Arithmetic-Logic Unit (ALU)

Arithmetic-Logic Unit: ALU

- *ALU*: Component that can perform various arithmetic (add, subtract, increment, etc.) and logic (AND, OR, etc.) operations, based on control inputs

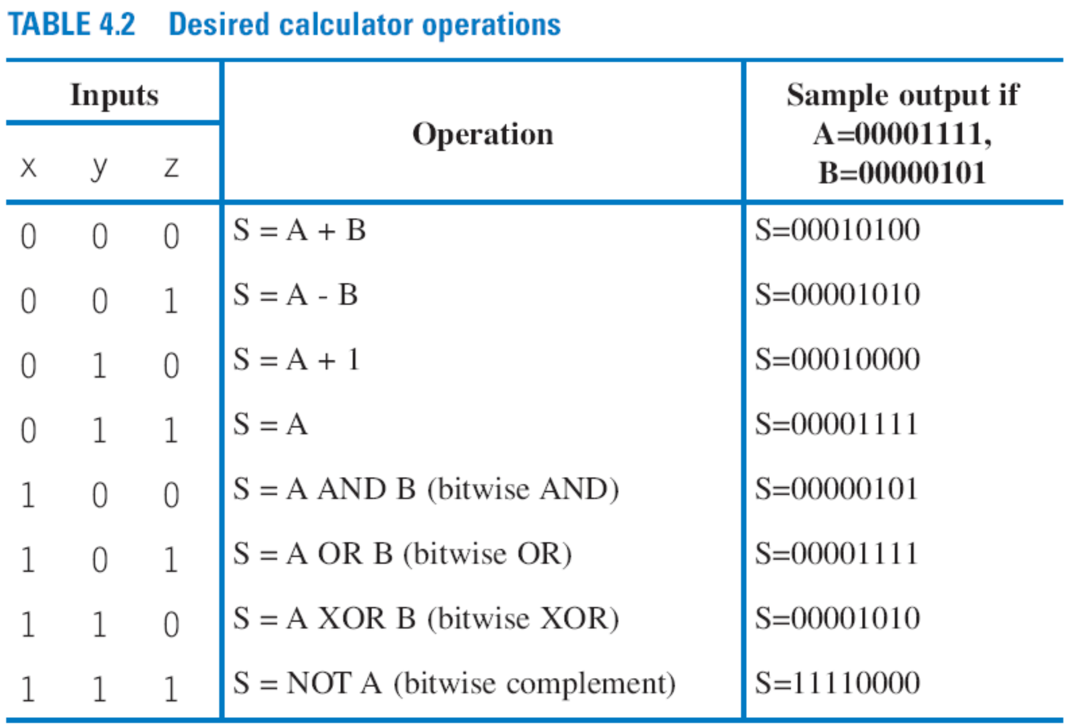


TABLE 4.2 Desired calculator operations

Inputs			Operation	Sample output if A=00001111, B=00000101
x	y	z		
0	0	0	S = A + B	S=00010100
0	0	1	S = A - B	S=00001010
0	1	0	S = A + 1	S=00010000
0	1	1	S = A	S=00001111
1	0	0	S = A AND B (bitwise AND)	S=00000101
1	0	1	S = A OR B (bitwise OR)	S=00001111
1	1	0	S = A XOR B (bitwise XOR)	S=00001010
1	1	1	S = NOT A (bitwise complement)	S=11110000

Multifunction Calculator without an ALU

- Can build using separate components for each operation, and muxes
  - Too many wires, also wastes power computing operations when only use one result at given time

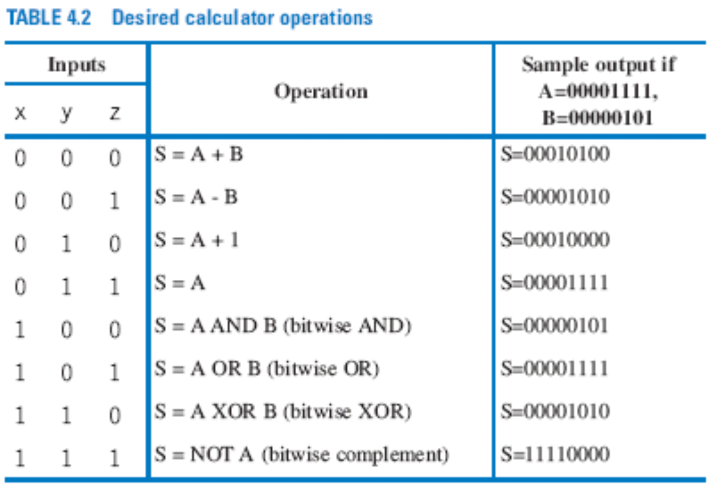
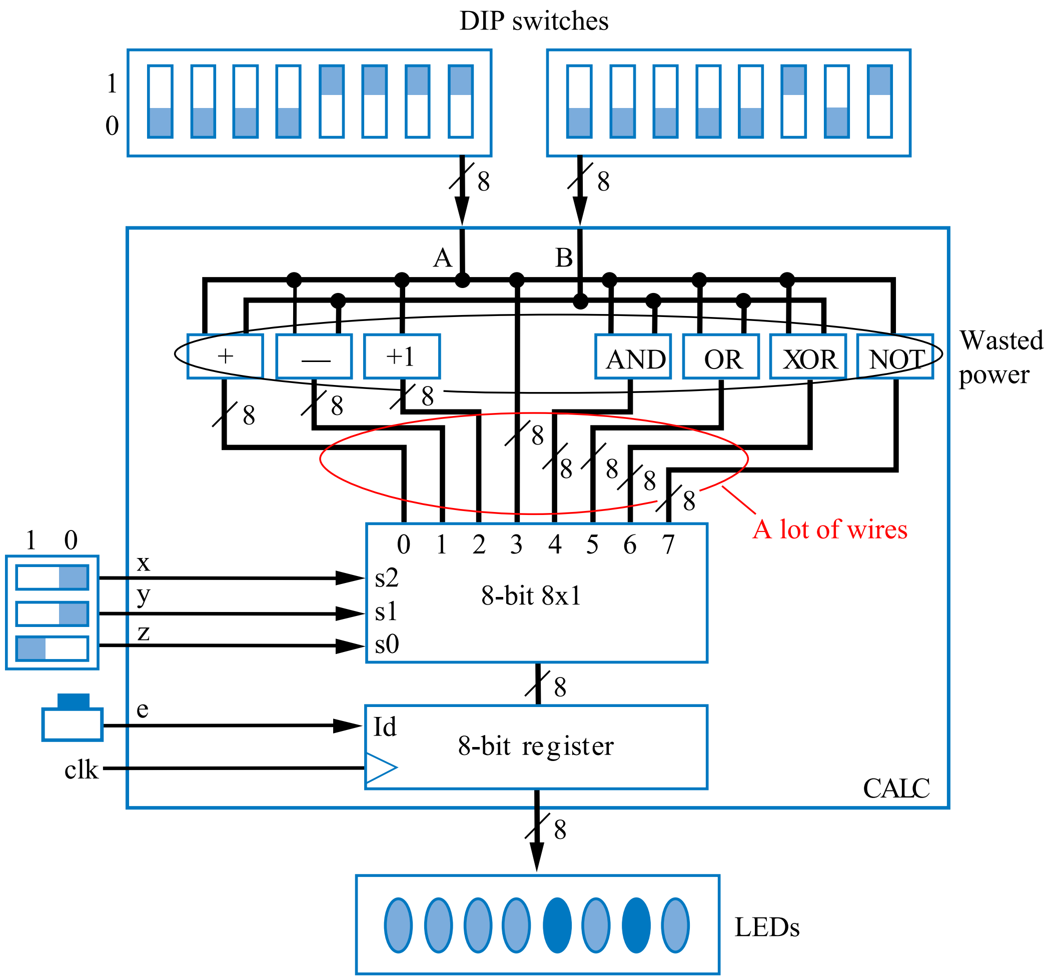


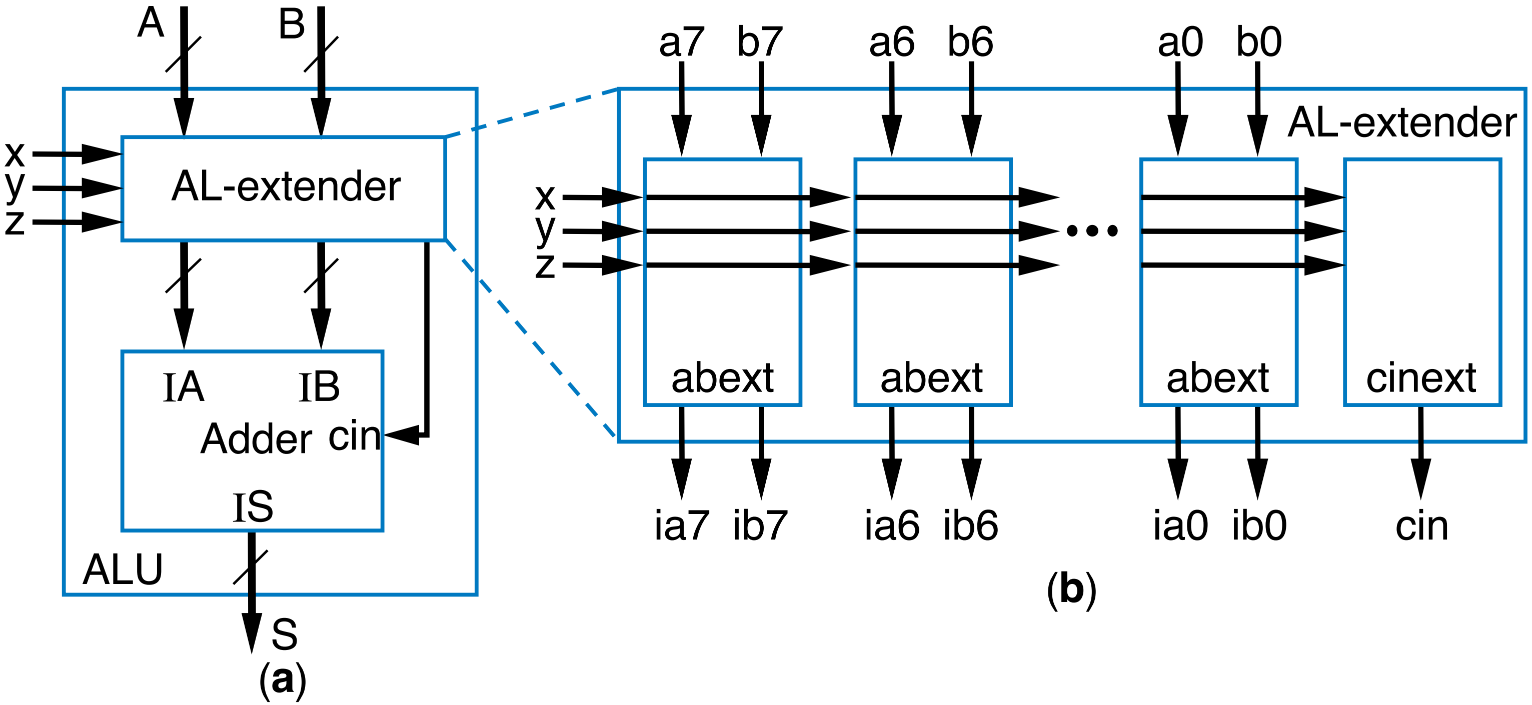
TABLE 4.2 Desired calculator operations

Inputs			Operation	Sample output if A=00001111, B=00000101
x	y	z		
0	0	0	S = A + B	S=00010100
0	0	1	S = A - B	S=00001010
0	1	0	S = A + 1	S=00010000
0	1	1	S = A	S=00001111
1	0	0	S = A AND B (bitwise AND)	S=00000101
1	0	1	S = A OR B (bitwise OR)	S=00001111
1	1	0	S = A XOR B (bitwise XOR)	S=00001010
1	1	1	S = NOT A (bitwise complement)	S=11110000



ALU

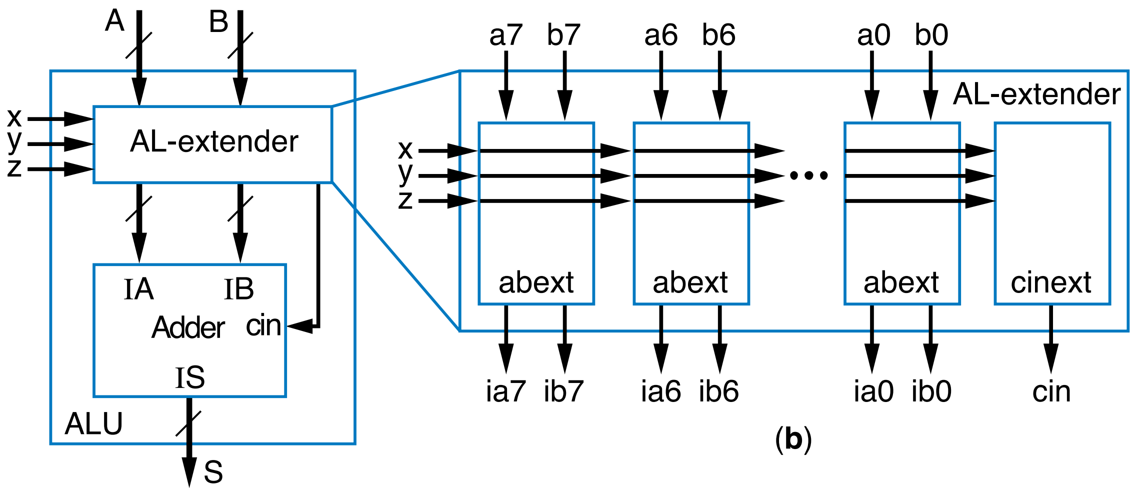
- More efficient design uses ALU
  - ALU design not just separate components multiplexed (same problem as previous slide)
  - Instead, ALU design uses single adder, plus logic in front of adder's A and B inputs
    - Logic in front is called an *arithmetic-logic extender*
  - Extender modifies A and B inputs so desired operation appears at output of the adder



Arithmetic-Logic Extender in Front of ALU

TABLE 4.2 Desired calculator operations

Inputs			Operation	Sample output if A=00001111, B=00000101
x	y	z		
0	0	0	S = A + B	S=00010100
0	0	1	S = A - B	S=00001010
0	1	0	S = A + 1	S=00010000
0	1	1	S = A	S=00001111
1	0	0	S = A AND B (bitwise AND)	S=00000101
1	0	1	S = A OR B (bitwise OR)	S=00001111
1	1	0	S = A XOR B (bitwise XOR)	S=00001010
1	1	1	S = NOT A (bitwise complement)	S=11110000

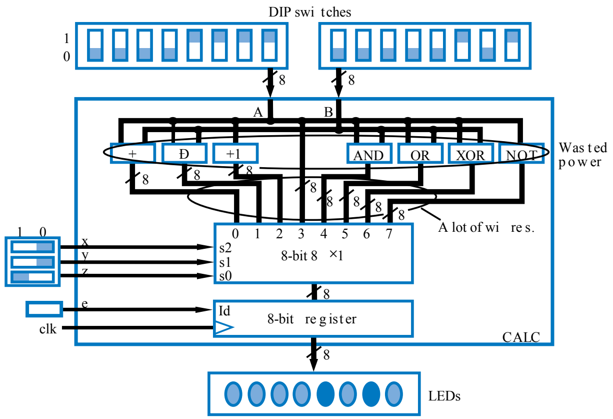


(a)

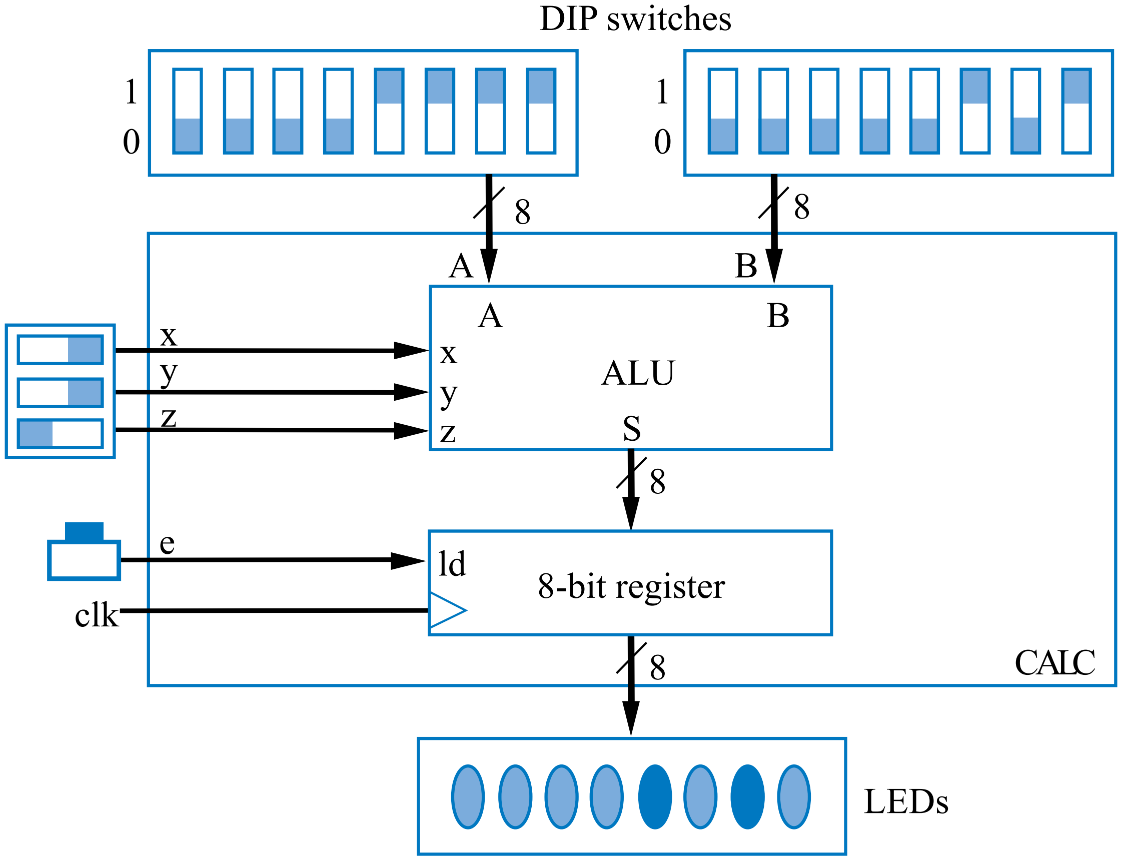
(b)

- xyz=000 Want S=A+B : just pass a to ia, b to ib, and set cin=0
- xyz=001 Want S=A-B : pass a to ia, b' to ib and set cin=1 (two's complement)
- xyz=010 Want S=A+1 : pass a to ia, set ib=0, and set cin=1
- xyz=011 Want S=A : pass a to ia, set ib=0, and set cin=0
- xyz=100 Want S=A AND B : set ia=a*b, b=0, and cin=0
- Others: likewise
- Based on above, create logic for ia(x,y,z,a,b) and ib(x,y,z,a,b) for each abext, and create logic for cin(x,y,z), to complete design of the AL-extender component

ALU Example: Multifunction Calculator



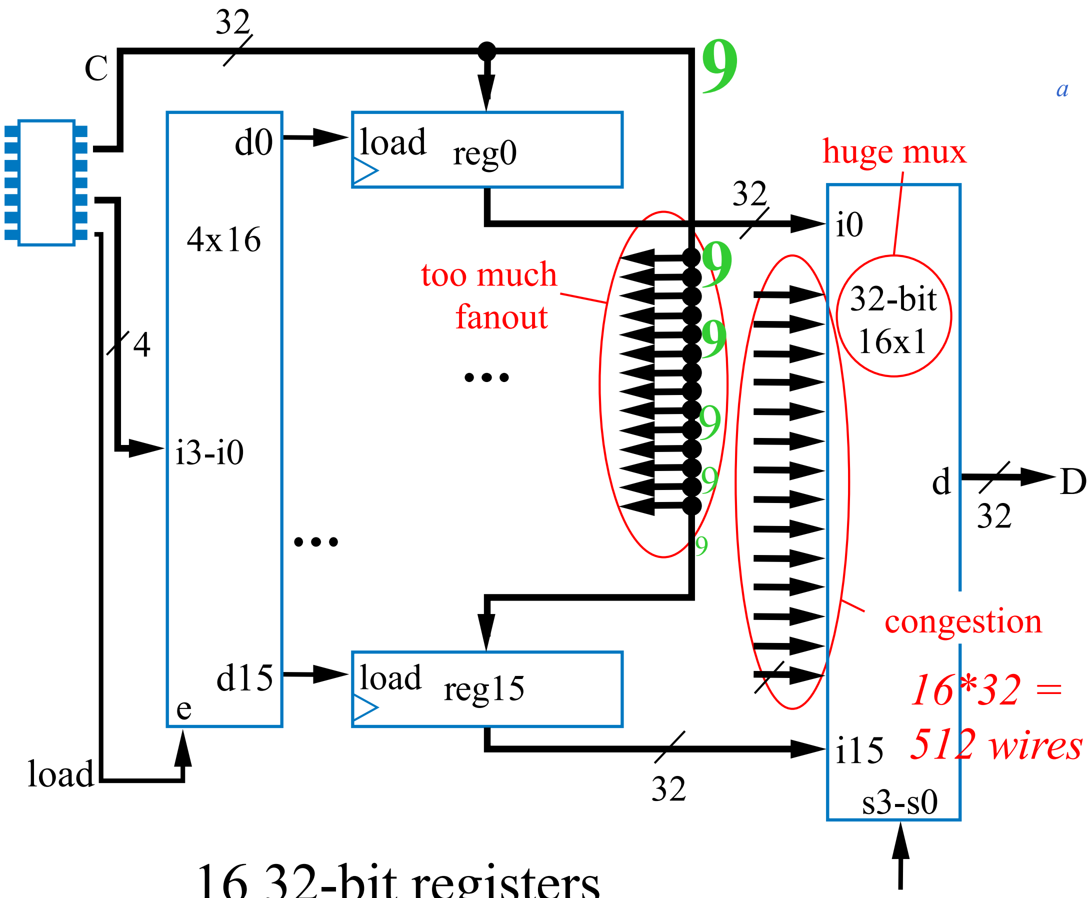
- Design using ALU is elegant and efficient
  - No mass of wires
  - No big waste of power



Register Files (Register Bank)

Register Files

- Accessing one of several registers is:
  - OK if just a few registers
  - Problematic when many
  - Ex: Earlier above-mirror display, with 16 registers
    - Much fanout (branching of wire): Weakens signal
    - Many wires: Congestion



16 32-bit registers begins to have fanout and wire problems

Register File

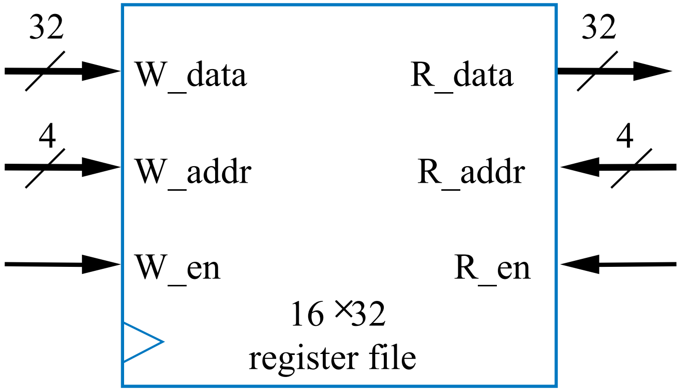
- MxN register file: Efficient design for one-at-a-time write/read of many registers
  - Consider 16 32-bit registers

Called “write port”

32-bit data to write

4-bit “address” specifies which register to write

Enable (load) line: Reg written on next clock



32-bit data that is read

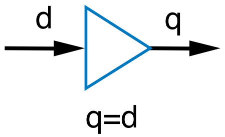
4-bit address to specifies which register to read

Enable read

“read port”

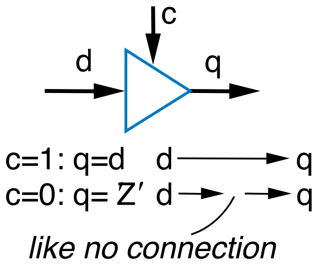
Register File

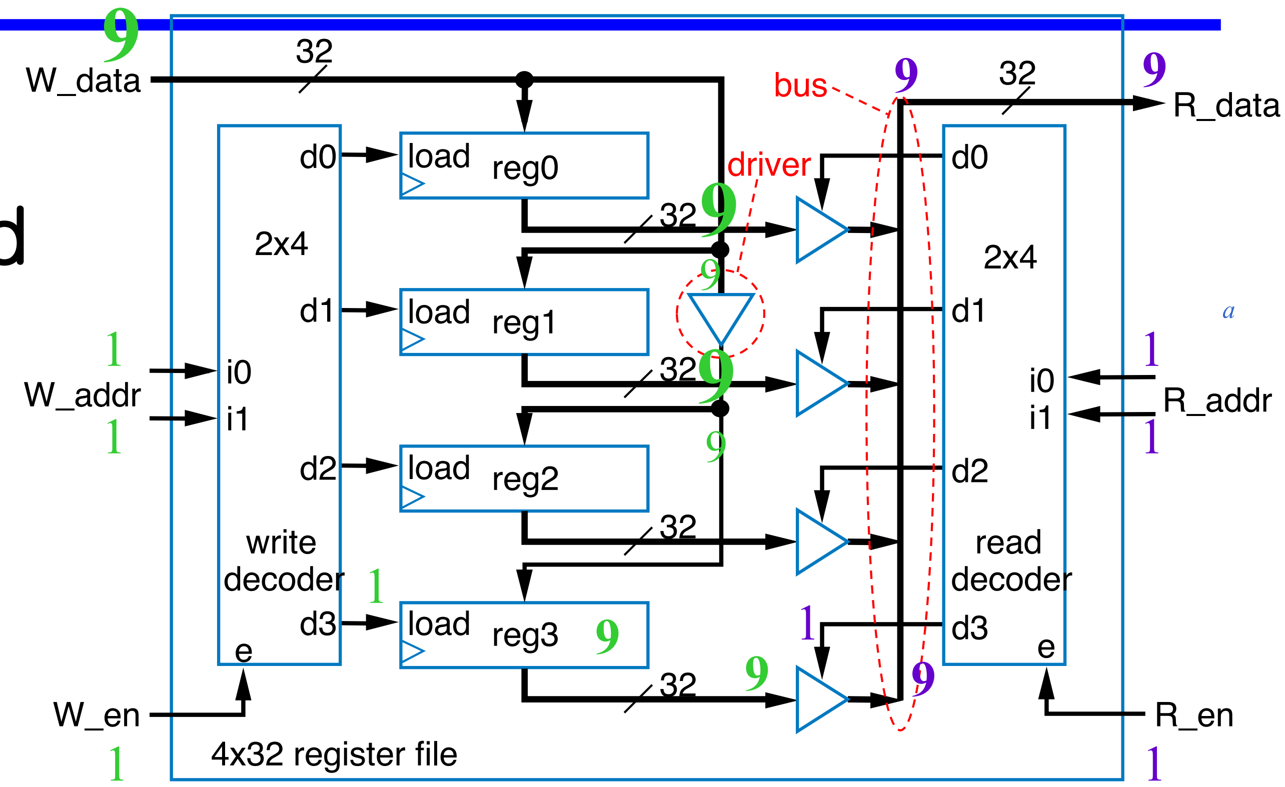
- Internal design uses drivers and bus driver



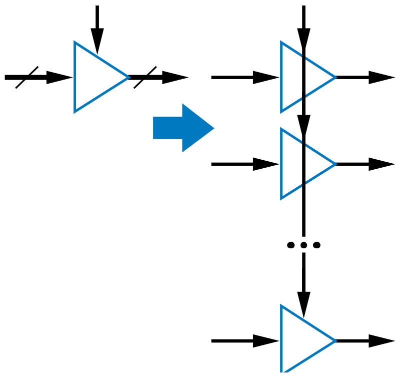
Boosts signal

three-state driver





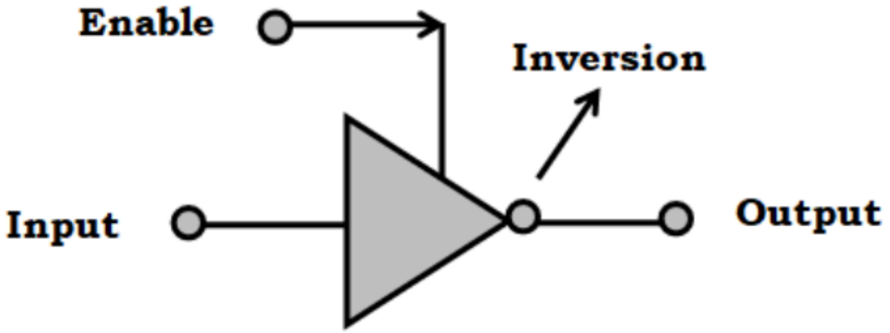
Internal design of 4x32 RF; 16x32 RF follows similarly

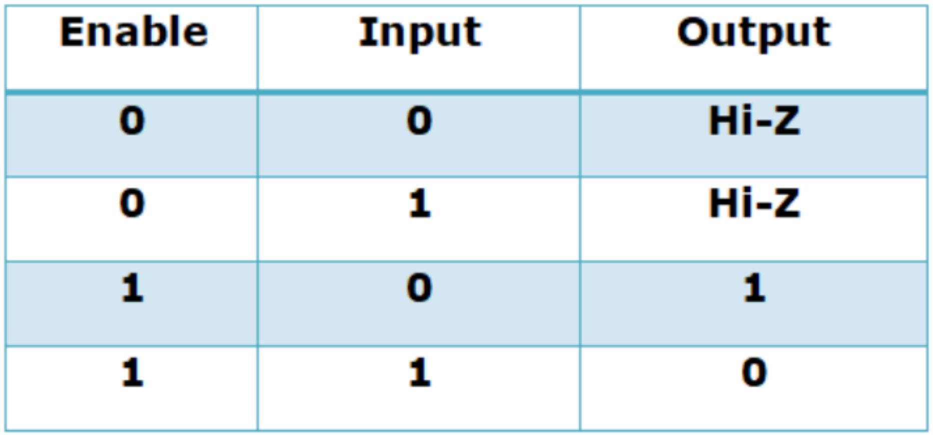


Note: Each driver in figure actually represents 32 1-bit drivers

Tri-state buffer

- Connect the input to output when enabled
- When disabled,
    - Output is not connected to any source
    - It is unstable state, and different from logical '0'

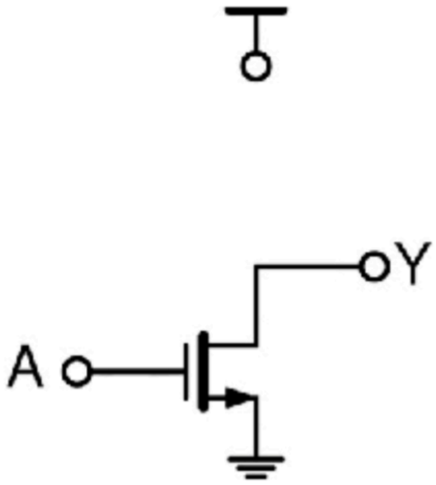


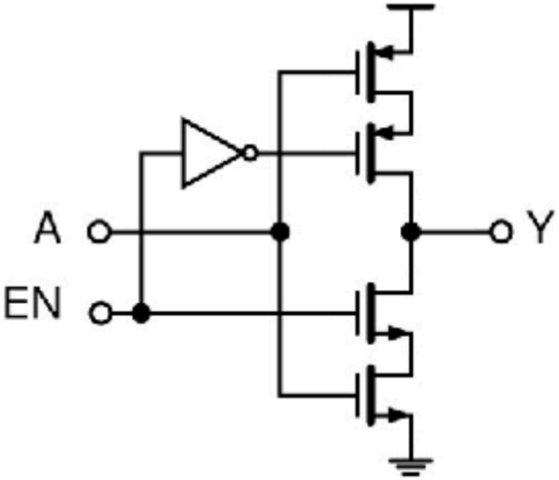


Enable	Input	Output
0	0	Hi-Z
0	1	Hi-Z
1	0	1
1	1	0

Tri-state buffer

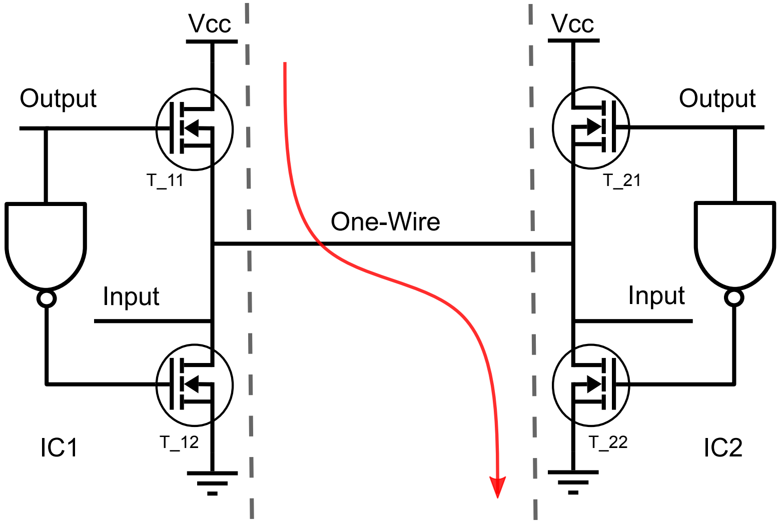
- High-Z (High impedance)
  - High enough impedance (resistance) to be regarded as disconnected

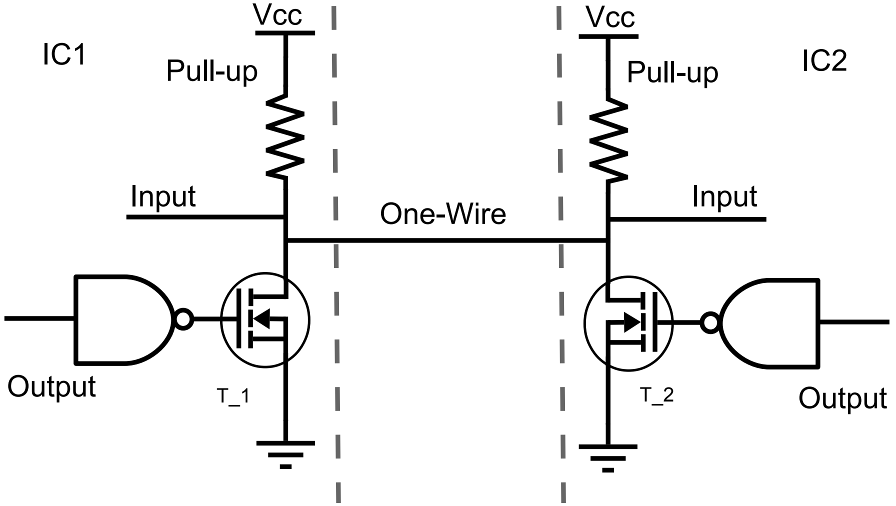




Open drain driver

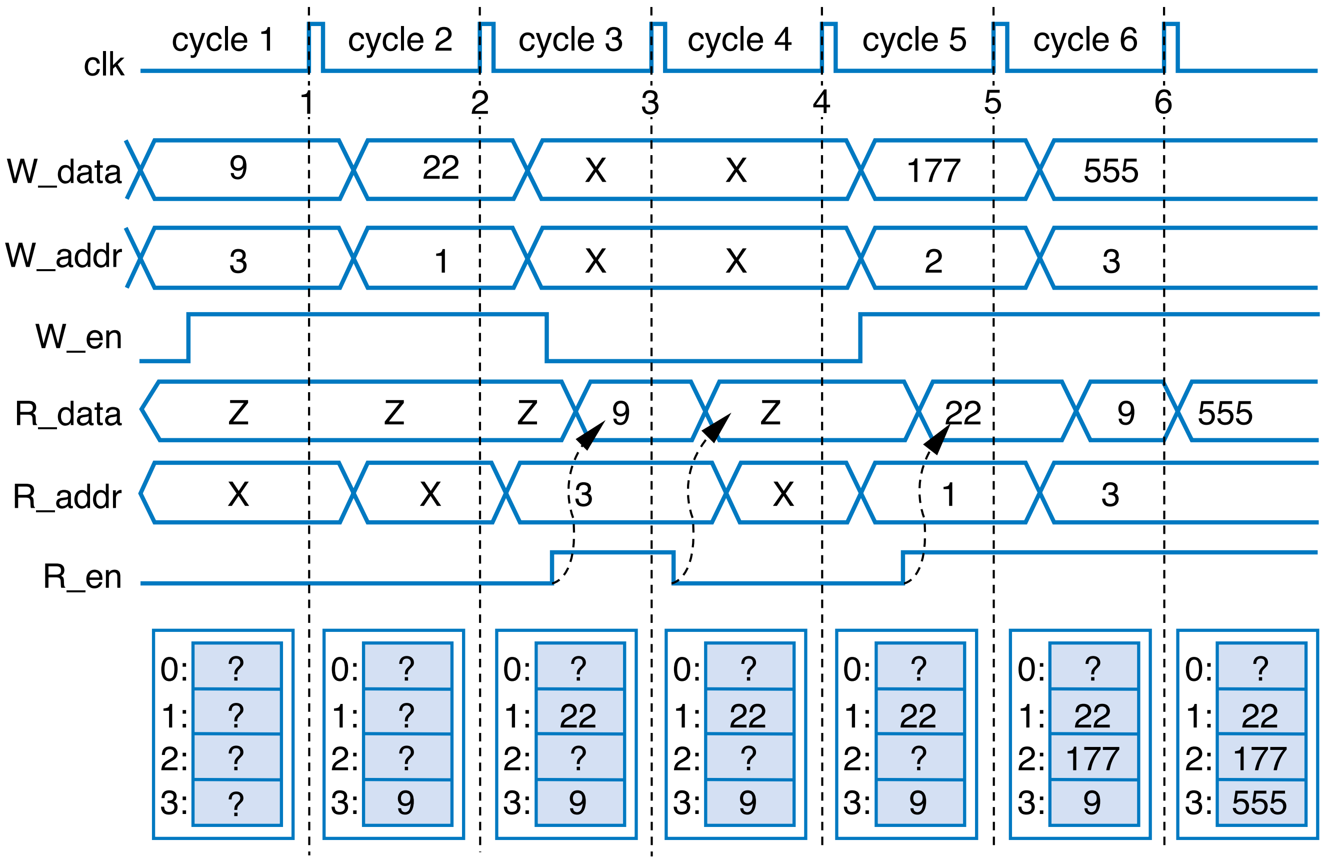
- Bus multiplexer
  - Aggregate output from multiple sources
- When multiple CMOS drivers are connected to one node,
  - Logical ‘0’ and ‘1’ values may exist at the same time
  - It causes logical conflict, and electrical short circuit
- How to avoid it?
  - Remove one side of driver capability.

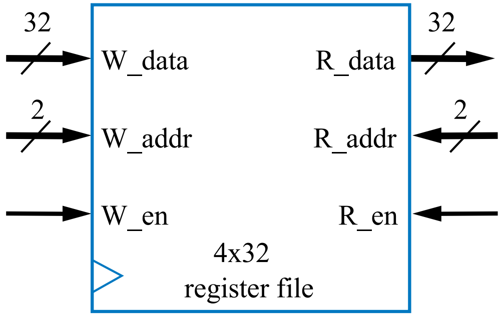




Register File Timing Diagram

- Can write one register and read one register each clock cycle
  - May be same register





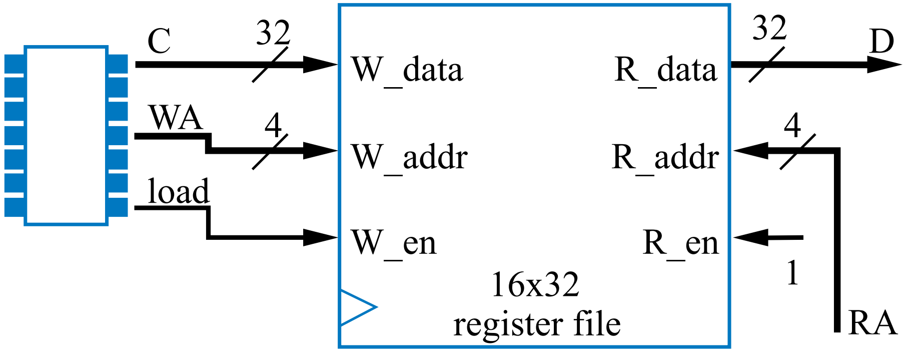
Chapter Summary

- Need datapath components to store and operate on multi-bit data
  - Also known as register-transfer-level (RTL) components
- Components introduced
  - Registers
  - Adders
  - Comparators
  - Multipliers
  - Subtractors
  - Arithmetic-Logic Units
  - Shifters
  - Counters and Timers
  - Register Files
- Next chapter combines knowledge of combinational logic design, sequential logic design, and datapath components, to build digital circuits that can perform general and powerful computations

Register-File Example: Above-Mirror Display

- 16 32-bit registers that can be written by car's computer, and displayed
  - Use 16x32 register file
  - Simple, elegant design

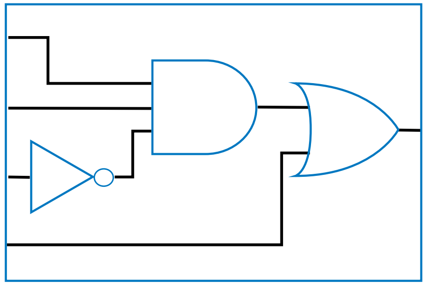
- Register file hides complexity internally
  - And because only one register needs to be written and/or read at a time, internal design is simple



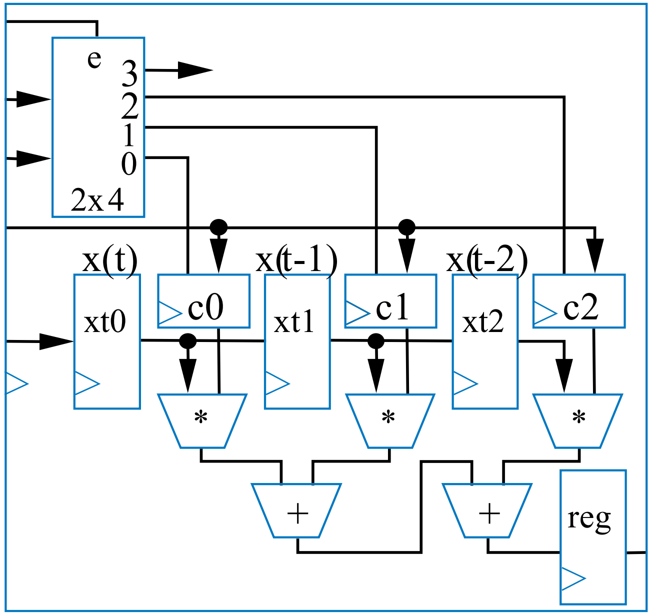
Programmable processor

Introduction

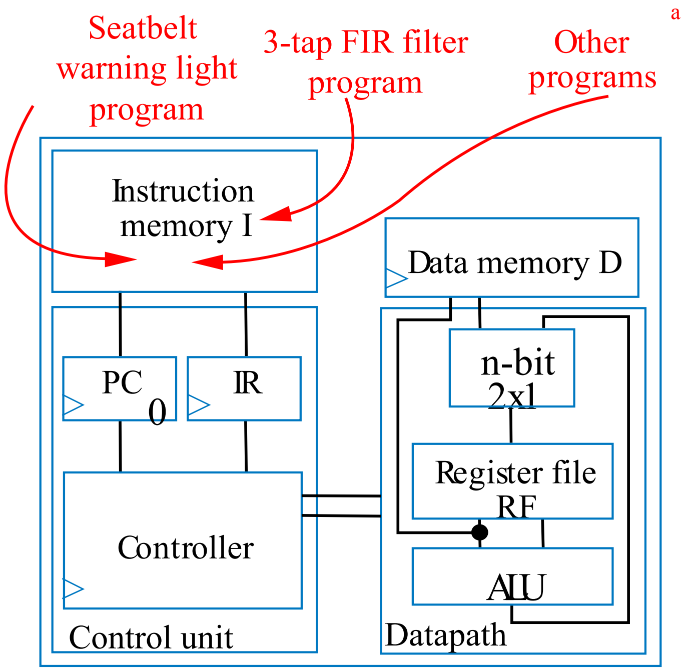
- Programmable (general-purpose) processor
  - Mass-produced, then programmed to implement different processing tasks
    - Well-known common programmable processors: Pentium, Sparc, PowerPC
    - Lesser-known but still common: ARM, MIPS, 8051, PIC, AVR
      - Low-cost embedded processors found in cell 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

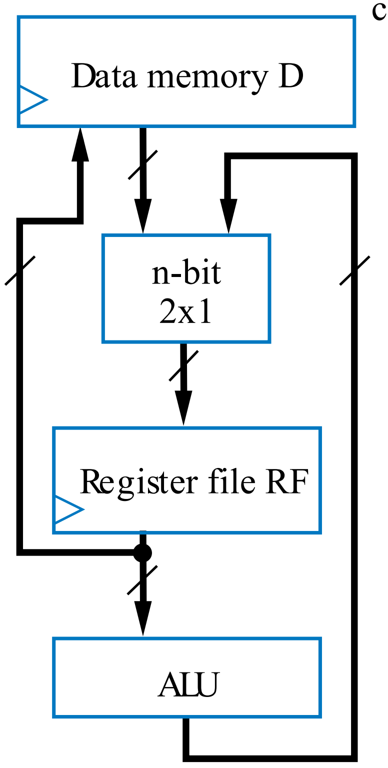


General-purpose processor

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

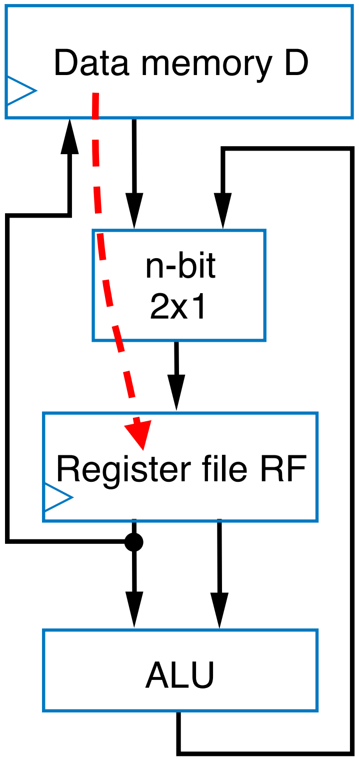


somehow connected to the outside world

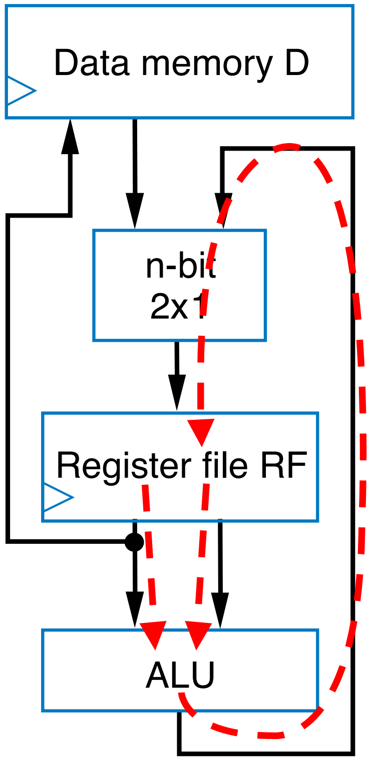
Datapath

Basic Datapath Operations

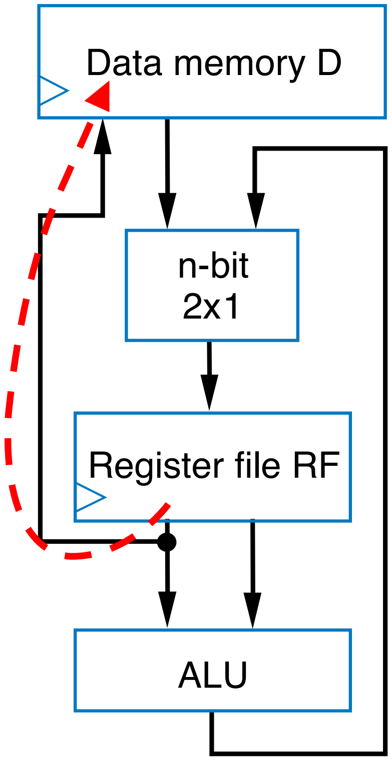
- Load operation: Load data from data memory to RF
- ALU operation: Transforms data by passing one or two RF register values through ALU, performing operation (ADD, SUB, AND, OR, etc.), and writing back into RF.
- Store operation: Stores RF register value back into data memory
- Each operation can be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



Load operation



ALU operation



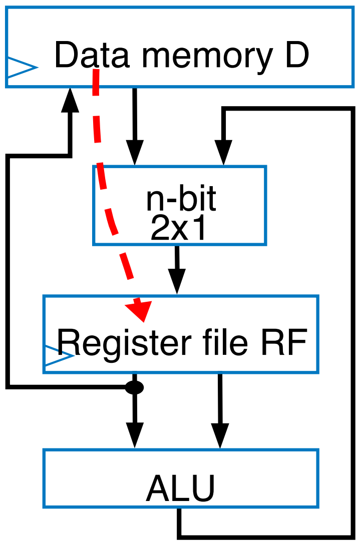
Store operation

a

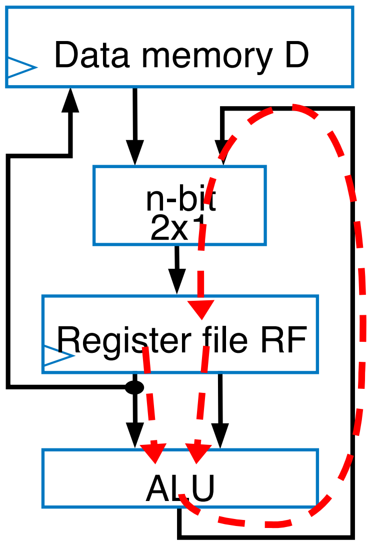
Basic Datapath Operations

- 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])

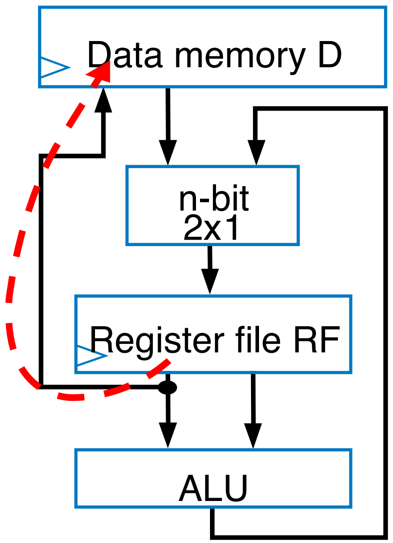
a



Load operation



ALU operation



Store operation

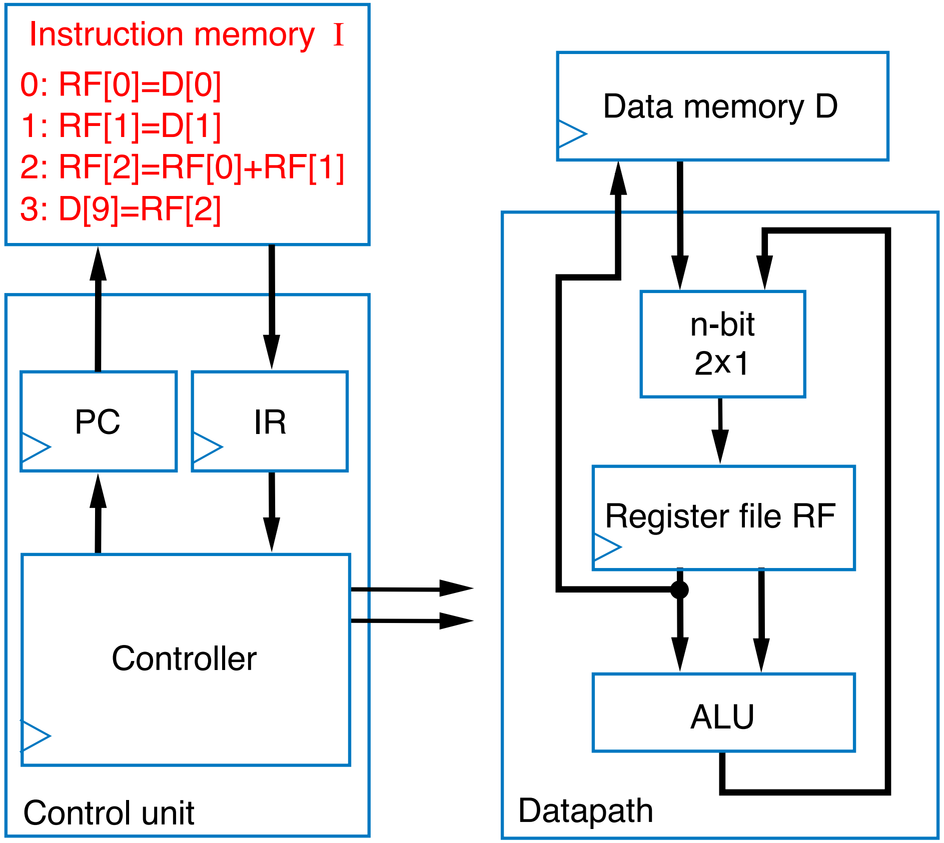
Basic Architecture – Control Unit

- 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]

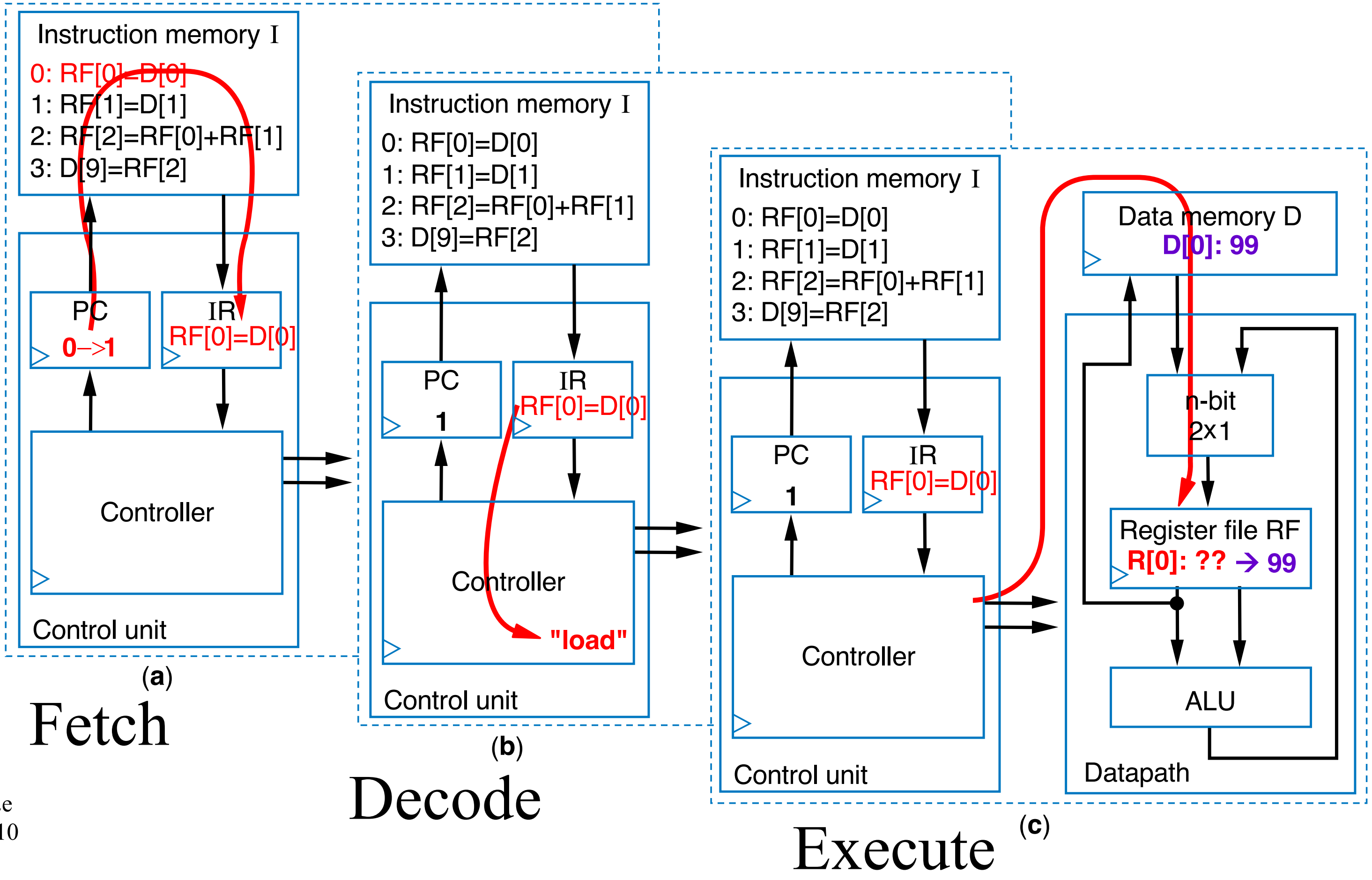
- Each operation is an *instruction*
  - Sequence of instructions – *program*
  - Looks cumbersome, but that's the world of programmable processors – Decomposing desired computations into processor-supported operations
  - Store program in *Instruction memory*
  - *Control unit* reads each instruction and executes it on the datapath
    - PC: Program counter – address of current instruction
    - IR: Instruction register – current instruction

a



Basic Architecture – Control Unit

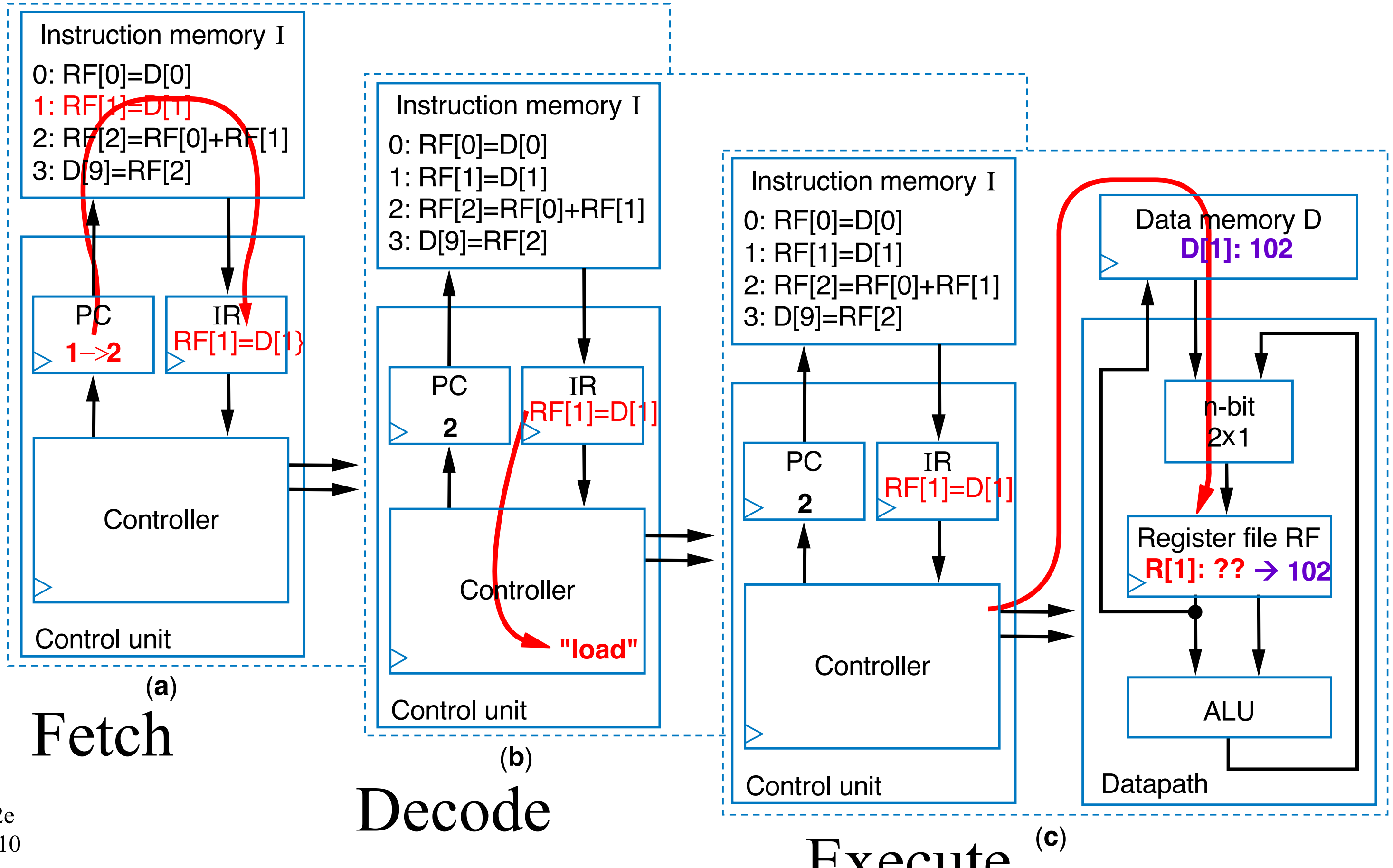
- To carry out *each instruction*, the control unit must:
  - Fetch – Read instruction from inst. mem.
  - Decode – Determine the operation and operands of the instruction
  - Execute – Carry out the instruction's operation using the datapath



a

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



(a)

Fetch

(b)

Decode

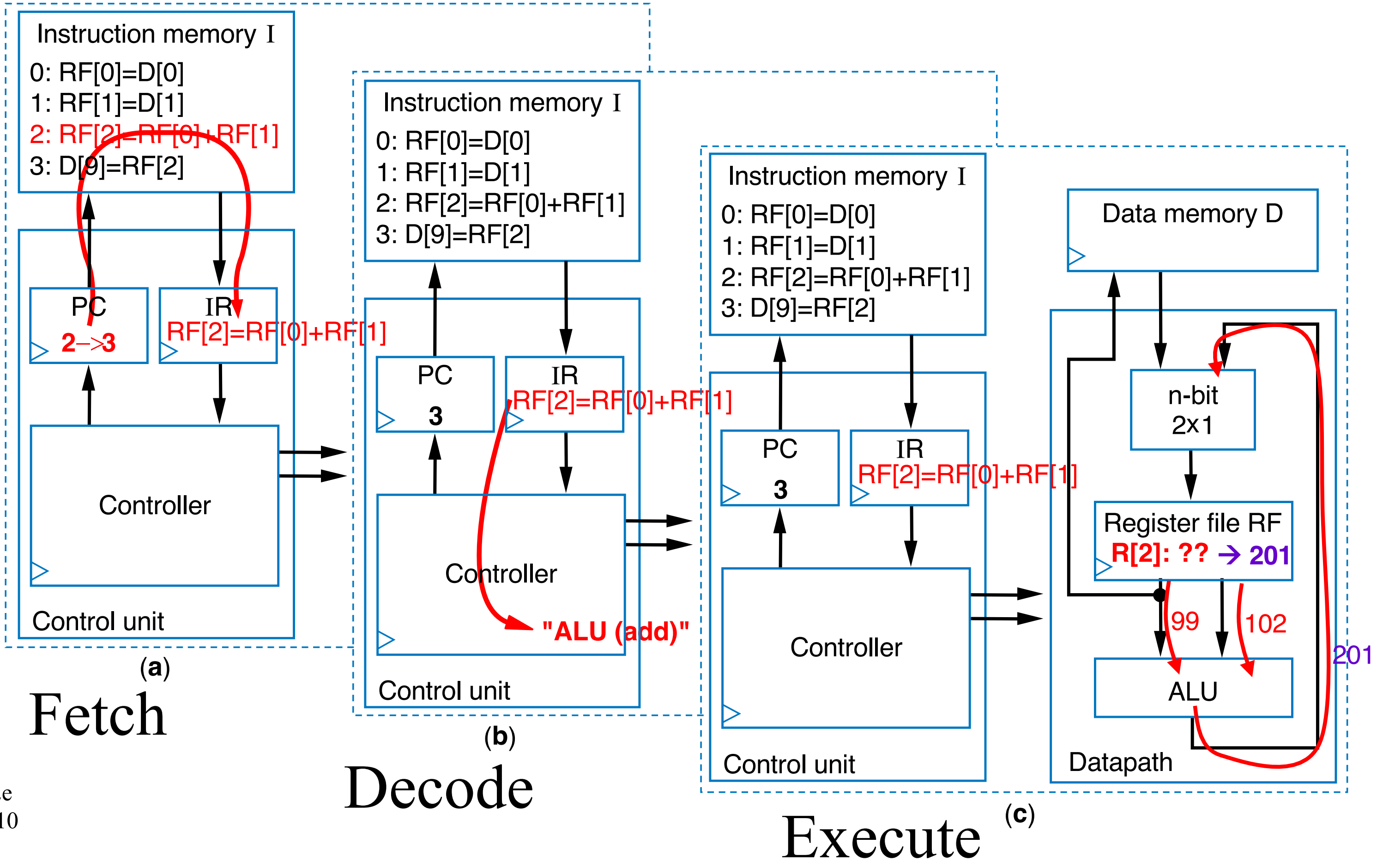
(c)

Execute

a

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

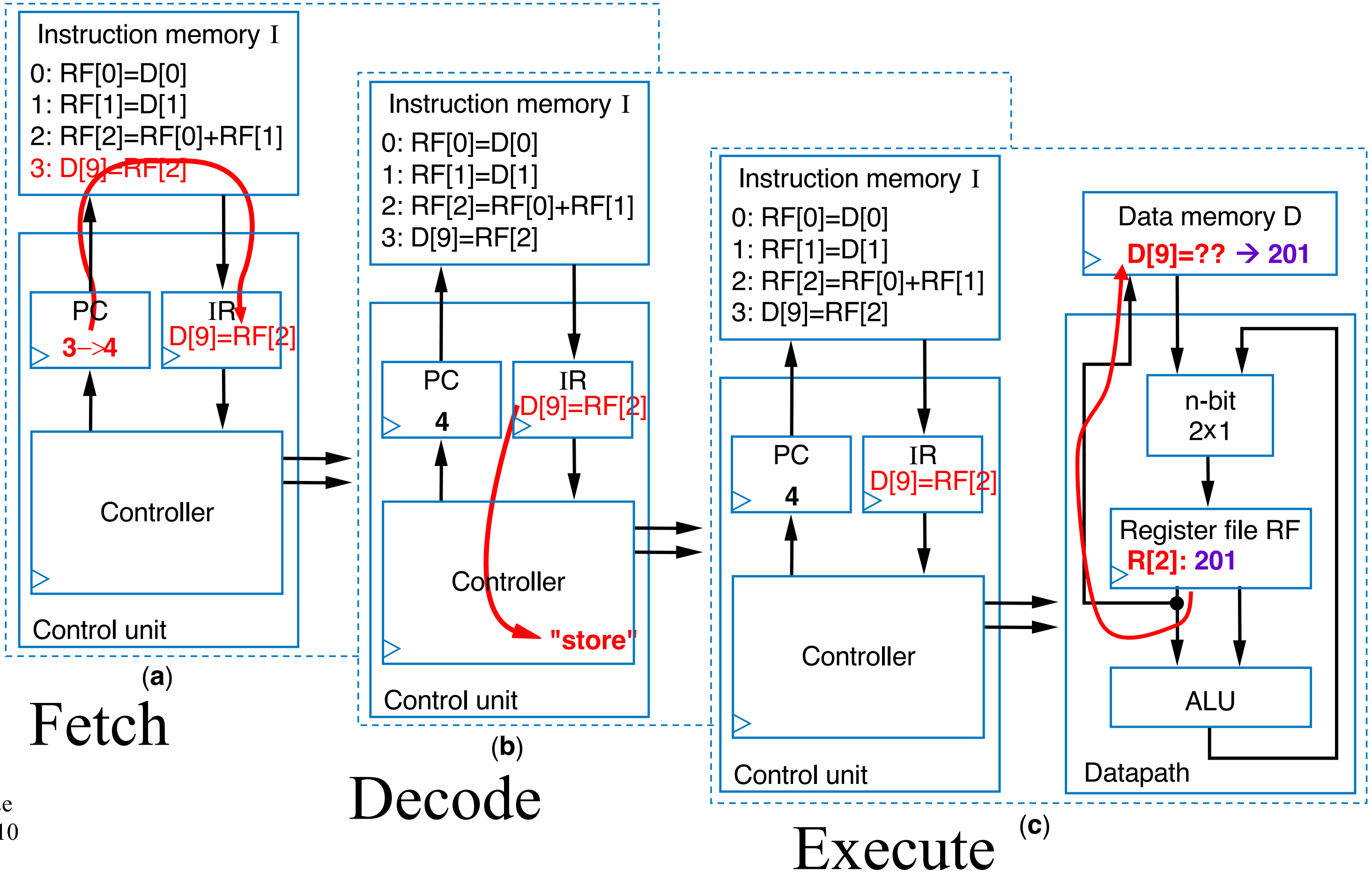


Instruction memory I
0: RF[0]=D[0]
1: RF[1]=D[1]
2: RF[2]=RF[0]+RF[1]
3: D[9]=RF[2]

a

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

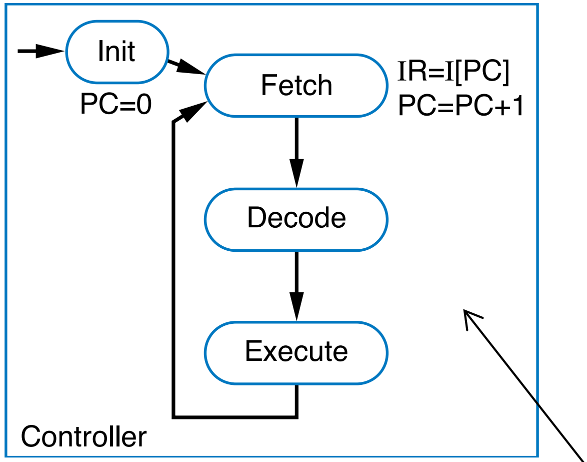


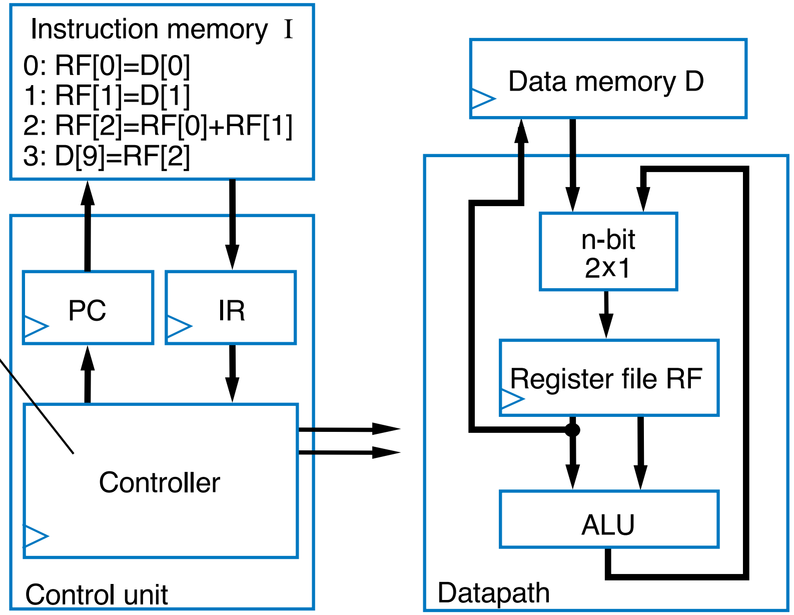
a

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]

a

- 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]

a

Number of Cycles

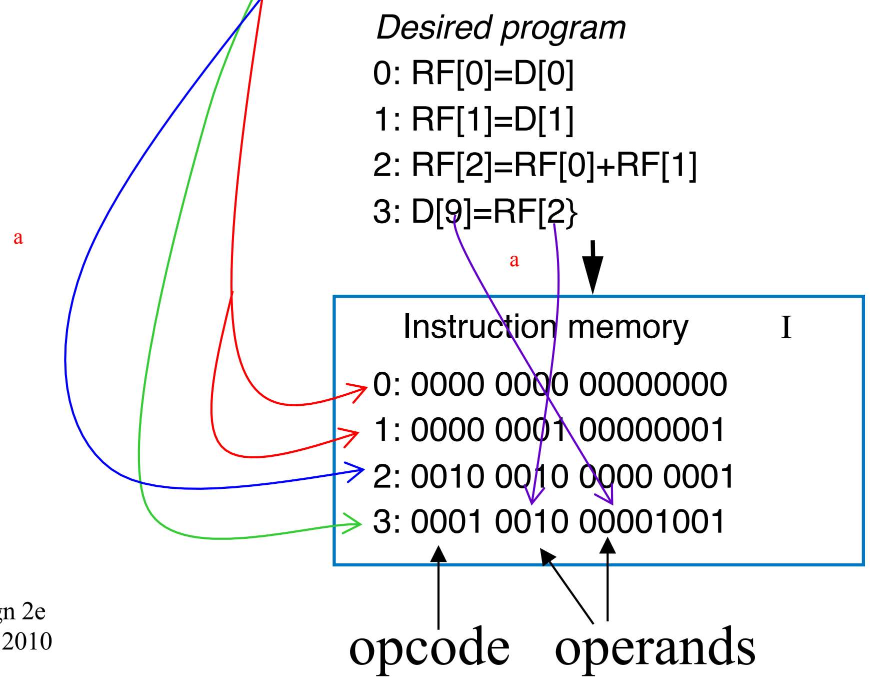
- Q: How many cycles are needed to execute six instructions using the earlier-described processor?

- A: Each instruction requires 3 cycles – 1 to fetch, 1 to decode, and 1 to execute
  - Thus, 6 instr * 3 cycles/instr = 18 cycles

a

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 $ra_3ra_2ra_1ra_0$ $rb_3rb_2rb_1rb_0$ $rc_3rc_2rc_1rc_0$**



Desired program

0: RF[0]=D[0]

1: RF[1]=D[1]

2: RF[2]=RF[0]+RF[1]

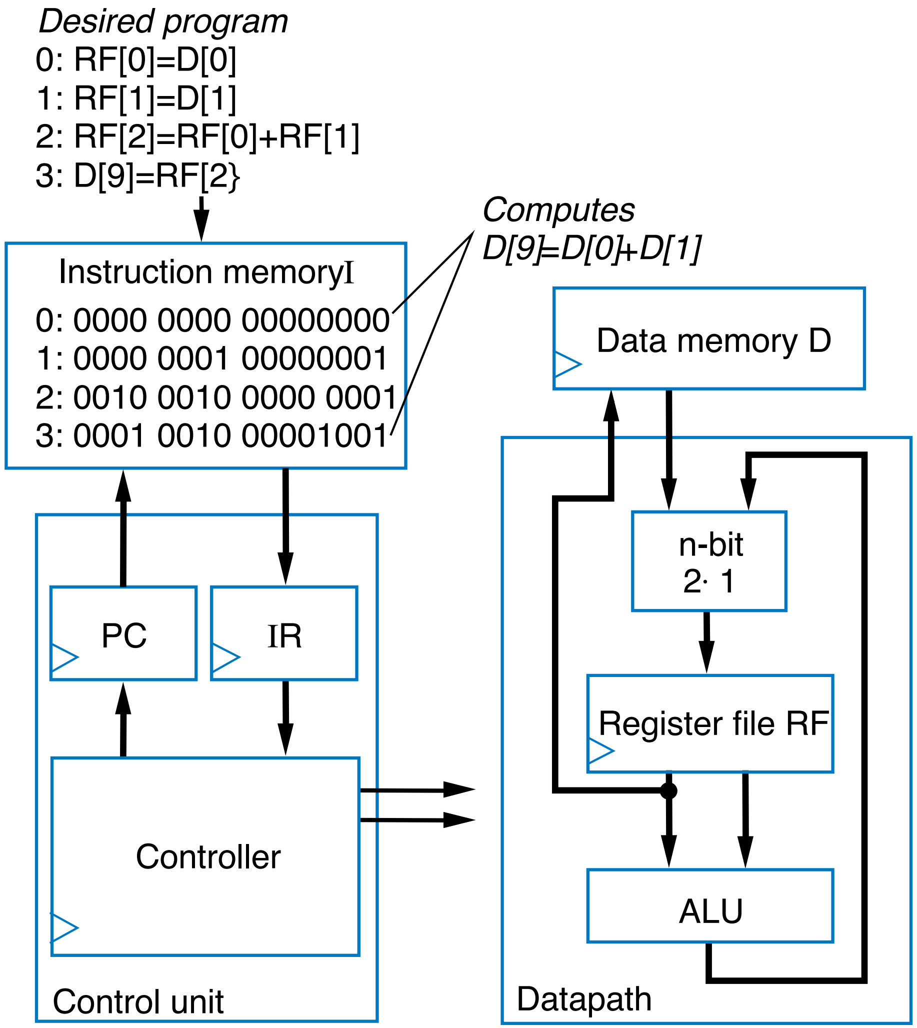
3: D[9]=RF[2}

Instruction memory

```
0: 0000 0000 00000000
1: 0000 0001 00000001
2: 0010 0010 0000 0001
3: 0001 0010 00001001
```

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

Program for Three-Instruction Processor



Desired program

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

Instruction memoryI

0: 0000 0000 00000000
1: 0000 0001 00000001
2: 0010 0010 0000 0001
3: 0001 0010 00001001

Computes D[9]=D[0]+D[1]

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 $ra_3ra_2ra_1ra_0$ $rb_3rb_2rb_1rb_0$ $rc_3rc_2rc_1rc_0$**

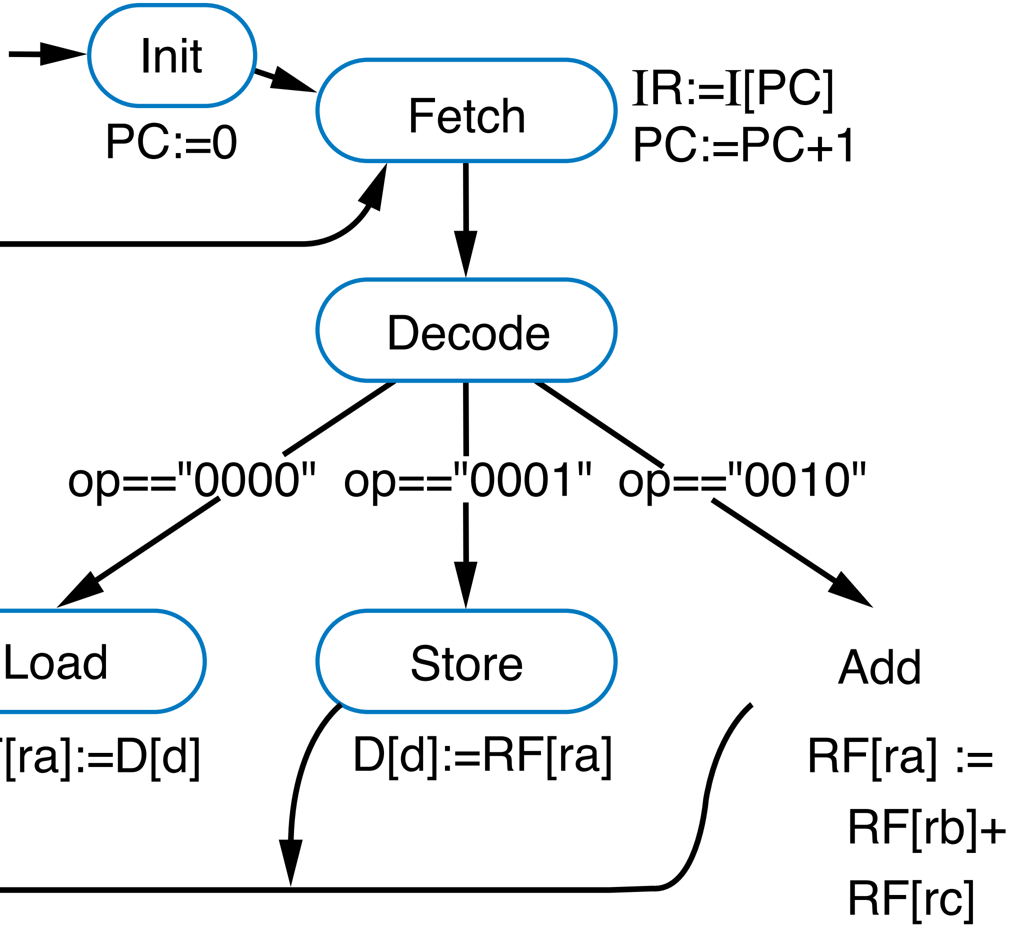
Assembly Code

- 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		
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
	machine code	assembly code

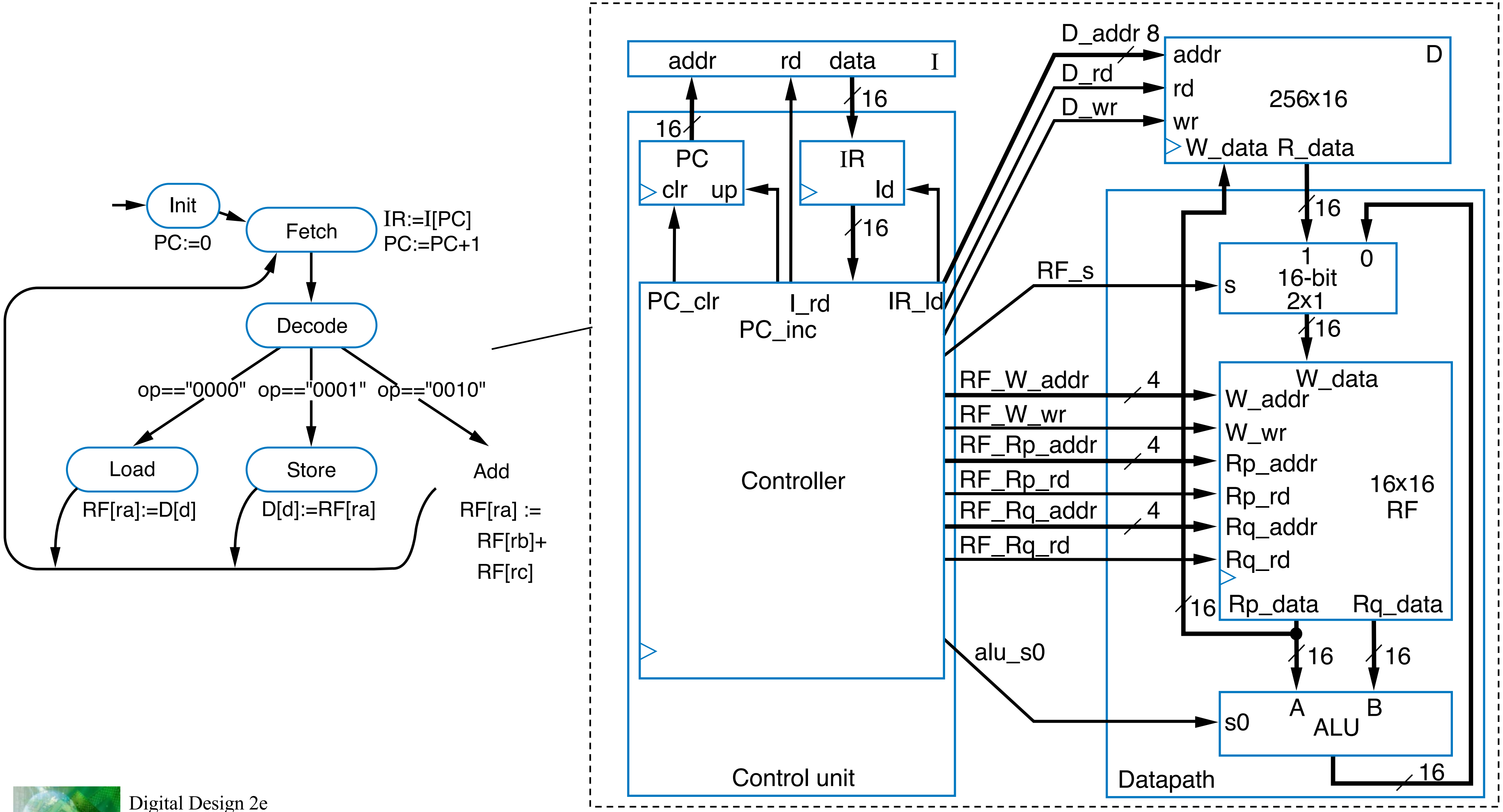
Control-Unit and Datapath for Three-Instruction Processor

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



Control-Unit and Datapath for Three-Instruction Processor

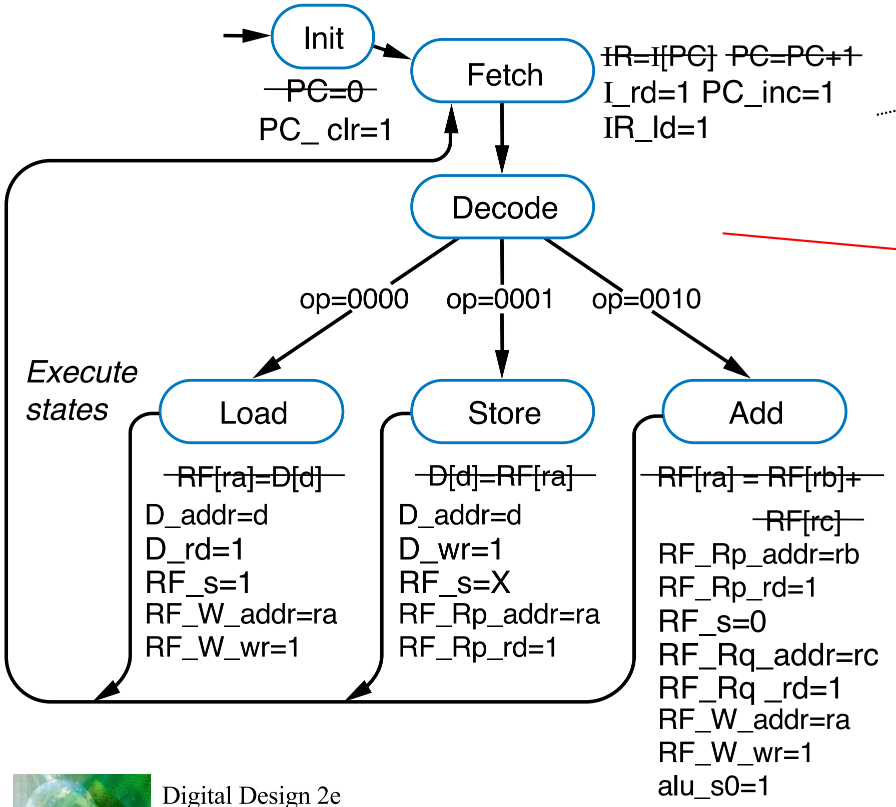
- Create detailed connections among components



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

a



Init

~~PC=0~~
PC_ clr=1

Fetch

~~IR=I[PC] PC=PC+1~~
I_rd=1 PC_inc=1
IR_ld=1

Decode

op=0000 op=0001 op=0010

Execute states

Load

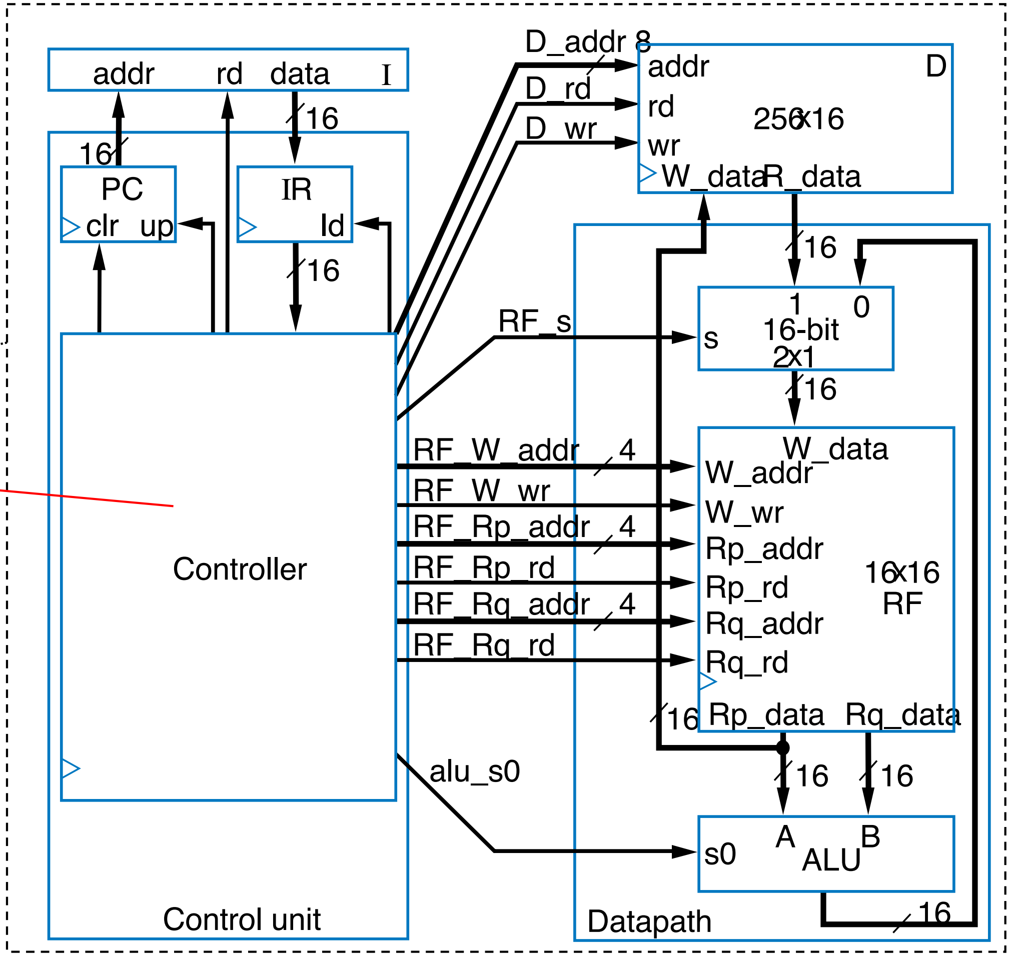
~~RF[ra]=D[d]~~
D_addr=d
D_rd=1
RF_s=1
RF_W_addr=ra
RF_W_wr=1

Store

~~D[d]=RF[ra]~~
D_addr=d
D_wr=1
RF_s=X
RF_Rp_addr=ra
RF_Rp_rd=1

Add

~~RF[ra] = RF[rb]+~~
~~RF[rc]~~
RF_Rp_addr=rb
RF_Rp_rd=1
RF_s=0
RF_Rq_addr=rc
RF_Rq _rd=1
RF_W_addr=ra
RF_W_wr=1
alu_s0=1



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

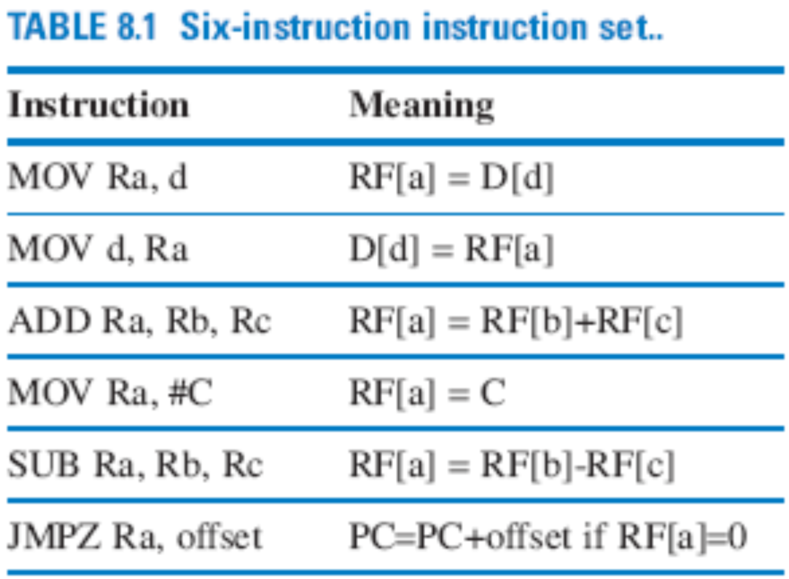


TABLE 8.1 Six-instruction instruction set..

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

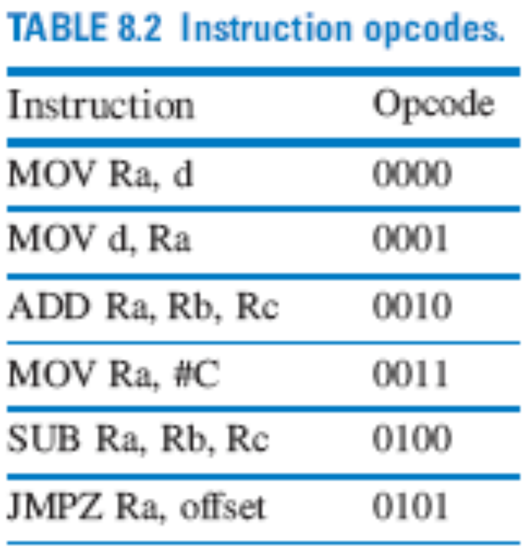


TABLE 8.2 Instruction opcodes.

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

Extending the Control-Unit and Datapath

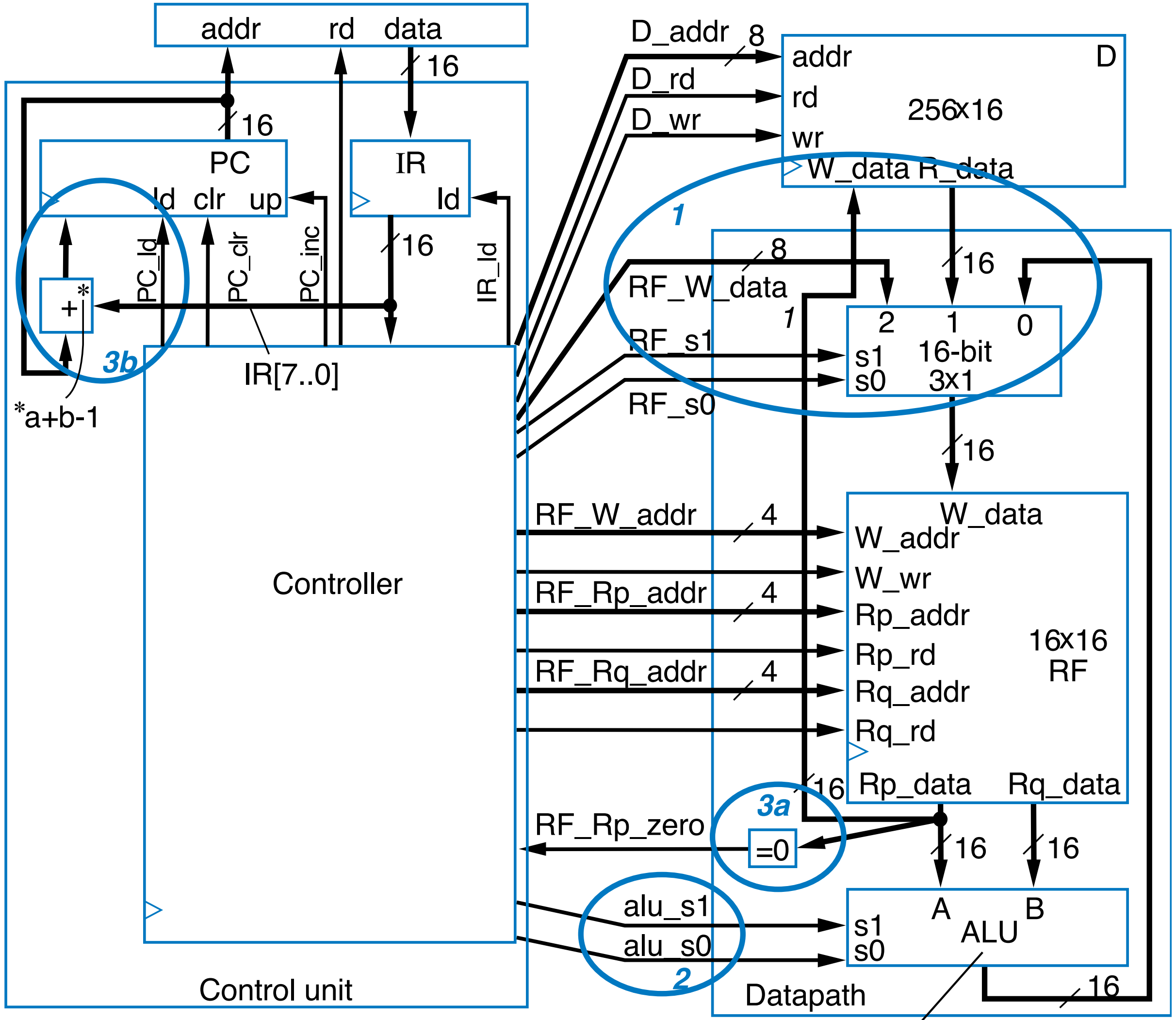
1: The *load constant* instruction requires that the register file be able to load data from *IR[7..0]*, in addition to data from data memory or the ALU output. Thus, we widen the register file's multiplexer from 2x1 to 3x1, add another mux control signal, and also create a new signal coming from the controller labeled *RF_W_data*, which will connect with *IR[7..0]*.

2: The subtract instruction requires that we use an ALU capable of subtraction, so we add another ALU control signal.

3: The jump-if-zero instruction requires that we be able to detect if a register is zero, and that we be able to add *IR[7..0]* to the *PC*.

3a: We insert a datapath component to detect if the register file's *Rp* read port is all zeros (that component would just be a NOR gate).

3b: We also upgrade the *PC* register so it can be loaded with *PC* plus *IR[7..0]*. The adder used for this also subtracts 1 from the sum, to compensate for the fact that the *Fetch* state already added 1 to the *PC*.

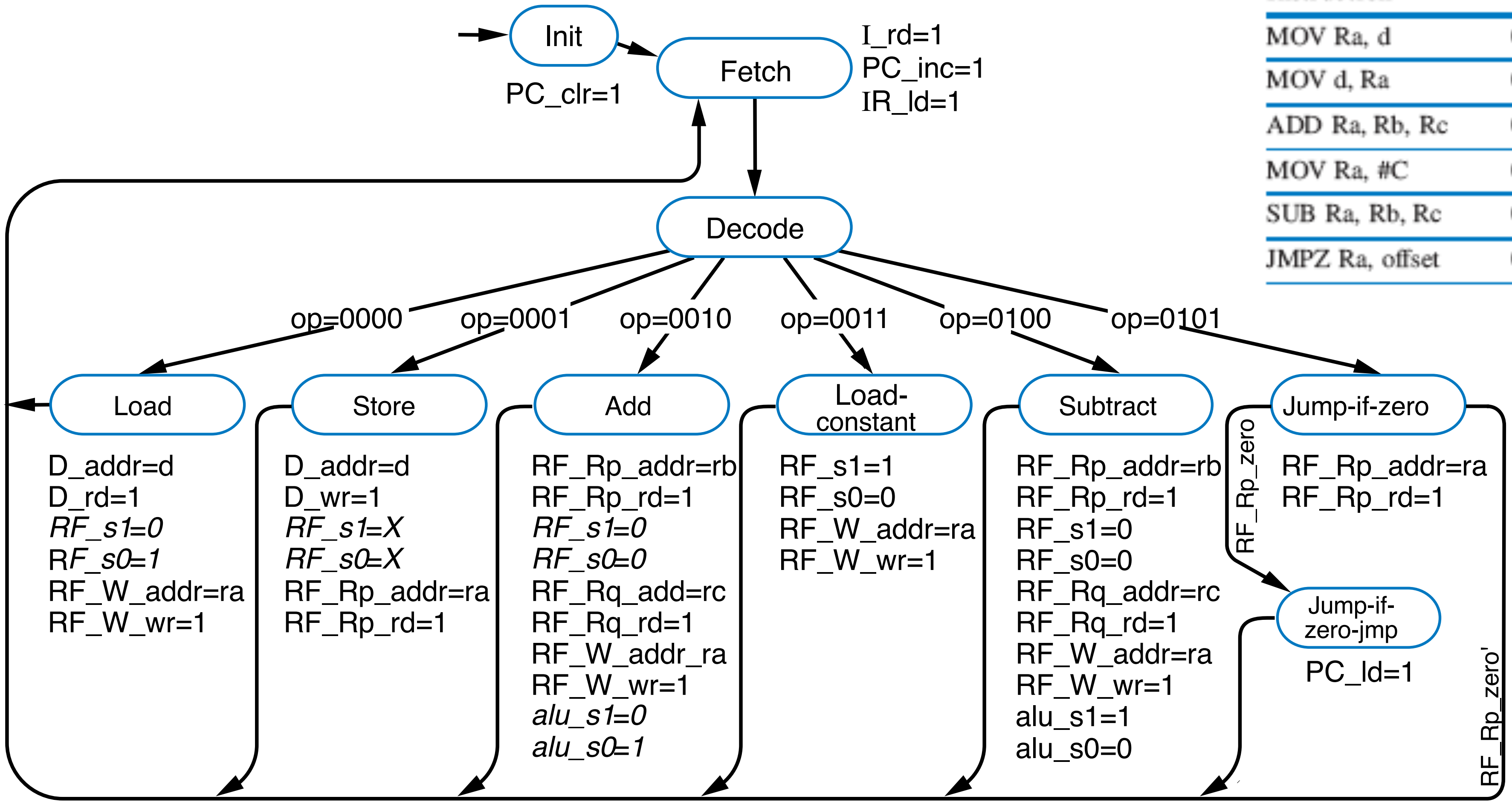


s1	s0	ALU operation
0	0	pass A through
0	1	A+B
1	0	A-B

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



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]

```
     MOV R0, #0; // initialize result to 0                      0011 0000 00000000
     MOV R1, #1; // constant 1 for incrementing result          0011 0001 00000001
     MOV R2, 4; // get data memory location 4                   0000 0010 00000100
     JMPZ R2, lab1; // if zero, skip next instruction           0101 0010 00000010
     ADD R0, R0, R1; // not zero, so increment result           0010 0000 0000 0001
lab1:MOV R2, 5; // get data memory location 5                   0000 0010 00000101
     JMPZ R2, lab2; // if zero, skip next instruction           0101 0010 00000010
     ADD R0, R0, R1; //not zero, so increment result            0010 0000 0000 0001
lab2:MOV 9, R0; // store result in data memory location 9       0001 0000 00001001
```

(**a**) (**b**)

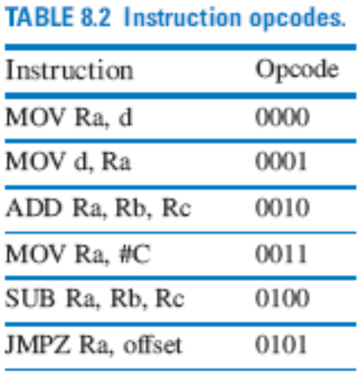
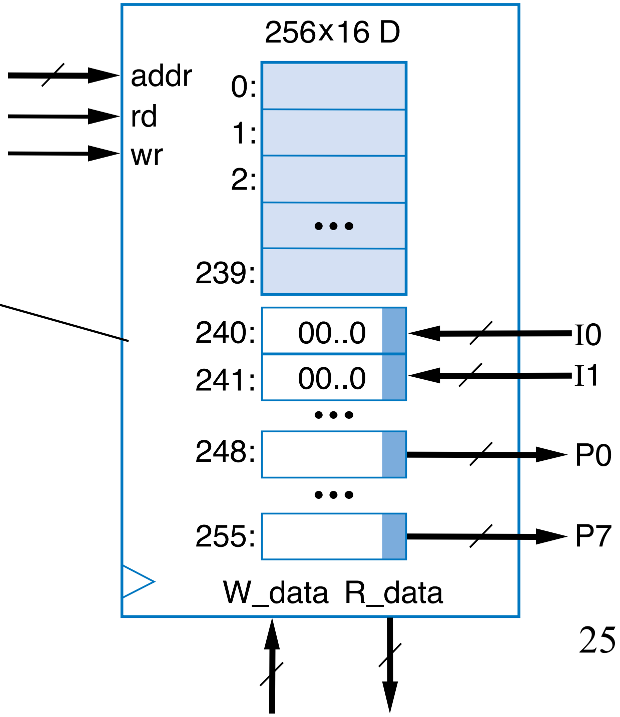


TABLE 8.2 Instruction opcodes.

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

Further Extensions to the Programmable Processor

- 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

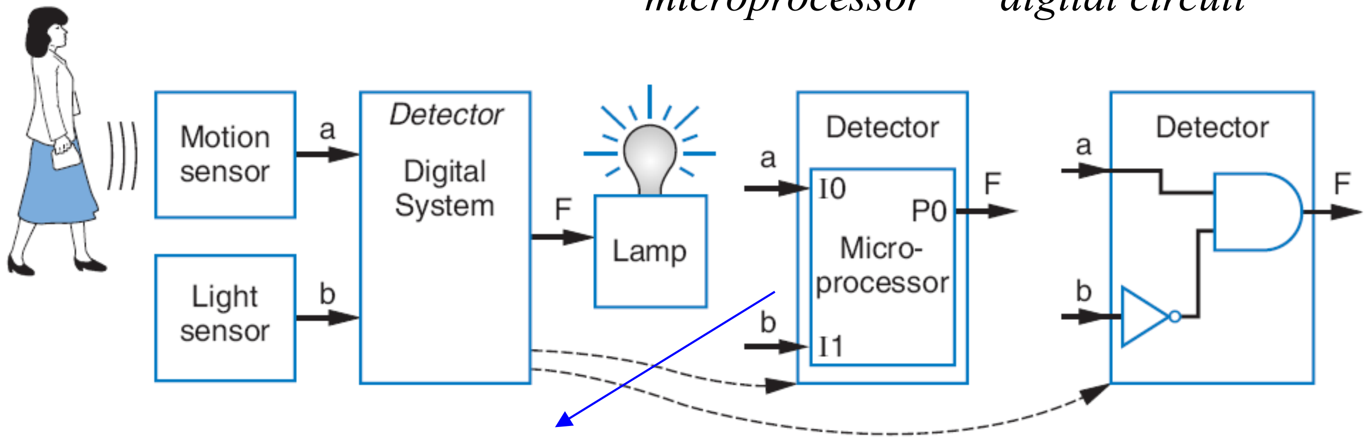


Program using I/O Extensions – Recall Chpt 1 C-Program Example

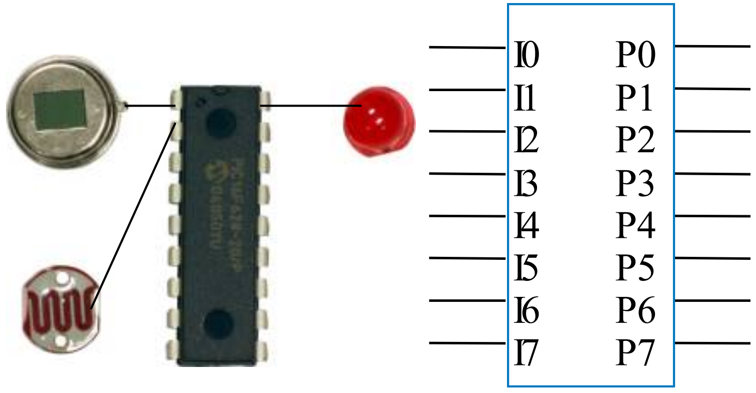
Desired motion-at-night detector

Programmed microprocessor

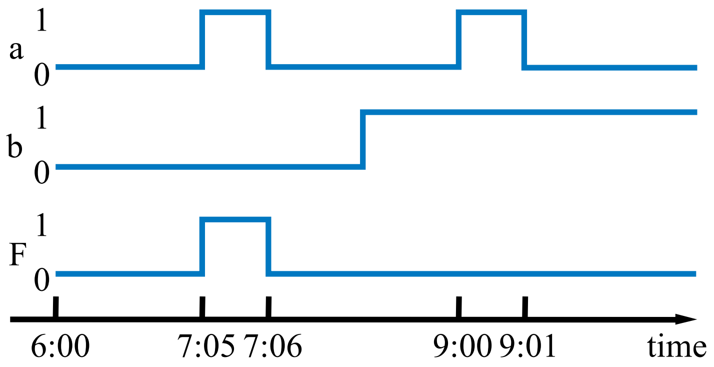
Custom designed digital circuit



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



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

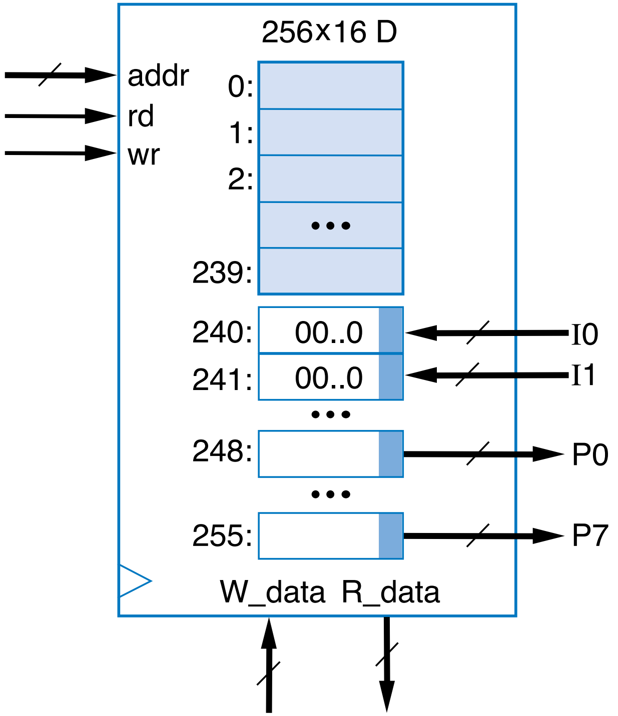


Program Using Input/Output Extensions

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