Lecture 03

Chapter 8:
Programmable Processors

Slides to accompany the textbook *Digital Design*, First Edition,
by Frank Vahid, John Wiley and Sons Publishers, 2007.
http://www.ddvahid.com

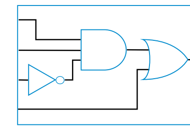
Any questions from last time?

Copyright © 2007 Frank Vahid

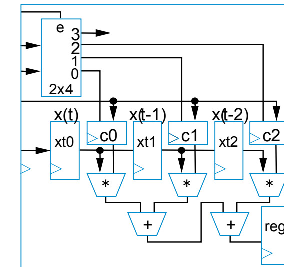
*Instructors of courses requiring Vahid's Digital Design textbook (published by John Wiley and Sons) have permission to modify and use these slides for customary course-related activities, subject to keeping this copyright notice in place and unmodified. These slides may be posted as unanimated pdf versions on publicly-accessible course websites.. PowerPoint source (or pdf with animations) may **not** be posted to publicly-accessible websites, but may be posted for students on internal protected sites or distributed directly to students by other electronic means. Instructors may make printouts of the slides available to students for a reasonable photocopying charge, without incurring royalties. Any other use requires explicit permission. Instructors may obtain PowerPoint source or obtain special use permissions from Wiley – see http://www.ddvahid.com for information.*

Review: Types of Processing Logic

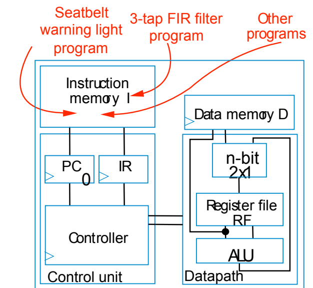
- Programmable (general-purpose) processor
  - Mass-produced, then programmed to implement different processing tasks
    - Well-known common programmable processors: Pentium, Sparc, PowerPC
    - Lesser-known but still common: ARM, MIPS, 8051, PIC
      - Low-cost embedded processors found in cell phones, blinking shoes, etc.
  - Instructive to design a very simple programmable processor
    - Real processors can be much more complex



Seatbelt warning light single-purpose processor



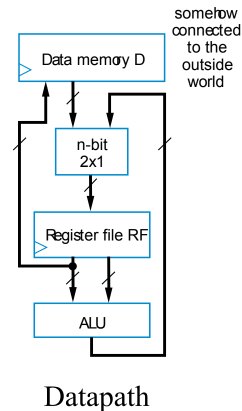
3-tap FIR filter single-purpose processor



General-purpose processor

Review: Basic Architecture

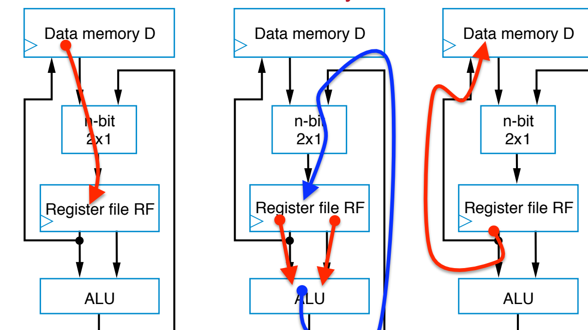
- Processing generally consists of:
  - Loading some data
  - Transforming that data
  - Storing that data
- *Basic datapath*: Useful circuit in a programmable processor
  - Can read/write data memory, where main data exists
  - Has register file to hold data locally
  - Has ALU to transform local data



Datapath

Review: Basic Datapath Operations

- Load operation: Load data from data memory to RF
- ALU operation: Transforms data by passing one or two RF register values through ALU, performing operation (ADD, SUB, AND, OR, etc.), and writing back into RF.
- Store operation: Stores RF register value back into data memory
- Each operation can be done in one clock cycle

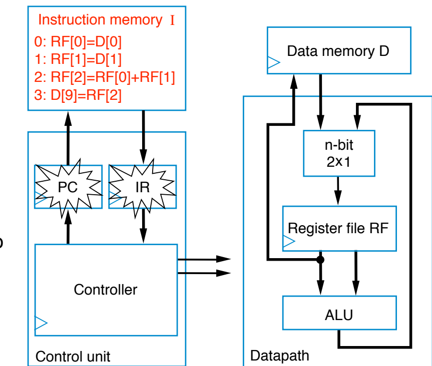


Load operation — ALU operation — Store operation

Review: What's a Clock Cycle?

Review: Basic Architecture – Control Unit

- 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



Foreshadowing:
What if we want ALU to add, subtract? How do we tell it what to do?

Digression:
HW vs. SW based approaches

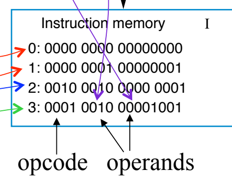
Review: Three-Instruction Programmable Processor

- Instruction Set – List of allowable instructions and their representation in memory, e.g.,
  - *Load* instruction—0000 $r_3r_2r_1r_0$ $d_7d_6d_5d_4d_3d_2d_1d_0$
  - *Store* instruction—0001 $r_3r_2r_1r_0$ $d_7d_6d_5d_4d_3d_2d_1d_0$
  - *Add* instruction—0010 $ra_3ra_2ra_1ra_0$ $rb_3rb_2rb_1rb_0$ $rc_3rc_2rc_1rc_0$

What does this tell you about data memory?

What does this tell us about the register file?

Desired program
0: RF[0]=D[0]
1: RF[1]=D[1]
2: RF[2]=RF[0]+RF[1]
3: D[9]=RF[2]



opcode operands

"Instruction" is an idea that helps abstract 1s, 0s, but still provides info. about HW

Instructions in 0s and 1s – *machine code*

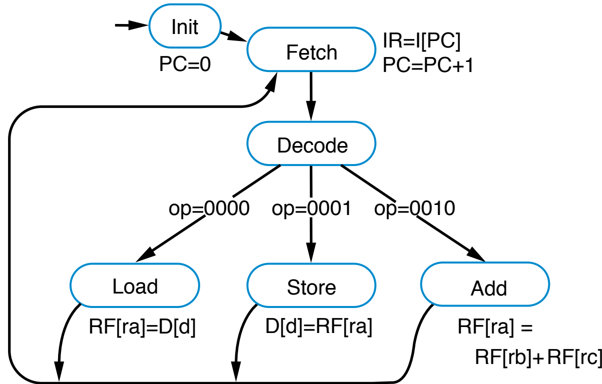
Review: Assembly Code

- Machine code (0s and 1s) hard to work with
- Assembly code – Uses mnemonics
  - *Load* instruction—**MOV Ra, d**
    - specifies the operation *RF[a]=D[d]*. *a* must be 0,1, ..., or 15—so *R0* means *RF[0]*, *R1* means *RF[1]*, etc. *d* must be 0, 1, ..., 255
  - • ***Store*** instruction—**MOV d, Ra**
    - specifies the operation *D[d]=RF[a]*
  - • ***Add*** instruction—**ADD Ra, Rb, Rc**
    - specifies the operation *RF[a]=RF[b]+RF[c]*

Desired program	machine code	assembly code
0: RF[0]=D[0]	0: 0000 0000 00000000	0: MOV R0, 0
1: RF[1]=D[1]	1: 0000 0001 00000001	1: MOV R1, 1
2: RF[2]=RF[0]+RF[1]	2: 0010 0010 0000 0001	2: ADD R2, R0, R1
3: D[9]=RF[2]	3: 0001 0010 00001001	3: MOV 9, R2

Control-Unit and Datapath for Three-Instruction Processor

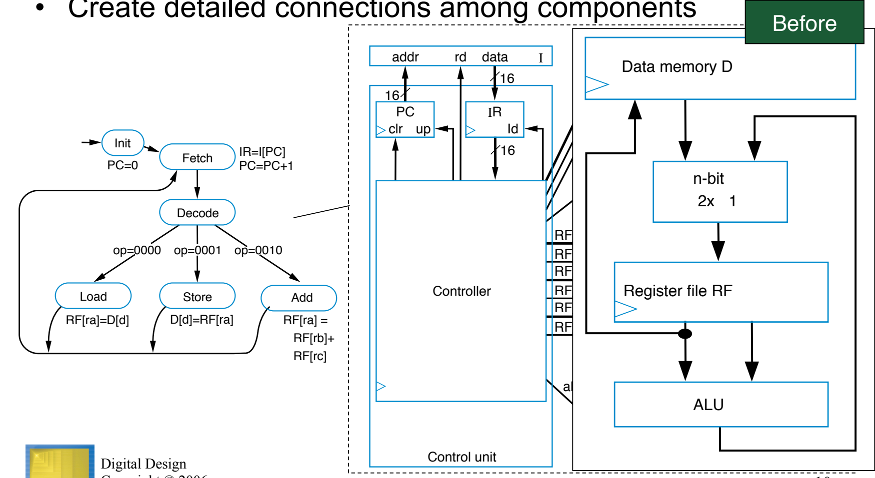
- To design the processor, we can begin with a high-level state machine description of the processor's behavior



Your 1st lab is about state machines.
They're important b/c they help to automate instruction processing.

Control-Unit and Datapath for Three-Instruction Processor

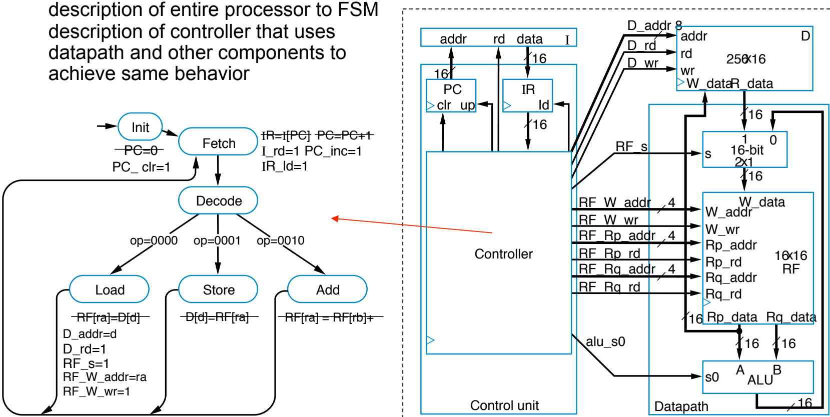
- Create detailed connections among components



Digital Design
Copyright © 2006
Frank Vahid

Control-Unit and Datapath for Three-Instruction Processor

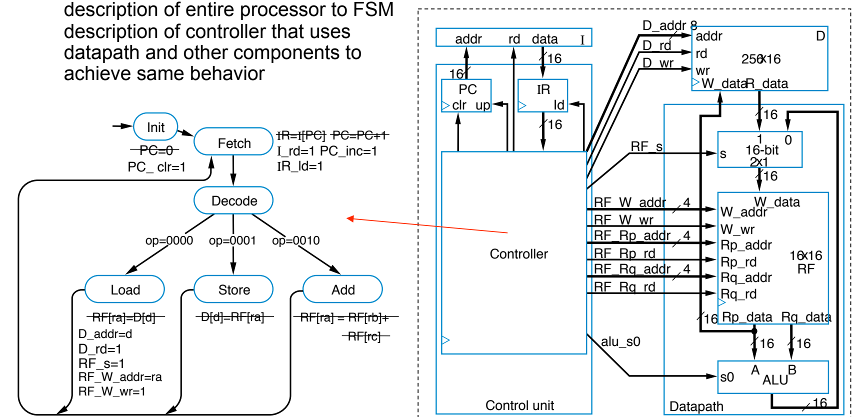
- Convert high-level state machine description of entire processor to FSM description of controller that uses datapath and other components to achieve same behavior



Digital Design
Copyright © 2006
Frank Vahid

Control-Unit and Datapath for Three-Instruction Processor

- Convert high-level state machine description of entire processor to FSM description of controller that uses datapath and other components to achieve same behavior



Digital Design
Copyright © 2006
Frank Vahid

Exercise: Understanding the Processor Design (1)

- Will the correct instruction be fetched if PC is incremented during the fetch cycle?
- While executing "MOV R1, 3", what is the content of PC and IR at the end of the 1st cycle, 2nd cycle, 3rd cycle, etc.?
- What if it takes more than 1 cycle for memory read?

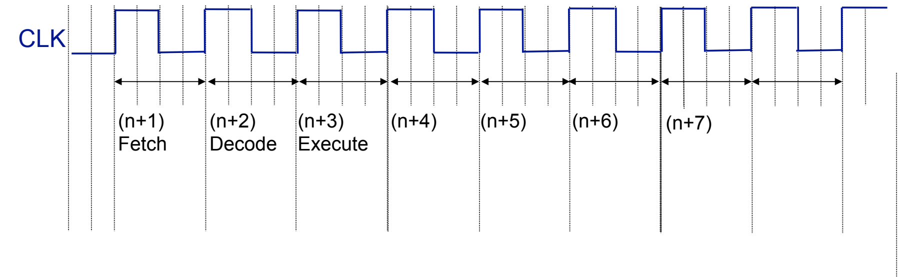
Exercise: Understanding the Processor Design (2)

Q1: D[8] = D[8] + RF[1] + RF[4]

…

I[15]:	**Add R2, R1, R4**	**RF[1] = 4**
I[16]:	**MOV R3, 8**	**RF[4] = 5**
I[17]:	**Add R2, R2, R3**	**D[8] = 7**

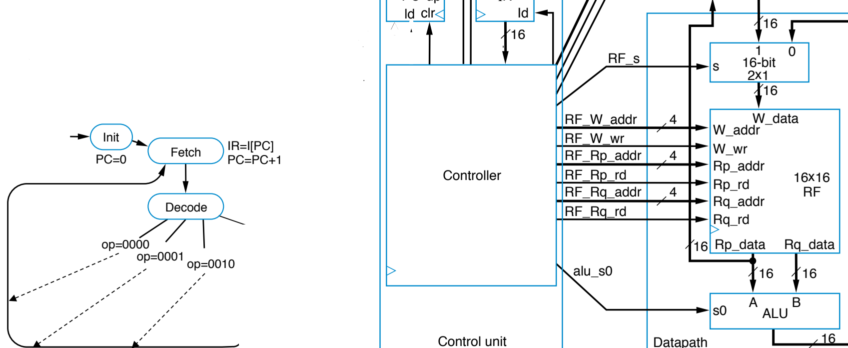
…



Exercise: Extending the Three-Instruction Processor

Add a instruction:

JMP: jump to a location specified by the 12-bit offset



A Six-Instruction Programmable Processor

- Let's add three more instructions:
  - ***Load-constant*** instruction—**0011** $r_3r_2r_1r_0$ $c_7c_6c_5c_4c_3c_2c_1c_0$
    - **MOV Ra, #c**—specifies the operation *RF[a]=c*
  - ***Subtract*** instruction—**0100** $ra_3ra_2ra_1ra_0$ $rb_3rb_2rb_1rb_0$ $rc_3rc_2rc_1rc_0$
    - **SUB Ra, Rb, Rc**—specifies the operation *RF[a]=RF[b] – RF[c]*
  - ***Jump-if-zero*** instruction—**0101** $ra_3ra_2ra_1ra_0$ $o_7o_6o_5o_4o_3o_2o_1o_0$
    - **JMPZ Ra, offset**—specifies the operation *PC = PC + offset* if *RF[a]* is 0

TABLE 8.1 Six-instruction instruction set.

Instruction	Meaning
MOV Ra, d	RF[a] = D[d]
MOV d, Ra	D[d] = RF[a]
ADD Ra, Rb, Rc	RF[a] = RF[b]+RF[c]
MOV Ra, #C	RF[a] = C
SUB Ra, Rb, Rc	RF[a] = RF[b]-RF[c]
JMPZ Ra, offset	PC=PC+offset if RF[a]=0

TABLE 8.2 Instruction opcodes.

Instruction	Opcode
MOV Ra, d	0000
MOV d, Ra	0001
ADD Ra, Rb, Rc	0010
MOV Ra, #C	0011
SUB Ra, Rb, Rc	0100
JMPZ Ra, offset	0101

Digital Design
Copyright © 2006
Frank Vahid

Program for the Six-Instruction Processor

- Example program – Count number of non-zero words in D[4] and D[5]
  - Result will be either 0, 1, or 2
  - Put result in D[9]

Digital Design
Copyright © 2006
Frank Vahid

Program for the Six-Instruction Processor

- Example program – Count number of non-zero words in D[4] and D[5]
  - Result will be either 0, 1, or 2
  - Put result in D[9]

Digital Design
Copyright © 2006
Frank Vahid

Program for the Six-Instruction Processor

- Example program – Count number of non-zero words in D[4] and D[5]
  - Result will be either 0, 1, or 2
  - Put result in D[9]

Digital Design
Copyright © 2006
Frank Vahid

Exercise: More on Programming

Q1: Compare the contents of D[4] and D[5]. If equal, D[3] =1, otherwise set D[3]=0.

TABLE 8.2 Instruction opcodes.

Instruction	Opcode
MOV Ra, d	0000
MOV d, Ra	0001
ADD Ra, Rb, Rc	0010
MOV Ra, #C	0011
SUB Ra, Rb, Rc	0100
JMPZ Ra, offset	0101

Exercise: More on Programming

Q1: Compare the contents of D[4] and D[5]. If equal, D[3] =1, otherwise set D[3]=0.

TABLE 8.2 Instruction opcodes.

Instruction	Opcode
MOV Ra, d	0000
MOV d, Ra	0001
ADD Ra, Rb, Rc	0010
MOV Ra, #C	0011
SUB Ra, Rb, Rc	0100
JMPZ Ra, offset	0101

Modifications to the Three-Instruction Processor

- ***Load-constant*** instruction

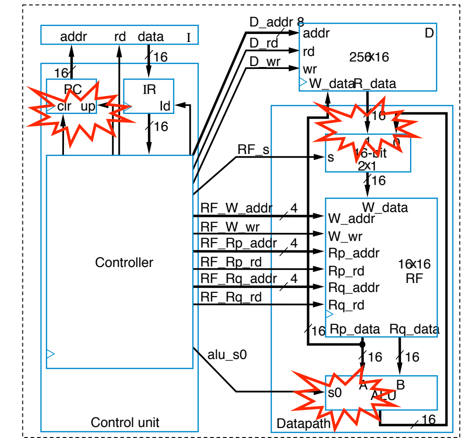
  0011 $r_3r_2r_1r_0c_7c_6c_5c_4c_3c_2c_1c_0$

- ***Subtract*** instruction

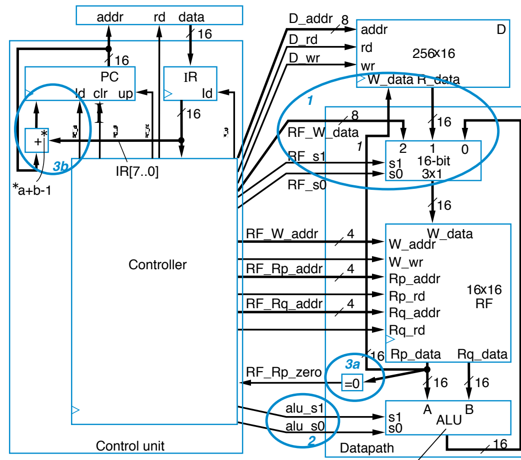
  0100 $ra_3ra_2ra_1ra_0$ $rb_3rb_2rb_1rb_0$ $rc_3rc_2rc_1rc_0$

- ***Jump-if-zero*** instruction

  0101 $ra_3ra_2ra_1ra_0$ $o_7o_6o_5o_4o_3o_2o_1o_0$

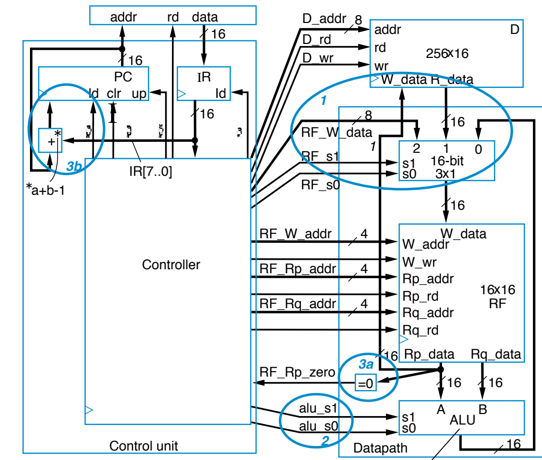


Extending the Control-Unit and Datapath



s1	s0	ALU operation
0	0	pass A through
0	1	A+B
1	0	A-B

Extending the Control-Unit and Datapath



s1	s0	ALU operation
0	0	pass A through
0	1	A+B
1	0	A-B

Controller FSM for the Six-Instruction Processor

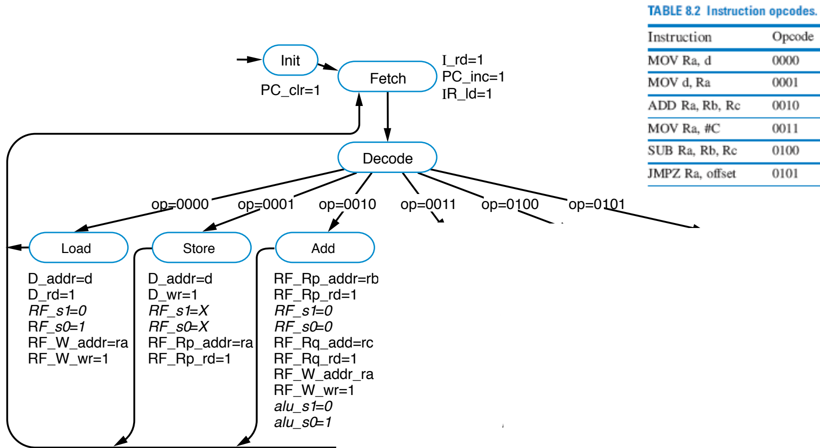
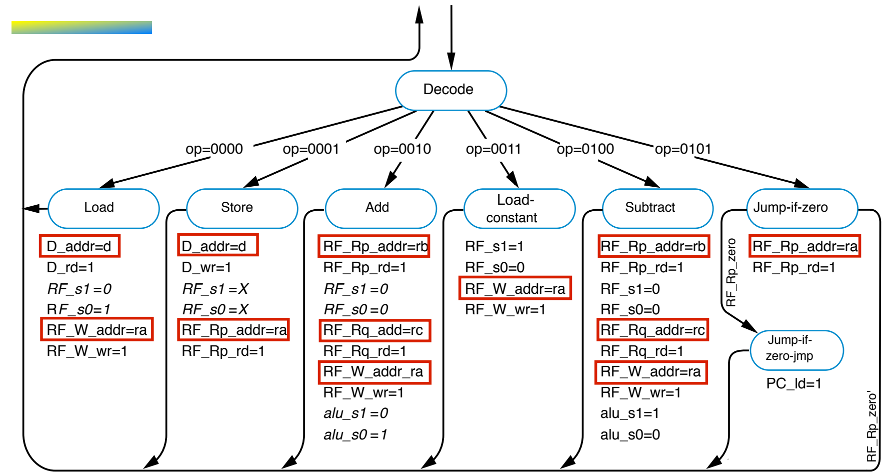


TABLE 8.2 Instruction opcodes.

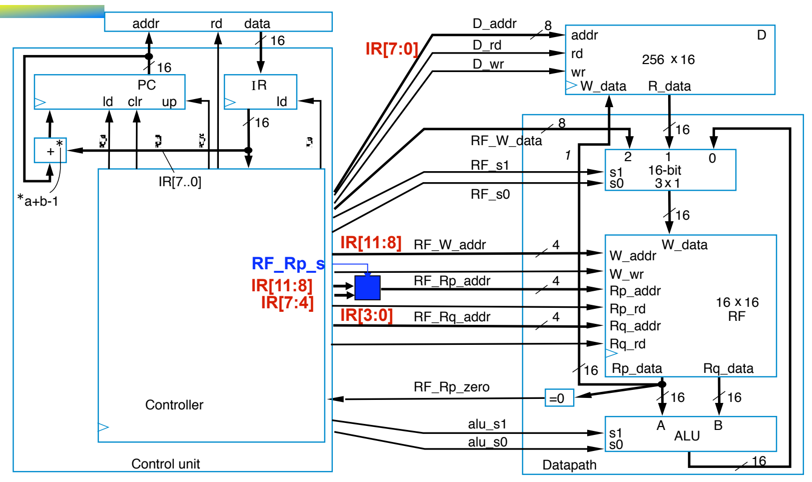
Instruction	Opcode
MOV Ra, d	0000
MOV d, Ra	0001
ADD Ra, Rb, Rc	0010
MOV Ra, #C	0011
SUB Ra, Rb, Rc	0100
JMPZ Ra, offset	0101

Digital Design
Copyright © 2006
Frank Vahid

More on Processor Implementation (1)



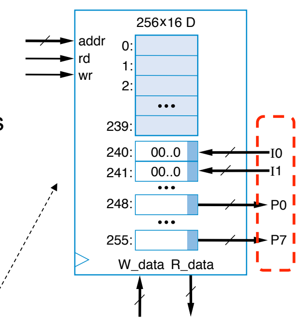
More on Processor Implementation (2)



Further Extensions to the Programmable Processor

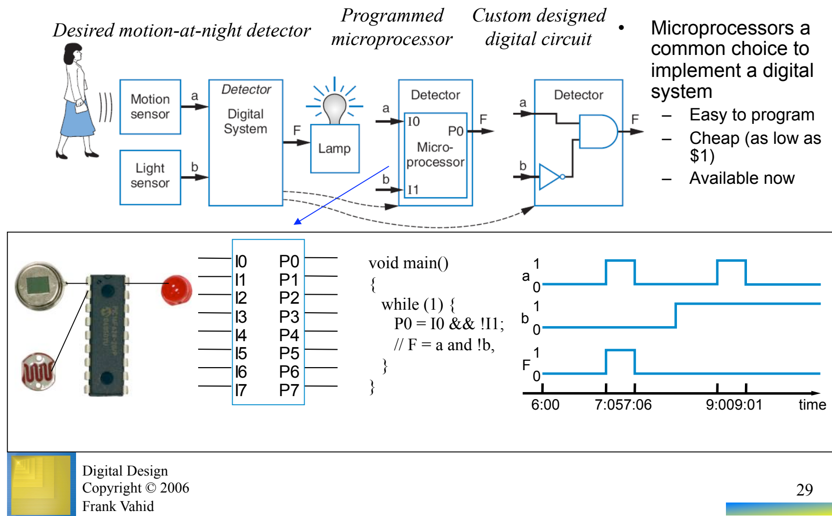
8.5

- Typical processor instruction set will contain dozens of data movement (e.g., loads, stores), ALU (e.g., add, sub), and flow-of-control (e.g., jump) instructions
  - Extending the control-unit/datapath follows similarly to previously-shown extensions
- Input/output extensions
  - Certain memory locations may actually be external pins
    - e.g, D[240] may represent 8-bit input I0, D[255] may represent 8-bit output P7



Digital Design
Copyright © 2006
Frank Vahid

Program using I/O Extensions



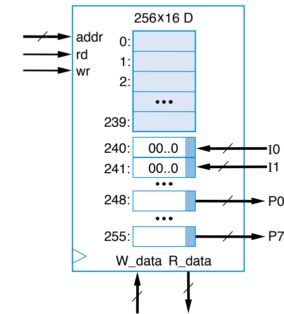
- Microprocessors a common choice to implement a digital system
  - Easy to program
  - Cheap (as low as $1)
  - Available now

Program Using Input/Output Extensions

Underlying assembly code for C expression I0 && !I1.

0: MOV R0, 240 // move *D[240]*, which is the value at pin *I0*, into *R0*
1: MOV R1, 241 // move *D[241]*, which is that value at pin *I1*, into *R1*
2: NOT R1, R1 // compute *!I1*, assuming existence of a complement instruction
3: AND R0, R0, R1 // compute *I0 && !I1*, assuming an AND instruction
4: MOV 248, R0 // move result to *D[248]*, which is pin *P0*

```
void main()
{
  while (1) {
    P0 = I0 && !I1;
    // F = a and !b,
  }
}
```

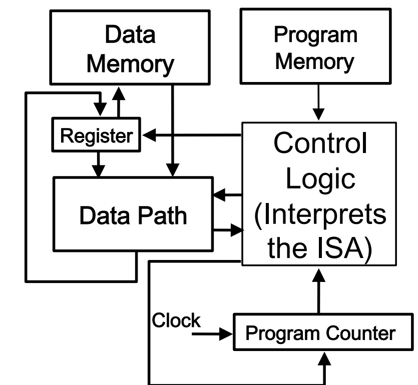


Chapter Summary

- Programmable processors are widely used
  - Easy availability, short design time
- Basic architecture
  - Datapath with register file and ALU
  - Control unit with PC, IR, and controller
  - Memories for instructions and data
  - Control unit fetches, decodes, and executes
- Three-instruction processor with machine-level programs
  - Extended to six instructions
  - Real processors have dozens or hundreds of instructions
  - Extended to access external pins
  - Modern processors are far more sophisticated
- Instructive to see how one general circuit (programmable processor) can execute variety of behaviors just by programming 0s and 1s into an instruction memory

Generic Computer Organization

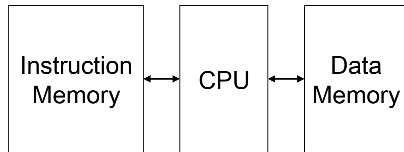
- **Stored Program Machine (vonNeumann Model)**
- **Instructions are represented as numbers**
- **Programs in memory are read or written just as normal data**



Memory Organization

Two main memory organizations:

HARVARD
ARCHITECTURE



PRINCETON
ARCHITECTURE