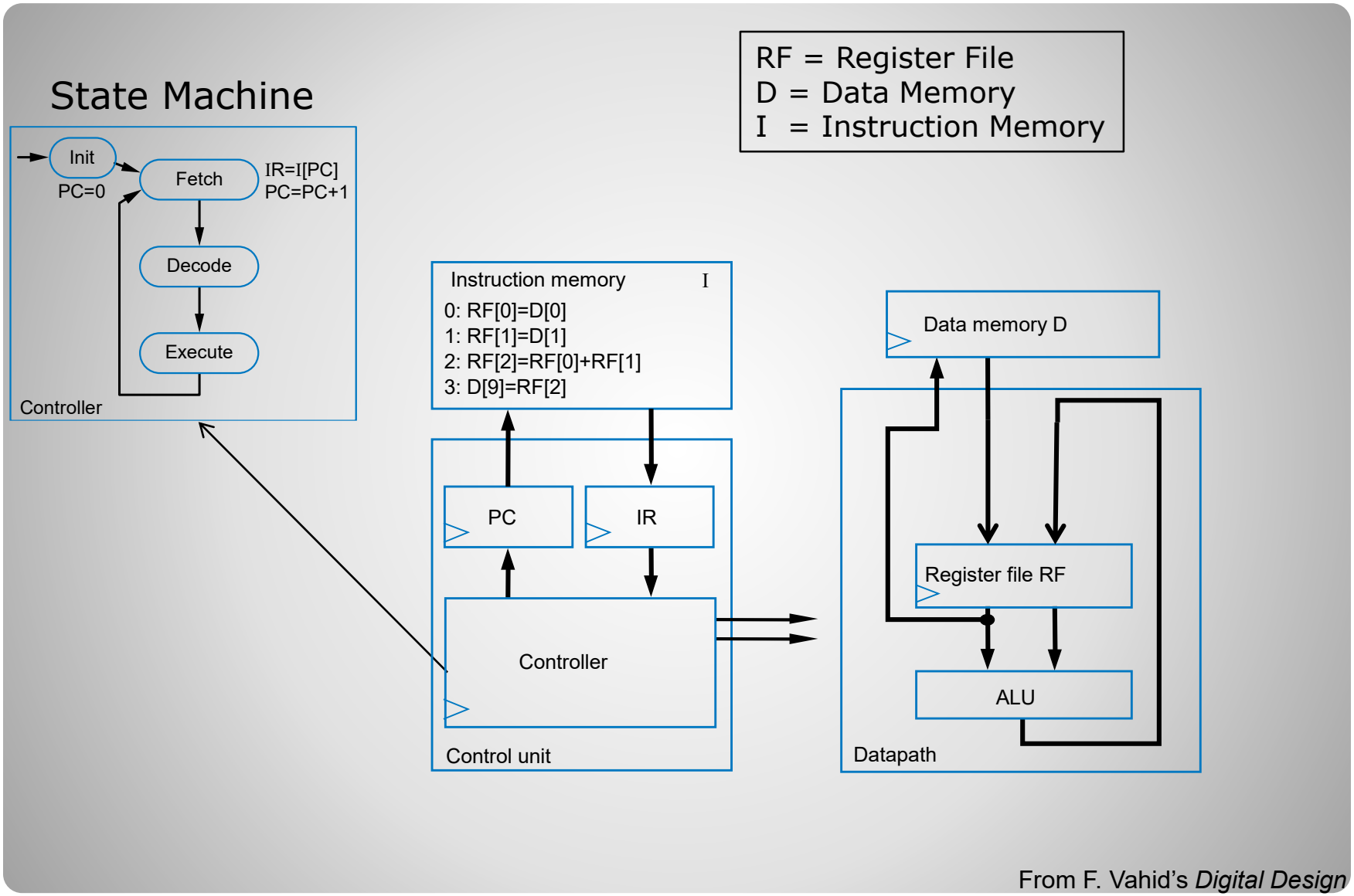
Programmable Processors

Course Project B

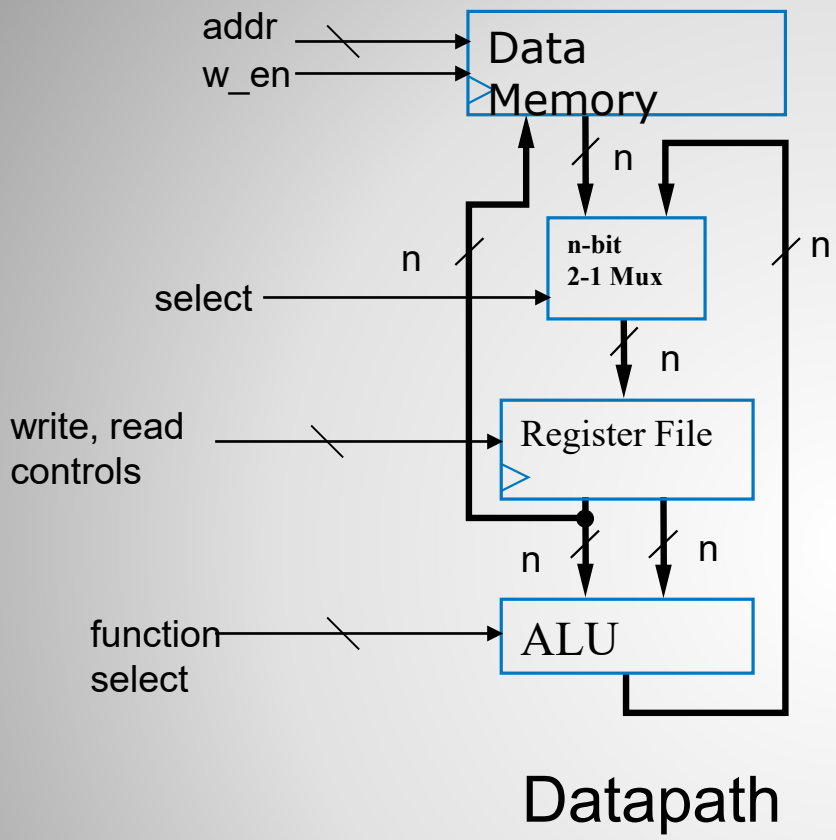
Programmable Processors

- Programmable processors
  also named: ‘stored program’ processors
- Different tasks can be performed by changing the program.
- Well-known examples: MIPS, ARM, Pentium, PIC, Atmel, AtMega



From F. Vahid's *Digital Design*

Basic Architecture

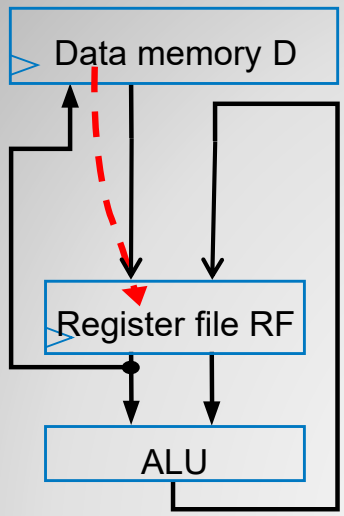


Datapath

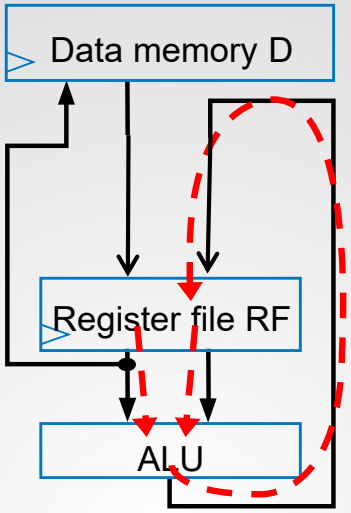
- Takes data from Data Memory
- Stores it in Register File
- Operates on it in ALU (puts it back into the Register File)
- Can store Register File data back into Data Memory
- Note the dual output Register File

Basic Architecture - Datapath

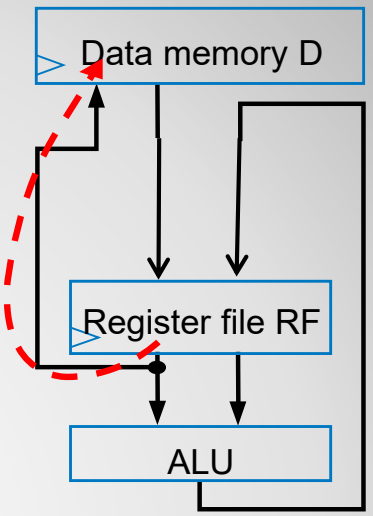
From F. Vahid's *Digital Design*



LOAD operation



ALU operation
(like an ADD)



STORE operation

D[9] = D[0] + D[1] – requires a sequence of four datapath operations:

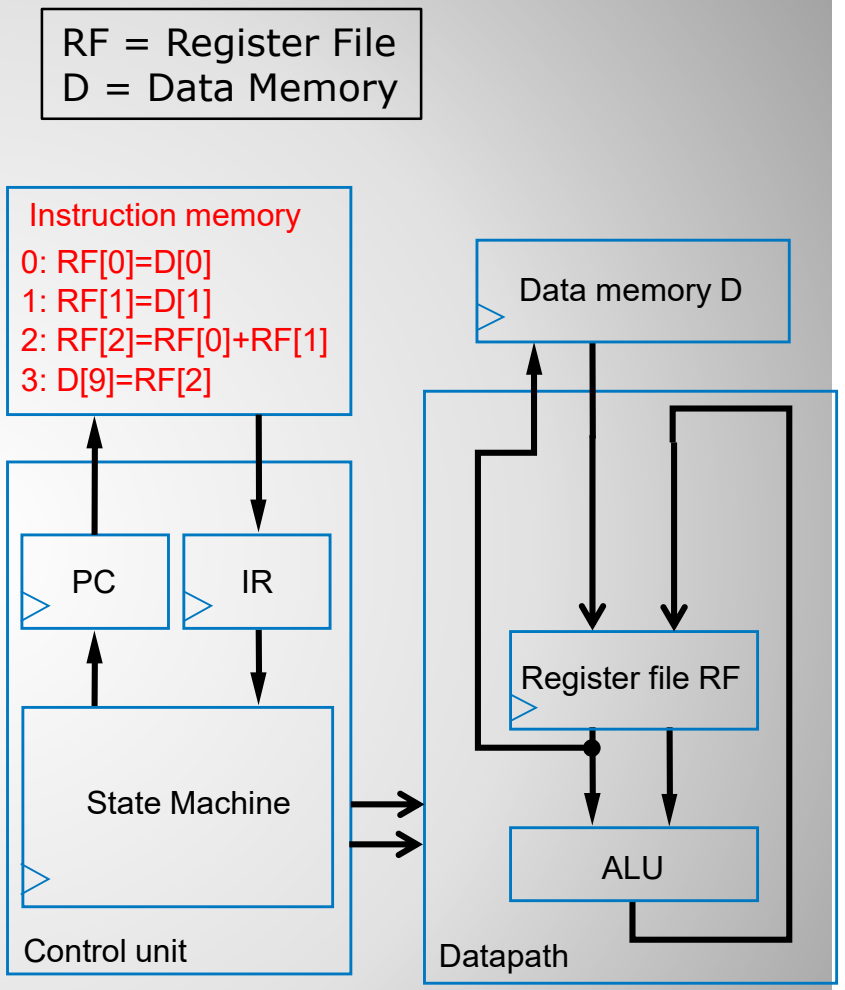
LOAD 0: RF[0] = D[0]
LOAD 1: RF[1] = D[1]
ADD 2: RF[2] = RF[0] + RF[1]
STORE 3: D[9] = RF[2]

Each line here represents one instruction; so it takes four instructions to do this add

From F. Vahid's *Digital Design*

Basic Datapath Operations

- D[9] = D[0] + D[1] – requires a sequence of four datapath operations:
  0: RF[0] = D[0]
  1: RF[1] = D[1]
  2: RF[2] = RF[0] + RF[1]
  3: D[9] = RF[2]
- Each operation is an *instruction*
  - Sequence of instructions = *program*
  - Looks cumbersome, but that's the world of programmable processors – Decomposing desired computations into processor-supported operations
  - Store program in *Instruction memory*
  - *Control unit* reads each instruction and executes it on the datapath
    - PC: Program counter – address of current instruction
    - IR: Instruction register – current instruction



Basic Architecture – Control

From F. Vahid's *Digital Design*

The Sizes of Things

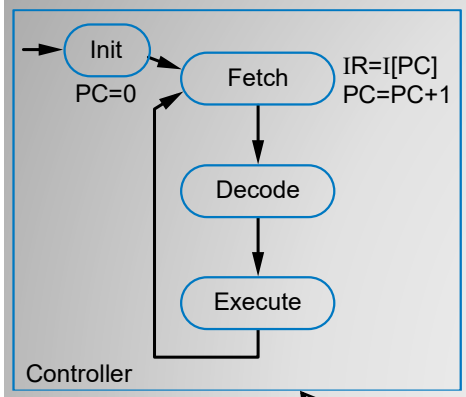
We will take

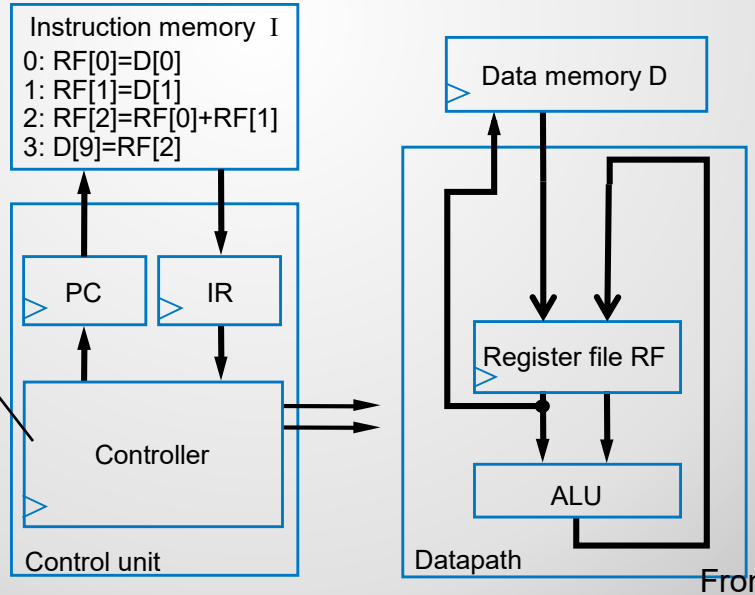
- Data memory (RAM): 256 X 16
- Instruction memory (ROM): 128 X 16
- Register file: 16 X 16
- ALU: Two 16-bit inputs, 16-bit output

Basic Architecture – Control

To summarize, the control unit processes each instruction in three stages:

1. first *fetching* the instruction by loading the current instruction into *IR* and incrementing the *PC* for the next fetch,
2. next *decoding* the instruction to determine its operation, and
3. finally *executing* the operation by setting the appropriate control lines for the datapath, if applicable. If the operation is a datapath operation, the operation may be one of three possible types:
   (a) *loading* a data memory location into a register file location,
   (b) transforming data using an *ALU* operation on register file locations and writing results back to a register file location, or
   (c) *storing* a register file location into a data memory location.





From F. Vahid's *Digital Design*

- Instruction Set – List of allowable instructions and their representation in memory
- Each of our instructions is 16 bits long
- Most of them contain some address information
- General form : **operation source destination**

NOOP instruction – **0000 0000 0000 0000**

STORE instruction – **0001 $r_3r_2r_1r_0$ $d_7d_6d_5d_4d_3d_2d_1d_0$**

LOAD instruction – **0010 $d_7d_6d_5d_4d_3d_2d_1d_0$ $r_3r_2r_1r_0$**

ADD instruction – **0011 $ra_3ra_2ra_1ra_0$ $rb_3rb_2rb_1rb_0$ $rc_3rc_2rc_1rc_0$**

SUBTRACT instruction – **0100 $ra_3ra_2ra_1ra_0$ $rb_3rb_2rb_1rb_0$ $rc_3rc_2rc_1rc_0$**

HALT instruction – **0101 0000 0000 0000**

'r's are Register File locations (4 bits each)
'd's are Data Memory locations (8 bits each)

Data memory (RAM): 256 X 16

Register file: 16 X 16

Six-Instruction Processor

NOOP Instruction

- **0000 0000 00000000**
- Always starts with **0000** (and contains nothing but zeros)
- Actually is just 0000 xxxx xxxxxxxx, since the 12 bits following 0000 are ignored.
- NoOp = No Operation
- Just takes up space (or time)
- Does not involve the Datapath
- If your program starts executing a string of 0s, nothing (too) bad will happen.

STORE Instruction

- **0001 $r_3r_2r_1r_0$ $d_7d_6d_5d_4d_3d_2d_1d_0$**
- Always starts with **0001**
- **$r_3r_2r_1r_0$** is a 4-bit Register File address (source)
- **$d_7d_6d_5d_4d_3d_2d_1d_0$** is a Data Memory address (destination)
- Moves 16 bits of data from the Register File to Data Memory
- Example: 0001 1111 0010 1001b (or 0x1F29) means "move the data at Register File 0xF (15d) to Data Memory location 0x29 (41d)

LOAD Instruction

- **0010 $d_7d_6d_5d_4d_3d_2d_1d_0$ $r_3r_2r_1r_0$**
- Always starts **0010**
- **$d_7d_6d_5d_4d_3d_2d_1d_0$** is a Data Memory address (source) – here d implies D memory
- **$r_3r_2r_1r_0$** is a 4-bit Register File address (destination)
- Moves 16 bits of data from Data Memory to the Register File
- Example : 0010 00001010 0111b (or 0x20A7) means “move the data at Data Memory location 0x0A (10d) to Register File 7.”

ADD Instruction

- **0011 $ra_3ra_2ra_1ra_0$ $rb_3rb_2rb_1rb_0$ $rc_3rc_2rc_1rc_0$**
- Always starts with **0011**
- **$ra_3ra_2ra_1ra_0$** is a 4-bit Register File address (source for A)
- **$rb_3rb_2rb_1rb_0$** is a 4-bit Register File address (source for B)
- **$rc_3rc_2rc_1rc_0$** is a 4-bit Register File address (destination)
- Adds the 16 bits of A to 16 bits of B, stores the result in C (all these in the Register File)
- Example: 0011 0001 0010 0011b (or 0x3123) adds the data at Register File 1 to the data at Register File 2 and stores the result in Register File 3.

SUBTRACT Instruction

- **0100 $ra_3ra_2ra_1ra_0$ $rb_3rb_2rb_1rb_0$ $rc_3rc_2rc_1rc_0$**
- Always starts with **0100**
- **$ra_3ra_2ra_1ra_0$** is a 4-bit Register File address (source for A)
- **$rb_3rb_2rb_1rb_0$** is a 4-bit Register File address (source for B)
- **$rc_3rc_2rc_1rc_0$** is a 4-bit Register File address (destination)
- Subtracts the 16 bits of B from the 16 bits of A, stores the result in C (all these in the Register File)
- Example: 0100 0001 0010 0011b (or 0x4123) subtracts the data at Register File 2 from the data at Register File 1 and stores the result in Register File 3.

HALT Instruction

- 0101 0000 00000000
- Always starts 0101 and remainder is all 0s
- Actually is 0100 xxxx xxxxxxxx, since all the bits following 0101 are ignored
- Causes the processor to just stop (**hard stop**)
- You need to reload it or reset it to get it going again.
- Does not involve the Datapath

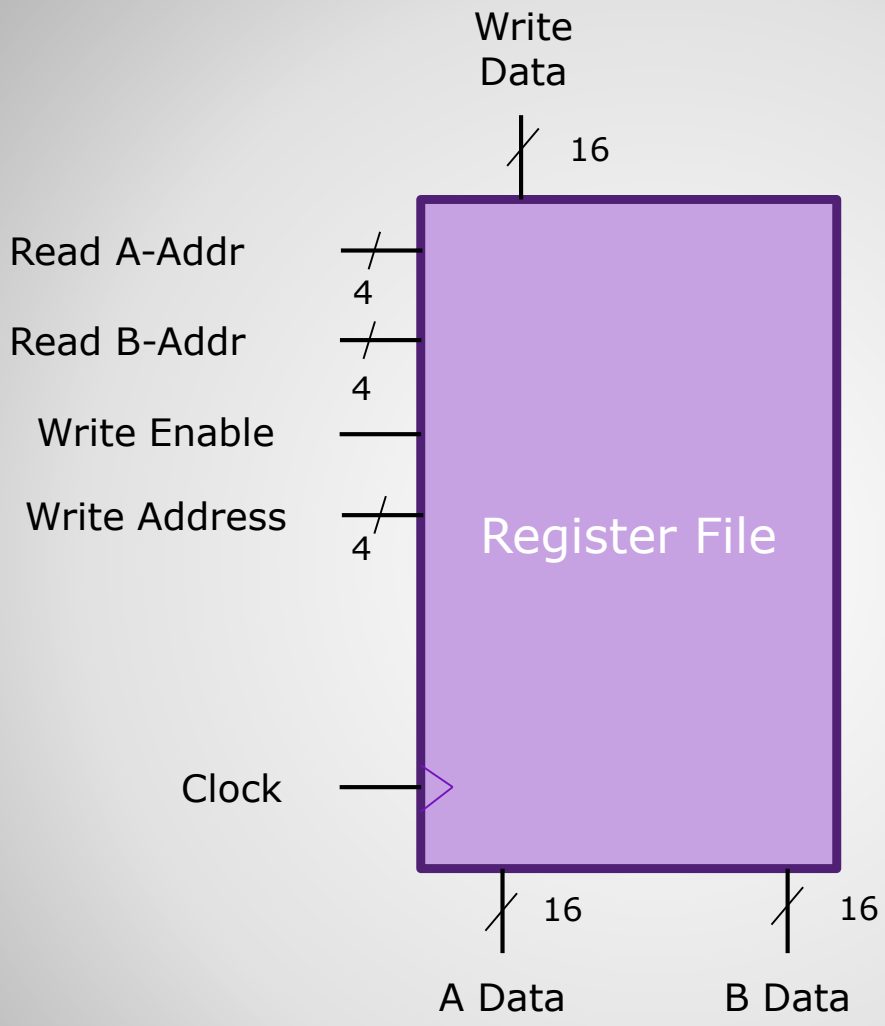
First Program

- Sample program:
  - 0: RF[0] = D[0]
  - 1: RF[1] = D[1]
  - 2: RF[2] = RF[0] + RF[1]
  - 3: D[9] = RF[2]
  - 4: HALT
- Becomes:

This is what gets stored in Instruction Memory

0x2000
0x2011
0x3012
0x1209
0x5000

LOAD 0 0	or 0010 00000000 0000b
LOAD 1 1	or 0010 00000001 0001b
ADD 0 1 2	or 0011 0000 0001 0010b
STORE 2 9	or 0001 0010 00001001b
HALT	or 0101 0000 00000000b



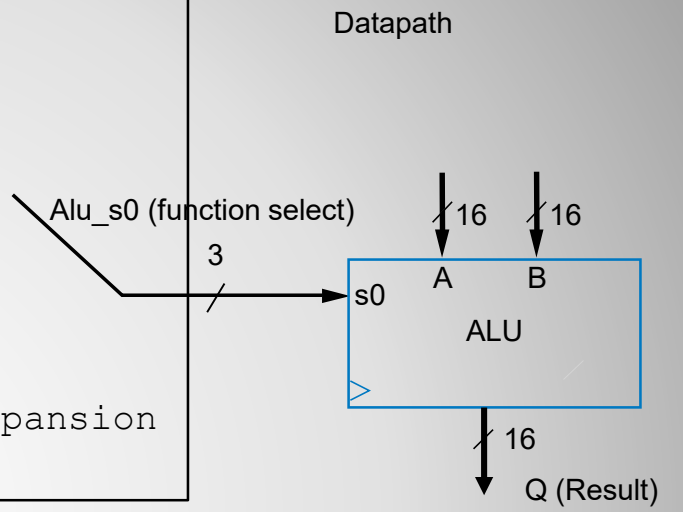
The Register File

```
// 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

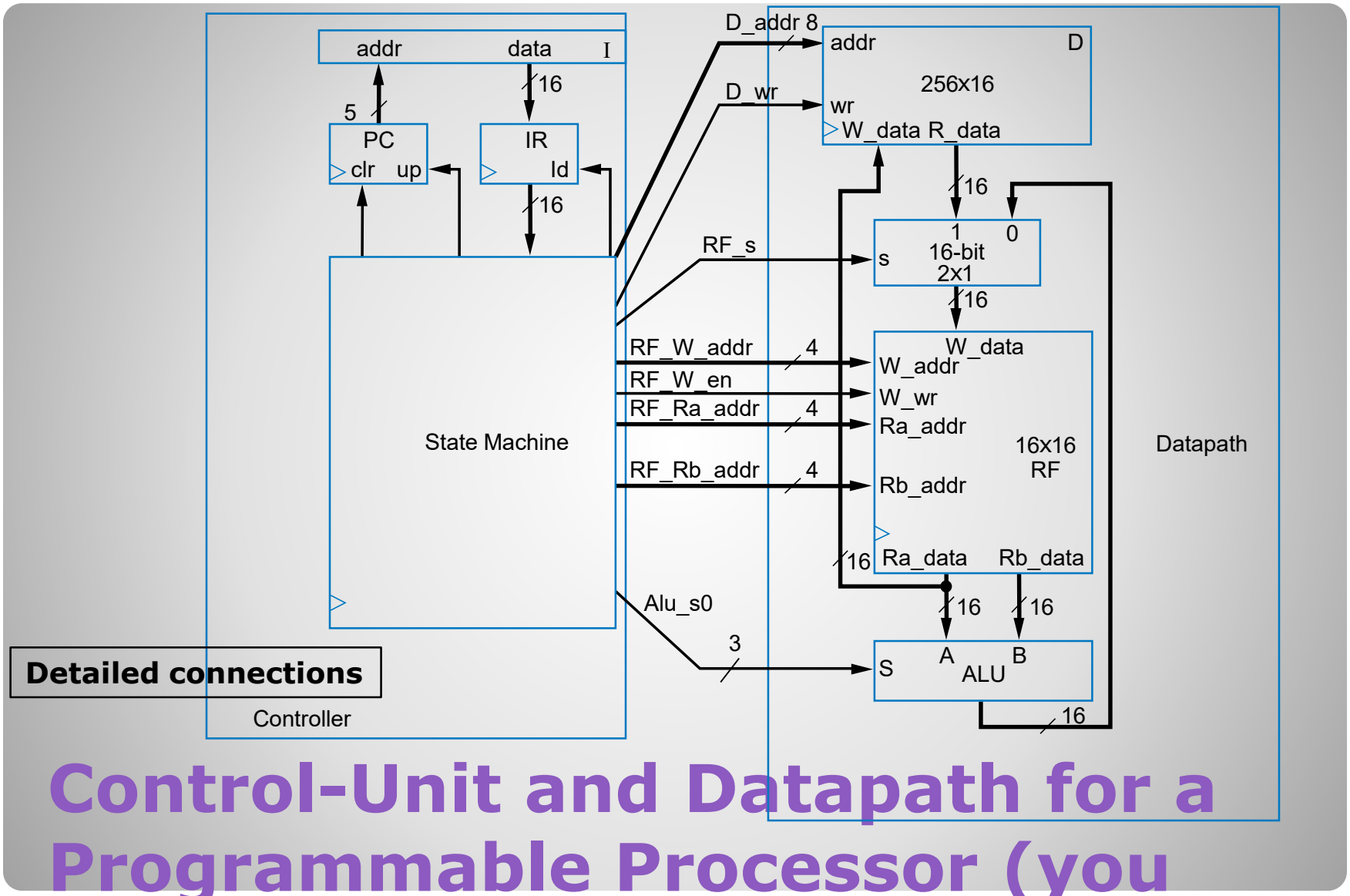
Suggested Register File Idea

```
// 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
```

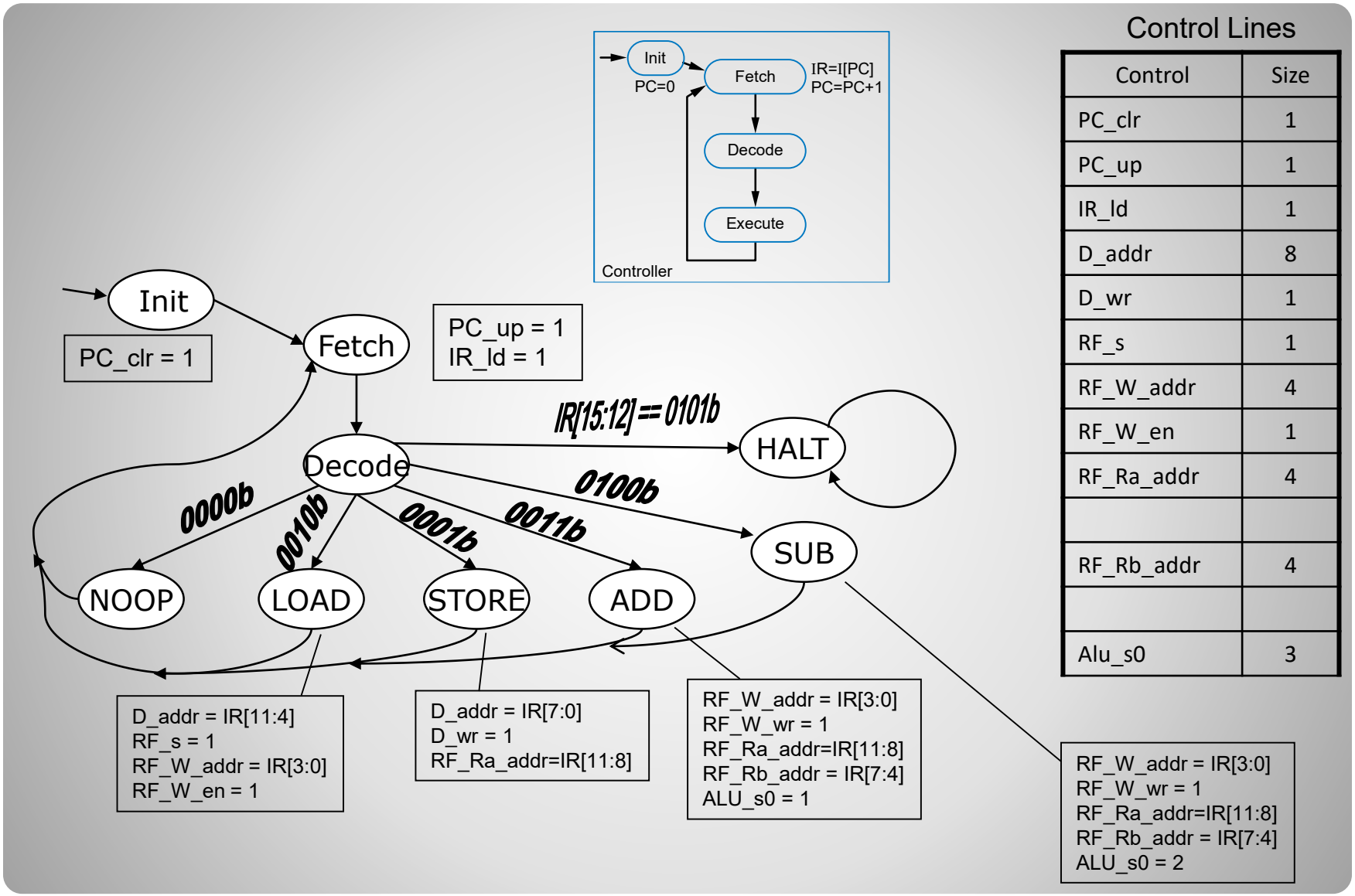



```
module ALU( A, B, Sel, Q );
  input [2:0] Sel;     // function select
  input [15:0] A, B;   // input data

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

The ALU



Control-Unit and Datapath for a Programmable Processor (you need some change for ProjectB)



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

Controller State Machine

State Machine Output Definitions

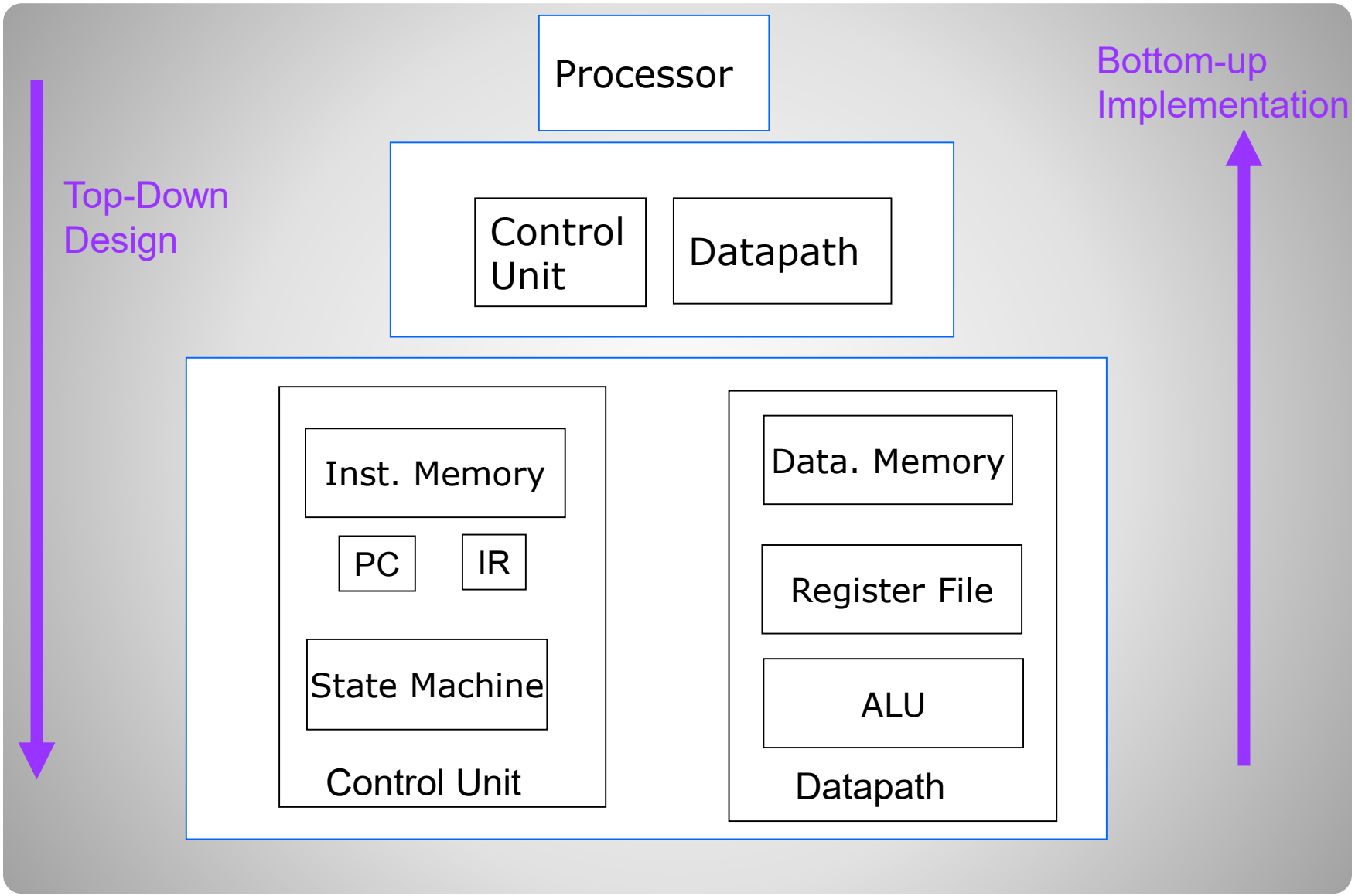
Variable	Meaning
PC_clr	Program counter (PC) clear command
PC_ld	PC register load command
PC_inc	PC increment command
D_addr	Data memory address (8 bits)
D_wr	Data memory write enable
RF_s	Mux select line
RF_Ra_addr	Register file A-side read address (4 bits)
RF_Rb_addr	Register file B-side read address (4 bits)
RF_Wen	Register file write enable
RF_W_Addr	Register file write address (4 bits)
ALU_s	ALU function select (3 bits)

State Machine State Outputs

State	Non-Zero Outputs
Init	PC_clr = 1
Fetch	IR_ld = 1
Decode	PC_inc = 1
Load	D_addr = IR[11:4], RF_s = 1, RF_A_addr = IR[3:0], RF_Wen = 1
Store	D_addr = IR[7:0], D_wr = 1, RF_A_addr = IR[11:8]
Add, Sub	RF_A_addr = IR[11:8], RF_B_addr = IR[7:4], RFWAddr = IR[3:0], RF_Wen = 1, ALU_s = alu function (1 for Add or 2 for Sub), RF_s = 0
Halt	

State Machine Considerations

- It is recommended to use the combinational loop/sequential loop version.
- Give all your states names using localparams or enums.
- Initialize all outputs to zero at the top of the combinational loop (so that each state has to set only non-zero outputs).
- Set the non-zero outputs in the appropriate state case.
- Sequential part should just check for reset and set state = next state (that's all).



Processor Module Layout

Procedure

- Write a module (lowest level).
- TEST IT!
- Repeat until you have written all low-level modules.
- Is your state machine recognized as such by Quartus?
- Write the modules next level up (Control Unit, Datapath).
- TEST THESE!
- Write the Processor by instantiating Control Unit module and Datapath module and wiring them together.
- At this point you should be able to test your entire processor in ModelSim.
- Final test for the Processor is running the program and inspecting Data Memory to make sure the correct value is stored there in the correct location.
- NOTE: You can do your unit testing on the DE2 or you can use ModelSim – It's probably better to test with ModelSim because you can more easily debug in this environment.