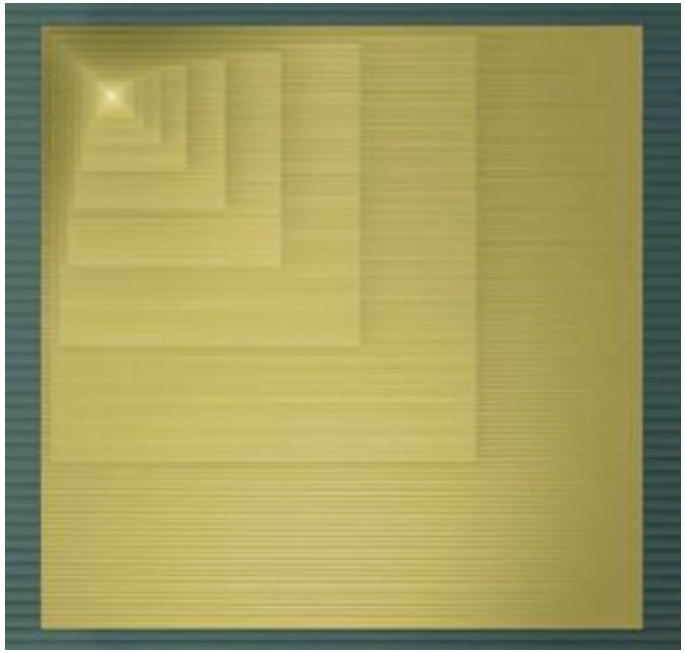
Preface

- Any questions from last time?
  - Will review 1st 22 slides
- A bit more motivation, information about me
  - Research
  - ND
- A bit more about this class…
  - Microsoft
- Later:
  - HW 1
  - Review session
  - MD McNally about Lab 1

(1)
Any questions from last time?



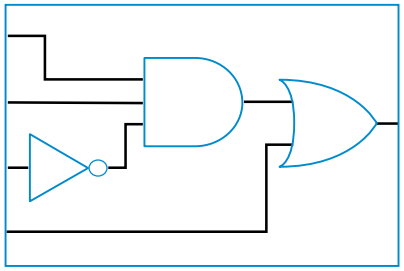
Digital Design

Chapter 8:
Programmable Processors

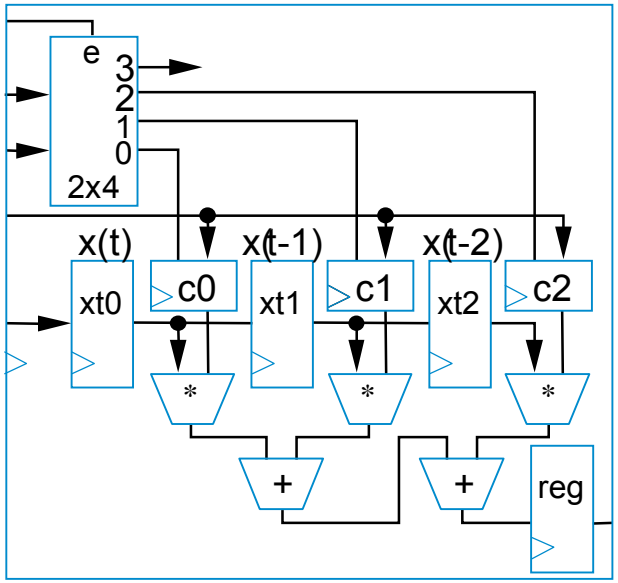
Slides to accompany the textbook *Digital Design*, First Edition,
by Frank Vahid, John Wiley and Sons Publishers, 2007.
http://www.ddvahid.com

Introduction

- 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

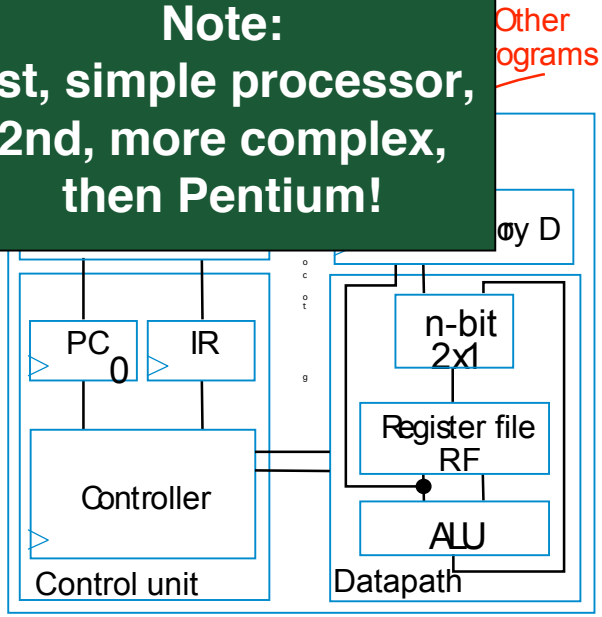


Seatbelt warning light single-purpose processor

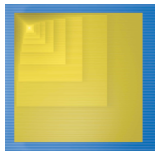


3-tap FIR filter single-purpose processor

Note:
1st, simple processor, 2nd, more complex, then Pentium!



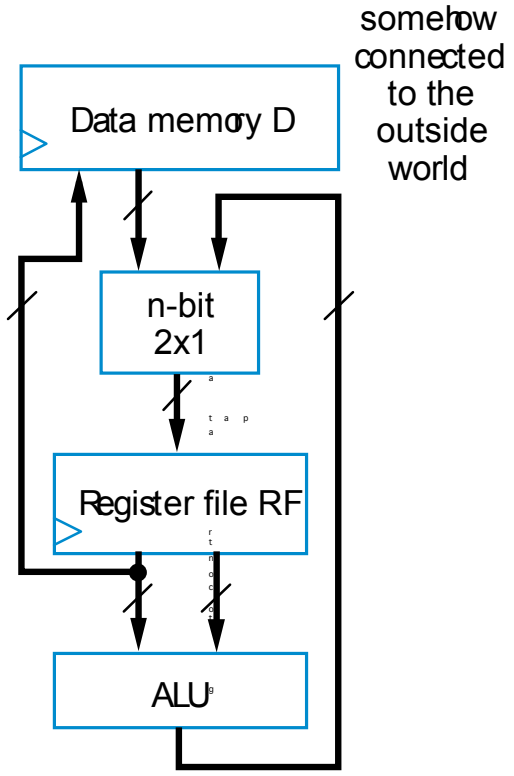
General-purpose processor



Note: Slides with animation are denoted with a small red "a" near the animated items

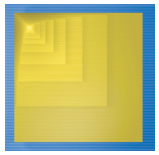
Basic Architecture

- 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

Compare to Slide15, Lecture 01.

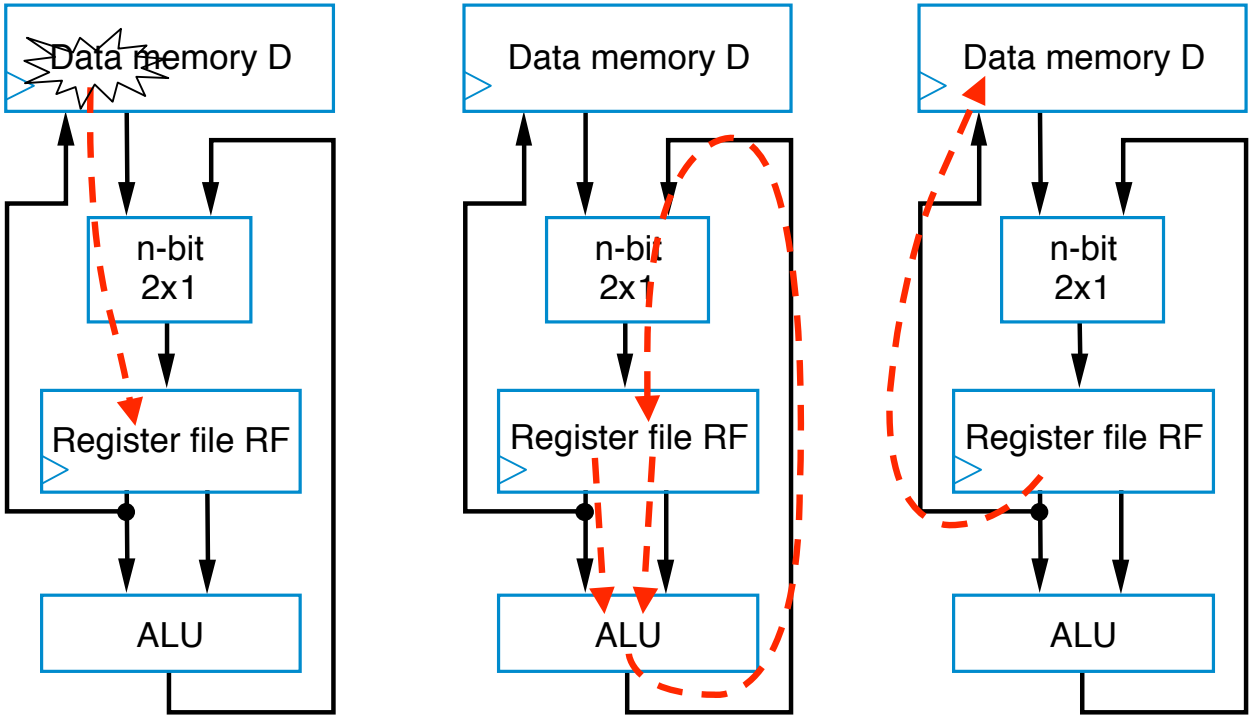


Basic Datapath Operations

- 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

Understand idea of register file?

Versus RAM?

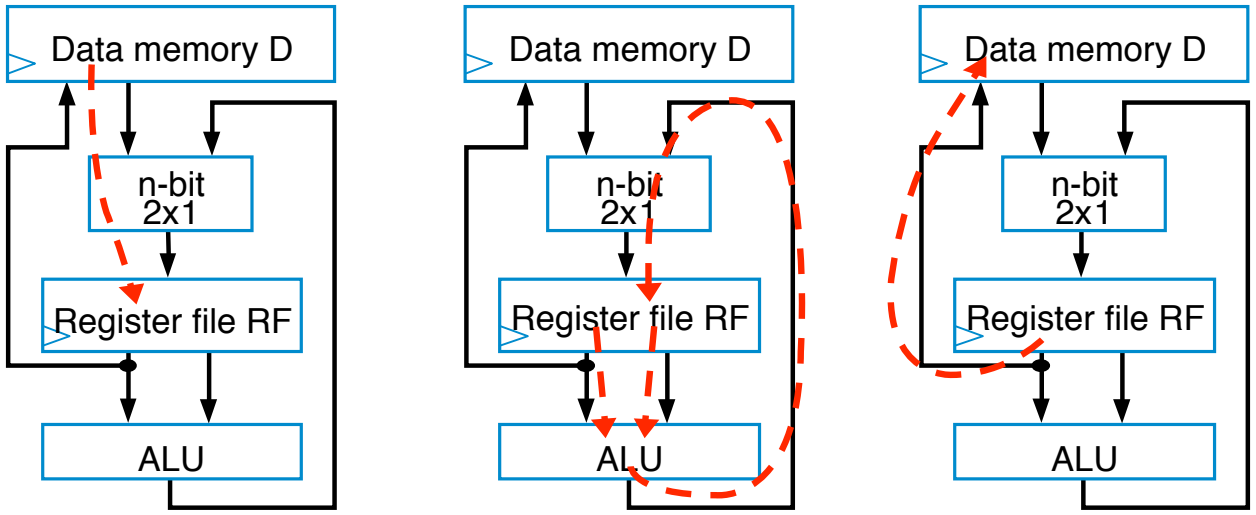


Load operation ALU operation Store operation

Basic Datapath Operations

- Q: Which are valid *single-cycle operations* for given datapath?
  - Move D[1] to RF[1] (i.e., RF[1] = D[1])
    - A: YES – That's a load operation
  - Store RF[1] to D[9] and store RF[2] to D[10]
    - A: NO – Requires two separate store operations
  - Add D[0] plus D[1], store result in D[9]
    - A: NO – ALU operation (ADD) only works with RF. Requires two load operations (e.g., RF[0]=D[0]; RF[1]=D[1], an ALU operation (e.g., RF[2]=RF[0]+RF[1]), and a store operation (e.g., D[9]=RF[2])

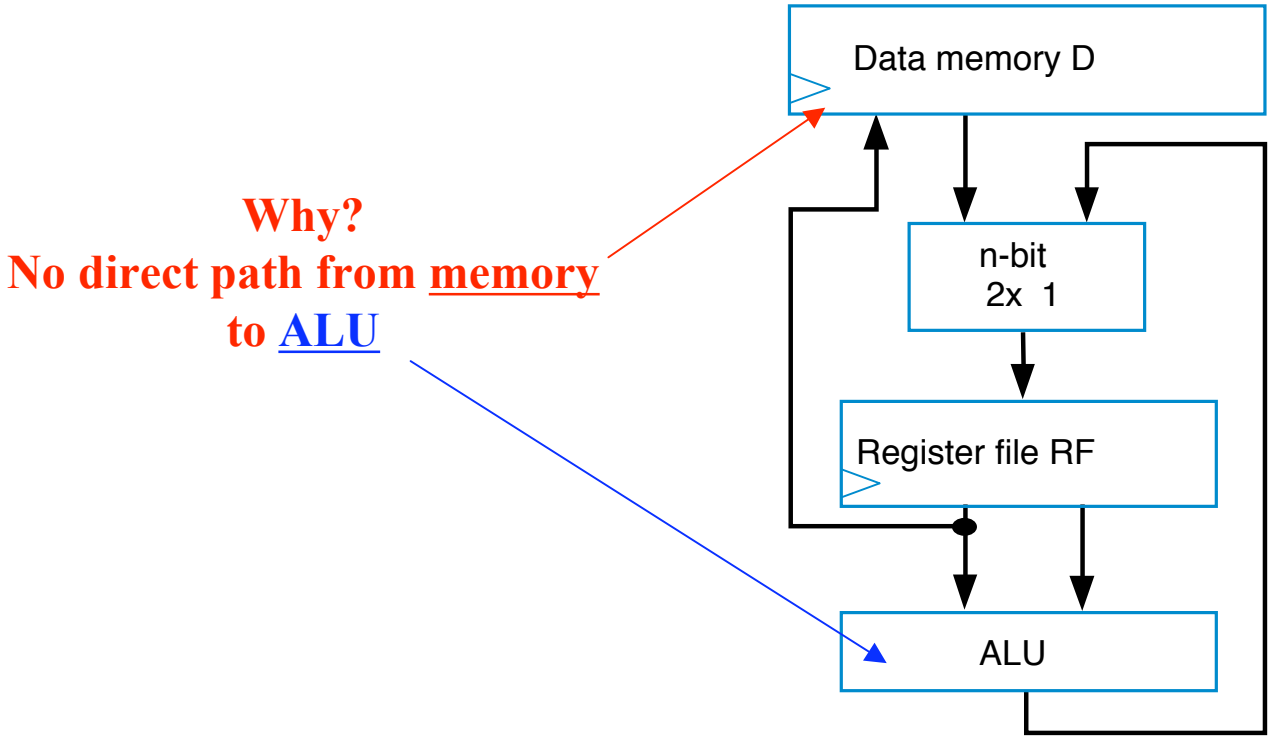
Let's do 1 more example…

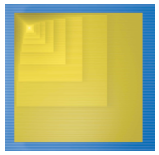


Load operation ALU operation Store operation

One more example…

- Q: Which are valid *single-cycle operations* for given datapath?
  - Add d(0) + R1, store in R2?





Exercise: Basic Datapath Operations

Q: How many cycles does each of the following take for given datapath?

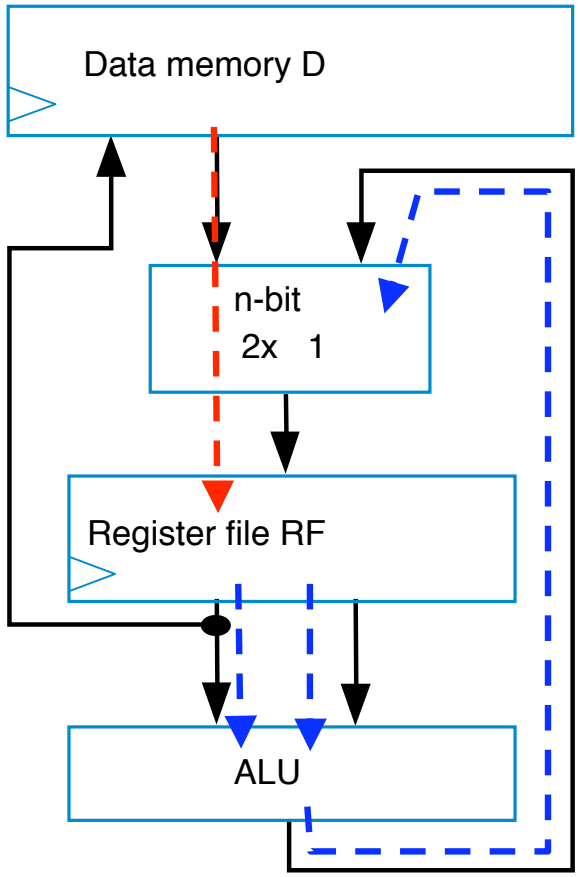
- Move RF[1] to RF[2]
- Add D[8] with RF[2] and store the result in RF[4]
- Add D[8] with RF[1], then add the result with RF[4], and store the final result in D[8]

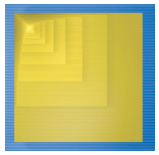
One more example…

- Q: How can we do: d(0) + R1, store in R2?

Need at least 2 instructions:

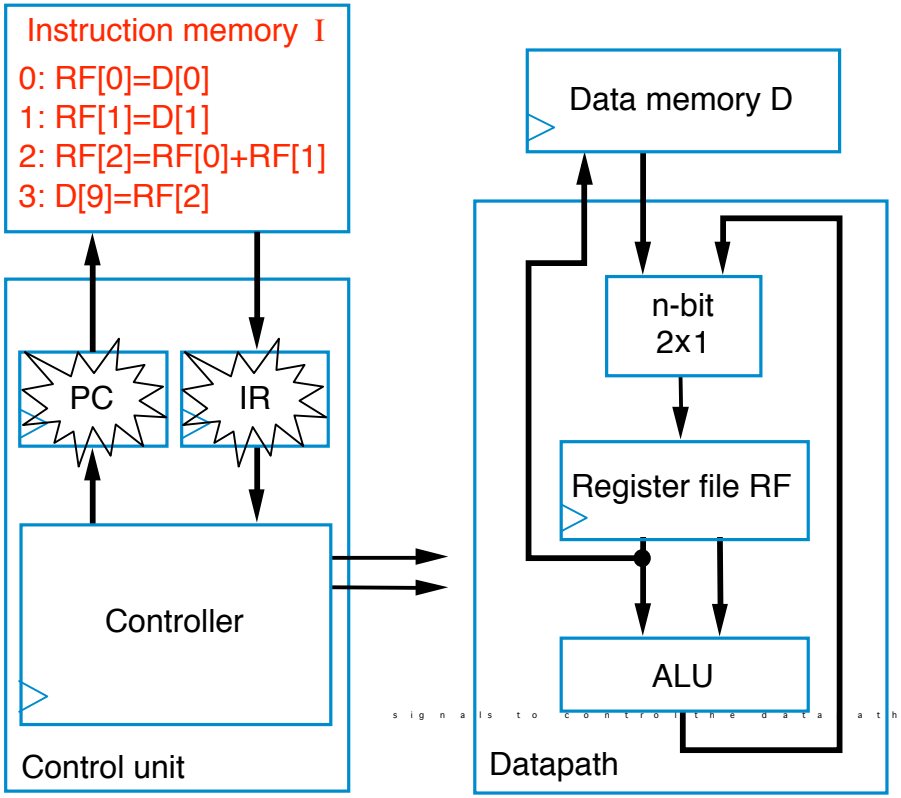
MOV R3, D(0) # LOAD D(0)
ADD R2, R1, R3 # Do the ADD





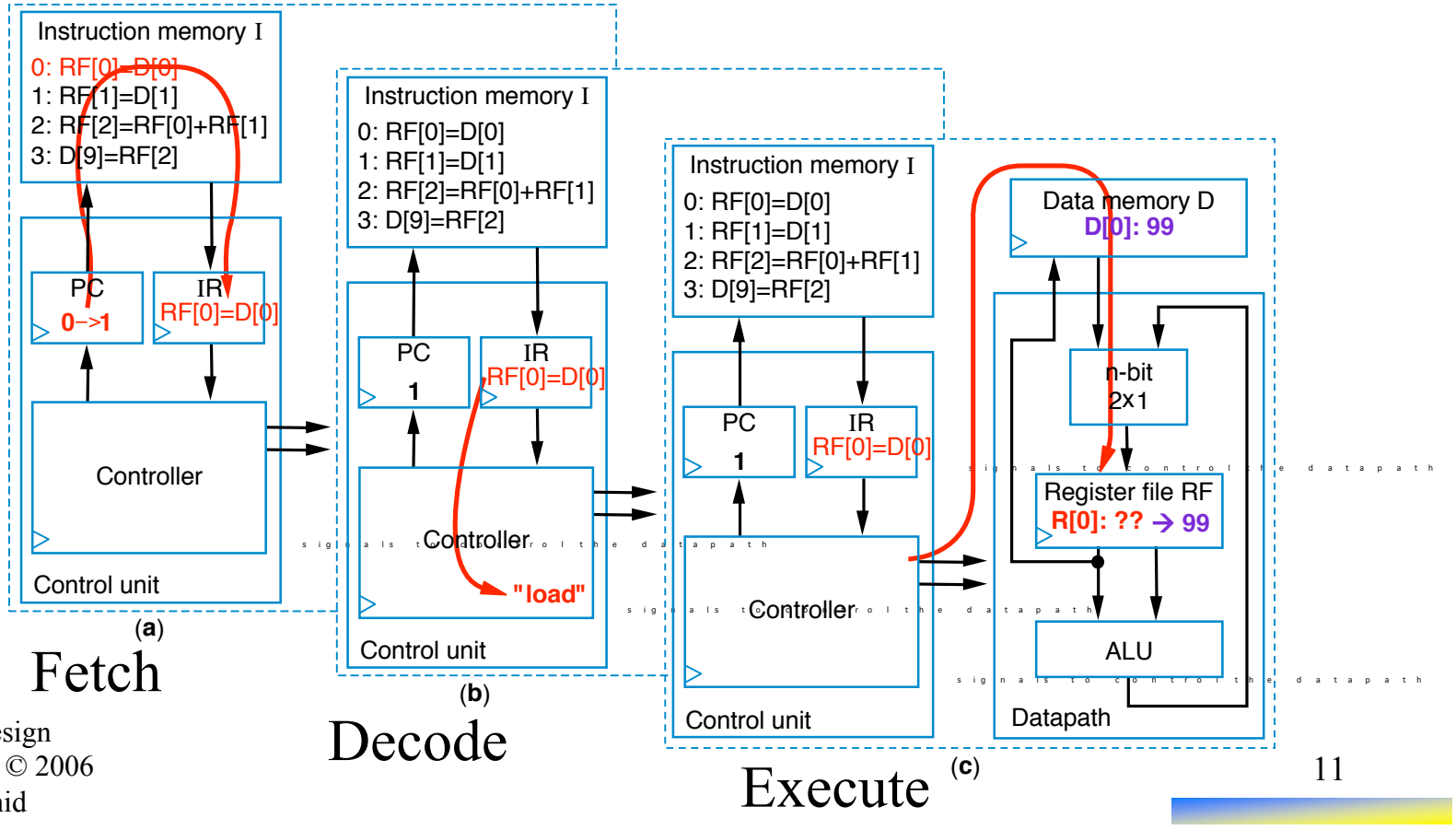
Basic Architecture – Control Unit

- 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



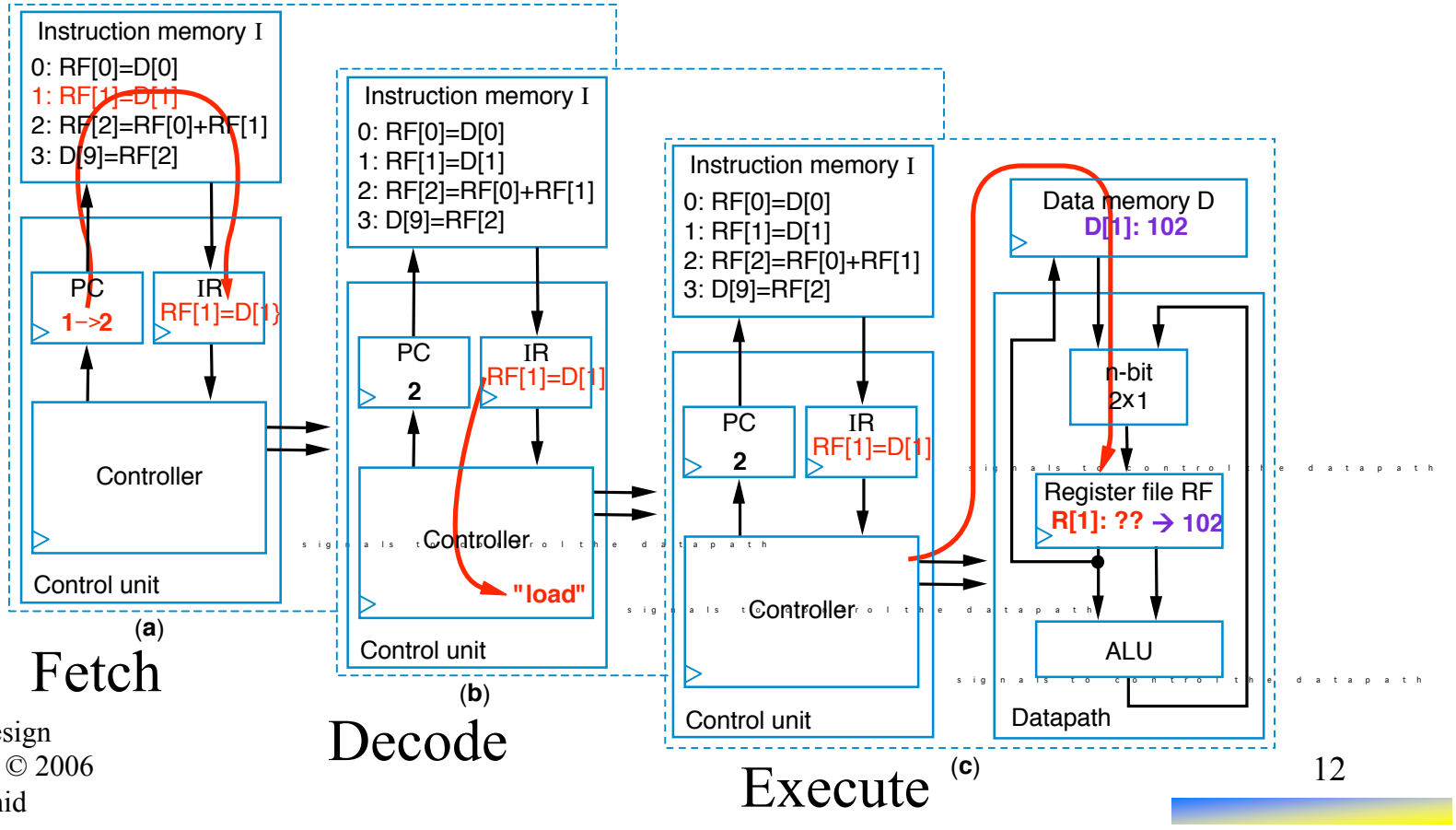
Basic Architecture – Control Unit

- To carry out *each instruction*, the control unit must:
  - Fetch – Read instruction from inst. mem.
  - Decode – Determine the operation and operands of the instruction
  - Execute – Carry out the instruction's operation using the datapath



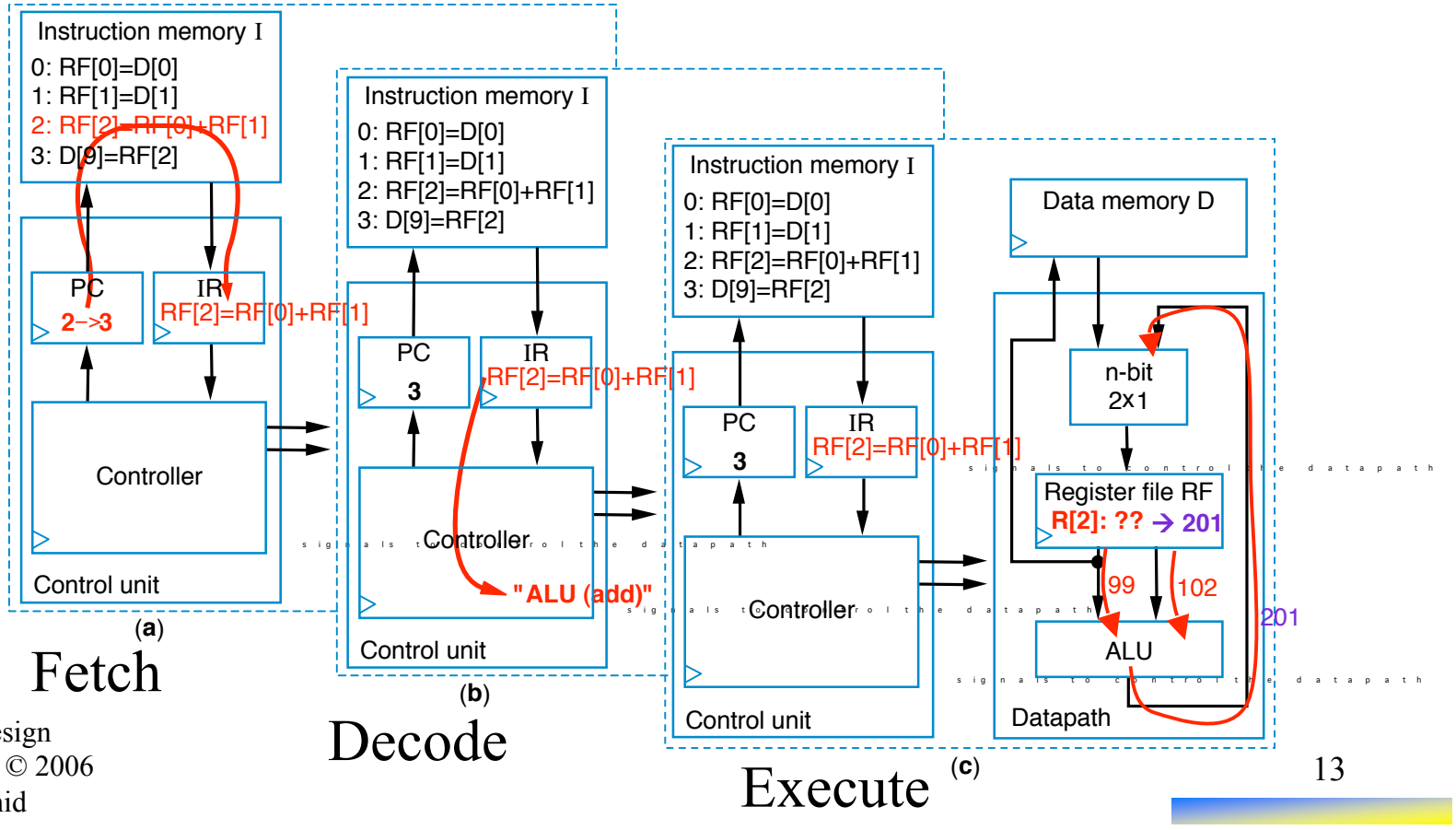
Basic Architecture – Control Unit

- To carry out *each instruction*, the control unit must:
  - Fetch – Read instruction from inst. mem.
  - Decode – Determine the operation and operands of the instruction
  - Execute – Carry out the instruction's operation using the datapath



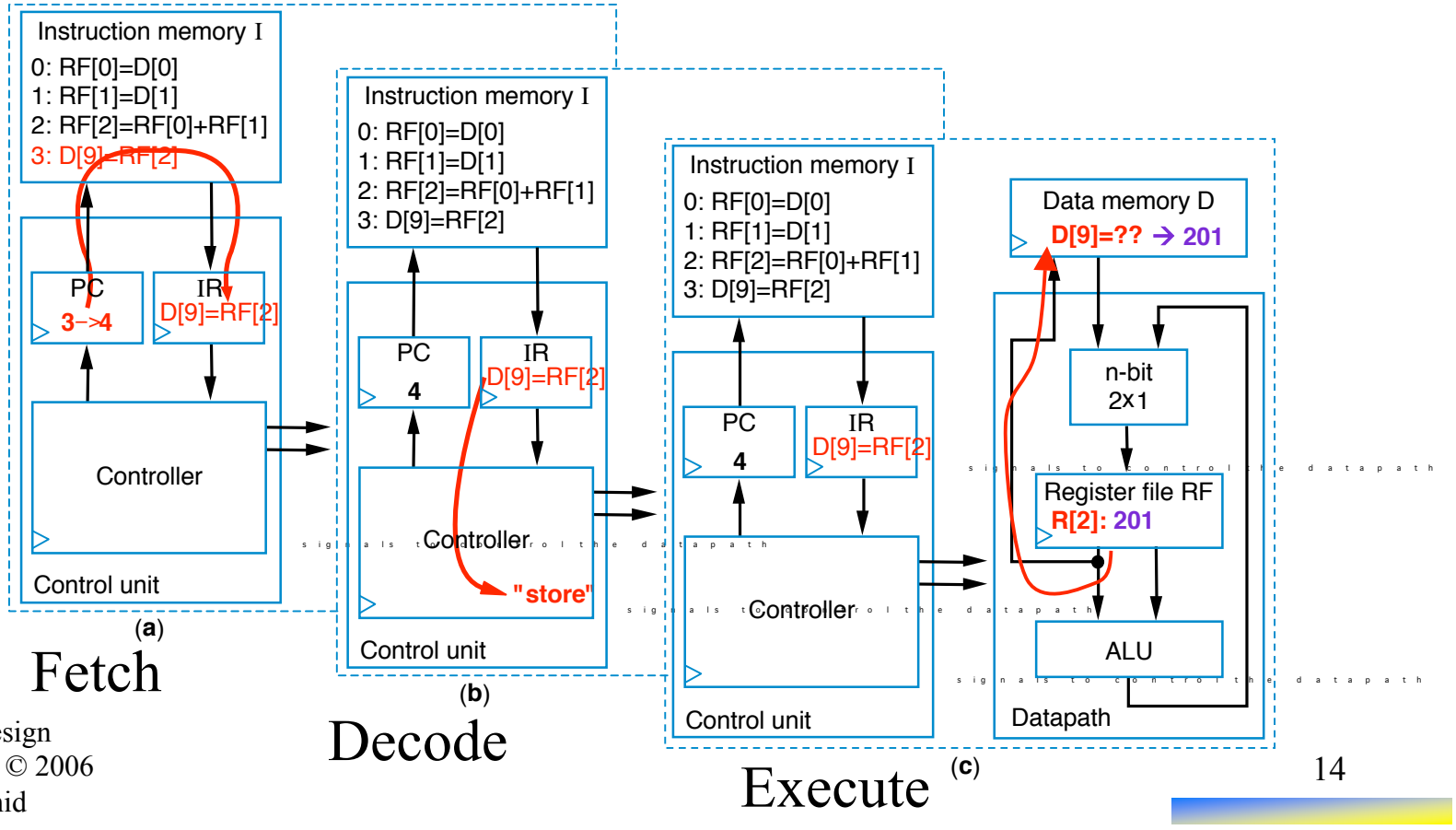
Basic Architecture – Control Unit

- To carry out *each instruction*, the control unit must:
  - Fetch – Read instruction from inst. mem.
  - Decode – Determine the operation and operands of the instruction
  - Execute – Carry out the instruction's operation using the datapath



Basic Architecture – Control Unit

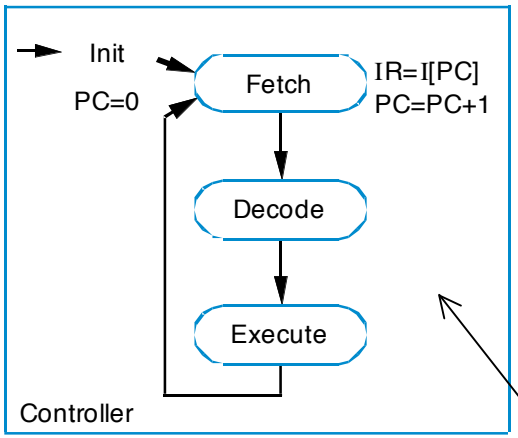
- To carry out *each instruction*, the control unit must:
  - Fetch – Read instruction from inst. mem.
  - Decode – Determine the operation and operands of the instruction
  - Execute – Carry out the instruction's operation using the datapath

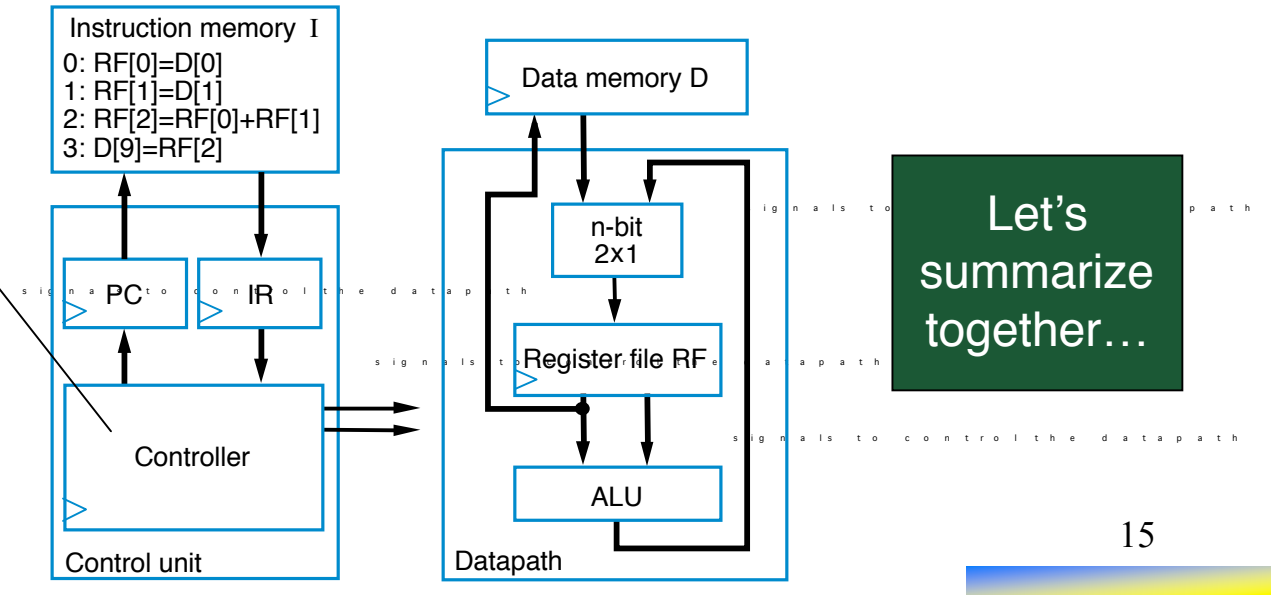


Basic Architecture – Control Unit

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.



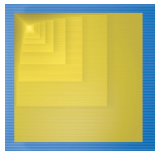


Creating a Sequence of Instructions

- Q: Create sequence of instructions to compute D[3] = D[0]+D[1]+D[2] on earlier-introduced processor
- A1: One possible sequence
  - First load data memory locations into register file
    - R[3] = D[0]
    - R[4] = D[1]
    - R[2] = D[2]

    (Note arbitrary register locations)
  - Next, perform the additions
    - R[1] = R[3] + R[4]
    - R[1] = R[1] + R[2]
  - Finally, store result
    - D[3] = R[1]
- A2: Alternative sequence
  - First load D[0] and D[1] and add them
    - R[1] = D[0]
    - R[2] = D[1]
    - R[1] = R[1] + R[2]
  - Next, load D[2] and add
    - R[2] = D[2]
    - R[1] = R[1] + R[2]
  - Finally, store result
    - D[3] = R[1]

Questions from last time?



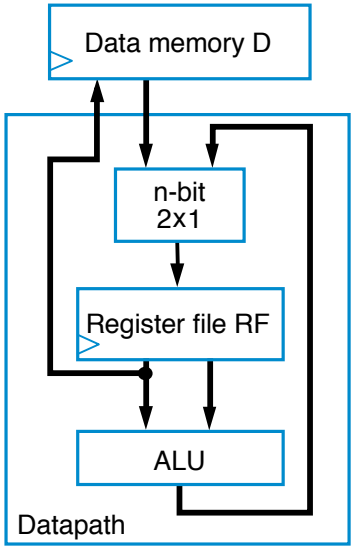
Exercise: Creating Instruction Sequences

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

Let's do 1 more example…

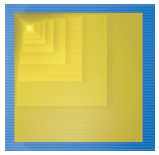
Q2: RF[2] = RF[1]

One more example…



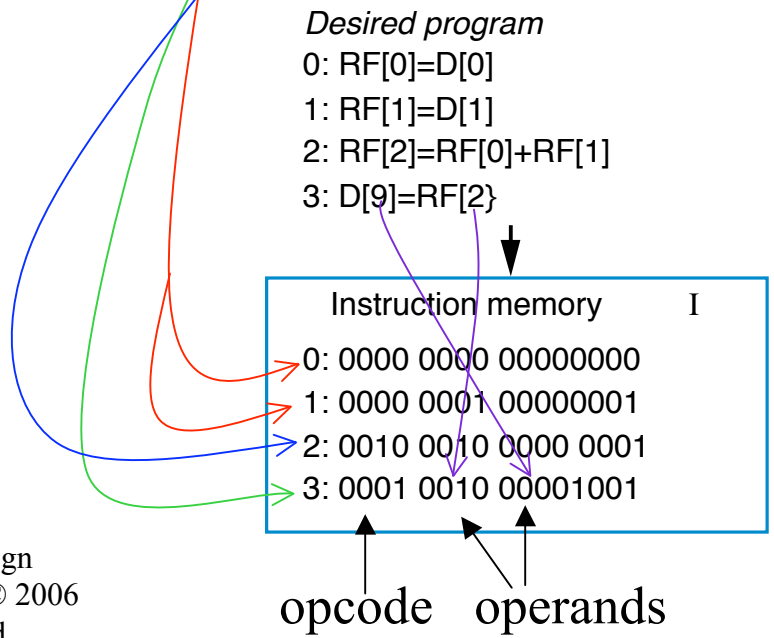
- R2 = R3 || R3 = R2 (swap) -- Thoughts?
  - MOV d(x), R2
  - MOV d(y), R3
  - MOV R2, d(y)
  - MOV R2, d(x)

- Could do in 3 … but only with added functionality:
  - How?



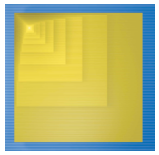
Three-Instruction Programmable Processor

- 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** $ra_3ra_2ra_1ra_0$ $rb_3rb_2rb_1rb_0$ $rc_3rc_2rc_1rc_0$



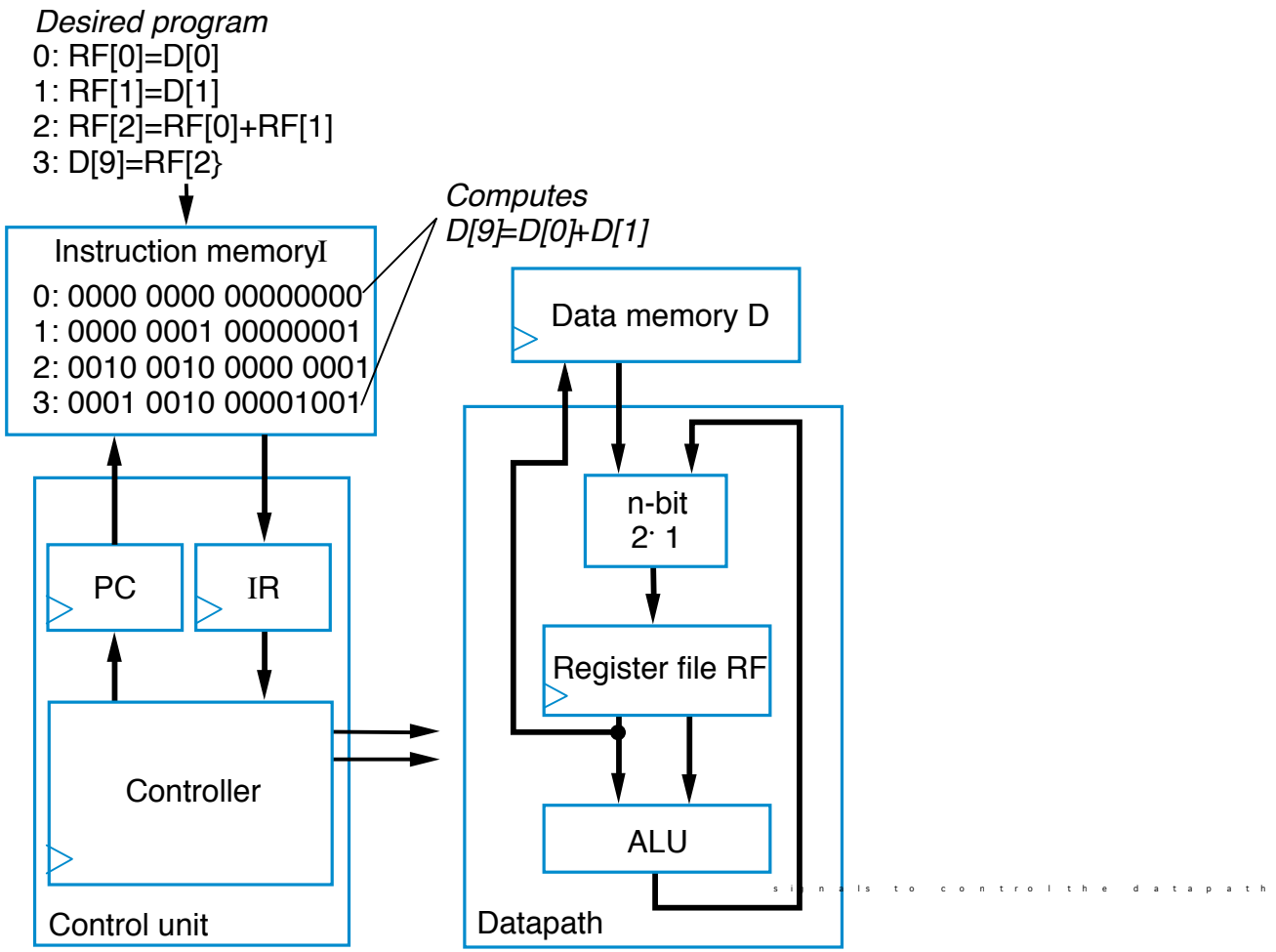
"Instruction" is an idea that helps abstract 1s, 0s, but still provides info. about HW

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



Program for Three-Instruction Processor

OPCODE = sequence of 1s, 0s that's fed to controller to tell datapath what to do



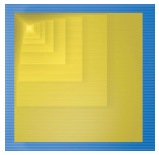
Program for Three-Instruction Processor

- Another example program in machine code
  - Compute D[5] = D[5] + D[6] + D[7]

```
0: 0000 0000 00000101  // RF[0] = D[5]
1: 0000 0001 00000110  // RF[1] = D[6]
2: 0000 0010 00000111  // RF[2] = D[7]
3: 0010 0000 0000 0001  // RF[0] = RF[0] + RF[1]
                        // which is D[5]+D[6]
4: 0010 0000 0000 0010  // RF[0] = RF[0] + RF[2]
                        // now D[5]+D[6]+D[7]
5: 0001 0000 00000101  // D[5] = RF[0]
```

- ***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 $ra_3ra_2ra_1ra_0$ $rb_3rb_2rb_1rb_0$ $rc_3rc_2rc_1rc_0$**

ignals to control the datapath



Assembly Code

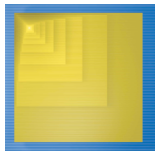
- Machine code (0s and 1s) hard to work with
- Assembly code – Uses mnemonics
  - ***Load*** instruction—**MOV Ra, d**
    - specifies the operation *RF[a]=D[d]*. *a* must be 0,1, ..., or 15—so *R0* means *RF[0]*, *R1* means *RF[1]*, etc. *d* must be 0, 1, ..., 255
  - • ***Store*** instruction—**MOV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

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

Desired program

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

machine code	assembly code
0: 0000 0000 00000000	0: MOV R0, 0
1: 0000 0001 00000001	1: MOV R1, 1
2: 0010 0010 0000 0001	2: ADD R2, R0, R1
3: 0001 0010 00001001	3: MOV 9, R2



Exercise: Creating Assembly Code

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

RF[2] = RF[1] + RF[4]

RF[3] = D[8]

RF[2] = RF[2] + RF[3]

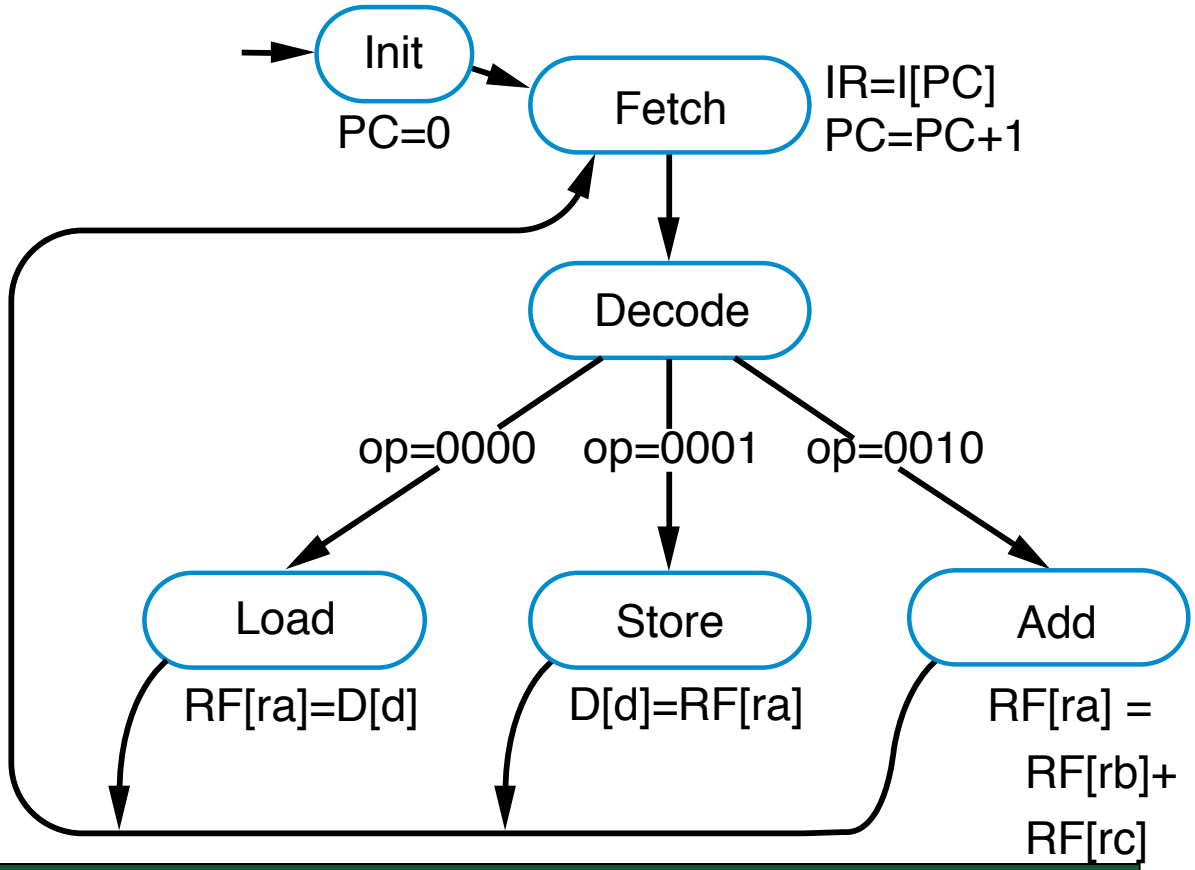
D[8] = RF[2]

Q2: RF[2] = RF[1]

RF[2] = RF[1] + 0 MOV R1, 0??

Control-Unit and Datapath for Three-Instruction Processor

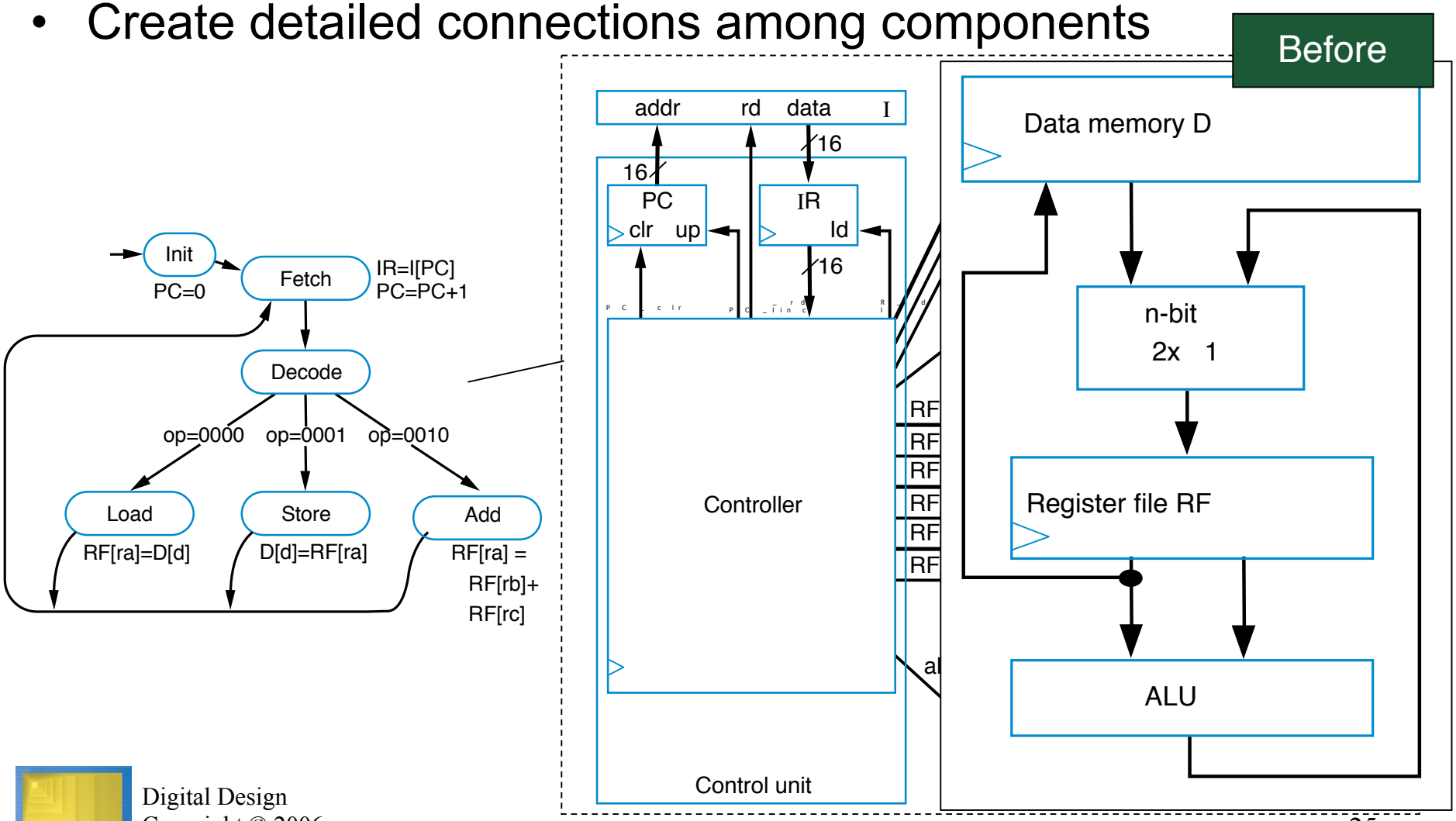
- To design the processor, we can begin with a high-level state machine description of the processor's behavior



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

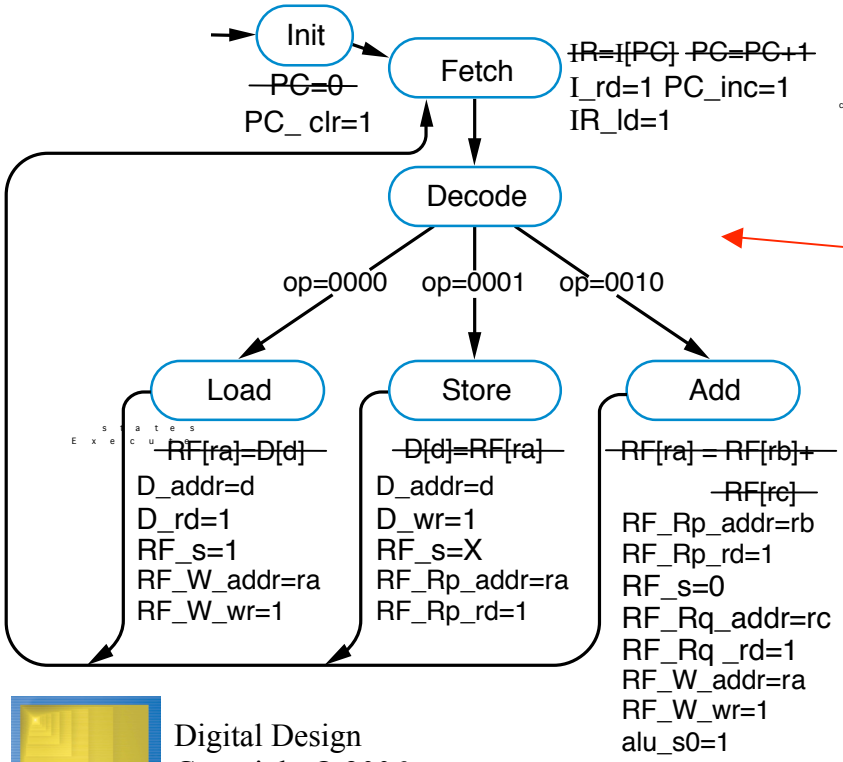
Control-Unit and Datapath for Three-Instruction Processor

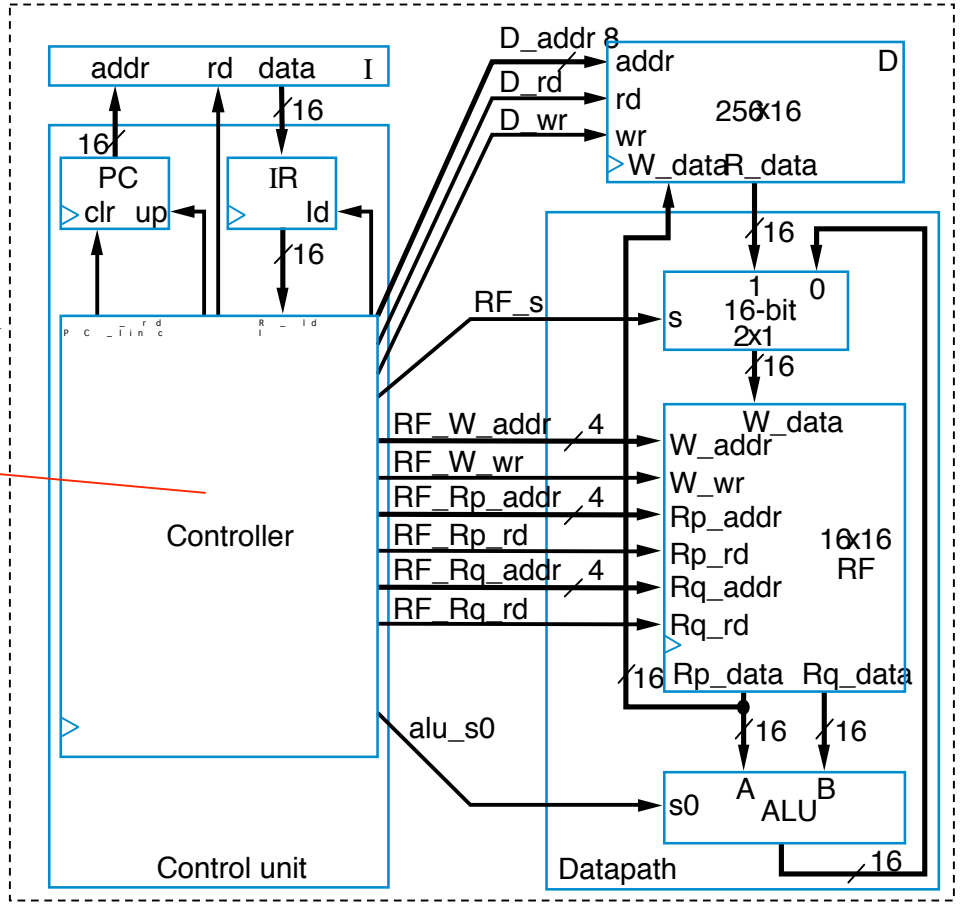
- Create detailed connections among components



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





Exercise: Understanding the Processor Design (1)

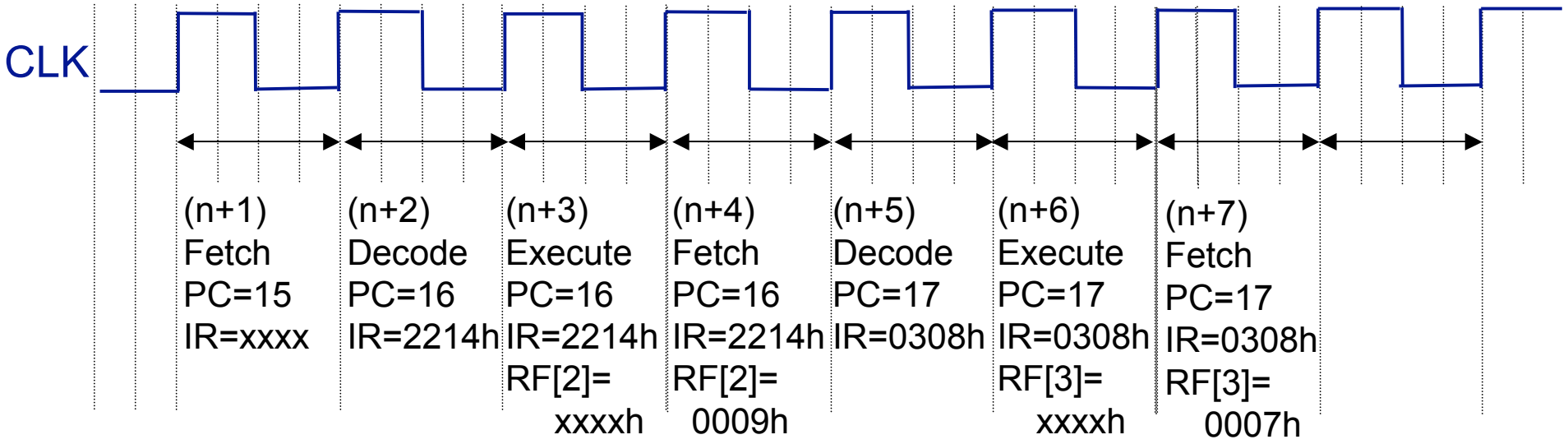
- Will the correct instruction be fetched if PC is incremented during the fetch cycle?
  - No, since PC will not be updated until the beginning of the next cycle
- While executing "MOV R1, 3", what is the content of PC and IR at the end of the 1st cycle, 2nd cycle, 3rd cycle, etc.?
  - 1st cycle: PC = 0, IR = xxxx
  - 2nd cycle: PC = 1, IR = I[0]
  - 3rd cycle: PC = 1, IR = I[0]
- What if it takes more than 1 cycle for memory read?
  - Cannot decode until IR is loaded

Exercise: Understanding the Processor Design (2)

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

```
…
I[15]:  Add   R2, R1,  R4
I[16]:  MOV R3, 8
I[17]:  Add   R2, R2, R3
…
```

RF[1] = 4
RF[4] = 5
D[8] = 7



Exercise: Extending the Three-Instruction Processor

Let's talk about A+B-1 a bit more…
Also, why is this instruction useful?

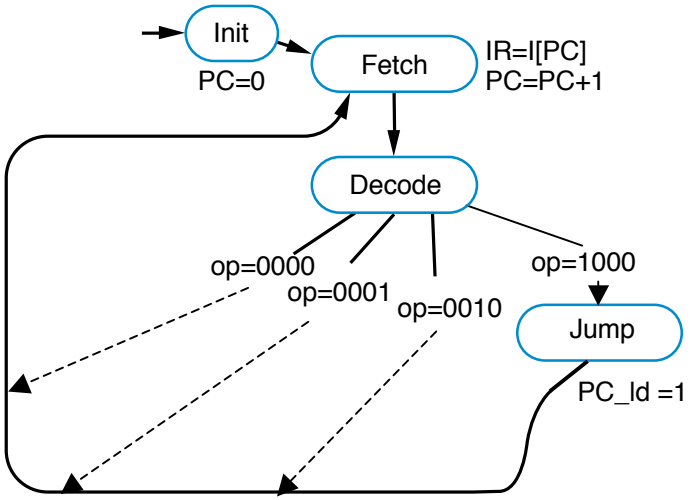
Add a instruction:

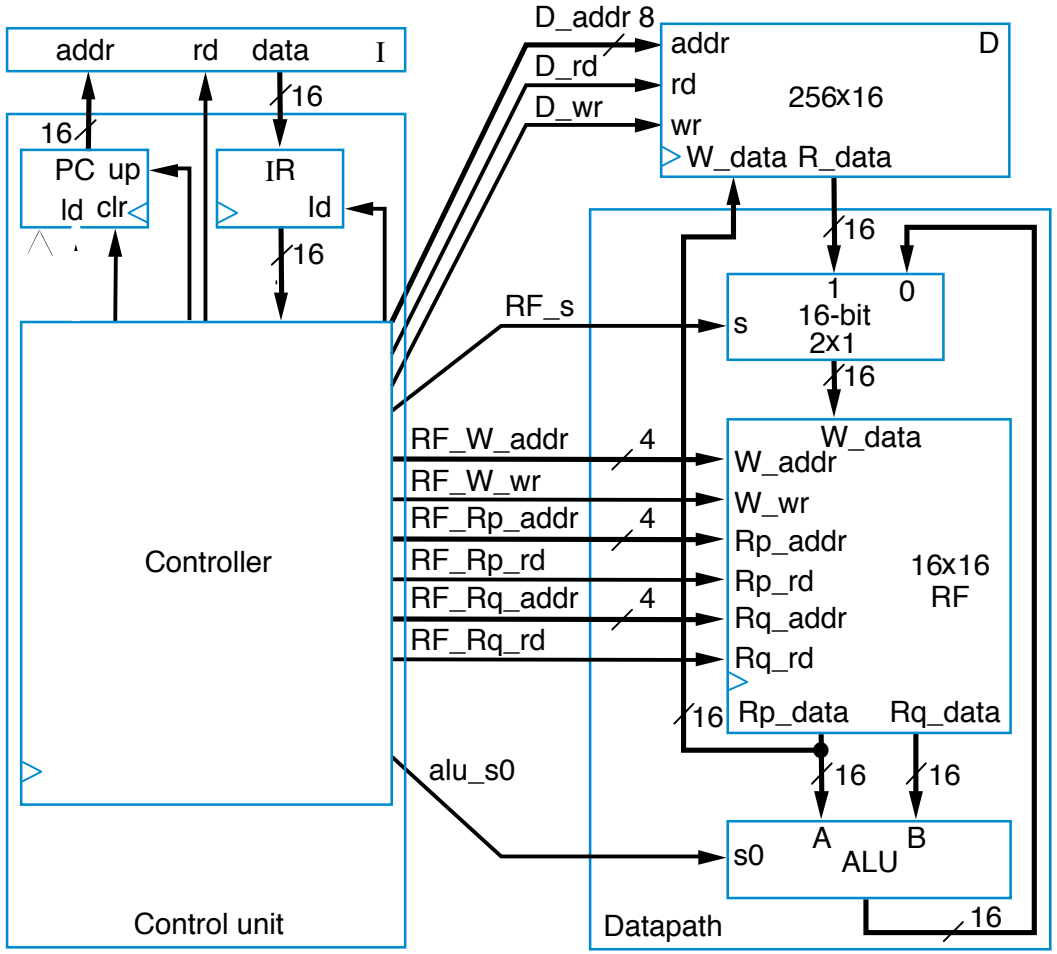
JMP: jump to a location specified by the 12-bit offset

RTL: PC = PC+I[15:4]

AS: JMP 3

MC: 1000000000000011





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