Chapter 3 General-Purpose Processors: Software

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 Sub-Operations

- Decode
  - Determine what the instruction means

- Fetch operands
  - Move data from memory to datapath register

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

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

Instruction Cycles

PC=100
Fetch Decode Fetch Exec Store results
clk

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 Exec Store results
clk

PC=101
Fetch Decode Fetch Exec Store results
clk

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

- **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**

Pipelining: Increasing Instruction Throughput

- **Non-pipelined**
  - Instruction execution
  - Throughput

- **Pipelined**
  - Instruction execution
  - 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 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

### Addressing Modes

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Operand field</th>
<th>Register-file contents</th>
<th>Memory contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>Data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register-direct</td>
<td>Register address</td>
<td>Data</td>
<td></td>
</tr>
<tr>
<td>Register indirect</td>
<td>Register address</td>
<td>Memory address</td>
<td>Data</td>
</tr>
<tr>
<td>Direct</td>
<td>Memory address</td>
<td>Data</td>
<td></td>
</tr>
<tr>
<td>Indirect</td>
<td>Memory address</td>
<td>Memory address</td>
<td>Data</td>
</tr>
</tbody>
</table>

A Simple (Trivial) Instruction Set

<table>
<thead>
<tr>
<th>Assembly instruct.</th>
<th>First byte</th>
<th>Second byte</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV Ra, direct</td>
<td>0000</td>
<td>Ra</td>
<td>direct =&gt; Ra = M(direct)</td>
</tr>
<tr>
<td>MOV direct, Ra</td>
<td>0001</td>
<td>Ra</td>
<td>M(direct) =&gt; Ra</td>
</tr>
<tr>
<td>MOV ij[Ra, Rm]</td>
<td>0010</td>
<td>Ra, Rm</td>
<td>M[Ra] =&gt; Rm</td>
</tr>
<tr>
<td>MOV Ra, simm.</td>
<td>0011</td>
<td>Ra</td>
<td>immediate =&gt; Ra = M[immediate]</td>
</tr>
<tr>
<td>ADD Ra, Rm</td>
<td>0100</td>
<td>Ra, Rm</td>
<td>Ra = Ra + Rm</td>
</tr>
<tr>
<td>SUB Ra, Rm</td>
<td>0101</td>
<td>Ra, Rm</td>
<td>Ra = Ra - Rm</td>
</tr>
<tr>
<td>JZ Ra, relative</td>
<td>0110</td>
<td>Ra</td>
<td>PC = PC + relative (only if Ra is 0)</td>
</tr>
</tbody>
</table>

Sample Programs

```c
C program
// Assume M is an array of int
int total = 0;
for (int i = 0; i < 10; i++)
  total += i;

Loop:
  JZ R1, Next; // Do if i=0
  int total = 0;
  for (int i = 0; i < 10; i++)
    total += i;
  // next instructions...

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? Are any special?
  - Only a direct concern for assembly-level programmers
- I/O
  - How communicate with external signals?
  - Commonly done over ports
- Interrupts
  - Causes processor to suspend execution and jump to an interrupt service routine (ISR)

Example: parallel port driver

<table>
<thead>
<tr>
<th>LPT Connection Pin</th>
<th>I/O Direction</th>
<th>Register Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Output</td>
<td>0th bit of register 02</td>
</tr>
<tr>
<td>2-9</td>
<td>Output</td>
<td>0th - 3rd bit of register 00</td>
</tr>
<tr>
<td>10,11,12,13,15</td>
<td>Input</td>
<td>3, 4, 5, 6th bit of register 01</td>
</tr>
<tr>
<td>14,16,17</td>
<td>Output</td>
<td>1, 2, 3th bit of register 02</td>
</tr>
</tbody>
</table>

- 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. The 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

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

Example: "cat" command

```
OS file_name "cat.c" — store file name
MOV R0, 1294 — system call "open" li
MOV R1, file name — address of file-name
INT 84 — causes a system call
SUB R0, 1 — if zero -> error
JE R0, 1 — execute FRET-0
JMP 60 — bypasses error cond.
LI $1, ... handle the error
```
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

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

Software Development Process

- **Compilers**
  - Cross compiler
    - Runs on one processor, but generates code for another

- **Assemblers**
- **Linkers**
- **Debuggers**
- **Profilers**

Instruction Set Simulator For A Simple Processor

```c
// Simple instruction set simulator

#include <stdio.h>

unsigned char reg[32], a, b;

void run_program(int num_bytes) {
    int i;
    while (i < num_bytes) {
        if (i < num_bytes / 2) {
            a = reg[i];
            b = reg[i + 1];
        }
        printf(":= %d %d
", a, b);

        switch (b) {
            case 0: reg[a] = reg[a] + reg[b]; break;
            case 1: reg[a] = reg[a] + reg[b]; break;
            case 2: reg[a] = reg[a] + reg[b]; break;
            case 3: reg[a] = reg[a] + reg[b]; break;
            default: return -1;
        }
    }
}
```
Testing and Debugging

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

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

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.tensilica.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
    - 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*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](http://www.eembc.org)
    - Suites of benchmarks: automotive, consumer electronics, networking, office automation, telecommunications

**General Purpose Processors**

<table>
<thead>
<tr>
<th>Processor</th>
<th>Clock speed</th>
<th>Power</th>
<th>Bus Width</th>
<th>MIPS</th>
<th>Power</th>
<th>Trans.</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel 8051</td>
<td>2 MHz</td>
<td>32</td>
<td>12</td>
<td>10K</td>
<td>15</td>
<td>$40</td>
<td></td>
</tr>
<tr>
<td>IBM PowerPC 750X</td>
<td>250 MHz</td>
<td>256</td>
<td>64</td>
<td>10K</td>
<td>15</td>
<td>$400</td>
<td></td>
</tr>
<tr>
<td>MIPS R3000</td>
<td>250 MHz</td>
<td>2</td>
<td>16</td>
<td>10K</td>
<td>15</td>
<td>$400</td>
<td></td>
</tr>
<tr>
<td>SiRF 64-bit</td>
<td>No</td>
<td>NA</td>
<td>64</td>
<td>1M</td>
<td>NA</td>
<td>NA</td>
<td></td>
</tr>
</tbody>
</table>

**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
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