

Embedded Systems Design: A Unified Hardware/Software Introduction

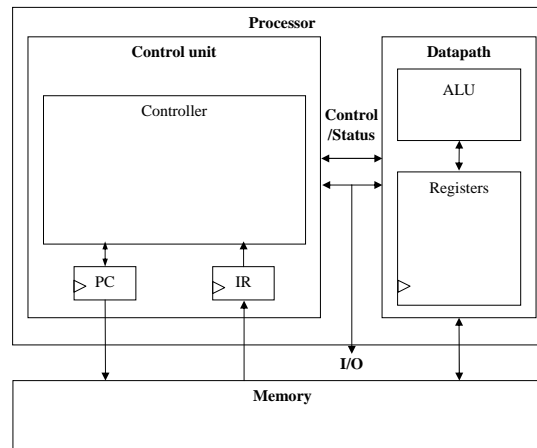
Chapter 3 General-Purpose Processors: Software

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

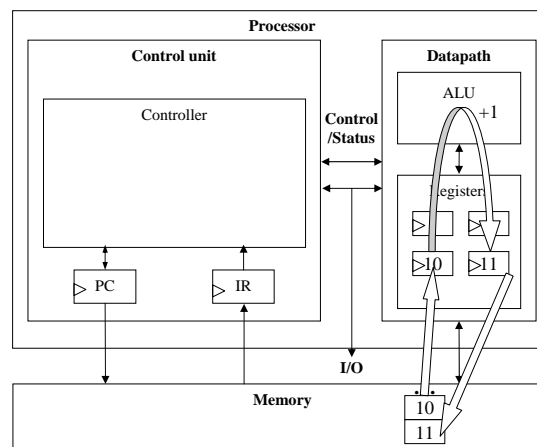


Embedded Systems Design: A Unified
Hardware/Software Introduction, (c) 2000 Vahid/Givargis

3

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

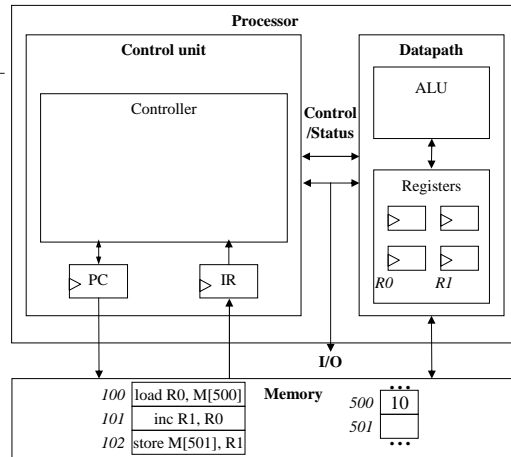


Embedded Systems Design: A Unified
Hardware/Software Introduction, (c) 2000 Vahid/Givargis

4

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

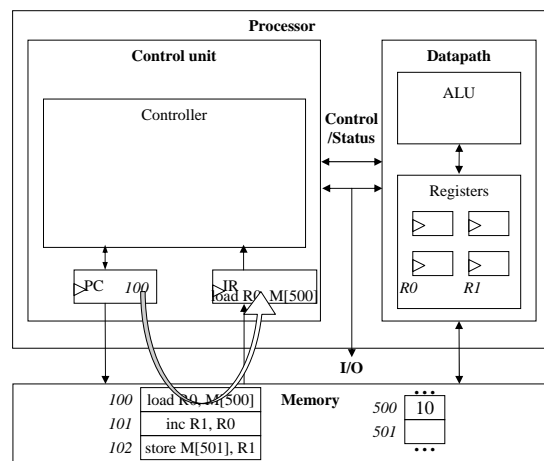


Embedded Systems Design: A Unified
Hardware/Software Introduction, (c) 2000 Vahid/Givargis

5

Control Unit Sub-Operations

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

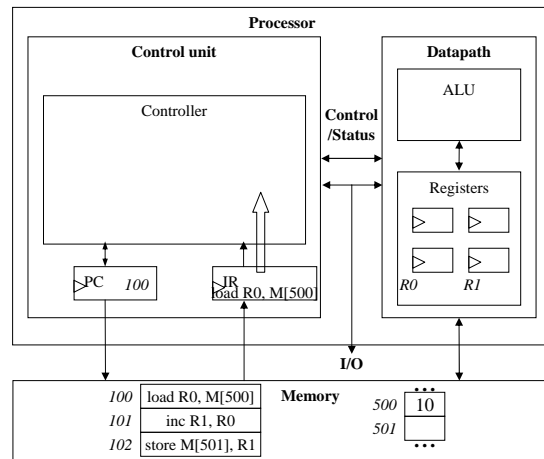


Embedded Systems Design: A Unified
Hardware/Software Introduction, (c) 2000 Vahid/Givargis

6

Control Unit Sub-Operations

- Decode
 - Determine what the instruction means

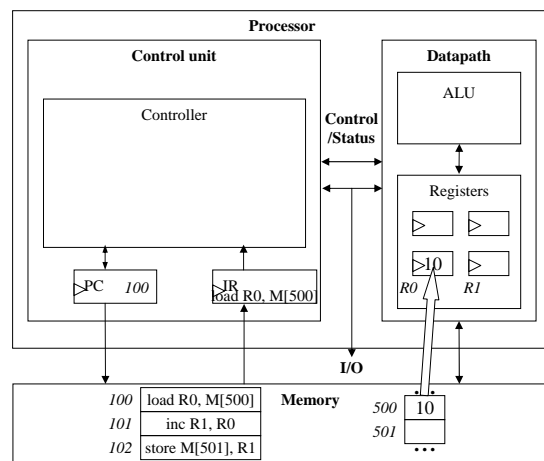


Embedded Systems Design: A Unified
Hardware/Software Introduction, (c) 2000 Vahid/Givargis

7

Control Unit Sub-Operations

- Fetch operands
 - Move data from memory to datapath register

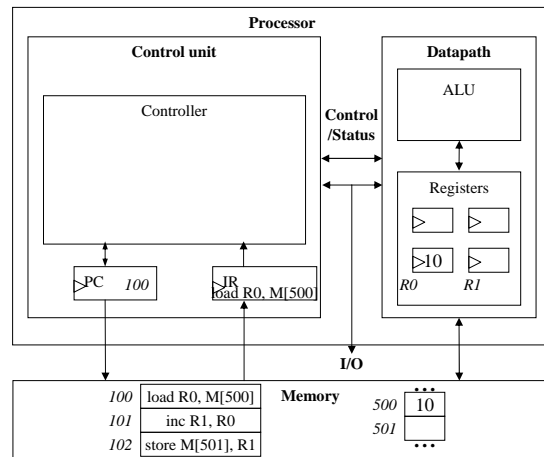


Embedded Systems Design: A Unified
Hardware/Software Introduction, (c) 2000 Vahid/Givargis

8

Control Unit Sub-Operations

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

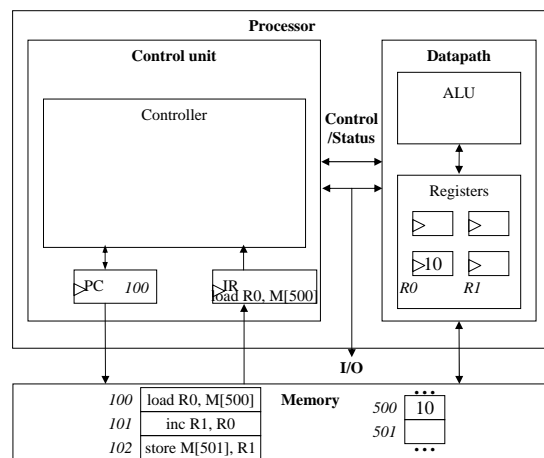


Embedded Systems Design: A Unified
Hardware/Software Introduction, (c) 2000 Vahid/Givargis

9

Control Unit Sub-Operations

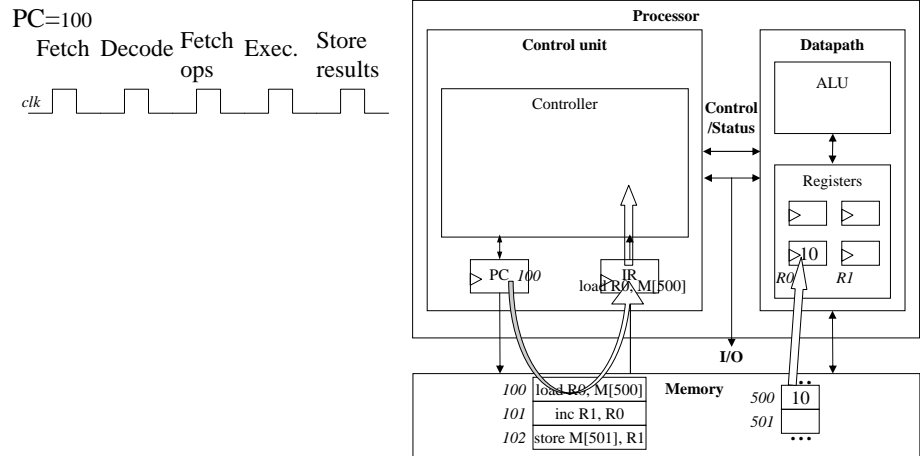
- Store results
 - Write data from register to memory
 - This particular instruction does nothing during this sub-operation



Embedded Systems Design: A Unified
Hardware/Software Introduction, (c) 2000 Vahid/Givargis

10

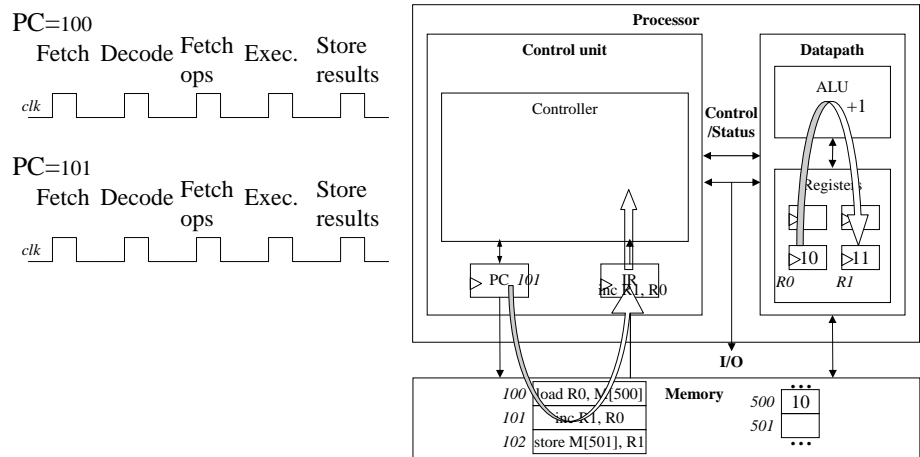
Instruction Cycles



Embedded Systems Design: A Unified
Hardware/Software Introduction, (c) 2000 Vahid/Givargis

11

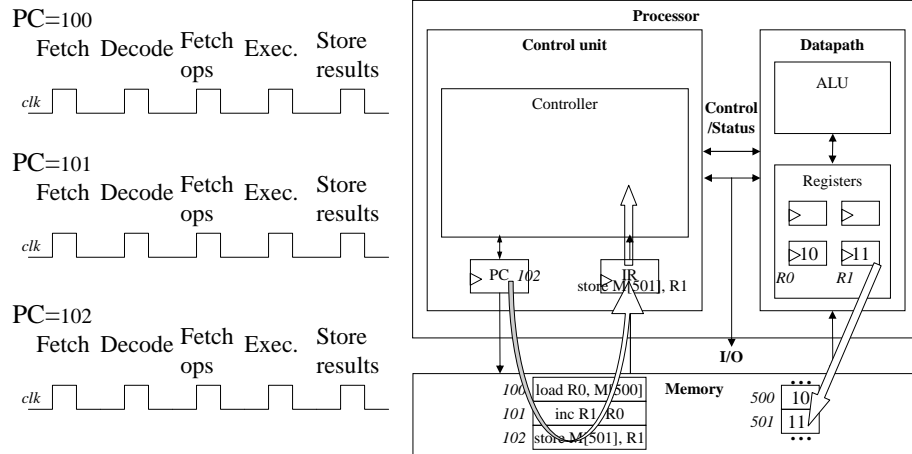
Instruction Cycles



Embedded Systems Design: A Unified
Hardware/Software Introduction, (c) 2000 Vahid/Givargis

12

Instruction Cycles

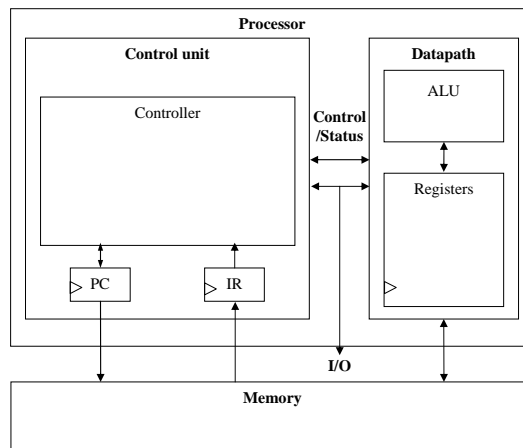


Embedded Systems Design: A Unified
Hardware/Software Introduction, (c) 2000 Vahid/Givargis

13

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

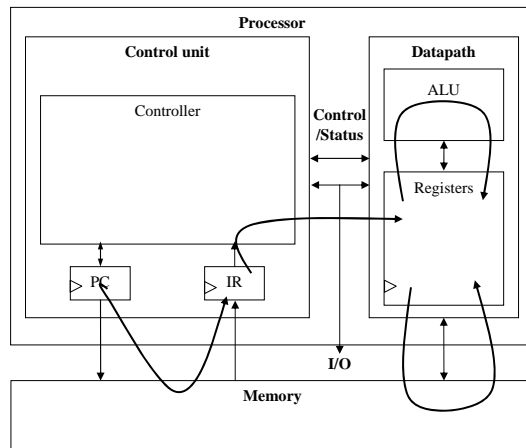


Embedded Systems Design: A Unified
Hardware/Software Introduction, (c) 2000 Vahid/Givargis

14

Architectural Considerations

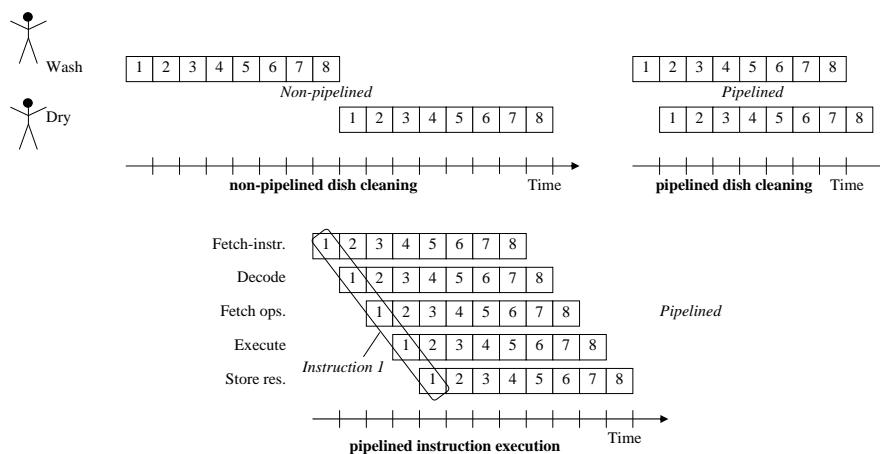
- Clock frequency
 - Inverse of clock period
 - Must be longer than longest register to register delay in entire processor
 - Memory access is often the longest



Embedded Systems Design: A Unified
Hardware/Software Introduction, (c) 2000 Vahid/Givargis

15

Pipelining: Increasing Instruction Throughput



Embedded Systems Design: A Unified
Hardware/Software Introduction, (c) 2000 Vahid/Givargis

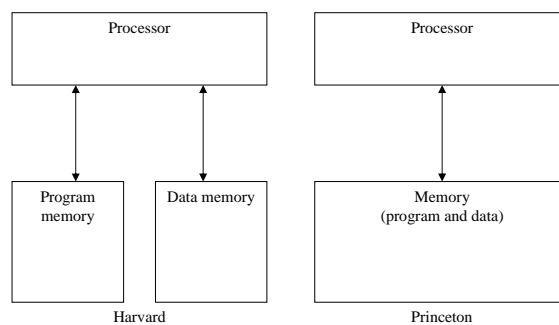
16

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

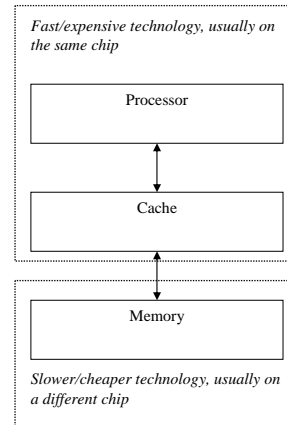
Two Memory Architectures

- 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

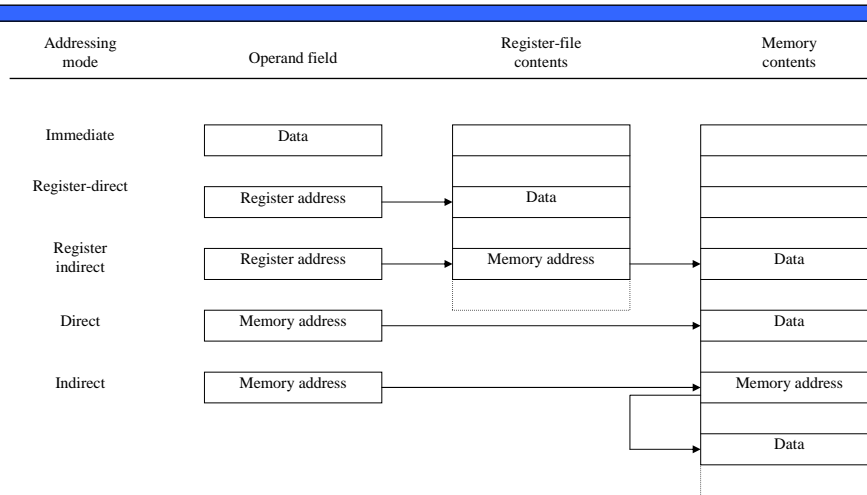
Instruction 1	opcode	operand1	operand2
Instruction 2	opcode	operand1	operand2
Instruction 3	opcode	operand1	operand2
Instruction 4	opcode	operand1	operand2
	...		

- **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

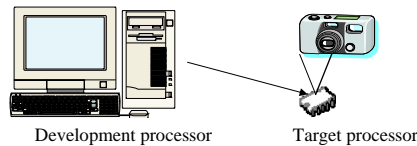


Programmer Considerations

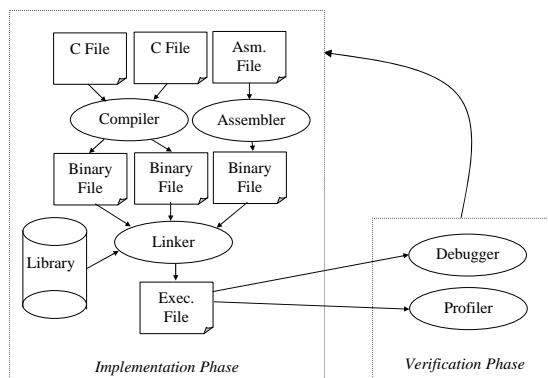
- Program and data memory space
 - Embedded processors often very limited
 - e.g., 64 Kbytes program, 256 bytes of RAM (expandable)
- Registers: How many are there?
 - Only a direct concern for assembly-level programmers
- I/O
 - How communicate with external signals?
- Interrupts

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;

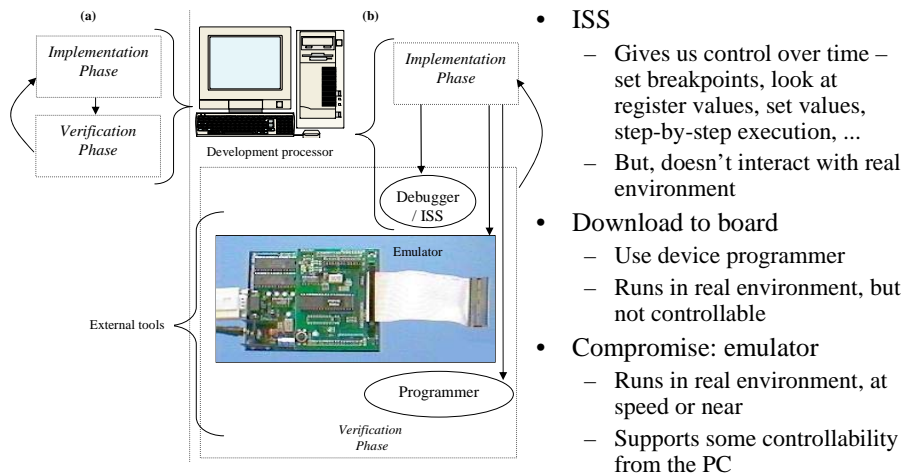
    while( ++pc < (num_bytes / 2) ) {
        fb = program[pc].first_byte;
        sb = program[pc].second_byte;
        switch( fb >> 4 ) {
            case 0: reg[fb & 0x0f] = memory[sb]; break;
            case 1: memory[sb] = reg[fb & 0x0f]; break;
            case 2: memory[reg[fb & 0x0f]] =
      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

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

*Embedded Systems Design: A Unified
Hardware/Software Introduction, (c) 2000 Vahid/Givargis*

29

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

*Embedded Systems Design: A Unified
Hardware/Software Introduction, (c) 2000 Vahid/Givargis*

30

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)

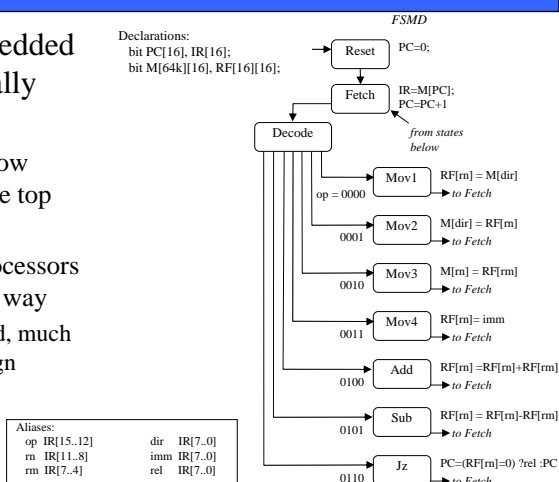
- For signal processing applications
 - Large amounts of digitized data, often streaming
 - Data transformations must be applied fast
 - e.g., cell-phone voice filter, digital TV, music synthesizer
- DSP features
 - Several instruction execution units
 - Multiple-accumulate single-cycle instruction, 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)

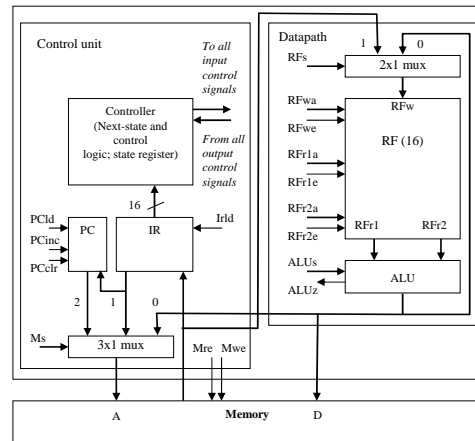
Designing a General Purpose Processor

- 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



Architecture of a Simple Microprocessor

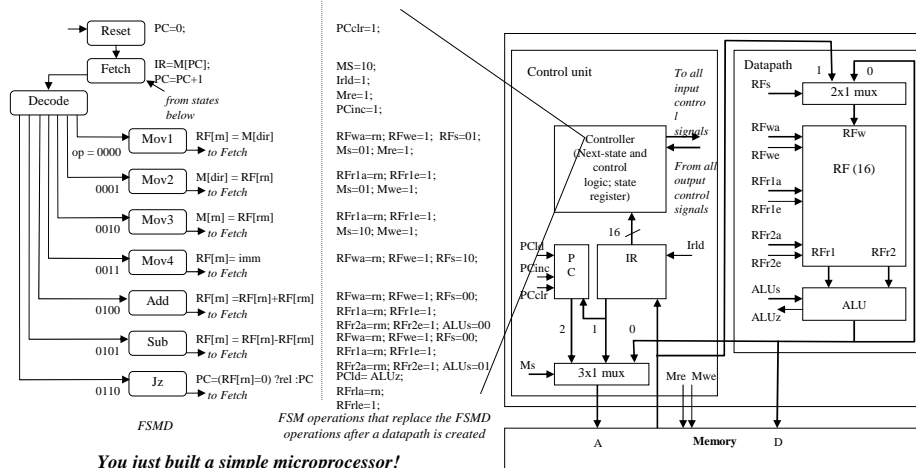
- Storage devices for each declared variable
 - register file holds each of the variables
- Functional units to carry out the FSM operations
 - One ALU carries out every required operation
- Connections added among the components' ports corresponding to the operations required by the FSM
- Unique identifiers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



*Embedded Systems Design: A Unified
Hardware/Software Introduction*, (c) 2000 Vahid/Givargis

35

A Simple Microprocessor



*Embedded Systems Design: A Unified
Hardware/Software Introduction*, (c) 2000 Vahid/Givargis

36