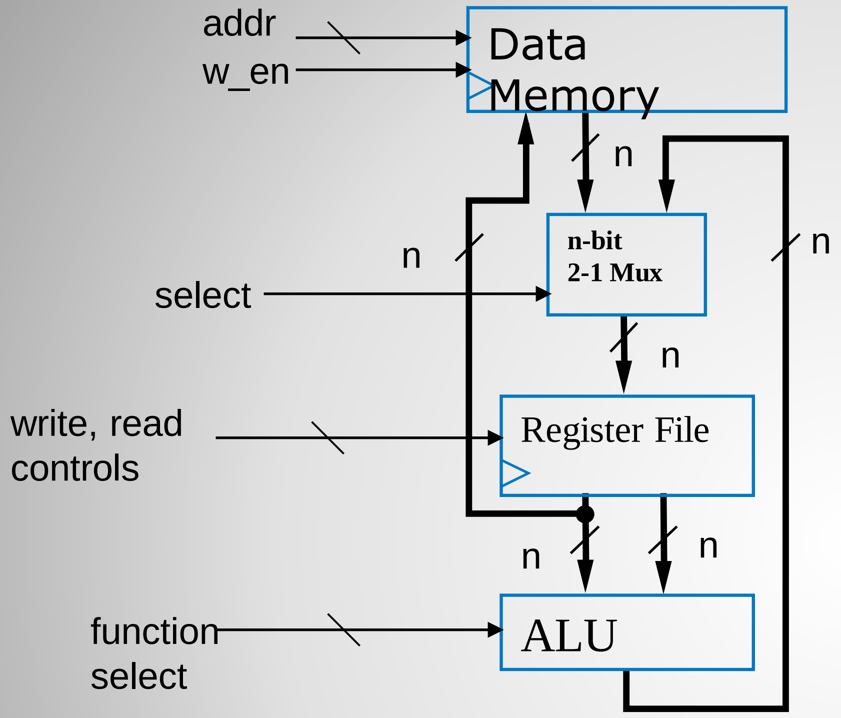
Programmable Processors

Programmable Processors

- So far we’ve looked at ‘hard-wired’ or special purpose processors (like the SAD processor).
- We’ll now look at programmable processors or ‘stored program’ processors.
- Different tasks can be performed by changing the program.
- Well-known examples: Pentium, PIC, ARM, Atmel AtMega, MIPS

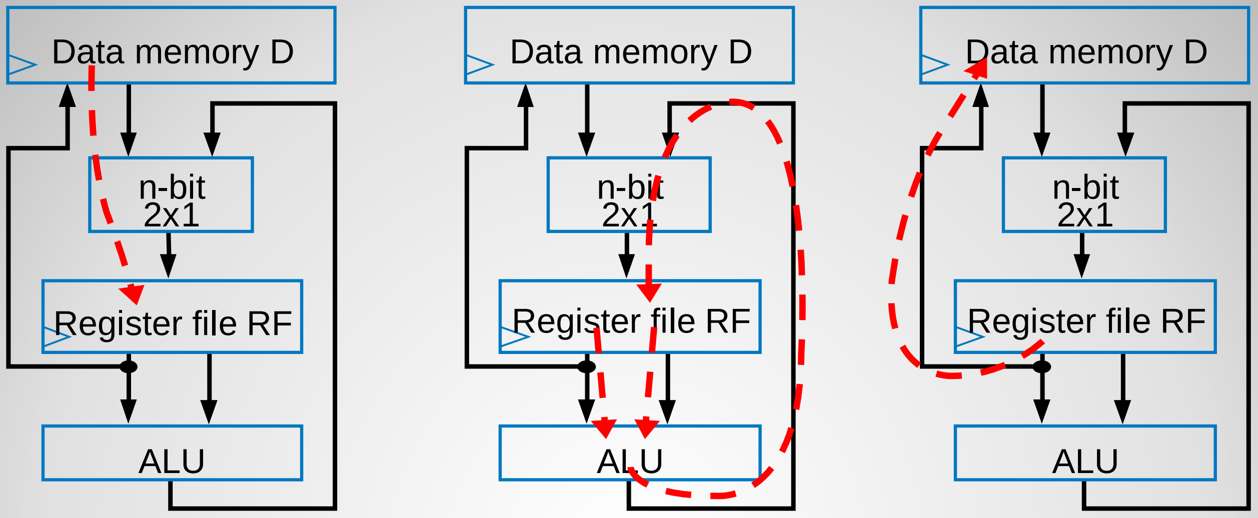


Datapath

- 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
- We've built every piece of this already

Datapath Side

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

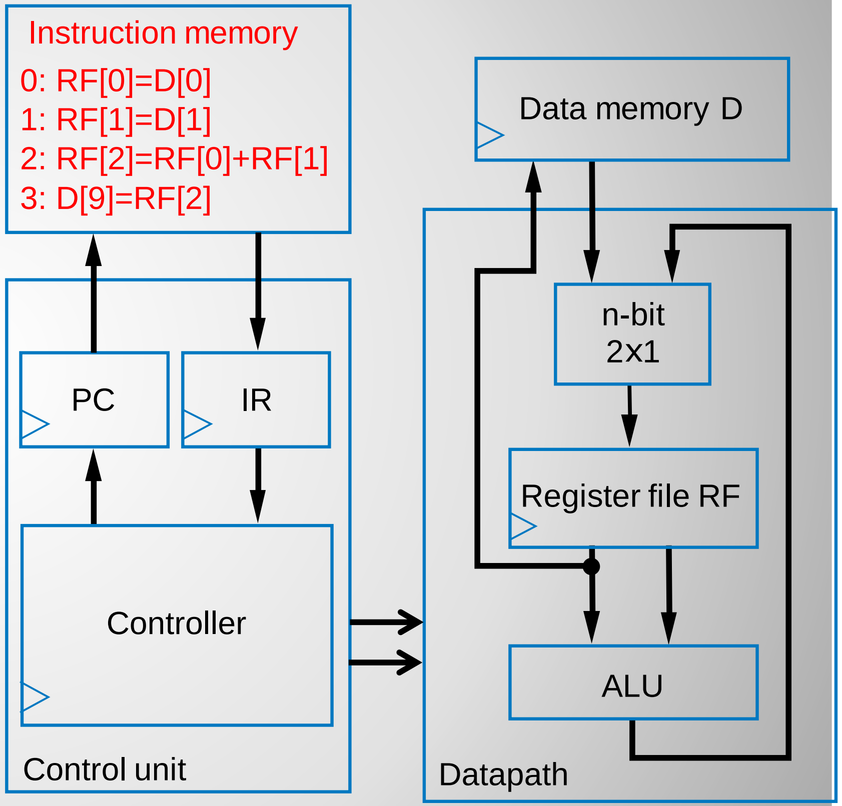
From F. Vahid's *Digital Design*

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

a

RF = Register File
D = Data Memory



Basic Architecture – Control Unit

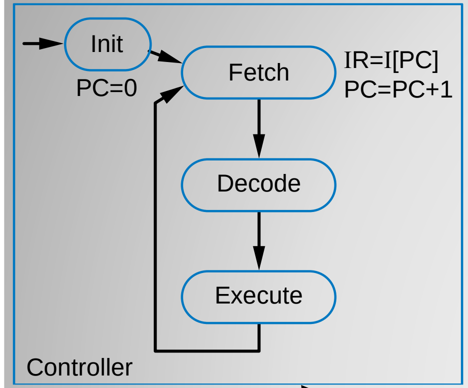
From F. Vahid's *Digital Design*

The Size of Things

- Data memory (RAM): 256 X 16
- Register file: 16 X 16 (dual output)
- ALU: Two 16-bit inputs, 16-bit output
- Instruction memory (ROM):32 X 16

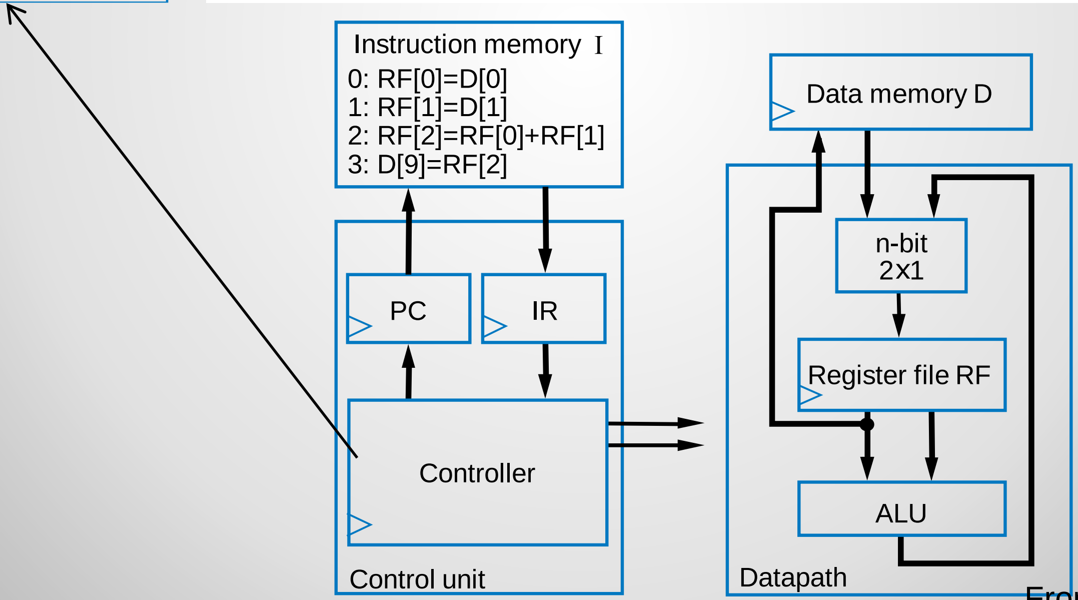
Basic Architecture – Control

State Machine



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*

Six-Instruction Processor

- 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)

- **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.

NOOP 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)

STORE Instruction

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)
- **$r_3r_2r_1r_0$** 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

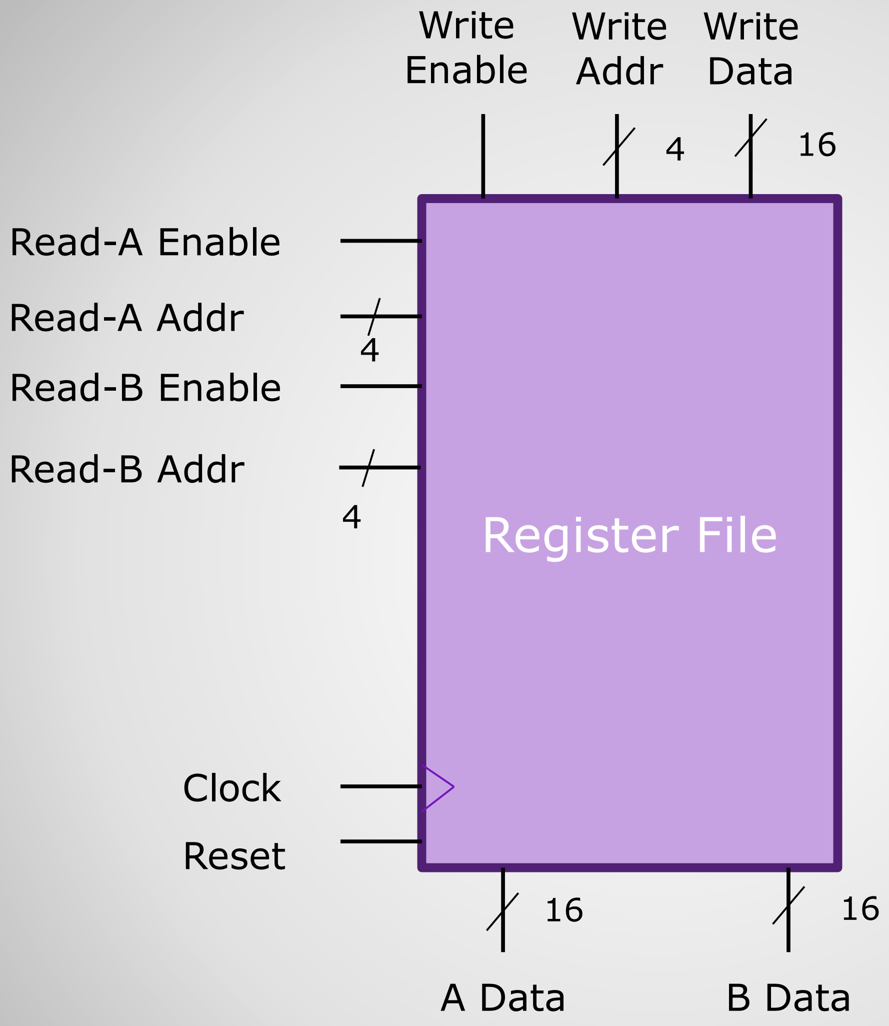
- Our program:
  - 0: RF[0] = D[0]
  - 1: RF[1] = D[1]
  - 2: RF[2] = RF[0] + RF[1]
  - 3: D[9] = RF[2]
  - 4: HALT

This is what gets stored in Instruction Memory

```
0x2000
0x2011
0x3012
0x1209
0x5000
```

- Becomes:

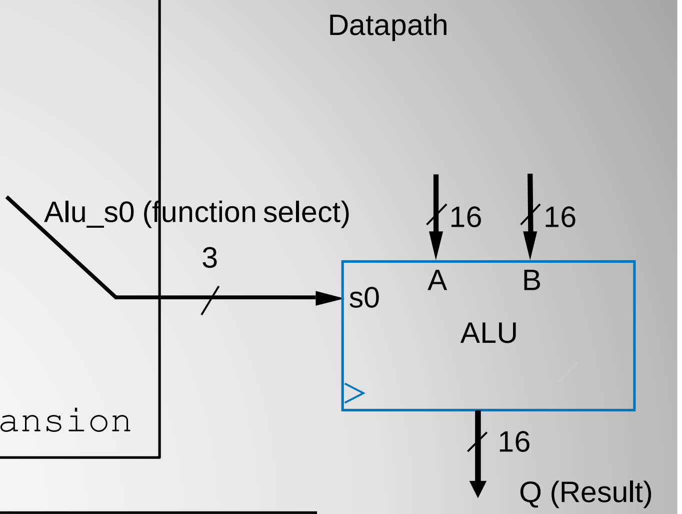
LOAD 0 0	or 0010 00000000 0000b
LOAD 1 1	or 0010 00000001 0001b
ADD 0 1 2	or 0011 0000 0001 0010b
STORE 2 9	or 0001 0010 00001001b
HALT	or 0101 0000 00000000b



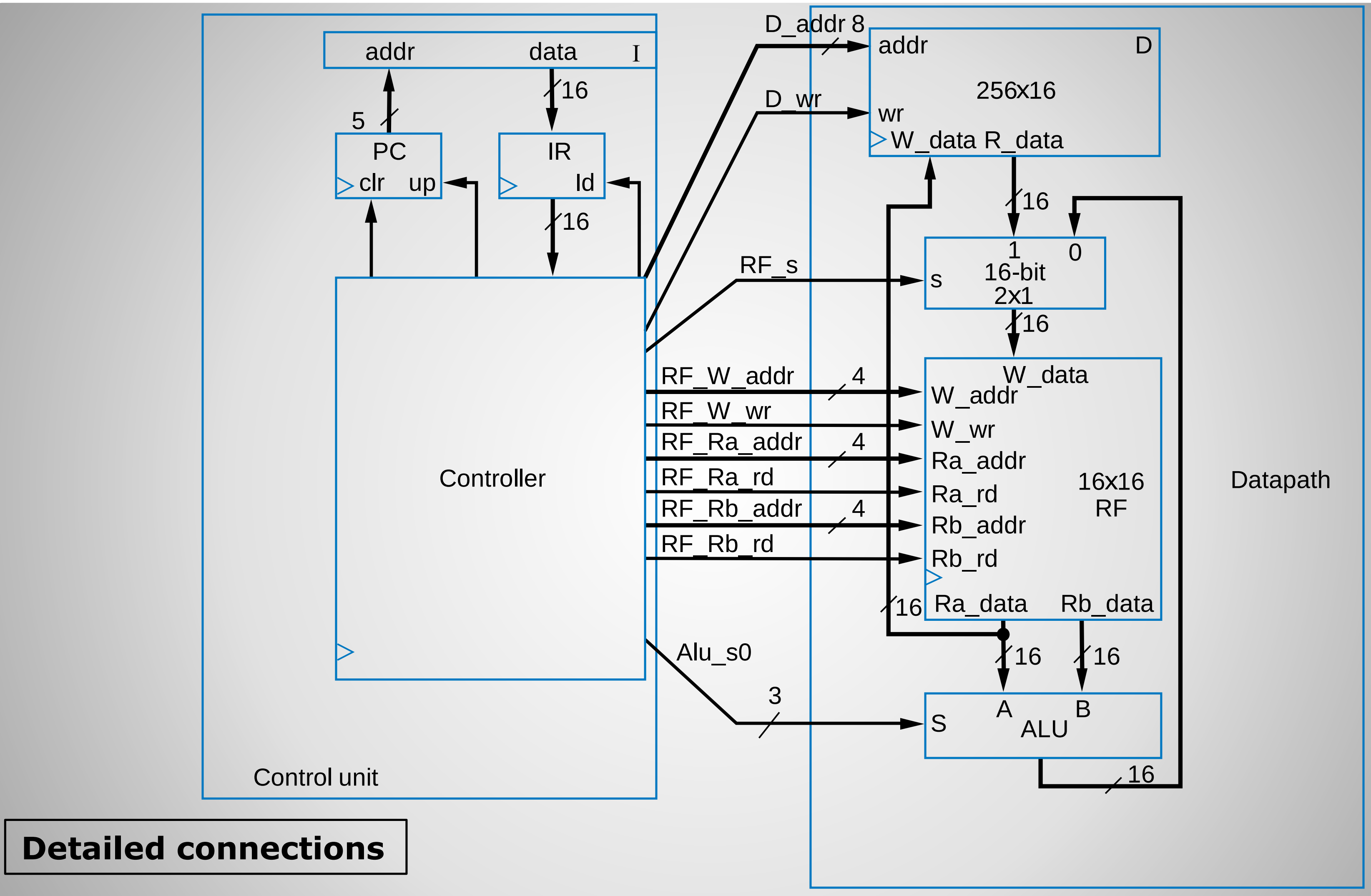
The Register File (from HW6)

The ALU (from HW6)

```
// 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;
// the additional functions are for future expansion
```



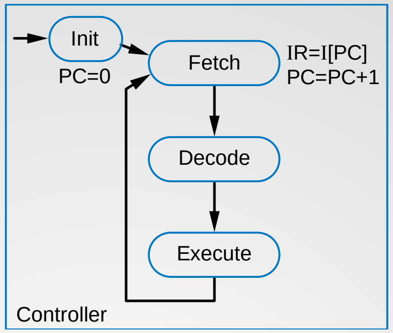
```
module ALU( A, B, Sel, Q );
  input [2:0] Sel;      // function select
  input [15:0] A, B;    // input data

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

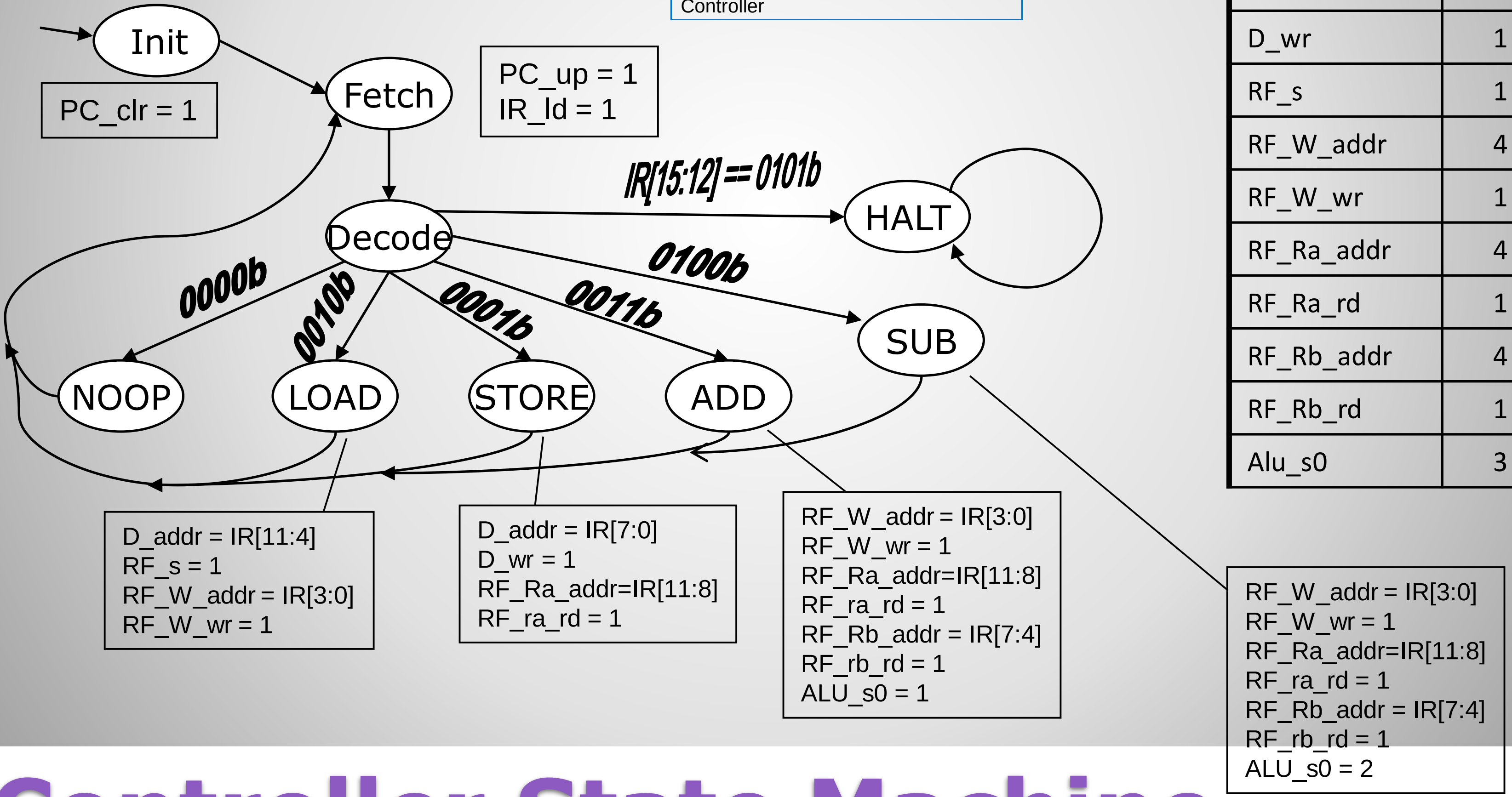


Detailed connections

Control-Unit and Datapath for Our Programmable Processor

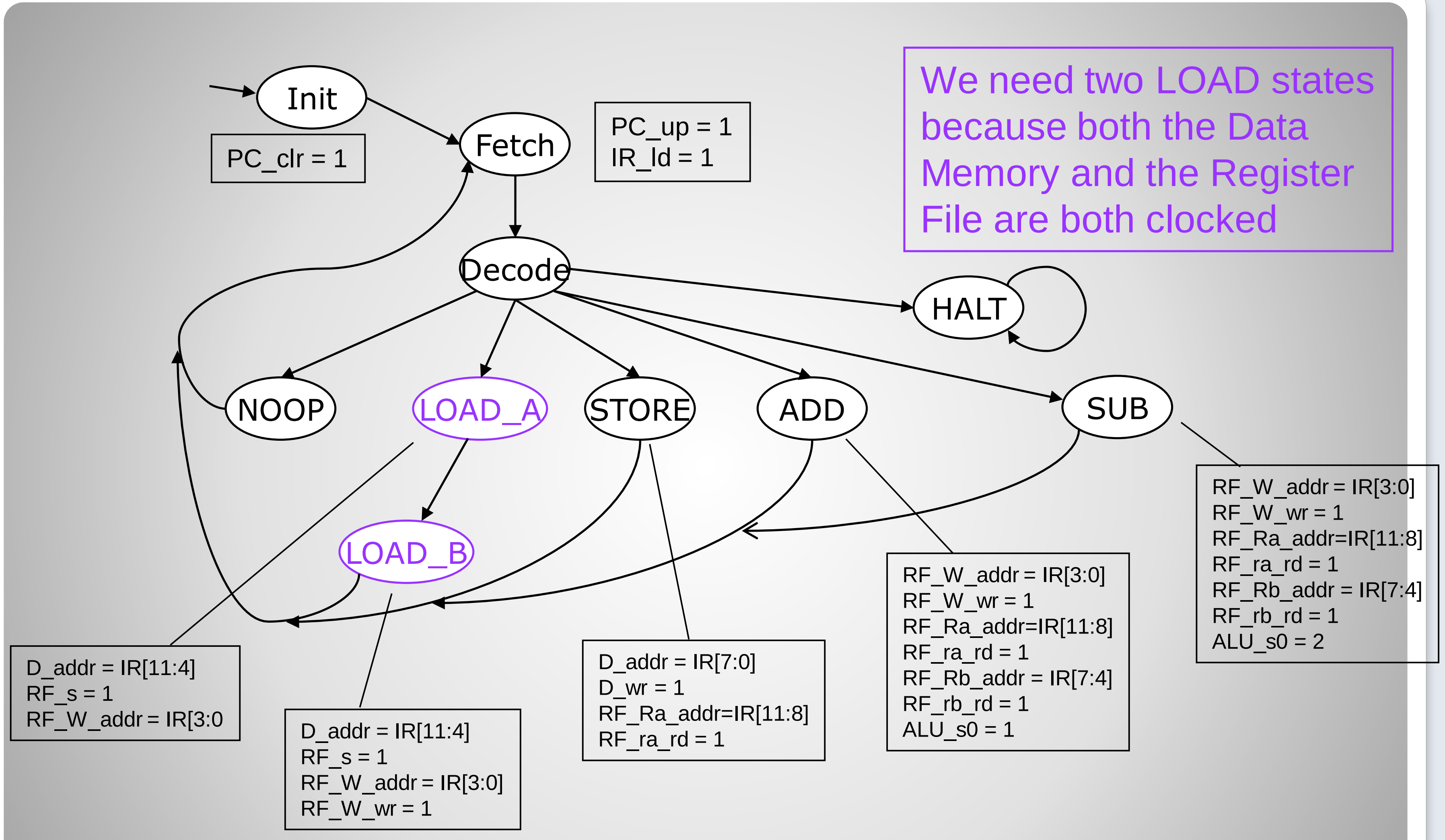
Controller State Machine





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_wr	1
RF_Ra_addr	4
RF_Ra_rd	1
RF_Rb_addr	4
RF_Rb_rd	1
Alu_s0	3

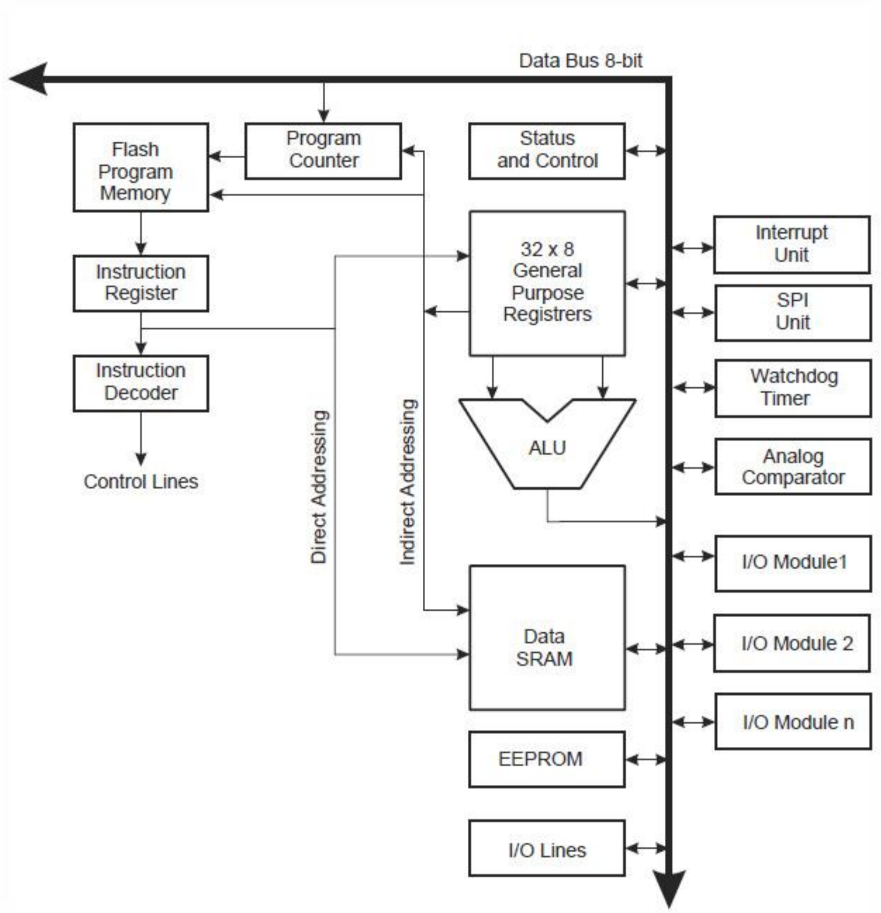


We need two LOAD states because both the Data Memory and the Register File are both clocked

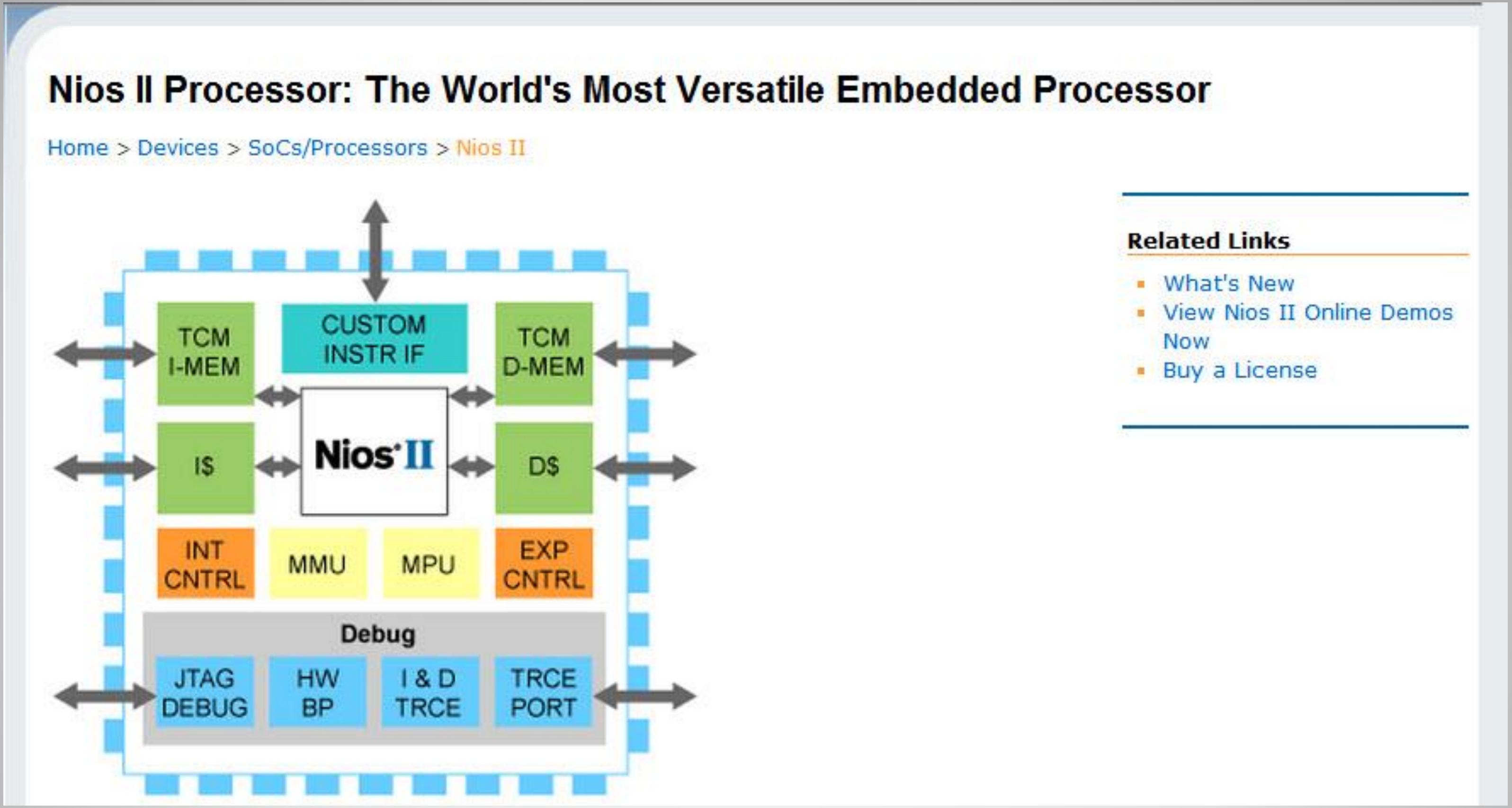
LOAD Anomaly

6.2 Architectural Overview

Figure 6-1. Block Diagram of the AVR Architecture

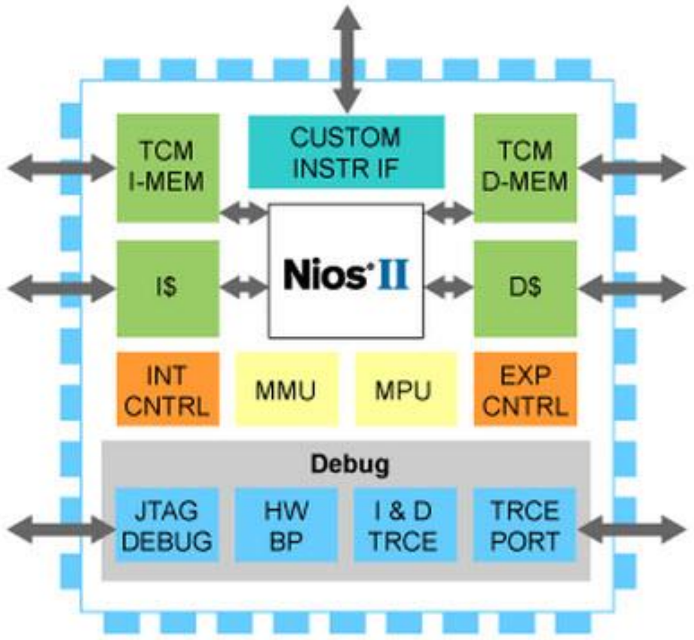


ATMEL 168 Microprocessor



Nios II Processor: The World's Most Versatile Embedded Processor

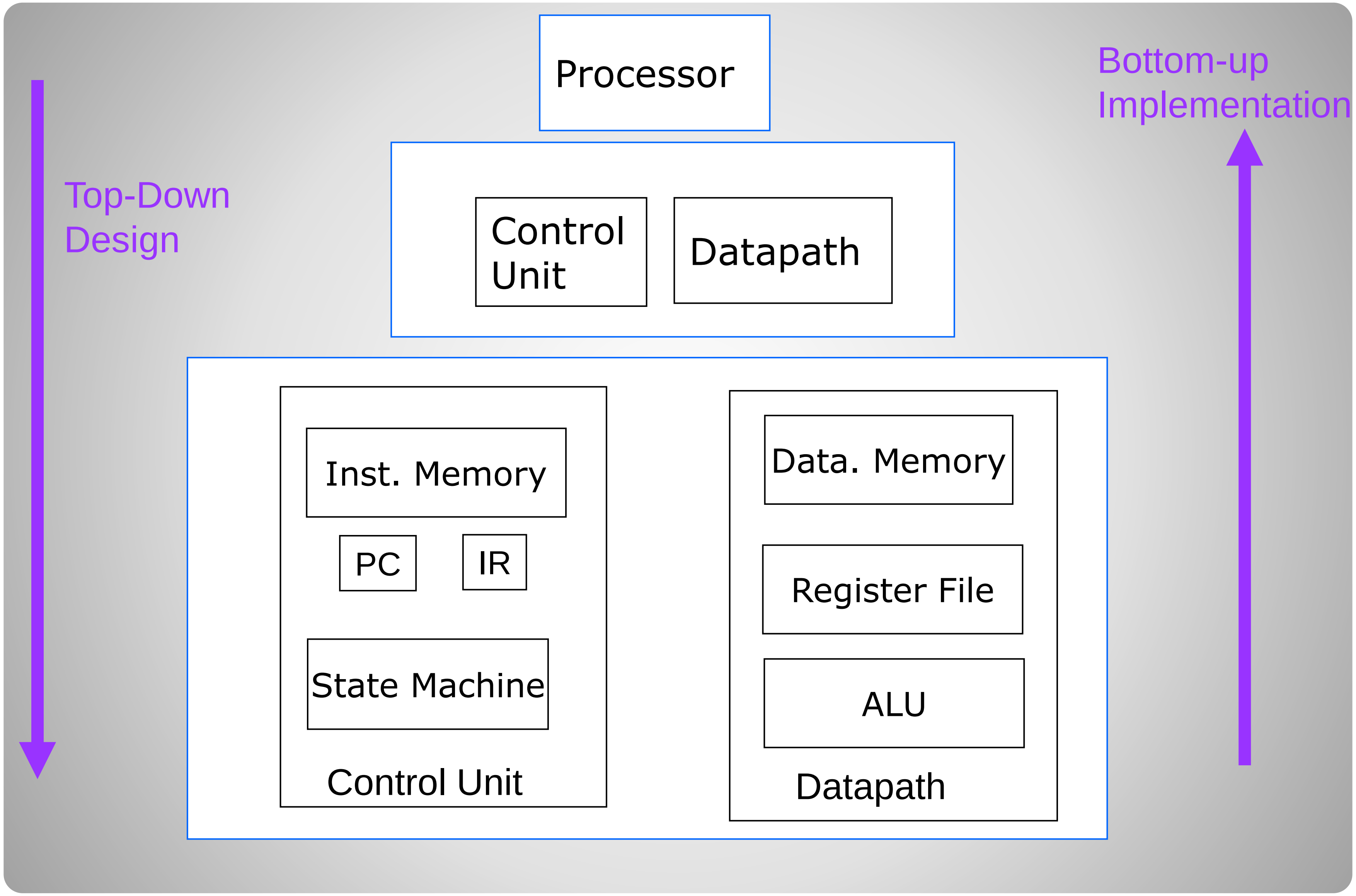
Home > Devices > SoCs/Processors > Nios II



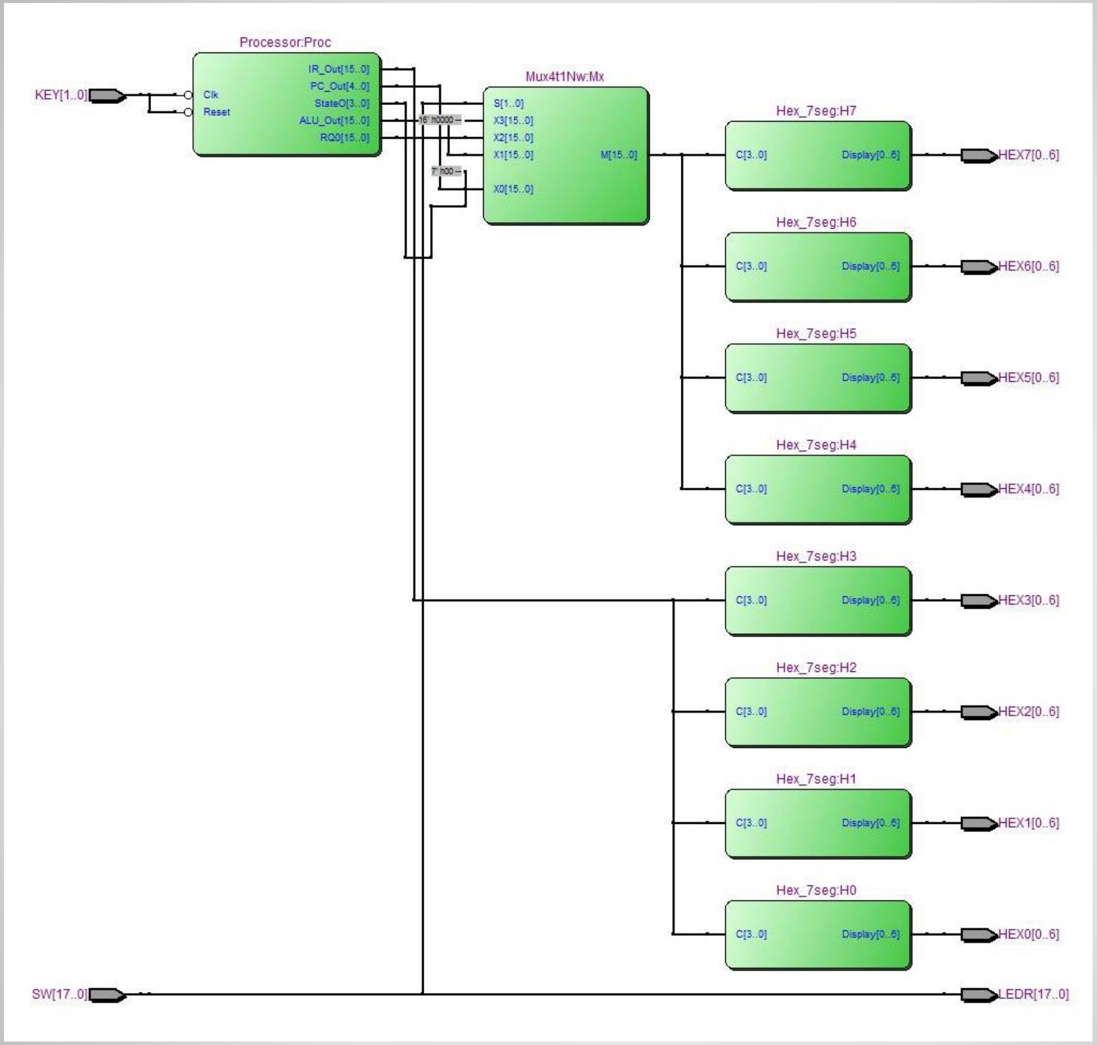
Related Links

- What's New
- View Nios II Online Demos Now
- Buy a License

Altera's 'Soft' Processor

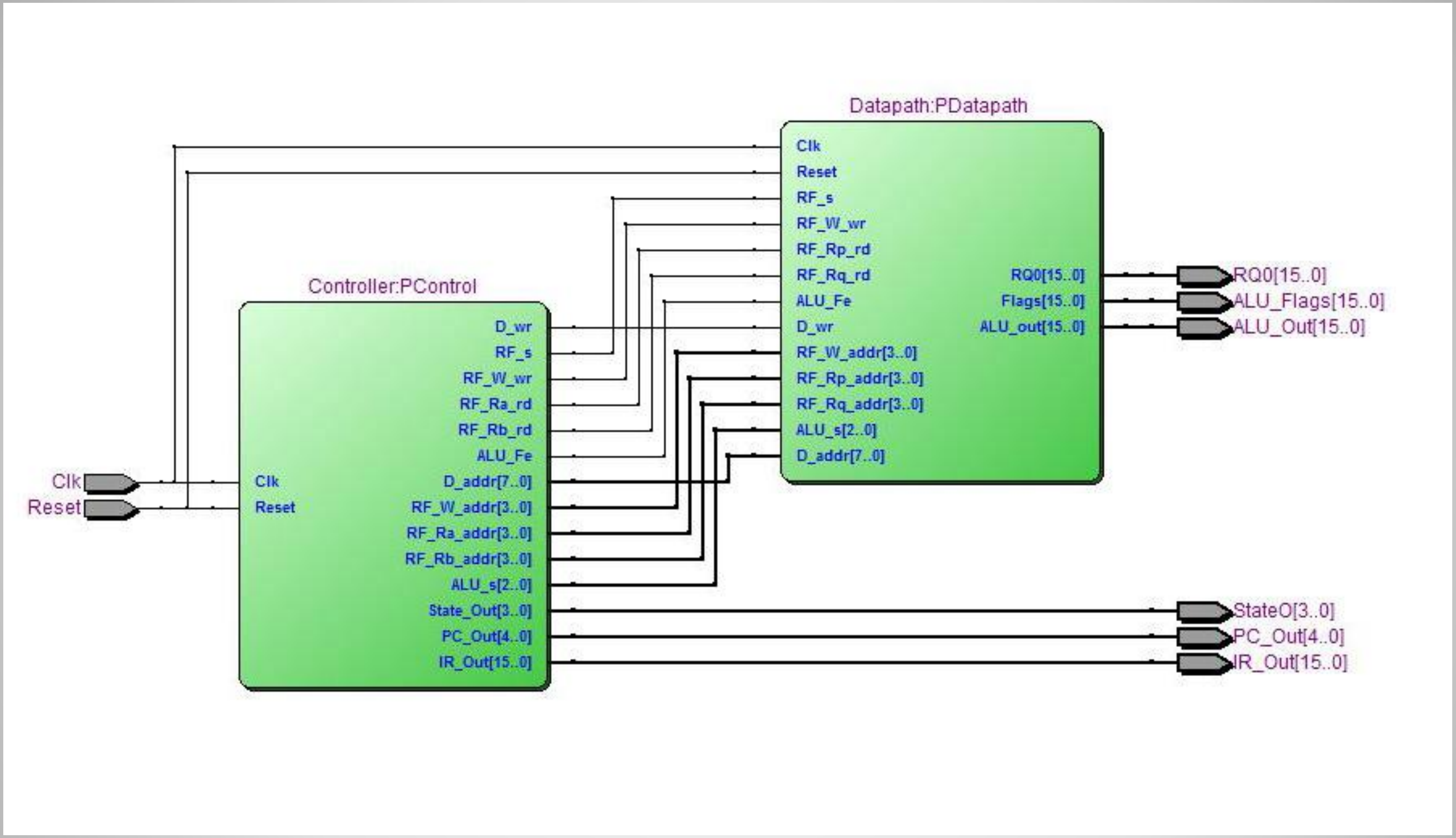


Processor Module Layout



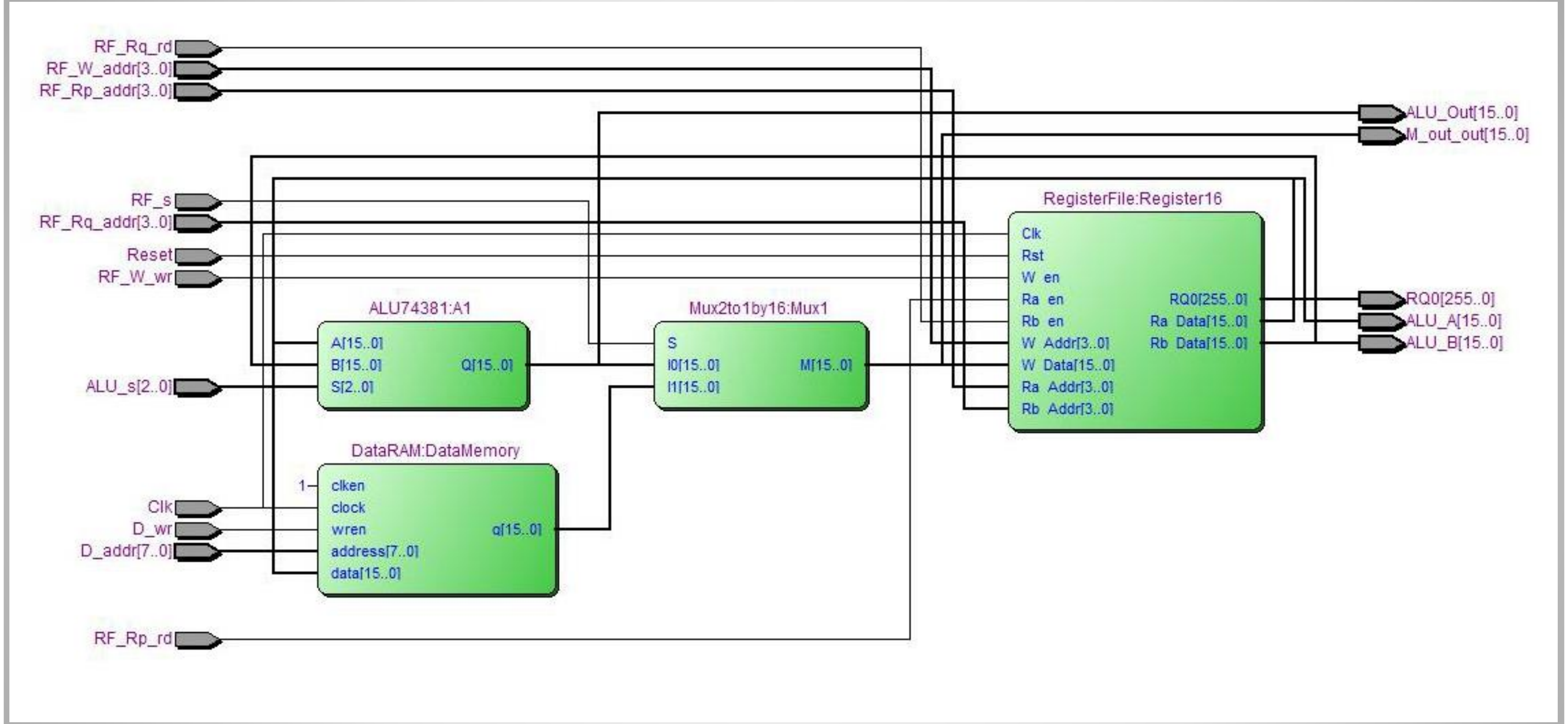
Top Level

Note: Contains Extra Debug Information



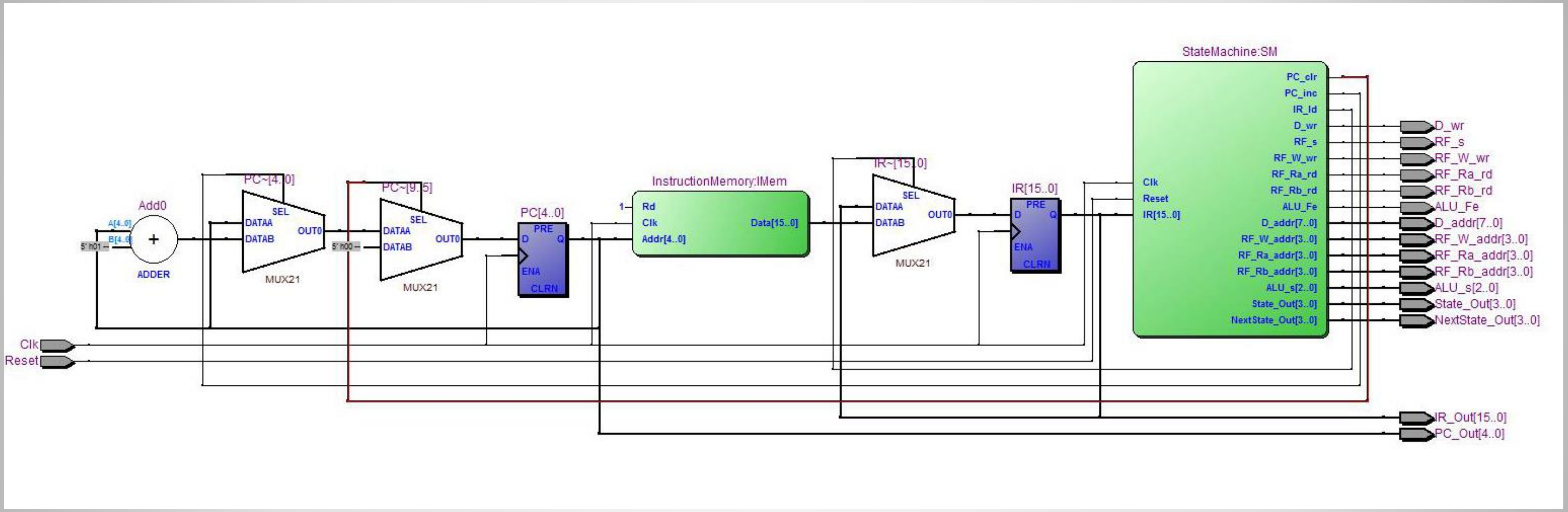
Note: Contains Extra Debug Information

Processor Level



Note: Contains Extra Debug Information

Datapath Level



Note: Contains Extra Debug Information

Controller Level

- Write a module (lowest level).
- TEST IT!
- Repeat until you have written all low-level modules.
- Make sure your state machine is 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. Remember you will need to create a ModelSim <u>project</u> not just a work library since you are using LPMs.
- Final test for the Processor is running the program and inspecting Data Memory to make sure the correct value is stored in location 5.
- 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.

Procedure