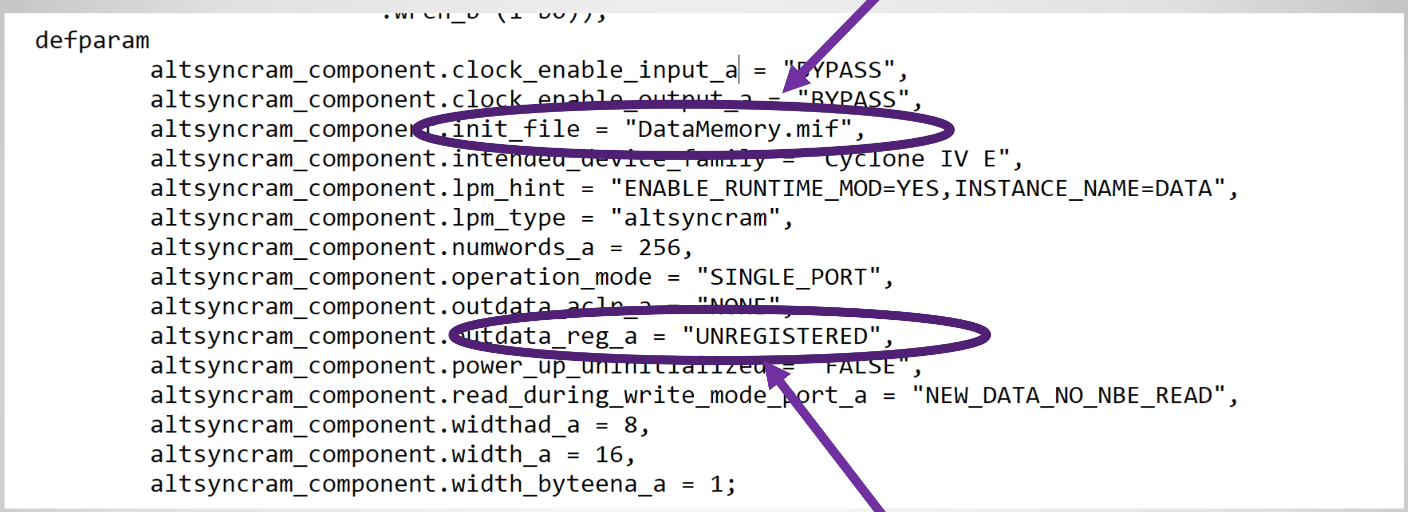
Combine Control Unit and DP in Processor Module and Test it!

- Test submodules from lowest level.
- Test your control unit module and ensure it works as expected
- Test your datapath module and ensure it works as expected
- Write the Processor by instantiating Control Unit module and Datapath module and wiring them together.
- Test the processor module by running runrtl.do
- At this point you should be able to notice the state machine works but ALU_A, ALU_B, and ALU_out are all zeros.
- Check your memory.v file; revise it as shown on next page
- NOTE: if testing with ModelSim doesn't work it won't help by trying testing on DE2 board.

Testing your Processor

Inside the memory.v file

Here should be the mif file you created for your RAM

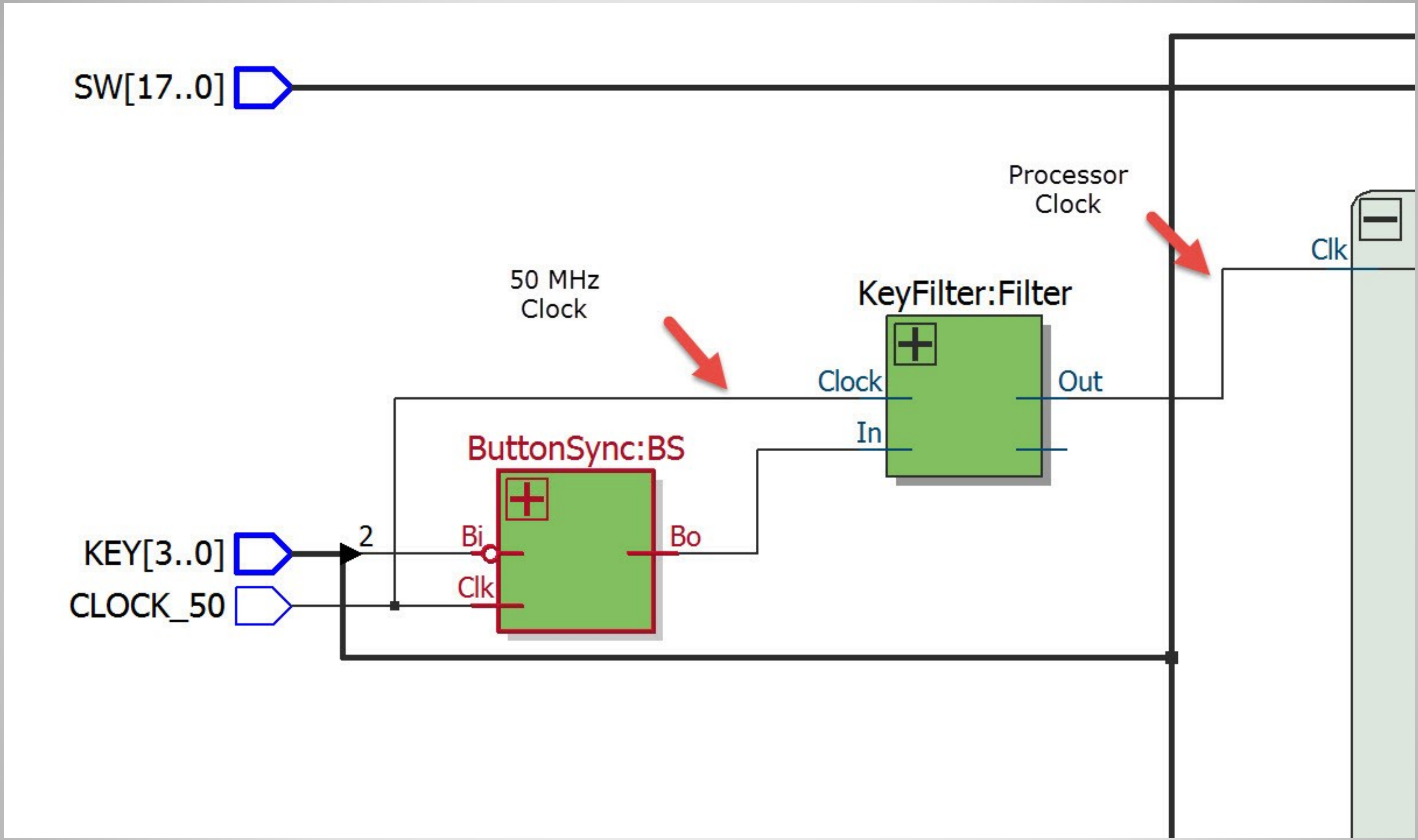


```
                        .wren_b (1'b0));
defparam
        altsyncram_component.clock_enable_input_a = "BYPASS",
        altsyncram_component.clock_enable_output_a = "BYPASS",
        altsyncram_component.init_file = "DataMemory.mif",
        altsyncram_component.intended_device_family = "Cyclone IV E",
        altsyncram_component.lpm_hint = "ENABLE_RUNTIME_MOD=YES,INSTANCE_NAME=DATA",
        altsyncram_component.lpm_type = "altsyncram",
        altsyncram_component.numwords_a = 256,
        altsyncram_component.operation_mode = "SINGLE_PORT",
        altsyncram_component.outdata_aclr_a = "NONE",
        altsyncram_component.outdata_reg_a = "UNREGISTERED",
        altsyncram_component.power_up_uninitialized = "FALSE",
        altsyncram_component.read_during_write_mode_port_a = "NEW_DATA_NO_NBE_READ",
        altsyncram_component.widthad_a = 8,
        altsyncram_component.width_a = 16,
        altsyncram_component.width_byteena_a = 1;
```

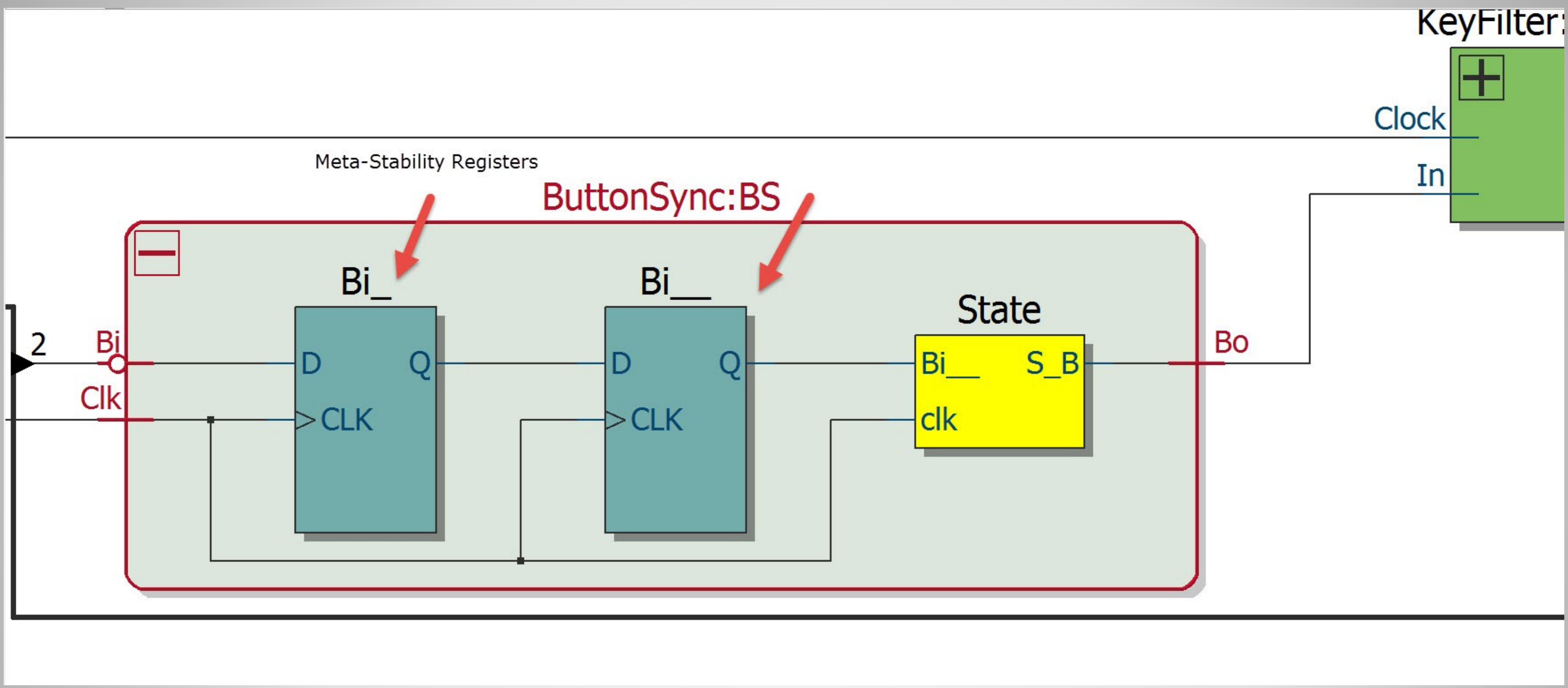
Here if showing “clock” change it to be “UNREGISTERED”

The Key Conditioner

Using ButtonSync and a Filter



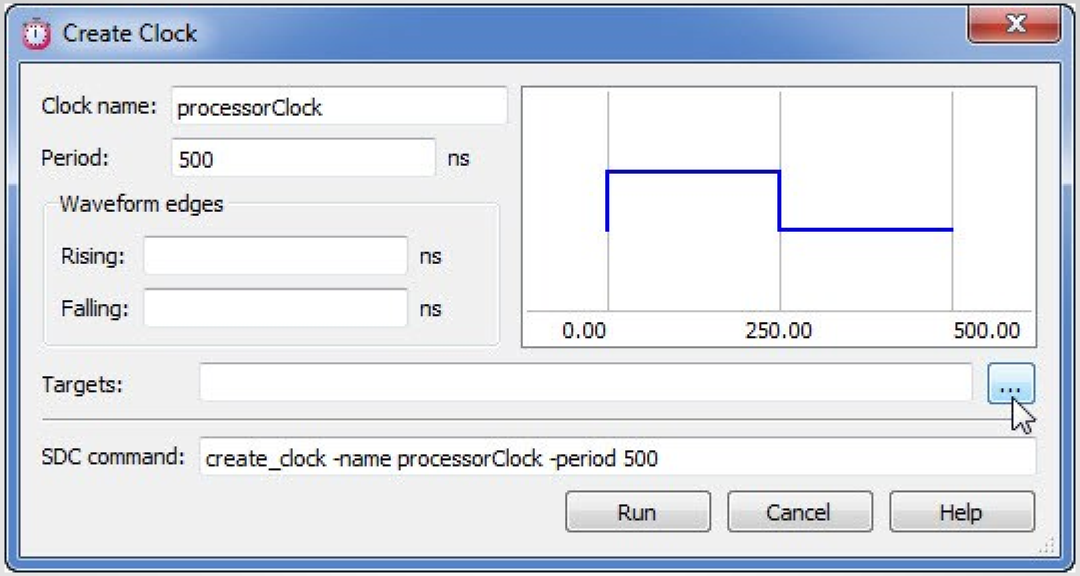
Key Conditioner RTL



ButtonSync RTL

TimeQuest: Two Clocks

Processor clock (from the KEY) and the 50 MHz clock for the filter circuit



Create Clock

Clock name: processorClock

Period: 500 ns

Waveform edges

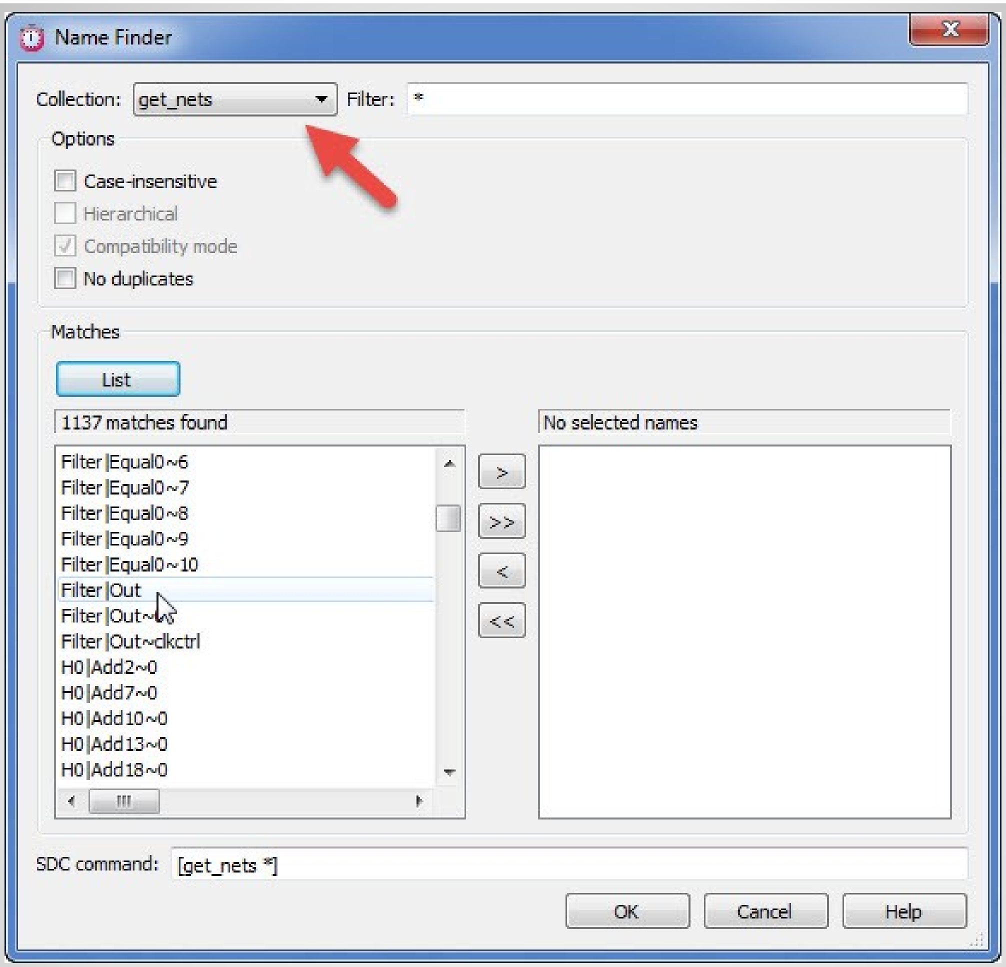
Rising: ns

Falling: ns

Targets:

SDC command:
```
create_clock -name processorClock -period 500
```

To Create Processor Clock



Name Finder

Collection: get_nets Filter: *

Options

- [ ] Case-insensitive
- [ ] Hierarchical
- [✓] Compatibility mode
- [ ] No duplicates

Matches

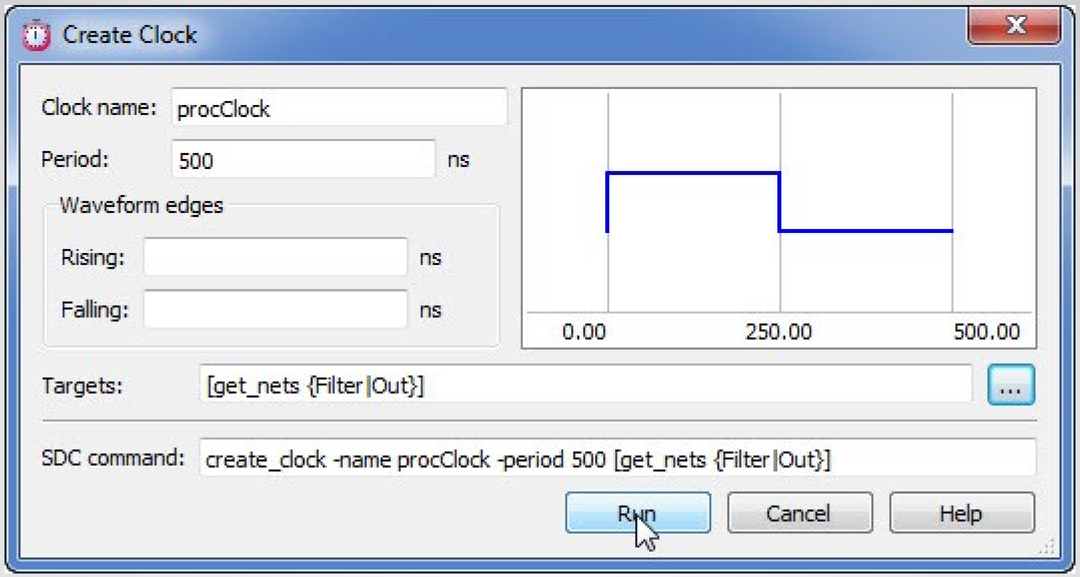
1137 matches found

```
Filter|Equal0~6
Filter|Equal0~7
Filter|Equal0~8
Filter|Equal0~9
Filter|Equal0~10
Filter|Out
Filter|Out~
Filter|Out~clkctrl
H0|Add2~0
H0|Add7~0
H0|Add10~0
H0|Add13~0
H0|Add18~0
```

No selected names

SDC command: [get_nets *]

To Find the Processor Clock



Create Clock

Clock name: procClock

Period: 500 ns

Waveform edges

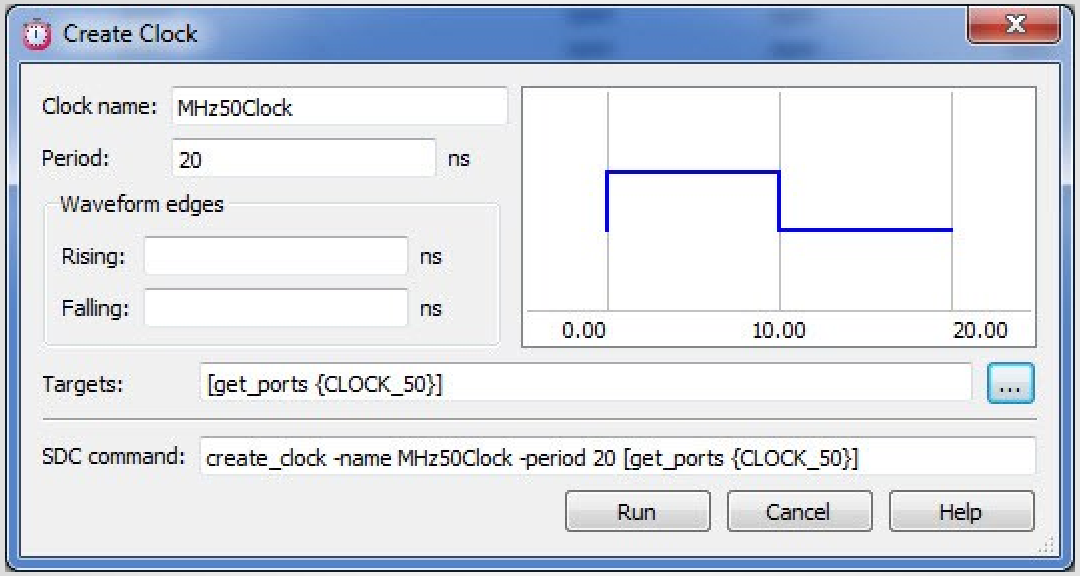
Rising: ns

Falling: ns

Targets: [get_nets {Filter|Out}]

SDC command: create_clock -name procClock -period 500 [get_nets {Filter|Out}]

Run to Create Processor Clock



Clock name: MHz50Clock

Period: 20 ns

Rising: ns

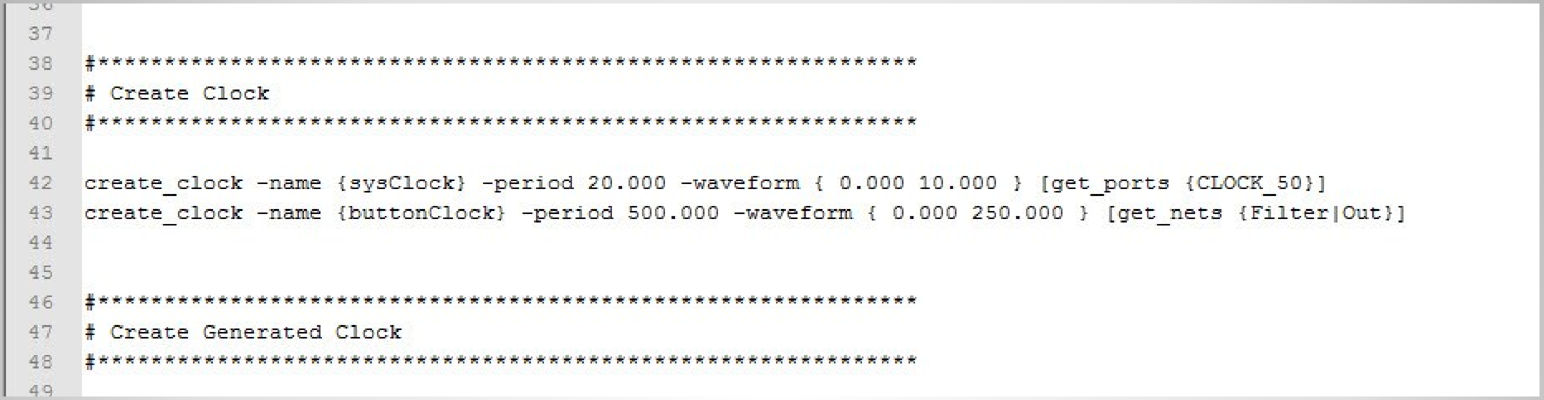
Falling: ns

Targets: [get_ports {CLOCK_50}]

SDC command:

```
create_clock -name MHz50Clock -period 20 [get_ports {CLOCK_50}]
```

Create 50 MHz Clock as Usual



```
#****************************************************************
# Create Clock
#****************************************************************

create_clock -name {sysClock} -period 20.000 -waveform { 0.000 10.000 } [get_ports {CLOCK_50}]
create_clock -name {buttonClock} -period 500.000 -waveform { 0.000 250.000 } [get_nets {Filter|Out}]


#****************************************************************
# Create Generated Clock
#****************************************************************

```

Your sdc File Should Show Two Clocks