

Embedded Systems Design: A Unified Hardware/Software Introduction

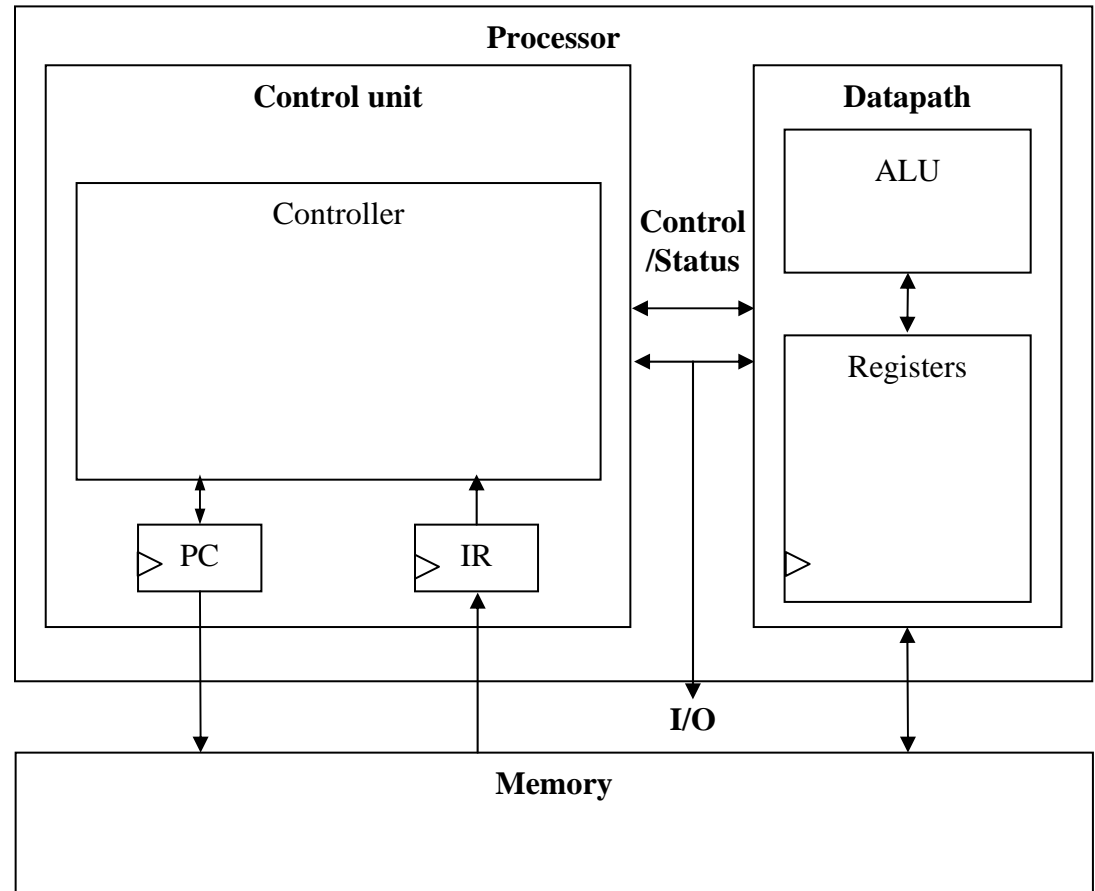
Chapter 3 General-Purpose Processors: Software

Introduction

- General-Purpose Processor
 - Processor designed for a variety of computation tasks
 - Low unit cost, in part because manufacturer spreads NRE over large numbers of units
 - Motorola sold half a billion 68HC05 microcontrollers *in 1996 alone*
 - Carefully designed since higher NRE is acceptable
 - Can yield good performance, size and power
 - Low NRE cost, short time-to-market/prototype, high flexibility
 - User just writes software; no processor design
 - a.k.a. “microprocessor” – “micro” used when they were implemented on one or a few chips rather than entire rooms

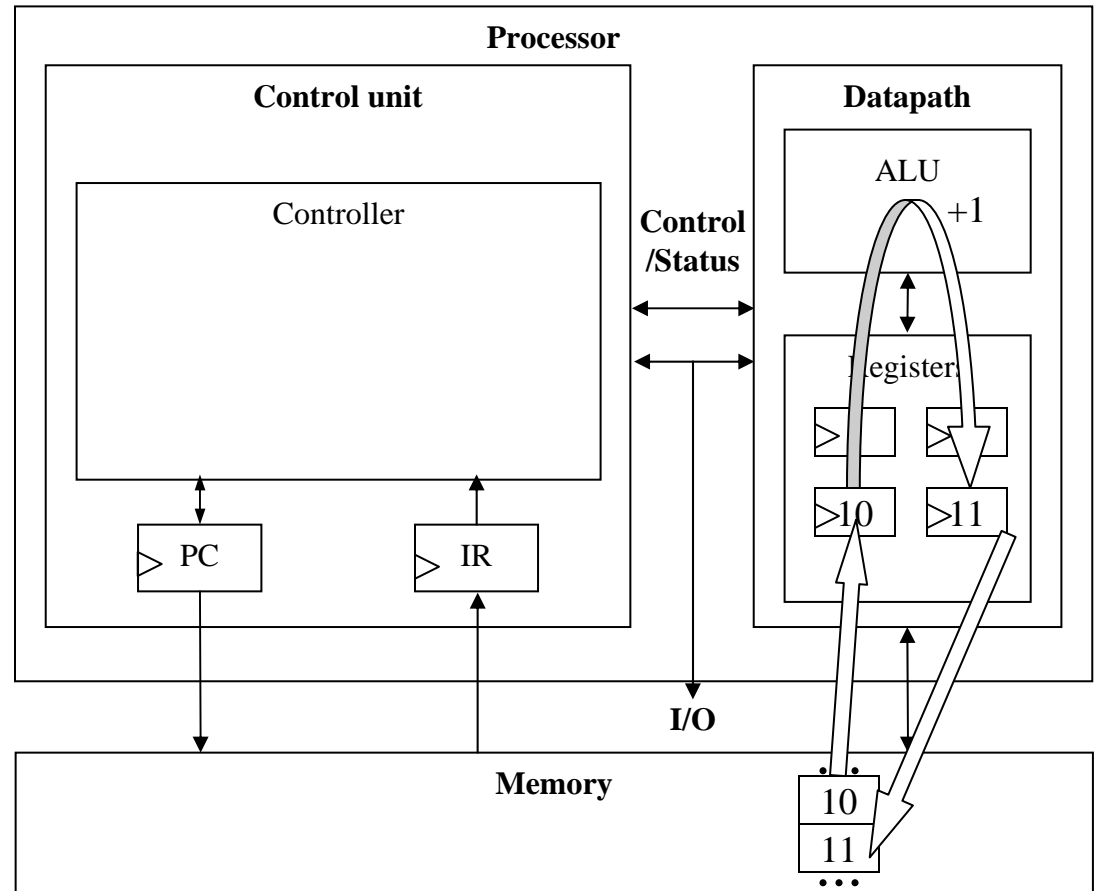
Basic Architecture

- Control unit and datapath
 - Note similarity to single-purpose processor
- Key differences
 - Datapath is general
 - Control unit doesn't store the algorithm – the algorithm is “programmed” into the memory



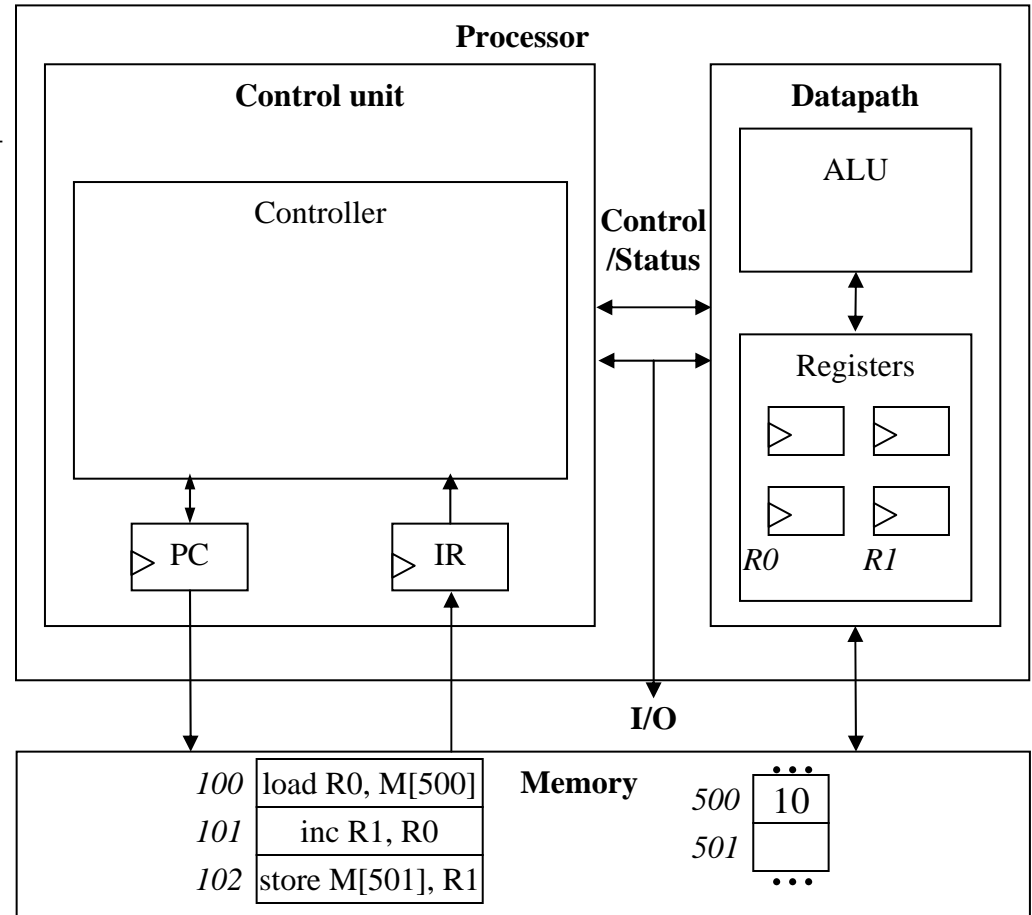
Datapath Operations

- Load
 - Read memory location into register
- ALU operation
 - Input certain registers through ALU, store back in register
- Store
 - Write register to memory location



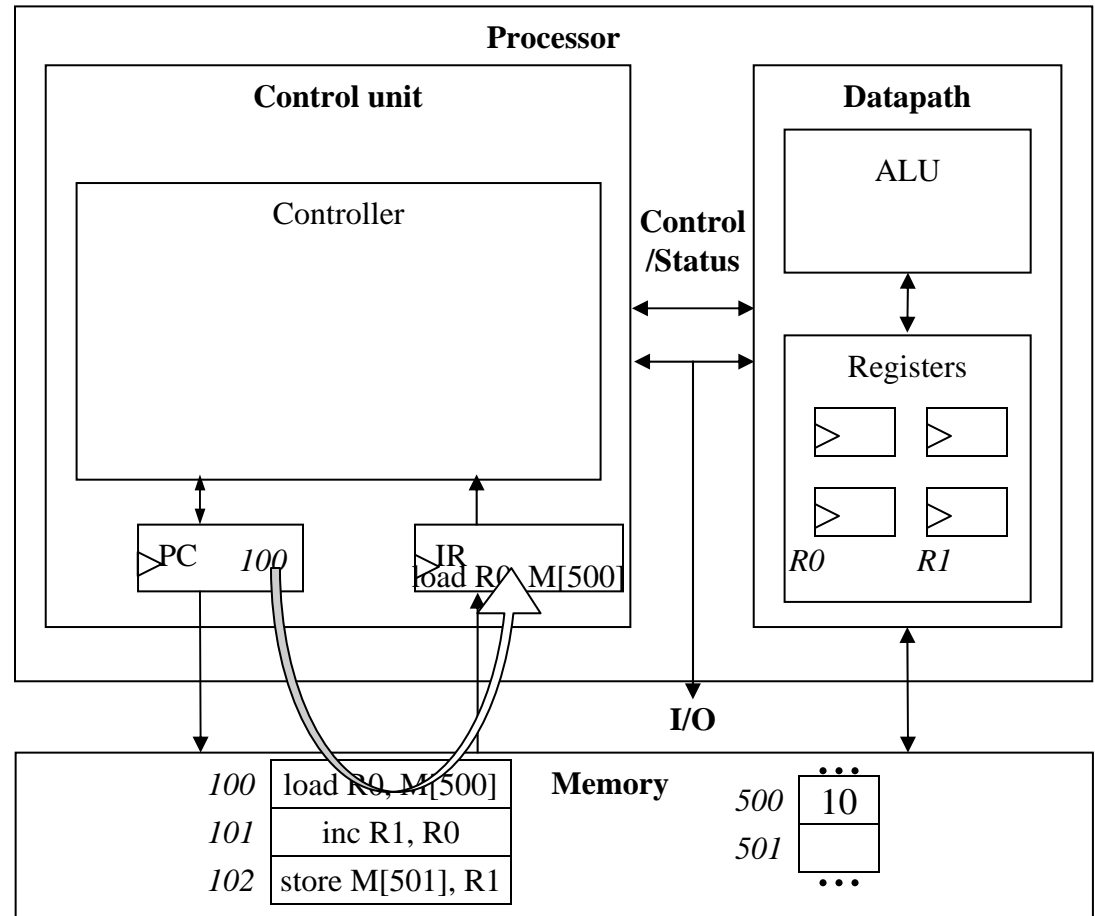
Control Unit

- Control unit: configures the datapath operations
 - Sequence of desired operations (“instructions”) stored in memory – “program”
- Instruction cycle – broken into several sub-operations, each one clock cycle, e.g.:
 - Fetch: Get next instruction into IR
 - Decode: Determine what the instruction means
 - Fetch operands: Move data from memory to datapath register
 - Execute: Move data through the ALU
 - Store results: Write data from register to memory



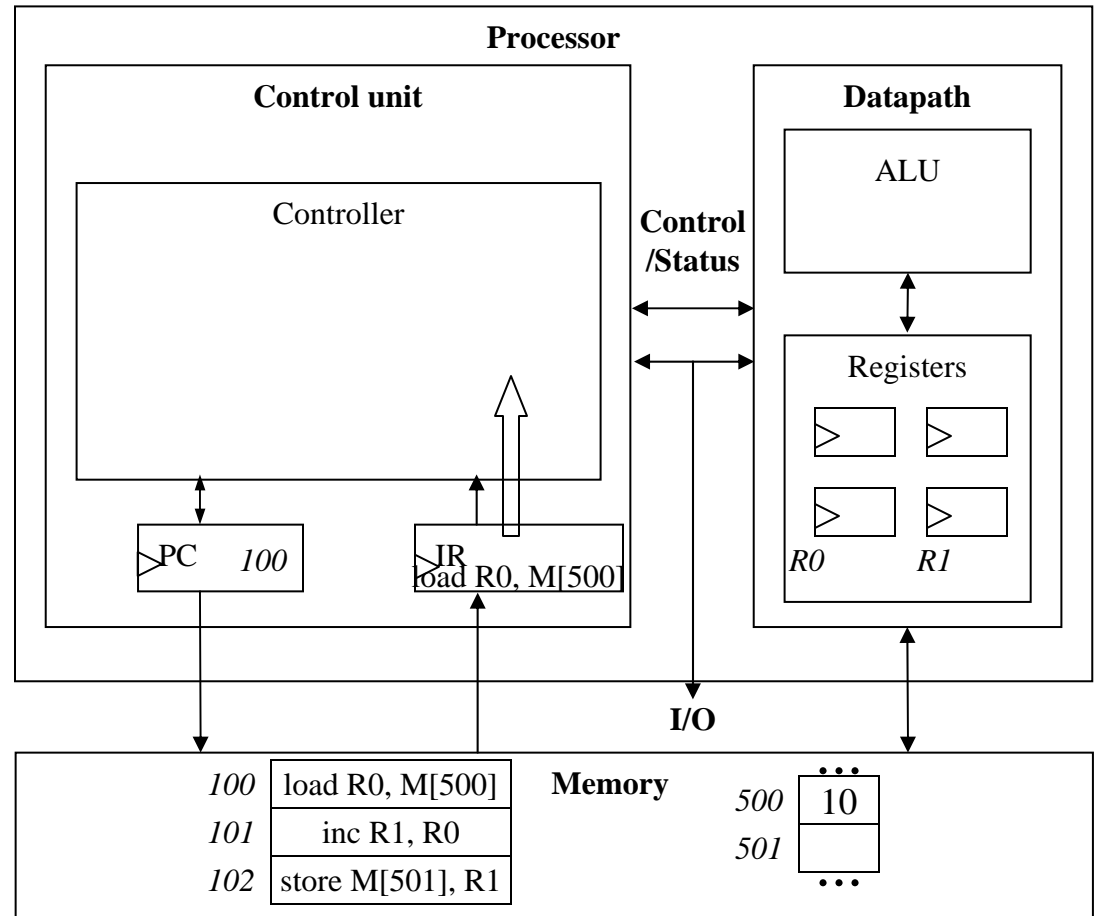
Control Unit Sub-Operations

- Fetch
 - Get next instruction into IR
 - PC: program counter, always points to next instruction
 - IR: holds the fetched instruction



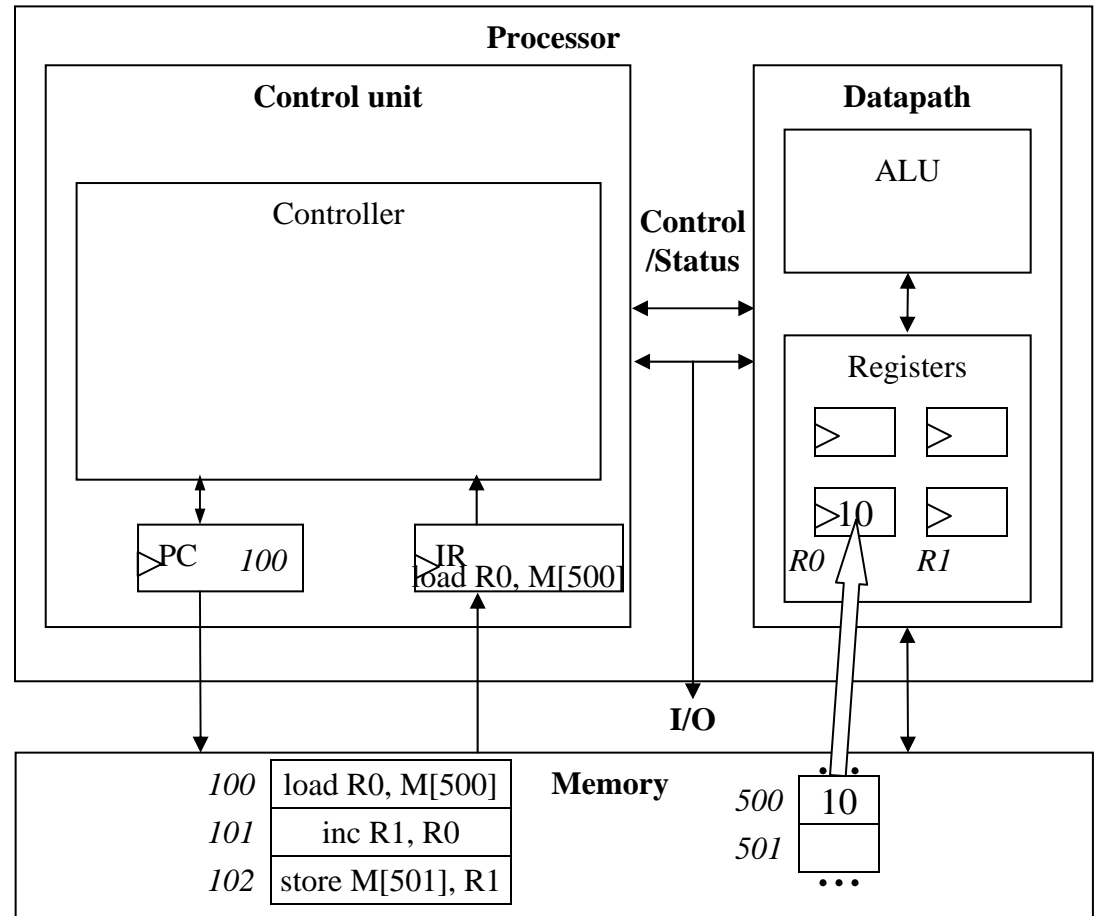
Control Unit Sub-Operations

- Decode
 - Determine what the instruction means



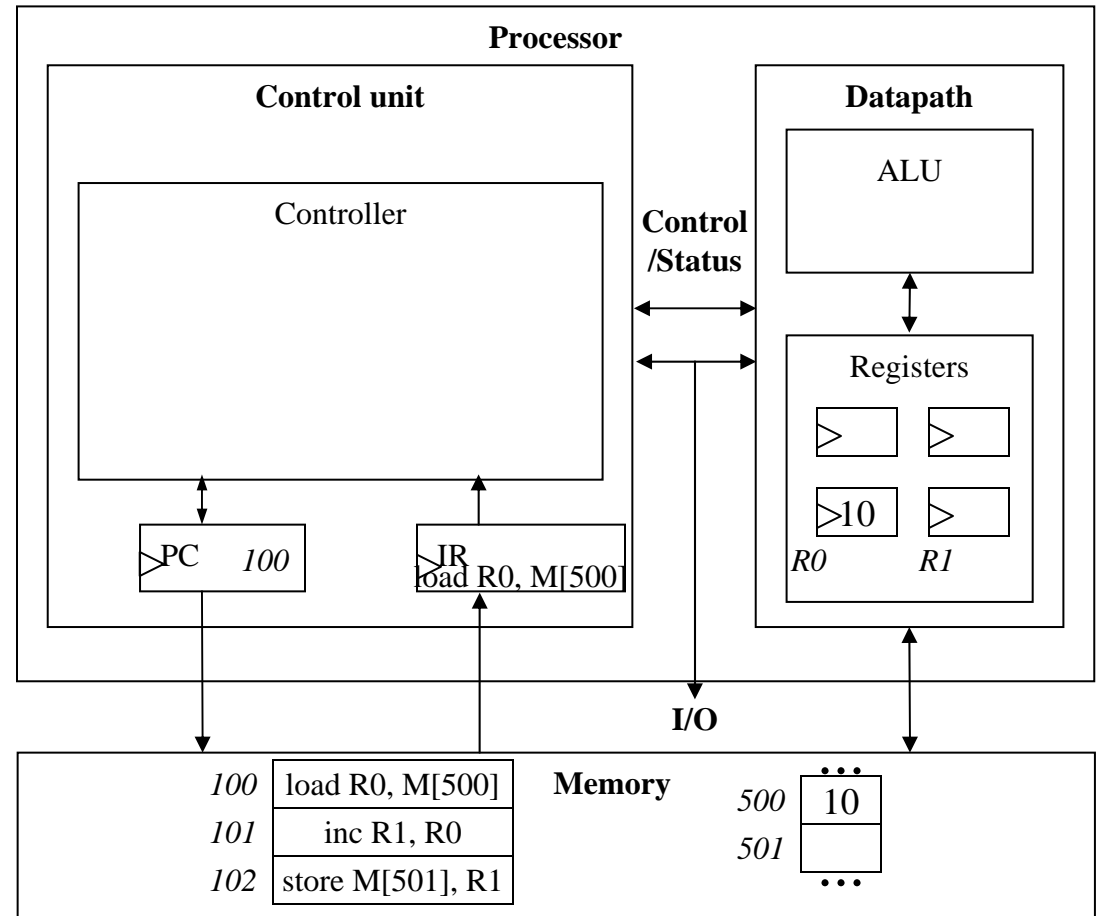
Control Unit Sub-Operations

- Fetch operands
 - Move data from memory to datapath register



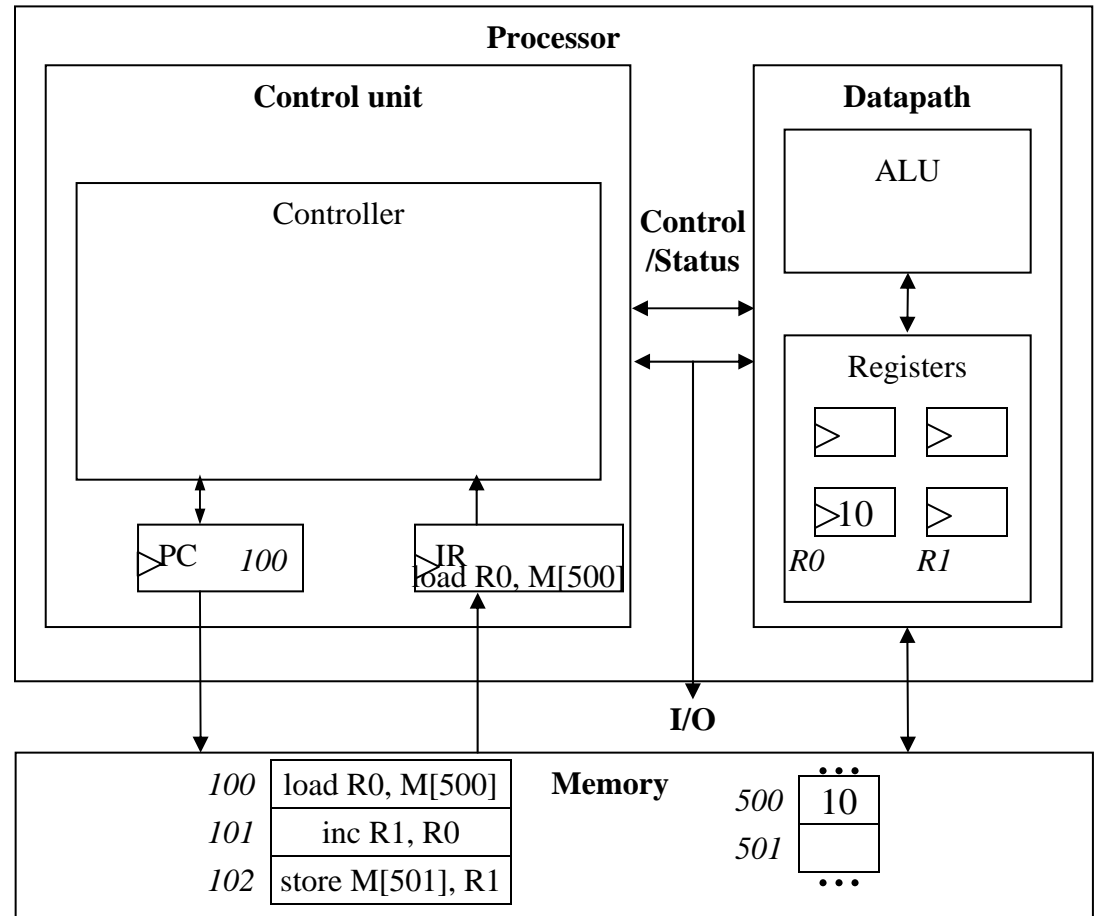
Control Unit Sub-Operations

- Execute
 - Move data through the ALU
 - This particular instruction does nothing during this sub-operation



Control Unit Sub-Operations

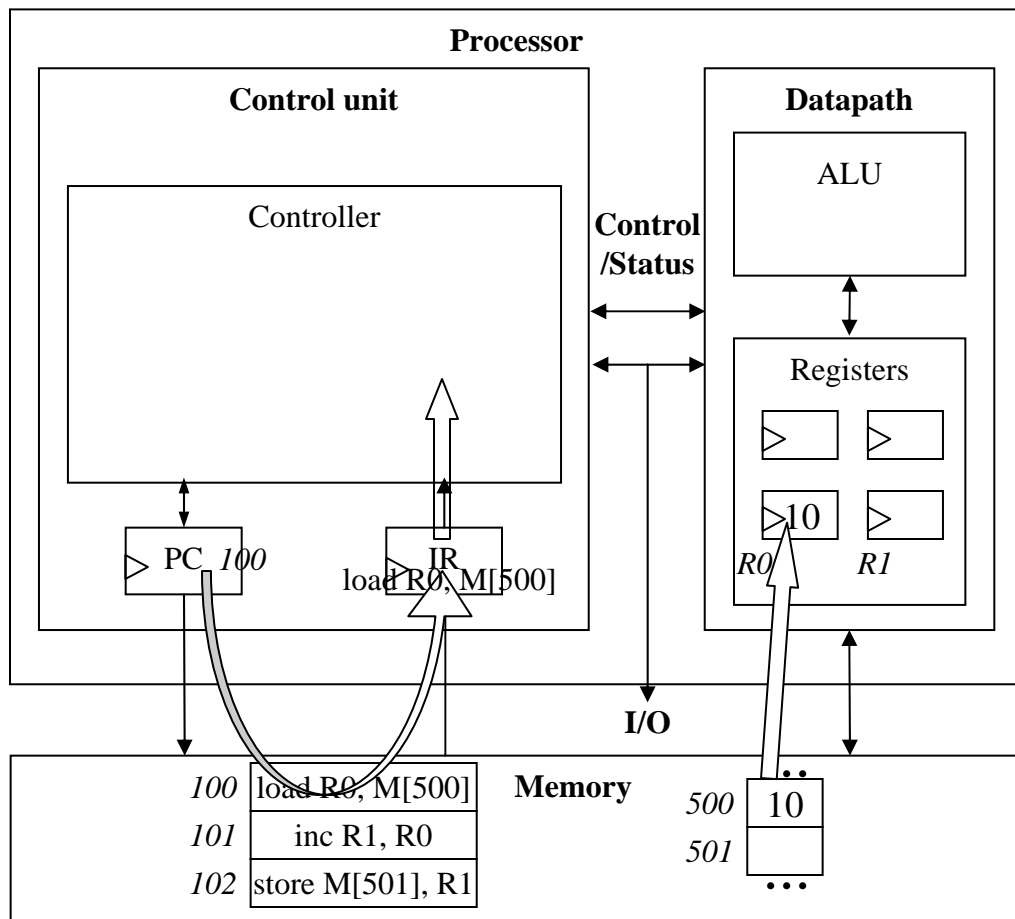
- Store results
 - Write data from register to memory
 - This particular instruction does nothing during this sub-operation



Instruction Cycles

PC=100

Fetch Decode Fetch ops Exec. Store results



Instruction Cycles

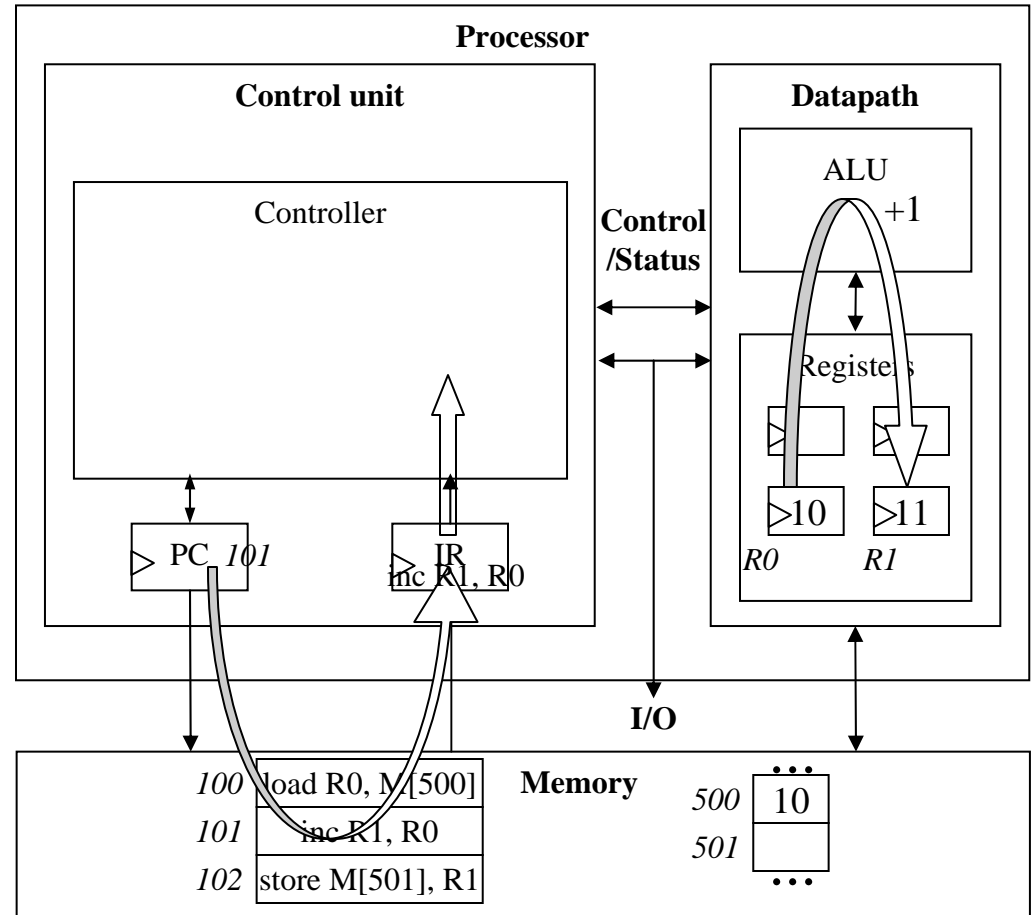
PC=100

Fetch Decode Fetch ops Exec. Store results

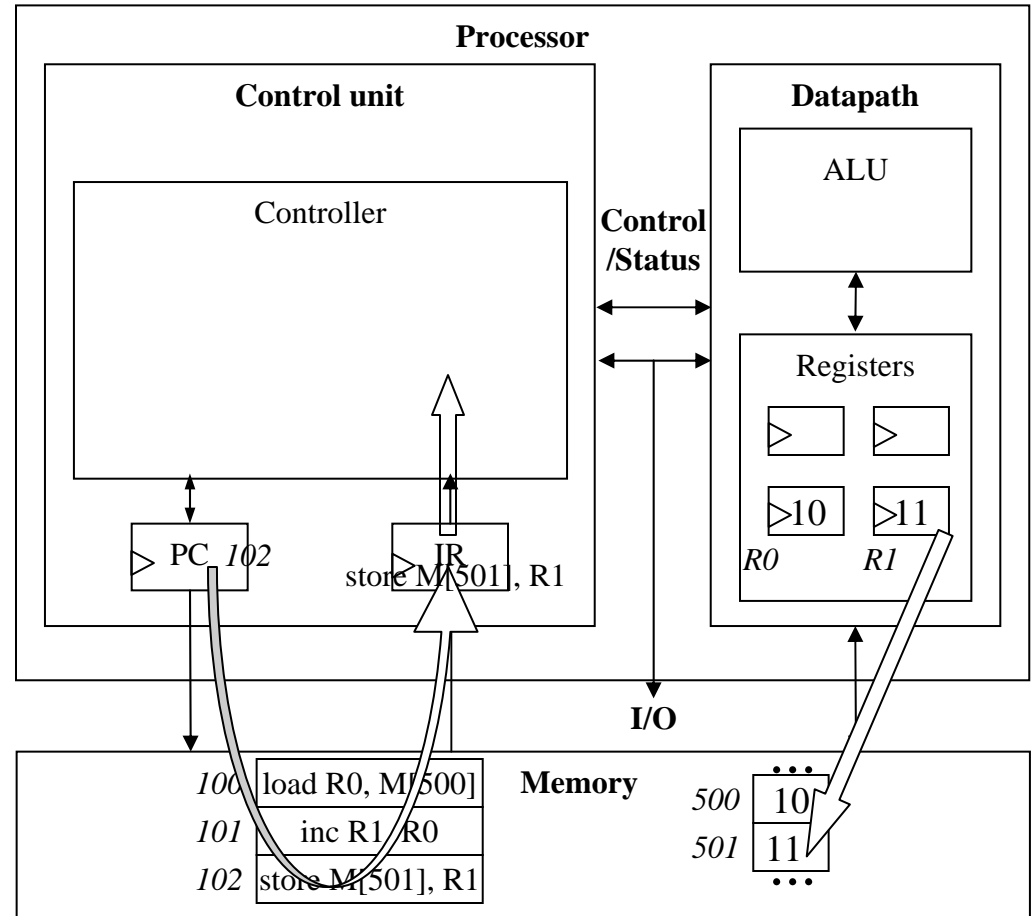
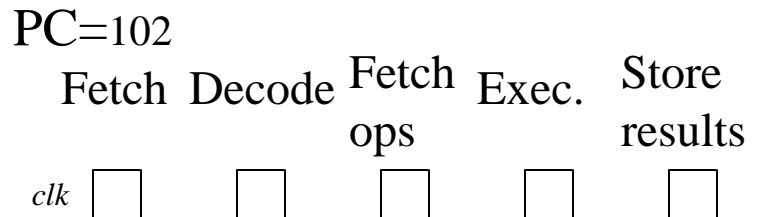
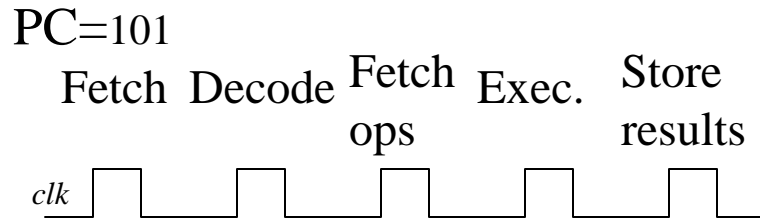
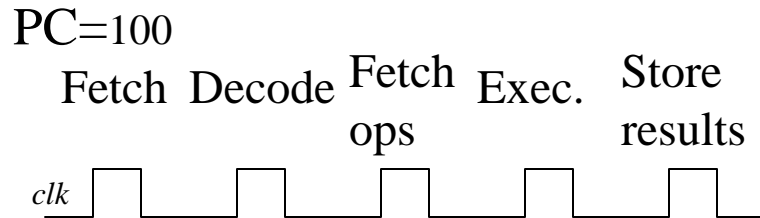


PC=101

Fetch Decode Fetch ops Exec. Store results

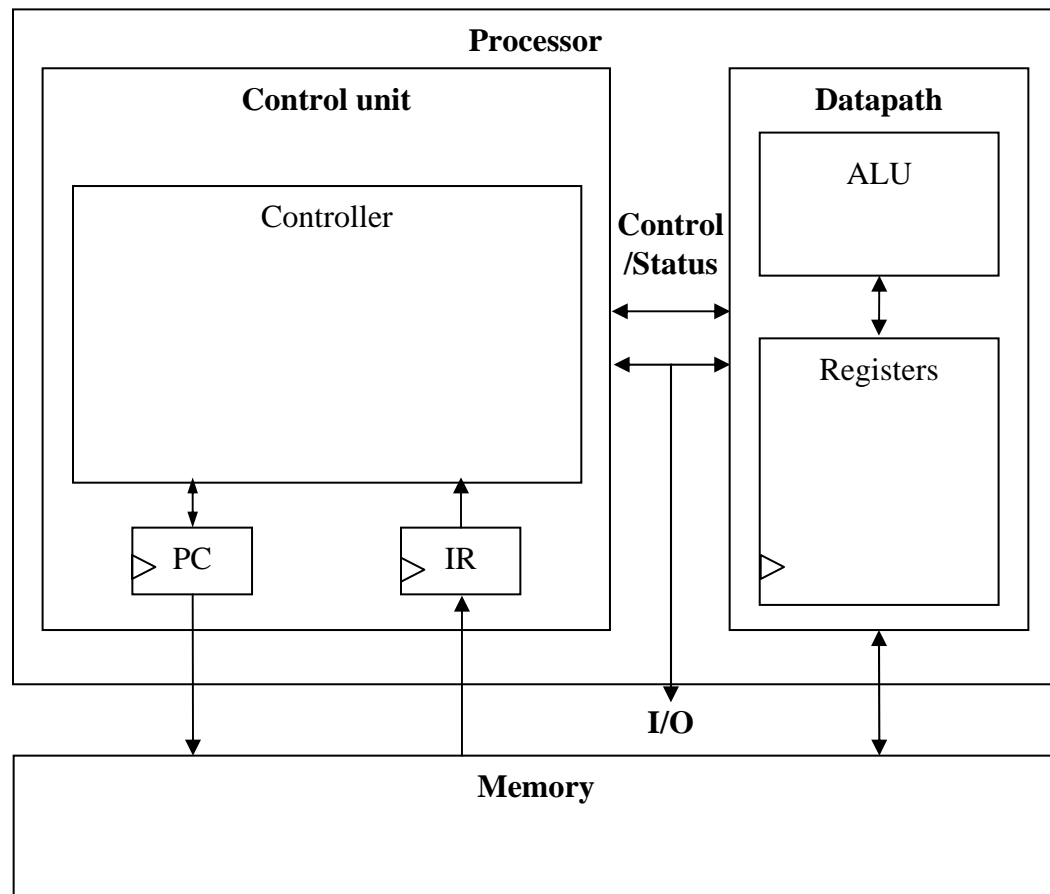


Instruction Cycles



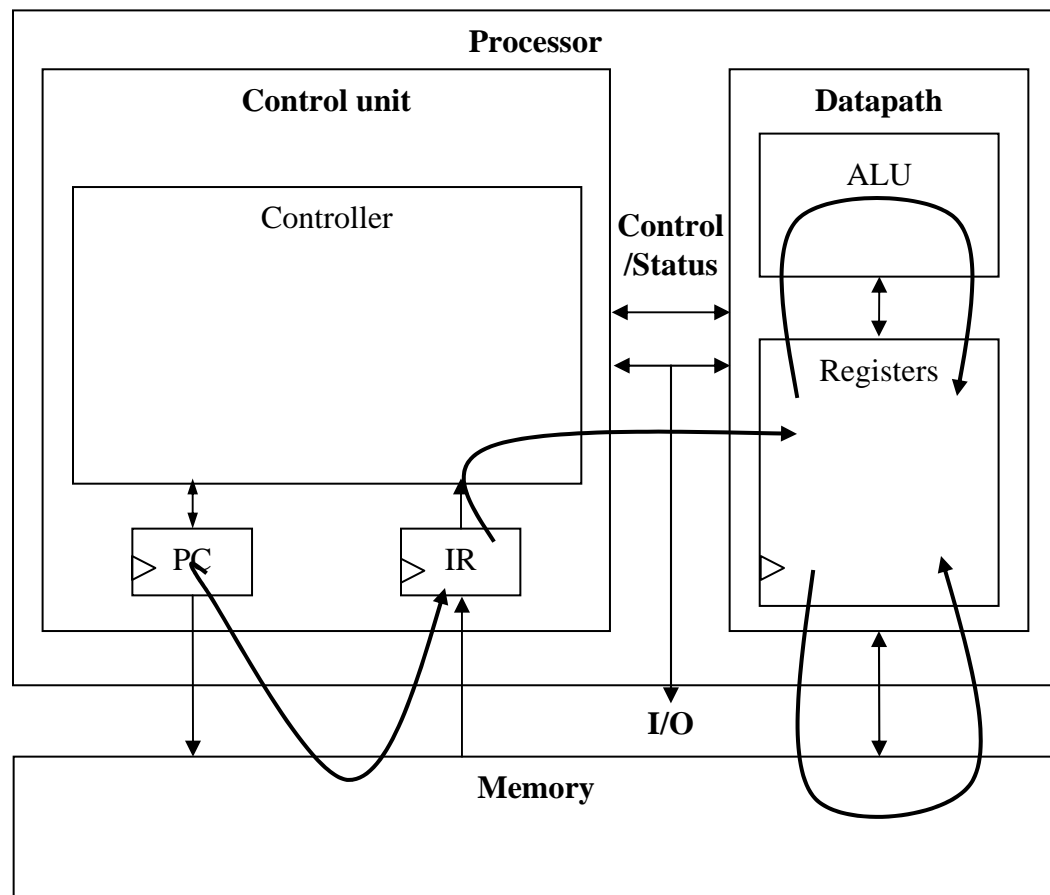
Architectural Considerations

- *N-bit* processor
 - N-bit ALU, registers, buses, memory data interface
 - Embedded: 8-bit, 16-bit, 32-bit common
 - Desktop/servers: 32-bit, even 64
- PC size determines address space

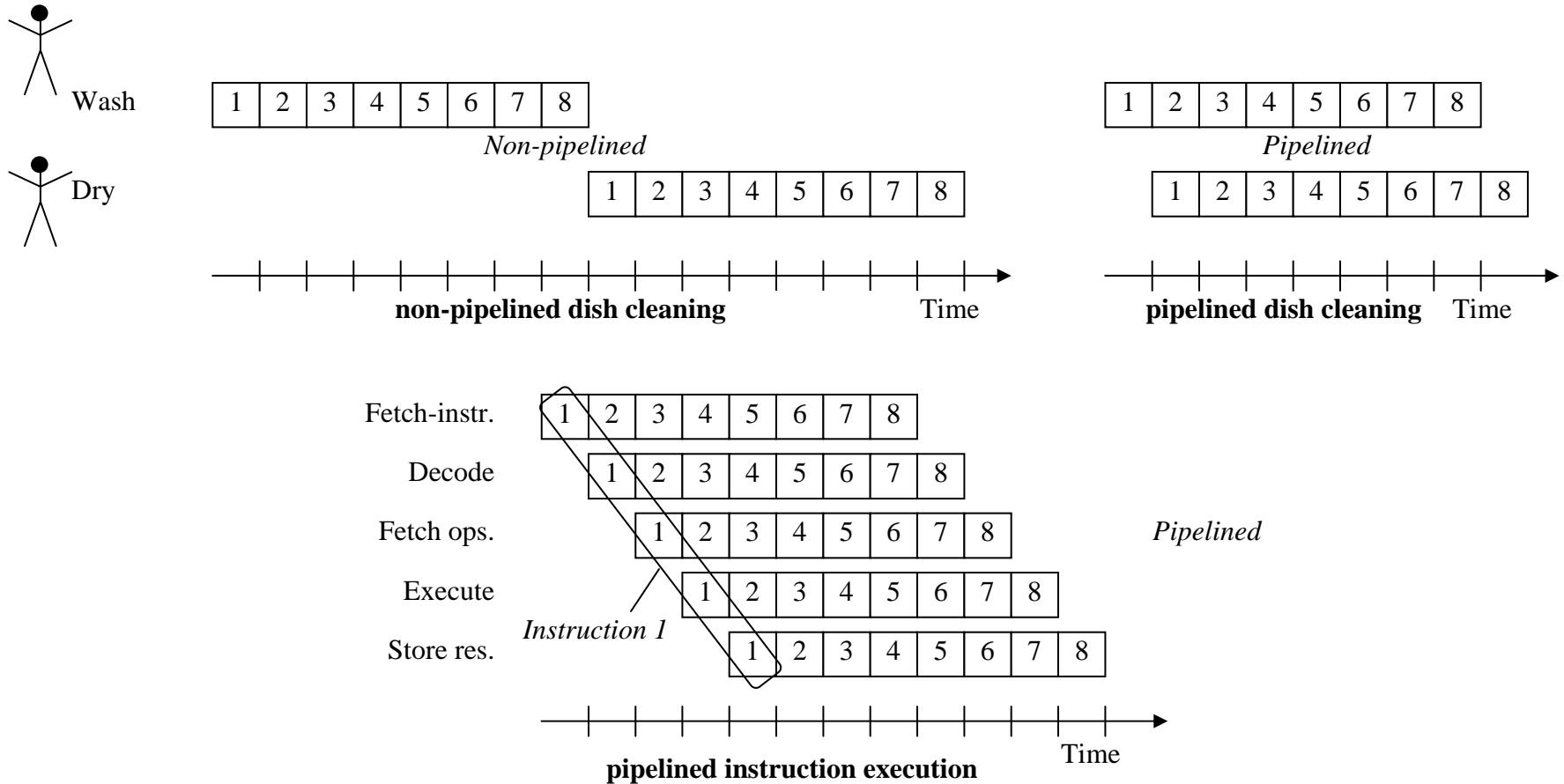


Architectural Considerations

- Clock frequency
 - Inverse of clock period
 - Must be longer than longest register to register delay in entire processor
 - Memory access is often the longest



Pipelining: Increasing Instruction Throughput

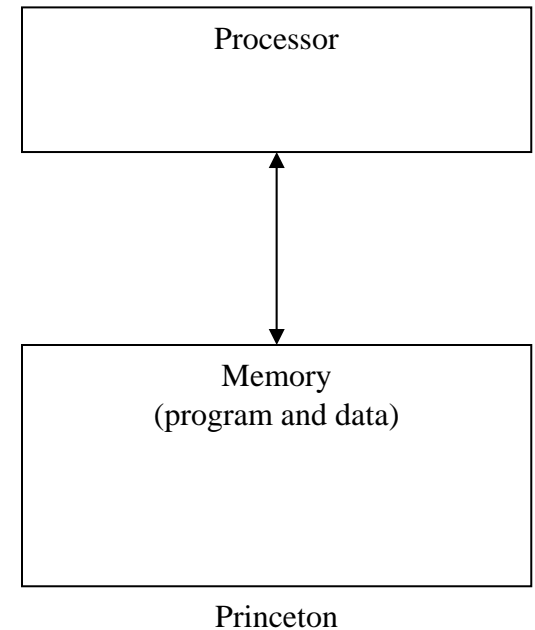
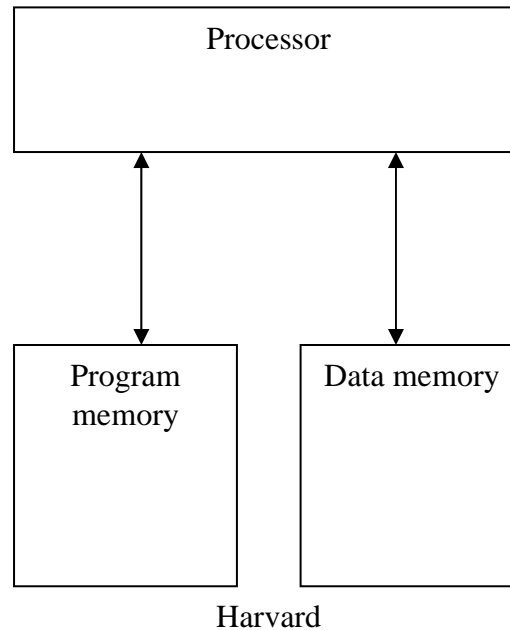


Superscalar and VLIW Architectures

- Performance can be improved by:
 - Faster clock (but there's a limit)
 - Pipelining: slice up instruction into stages, overlap stages
 - *Multiple ALUs* to support more than one instruction stream
 - Superscalar
 - Scalar: non-vector operations
 - Fetches instructions in batches, executes as many as possible
 - May require extensive hardware to detect independent instructions
 - VLIW: each word in memory has multiple independent instructions
 - Relies on the compiler to detect and schedule instructions
 - Currently growing in popularity

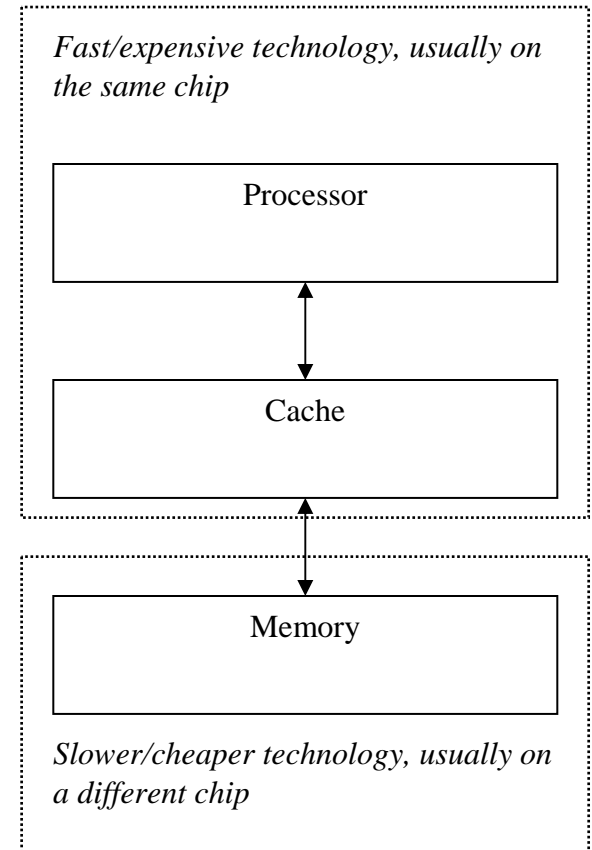
Two Memory Architectures

- Princeton
 - Fewer memory wires
- Harvard
 - Simultaneous program and data memory access



Cache Memory

- Memory access may be slow
- Cache is small but fast memory close to processor
 - Holds copy of part of memory
 - Hits and misses



Programmer's View

- Programmer doesn't need detailed understanding of architecture
 - Instead, needs to know what instructions can be executed
- Two levels of instructions:
 - Assembly level
 - Structured languages (C, C++, Java, etc.)
- Most development today done using structured languages
 - But, some assembly level programming may still be necessary
 - Drivers: portion of program that communicates with and/or controls (drives) another device
 - Often have detailed timing considerations, extensive bit manipulation
 - Assembly level may be best for these

Assembly-Level Instructions

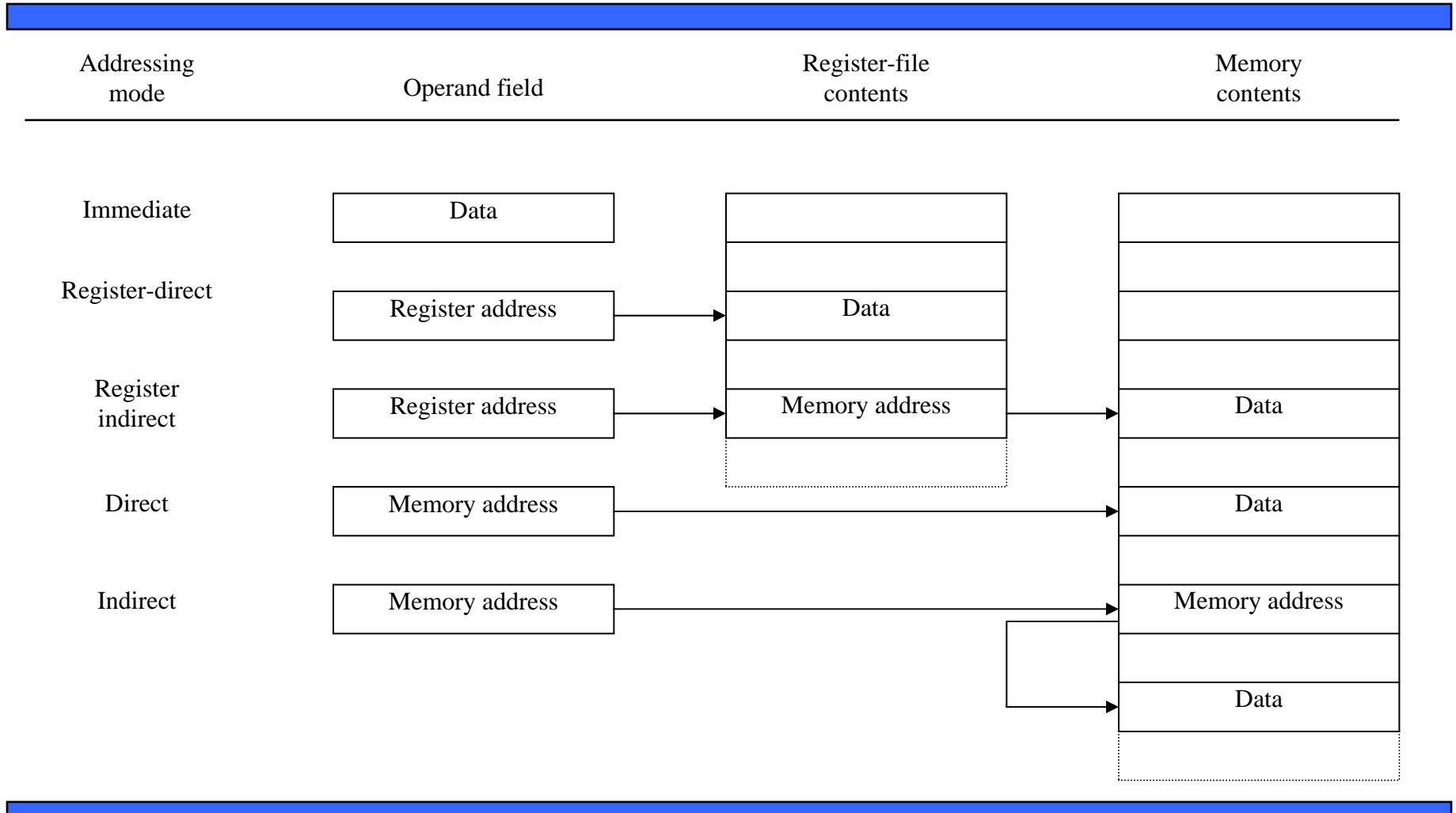
Instruction 1	opcode	operand1	operand2
Instruction 2	opcode	operand1	operand2
Instruction 3	opcode	operand1	operand2
Instruction 4	opcode	operand1	operand2
...			

- **Instruction Set**
 - Defines the legal set of instructions for that processor
 - Data transfer: memory/register, register/register, I/O, etc.
 - Arithmetic/logical: move register through ALU and back
 - Branches: determine next PC value when not just PC+1

A Simple (Trivial) Instruction Set

Assembly instruct.	First byte		Second byte	Operation
MOV Rn, direct	0000	Rn	direct	$Rn = M(\text{direct})$
MOV direct, Rn	0001	Rn	direct	$M(\text{direct}) = Rn$
MOV @Rn, Rm	0010	Rn	Rm	$M(Rn) = Rm$
MOV Rn, #immed.	0011	Rn	immediate	$Rn = \text{immediate}$
ADD Rn, Rm	0100	Rn	Rm	$Rn = Rn + Rm$
SUB Rn, Rm	0101	Rn	Rm	$Rn = Rn - Rm$
JZ Rn, relative	0110	Rn	relative	$PC = PC + \text{relative}$ (only if Rn is 0)
	opcode		operands	

Addressing Modes



Sample Programs

C program

```
int total = 0;
for (int i=10; i!=0; i--)
    total += i;
// next instructions...
```

Equivalent assembly program

```
0    MOV R0, #0;      // total = 0
1    MOV R1, #10;     // i = 10
2    MOV R2, #1;      // constant 1
3    MOV R3, #0;      // constant 0

Loop: JZ R1, Next;     // Done if i=0
5    ADD R0, R1;       // total += i
6    SUB R1, R2;       // i--
7    JZ R3, Loop;      // Jump always

Next: // next instructions...
```

- Try some others
 - Handshake: Wait until the value of M[254] is not 0, set M[255] to 1, wait until M[254] is 0, set M[255] to 0 (assume those locations are ports).
 - (Harder) Count the occurrences of zero in an array stored in memory locations 100 through 199.

Programmer Considerations

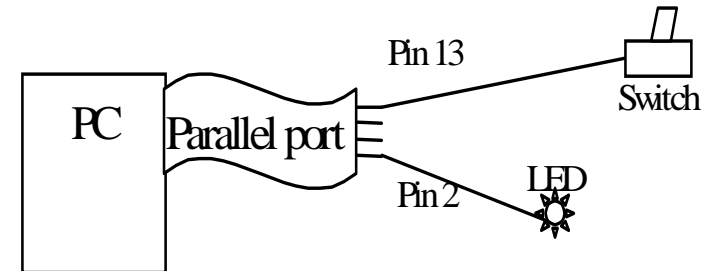
- Program and data memory space
 - Embedded processors often very limited
 - e.g., 64 Kbytes program, 256 bytes of RAM (expandable)
- Registers: How many are there?
 - Only a direct concern for assembly-level programmers
- I/O
 - How communicate with external signals?
- Interrupts

Microprocessor Architecture Overview

- If you are using a particular microprocessor, now is a good time to review its architecture

Example: parallel port driver

LPT Connection Pin	I/O Direction	Register Address
1	Output	0 th bit of register #2
2-9	Output	0 th bit of register #2
10,11,12,13,15	Input	6,7,5,4,3 th bit of register #1
14,16,17	Output	1,2,3 th bit of register #2



- Using assembly language programming we can configure a PC parallel port to perform digital I/O
 - write and read to three special registers to accomplish this table provides list of parallel port connector pins and corresponding register location
 - Example : parallel port monitors the input switch and turns the LED on/off accordingly

Parallel Port Example

```
; This program consists of a sub-routine that reads
; the state of the input pin, determining the on/off state
; of our switch and asserts the output pin, turning the LED
; on/off accordingly
.386
```

```
CheckPort      proc
    push  ax                ; save the content
    push  dx                ; save the content
    mov   dx, 3BCh + 1      ; base + 1 for register #1
    in    al, dx            ; read register #1
    and   al, 10h           ; mask out all but bit # 4
    cmp   al, 0             ; is it 0?
    jne   SwitchOn          ; if not, we need to turn the LED on
```

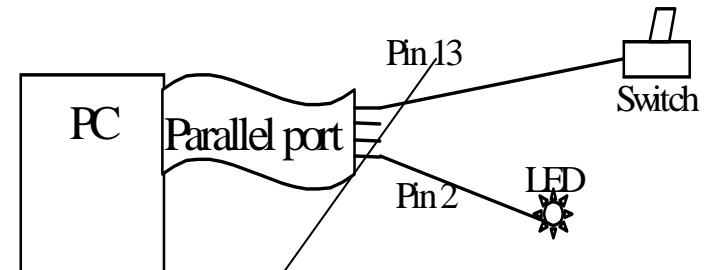
```
SwitchOff:
    mov   dx, 3BCh + 0      ; base + 0 for register #0
    in    al, dx            ; read the current state of the port
    and   al, f7h           ; clear first bit (masking)
    out   dx, al            ; write it out to the port
    jmp   Done              ; we are done
```

```
SwitchOn:
    mov   dx, 3BCh + 0      ; base + 0 for register #0
    in    al, dx            ; read the current state of the port
    or    al, 01h          ; set first bit (masking)
    out   dx, al            ; write it out to the port
```

```
Done:  pop  dx              ; restore the content
       pop  ax              ; restore the content
CheckPort      endp
```

```
extern "C" CheckPort(void);    // defined in
                                // assembly

void main(void) {
    while( 1 ) {
        CheckPort();
    }
}
```



LPT Connection Pin	I/O Direction	Register Address
1	Output	0 th bit of register #2
2-9	Output	0 th bit of register #2
10,11,12,13,15	Input	6,7,5,4,3 th bit of register #1
14,16,17	Output	1,2,3 th bit of register #2

Operating System

- Optional software layer providing low-level services to a program (application).
 - File management, disk access
 - Keyboard/display interfacing
 - Scheduling multiple programs for execution
 - Or even just multiple threads from one program
 - Program makes system calls to the OS

```
DB file_name "out.txt" -- store file name

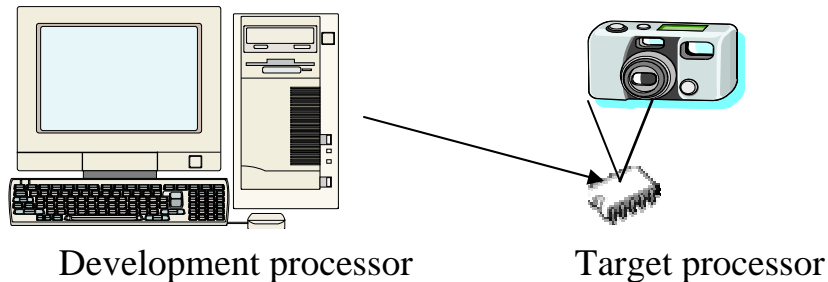
MOV R0, 1324           -- system call "open" id
MOV R1, file_name      -- address of file-name
INT 34                 -- cause a system call
JZ  R0, L1              -- if zero -> error

    . . . read the file
JMP L2                  -- bypass error cond.
L1:
    . . . handle the error

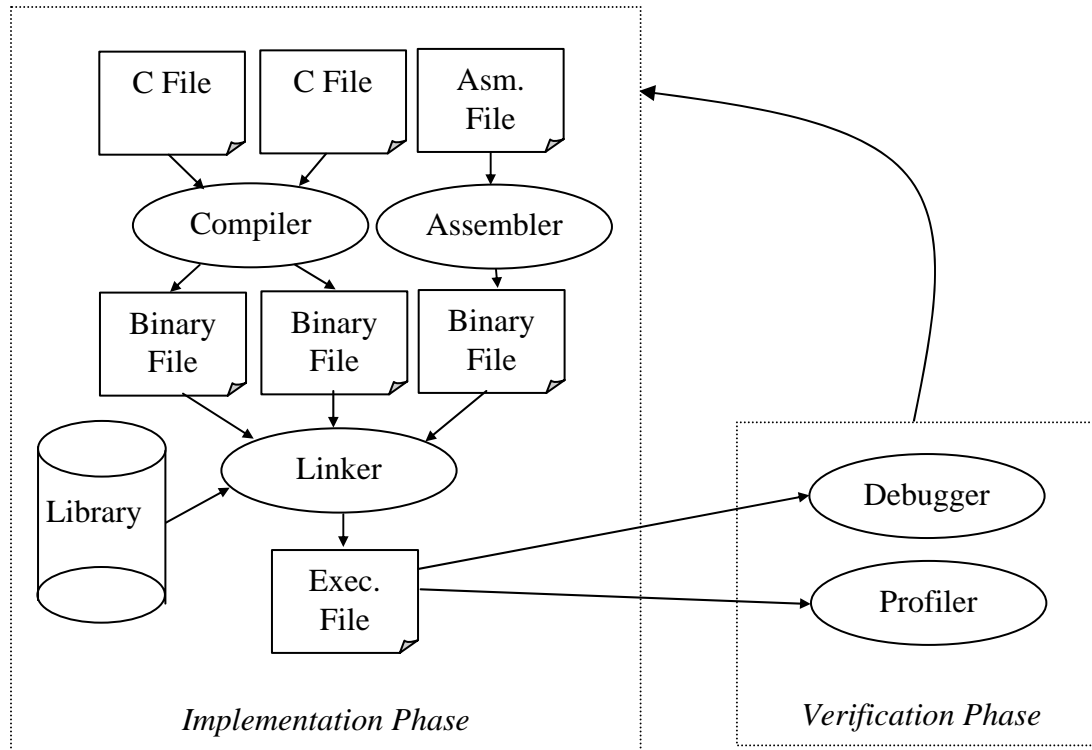
L2:
```

Development Environment

- Development processor
 - The processor on which we write and debug our programs
 - Usually a PC
- *Target processor*
 - The processor that the program will run on in our embedded system
 - Often different from the development processor



Software Development Process



- Compilers
 - Cross compiler
 - Runs on one processor, but generates code for another
- Assemblers
- Linkers
- Debuggers
- Profilers

Running a Program

- If development processor is different than target, how can we run our compiled code? Two options:
 - Download to target processor
 - Simulate
- Simulation
 - One method: Hardware description language
 - But slow, not always available
 - Another method: *Instruction set simulator (ISS)*
 - Runs on development processor, but executes instructions of target processor

Instruction Set Simulator For A Simple Processor

```
#include <stdio.h>
typedef struct {
    unsigned char first_byte, second_byte;
} instruction;

instruction program[1024]; //instruction memory
unsigned char memory[256]; //data memory

void run_program(int num_bytes) {

    int pc = -1;
    unsigned char reg[16], fb, sb;

    while( ++pc < (num_bytes / 2) ) {
        fb = program[pc].first_byte;
        sb = program[pc].second_byte;
        switch( fb >> 4 ) {
            case 0: reg[fb & 0x0f] = memory[sb]; break;
            case 1: memory[sb] = reg[fb & 0x0f]; break;
            case 2: memory[reg[fb & 0x0f]] =
                reg[sb >> 4]; break;
            case 3: reg[fb & 0x0f] = sb; break;
            case 4: reg[fb & 0x0f] += reg[sb >> 4]; break;
            case 5: reg[fb & 0x0f] -= reg[sb >> 4]; break;
            case 6: pc += sb; break;
            default: return -1;
        }
    }
}
```

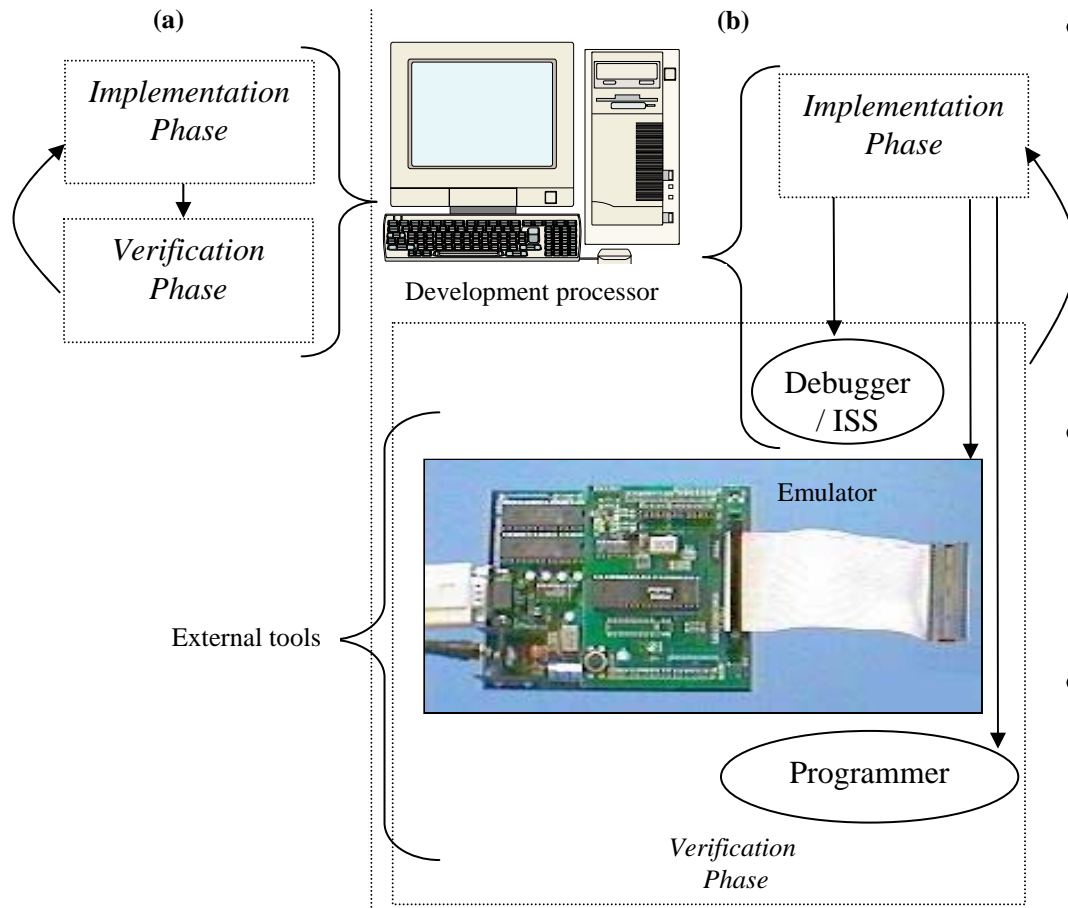
```
}
}
return 0;
}

int main(int argc, char *argv[]) {

    FILE* ifs;

    If( argc != 2 ||
        (ifs = fopen(argv[1], "rb") == NULL ) {
        return -1;
    }
    if (run_program(fread(program,
        sizeof(program) == 0) {
        print_memory_contents();
        return(0);
    }
    else return(-1);
}
```

Testing and Debugging



- ISS
 - Gives us control over time – set breakpoints, look at register values, set values, step-by-step execution, ...
 - But, doesn't interact with real environment
- Download to board
 - Use device programmer
 - Runs in real environment, but not controllable
- Compromise: emulator
 - Runs in real environment, at speed or near
 - Supports some controllability from the PC

Application-Specific Instruction-Set Processors (ASIPs)

- General-purpose processors
 - Sometimes too general to be effective in demanding application
 - e.g., video processing – requires huge video buffers and operations on large arrays of data, inefficient on a GPP
 - But single-purpose processor has high NRE, not programmable
- ASIPs – targeted to a particular domain
 - Contain architectural features specific to that domain
 - e.g., embedded control, digital signal processing, video processing, network processing, telecommunications, etc.
 - Still programmable

A Common ASIP: Microcontroller

- For embedded control applications
 - Reading sensors, setting actuators
 - Mostly dealing with events (bits): data is present, but not in huge amounts
 - e.g., VCR, disk drive, digital camera (assuming SPP for image compression), washing machine, microwave oven
 - Microcontroller features
 - On-chip peripherals
 - Timers, analog-digital converters, serial communication, etc.
 - Tightly integrated for programmer, typically part of register space
 - On-chip program and data memory
 - Direct programmer access to many of the chip's pins
 - Specialized instructions for bit-manipulation and other low-level operations
-

Another Common ASIP: Digital Signal Processors (DSP)

- For signal processing applications
 - Large amounts of digitized data, often streaming
 - Data transformations must be applied fast
 - e.g., cell-phone voice filter, digital TV, music synthesizer
- DSP features
 - Several instruction execution units
 - Multiple-accumulate single-cycle instruction, other instrs.
 - Efficient vector operations – e.g., add two arrays
 - Vector ALUs, loop buffers, etc.

Trend: Even More Customized ASIPs

- In the past, microprocessors were acquired as chips
- Today, we increasingly acquire a processor as Intellectual Property (IP)
 - e.g., synthesizable VHDL model
- Opportunity to add a custom datapath hardware and a few custom instructions, or delete a few instructions
 - Can have significant performance, power and size impacts
 - Problem: need compiler/debugger for customized ASIP
 - Remember, most development uses structured languages
 - One solution: automatic compiler/debugger generation
 - e.g., www.tensillica.com
 - Another solution: retargettable compilers
 - e.g., www.improvsys.com (customized VLIW architectures)

Selecting a Microprocessor

- Issues
 - Technical: speed, power, size, cost
 - Other: development environment, prior expertise, licensing, etc.
- Speed: how evaluate a processor's speed?
 - Clock speed – but instructions per cycle may differ
 - Instructions per second – but work per instr. may differ
 - Dhrystone: Synthetic benchmark, developed in 1984. Dhrystones/sec.
 - MIPS: 1 MIPS = 1757 Dhrystones per second (based on Digital's VAX 11/780). A.k.a. Dhrystone MIPS. Commonly used today.
 - So, 750 MIPS = $750 \times 1757 = 1,317,750$ Dhrystones per second
 - SPEC: set of more realistic benchmarks, but oriented to desktops
 - EEMBC – EDN Embedded Benchmark Consortium, www.eembc.org
 - Suites of benchmarks: automotive, consumer electronics, networking, office automation, telecommunications

General Purpose Processors

Processor	Clock speed	Periph.	Bus Width	MIPS	Power	Trans.	Price
General Purpose Processors							
Intel PIII	1GHz	2x16 K L1, 256K L2, MMX	32	~900	97W	~7M	\$900
IBM PowerPC 750X	550 MHz	2x32 K L1, 256K L2	32/64	~1300	5W	~7M	\$900
MIPS R5000	250 MHz	2x32 K 2 way set assoc.	32/64	NA	NA	3.6M	NA
StrongARM SA-110	233 MHz	None	32	268	1W	2.1M	NA
Microcontroller							
Intel 8051	12 MHz	4K ROM, 128 RAM, 32 I/O, Timer, UART	8	~1	~0.2W	~10K	\$7
Motorola 68HC811	3 MHz	4K ROM, 192 RAM, 32 I/O, Timer, WDT, SPI	8	~.5	~0.1W	~10K	\$5
Digital Signal Processors							
TI C5416	160 MHz	128K, SRAM, 3 T1 Ports, DMA, 13 ADC, 9 DAC	16/32	~600	NA	NA	\$34
Lucent DSP32C	80 MHz	16K Inst., 2K Data, Serial Ports, DMA	32	40	NA	NA	\$75

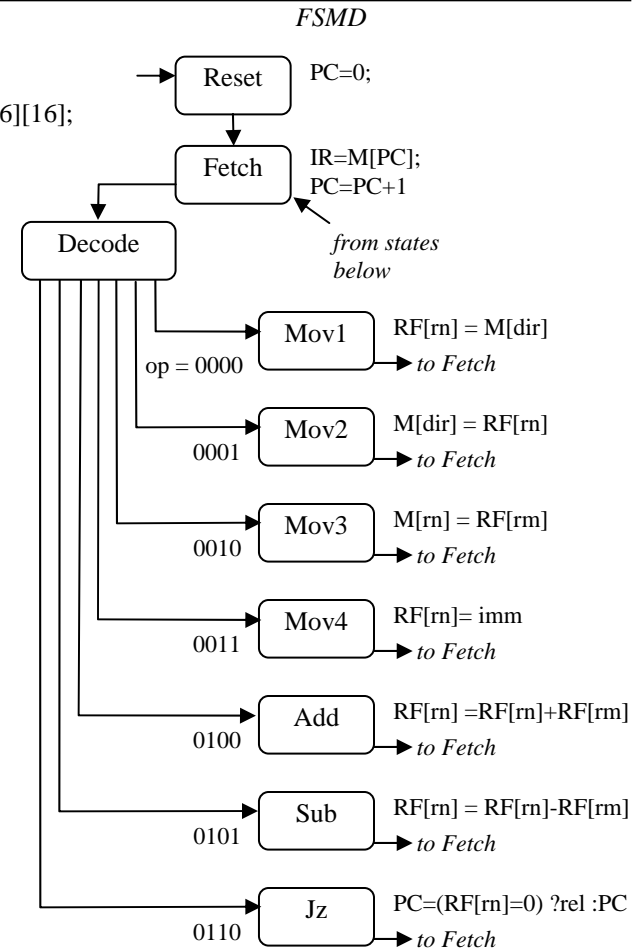
Sources: Intel, Motorola, MIPS, ARM, TI, and IBM Website/Datasheet; Embedded Systems Programming, Nov. 1998

Designing a General Purpose Processor

- Not something an embedded system designer normally would do
 - But instructive to see how simply we can build one top down
 - Remember that real processors aren't usually built this way
 - Much more optimized, much more bottom-up design

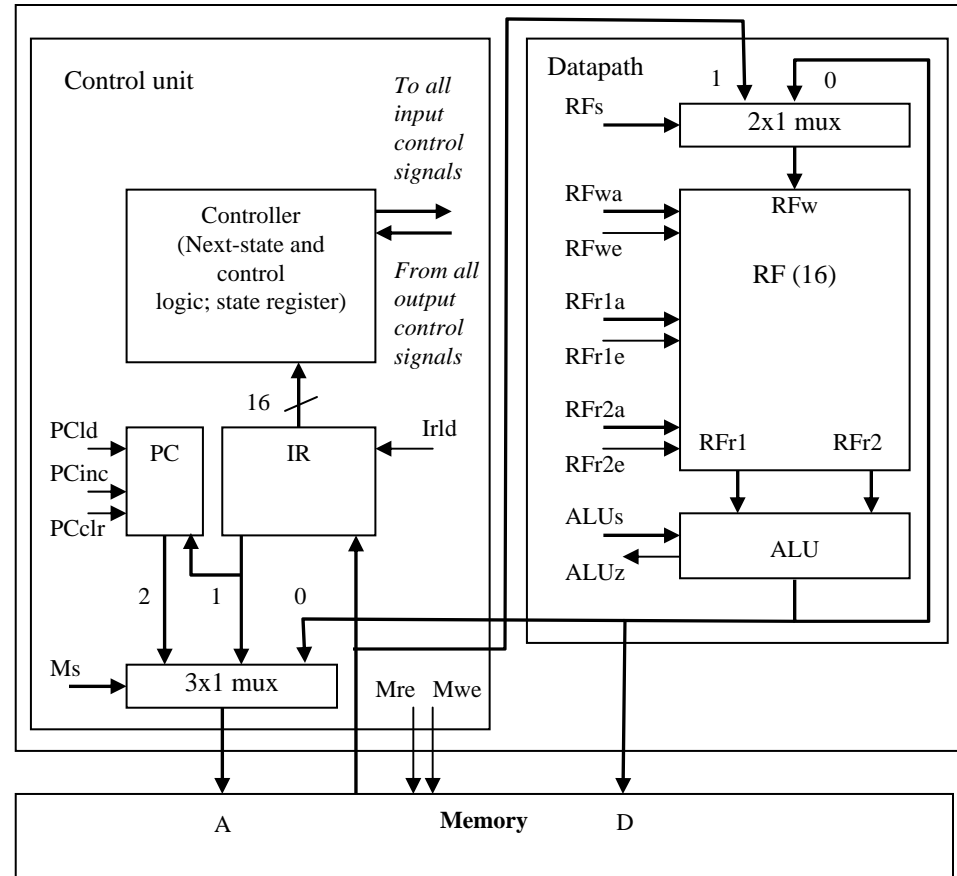
Declarations:
 bit PC[16], IR[16];
 bit M[64k][16], RF[16][16];

Aliases:			
op	IR[15..12]	dir	IR[7..0]
rn	IR[11..8]	imm	IR[7..0]
rm	IR[7..4]	rel	IR[7..0]

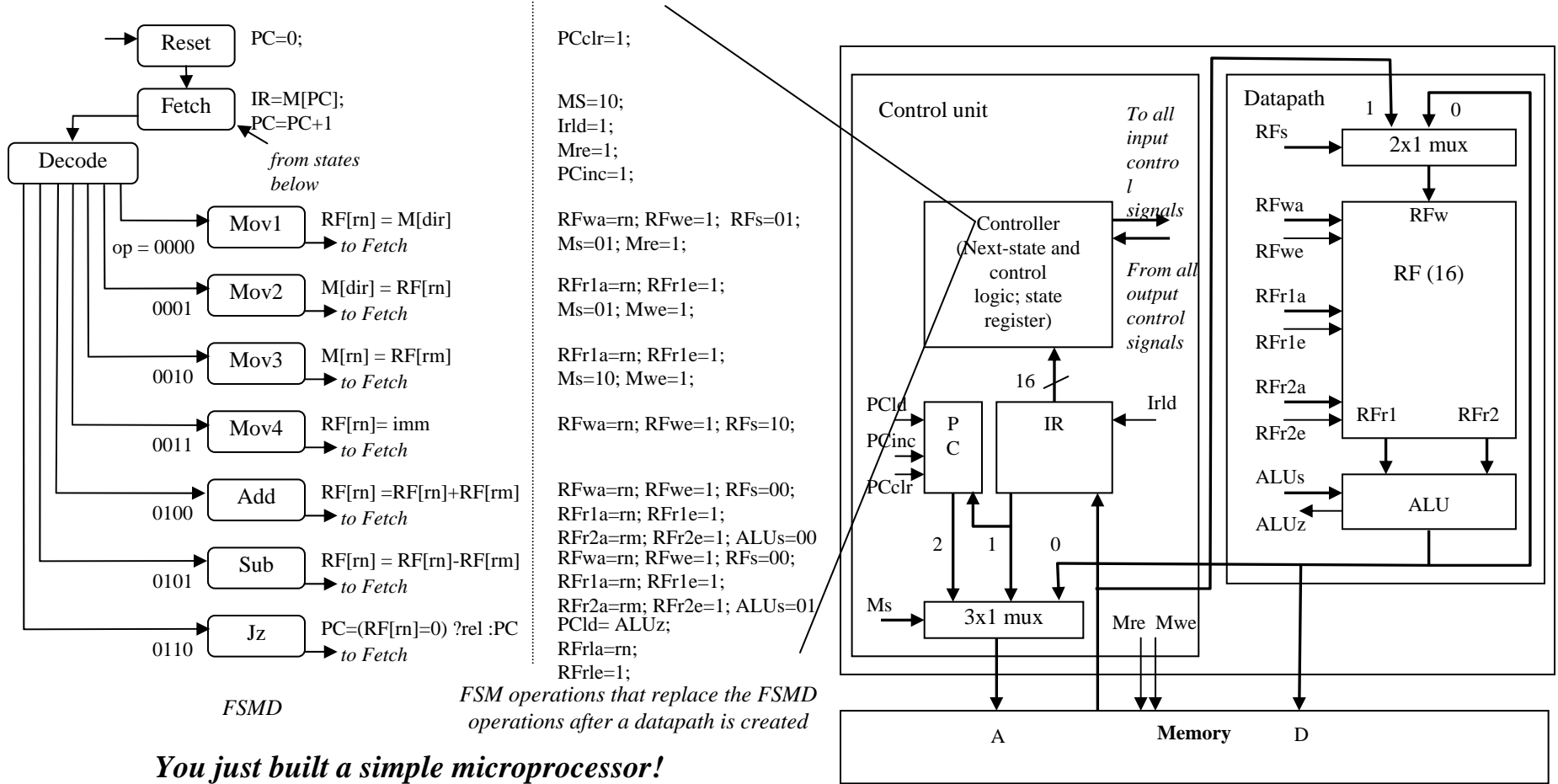


Architecture of a Simple Microprocessor

- Storage devices for each declared variable
 - register file holds each of the variables
- Functional units to carry out the FSM operations
 - One ALU carries out every required operation
- Connections added among the components' ports corresponding to the operations required by the FSM
- Unique identifiers created for every control signal



A Simple Microprocessor



Chapter Summary

- General-purpose processors
 - Good performance, low NRE, flexible
- Controller, datapath, and memory
- Structured languages prevail
 - But some assembly level programming still necessary
- Many tools available
 - Including instruction-set simulators, and in-circuit emulators
- ASIPs
 - Microcontrollers, DSPs, network processors, more customized ASIPs
- Choosing among processors is an important step
- Designing a general-purpose processor is conceptually the same as designing a single-purpose processor