Chapter 1: Background

System Software

Department of Computer Science
National Chiao-Tung University
Objectives

• In this section, you will
  – Understand the design concept of registers, instruction sets, instruction formats, and addressing modes
  – Understand how instructions are encoded as byte streams
  – Understand how different addressing modes work
  – See different design philosophies in designing microprocessors
Introduction

• System software
  – Programs that support the operation of a computer
  – User could focus on problems to be solved without bothering with how machines work

• Typical system software
  – Assemblers
  – Linkers and loaders
  – Compilers
  – Operating systems
  – ...
Relation to Machine Architecture

• System software is, inherently, dependent on machine architecture
  – Addressing modes $\leftrightarrow$ assemblers
  – Register configurations $\leftrightarrow$ compilers
  – ...

• Machine-independent concepts are also parts of system software
  – One-pass and multi-pass assembling algorithms
  – The resolving of references to external symbols
  – Grammar parsing for compilers
Relation to Machine Architecture

• Course materials are twofold
  – “General” machine-dependent issues
    • A hypothetical machine “Simplified Instructional Computer” (SIC) is adopted as the target machine
    • To avoid some machine-unique quirks
  – Specific machine-dependent issues
    • How the fundamental theories could be applied to real machines
    • Escaping from implementation details, design philosophies and comparison among them are focused
SIC Machine Architecture

• Memory
  – 1 byte = 8 bits
  – 1 word = 3 bytes = 24 bits
  – 15 bits are used to address $2^{15}$ bytes

• Registers

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Number</th>
<th>Special use</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>Accumulator, for arithmetic operations</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>Index register, for addressing</td>
</tr>
<tr>
<td>L</td>
<td>2</td>
<td>Linkage register, for JSUB</td>
</tr>
<tr>
<td>PC</td>
<td>8</td>
<td>Program counter</td>
</tr>
<tr>
<td>SW</td>
<td>9</td>
<td>Status word (include condition codes)</td>
</tr>
</tbody>
</table>
SIC Machine Architecture

- **Data Formats**
  - 24-bits integers, 2’s complements are used for signed integers
  - 8-bit ASCII codes for characters

- **Instruction Formats**
  - All instructions are of 24 bits
    - Flag x is used to indicate whether or not indexed-addressing mode is used
SIC Machine Architecture

- Two modes to calculate target addresses
  - The indexed mode is for array manipulations
  - $x$ denotes the $x$ bit in an instruction
  - $(X)$ denotes the content of register $X$

<table>
<thead>
<tr>
<th>Mode</th>
<th>Indication</th>
<th>Target Address Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>$x=0$</td>
<td>$TA = \text{address}$</td>
</tr>
<tr>
<td>Indexed</td>
<td>$x=1$</td>
<td>$TA = \text{address} + (X)$</td>
</tr>
</tbody>
</table>
SIC Machine Architecture

- **Instruction Set**
  - Arithmetic operations
    - ADD, SUB, MUL, DIV
    - Binary operations are always performed over register A and a memory location
  - Load/Store
    - LDA, LDX, STA, STX
  - Conditional branches
    - COMP, JLT, JEQ, JGT
    - The result of comparison is first put in CC of register SW, and then jump instructions check CC to see if jumps are needed.
  - Subroutine calls
    - JSUB, RSUB
    - The return address (for PC) is stored in register L
    - How nested subroutine calls could be implemented?
      - Many machines implement a stack register
SIC Machine Architecture

• Input and Output
  – 8 bits in target addresses are used to address I/O devices (ports)
  – Input and output are performed in terms of bytes
  – Dedicate opcodes for I/O operations
    • RD  \(\rightarrow\) read 1 byte from device
    • WD  \(\rightarrow\) write 1 byte from device
    • TD  \(\rightarrow\) test whether device is ready or not (use CC)
SIC/XE Machine Architecture

• SIC evolves to SIC/XE so as to provide
  – A large memory space
  – A large collection of registers
  – Many addressing modes
  – Powerful instructions

• The above items are shared by the evolvements of most modern microprocessors
SIC/XE Machine Architecture

- Memory
  - $2^{15} \rightarrow 2^{20}$ bytes

- Registers
  - Introducing general-purpose registers
  - A new floating-point processor

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Number</th>
<th>Special use</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>3</td>
<td>Base register, for addressing</td>
</tr>
<tr>
<td>S</td>
<td>4</td>
<td>General-purpose register</td>
</tr>
<tr>
<td>T</td>
<td>5</td>
<td>General-purpose register</td>
</tr>
<tr>
<td>F</td>
<td>6</td>
<td>Floating-point accumulator (48bits)</td>
</tr>
</tbody>
</table>
SIC/XE Machine Architecture

• Data Formats:
  – Integers: the same as those in SIC
  – Floating-point numbers:

\[ f \times 2^{(e-1024)} \]
SIC/XE Machine Architecture

• Instruction Formats
  – Format 1: no memory reference
  – Format 2: register-to-register
  – Format 3: 1) relative addressing 2) compatibility
  – Format 4: instruction for extended memory accesses
SIC/XE Machine Architecture

- Decoding instruction addressing modes
  - $e$: 0 → format 3, 1 → format 4
  - $(n,i)$
    - $(1,0)$: indirect addressing (e.g., pointers in C)
    - $(0,1)$: immediate addressing
  - $(1,1)$: just to indicate an SIC/XE instruction
  - $(0,0)$: indicating an SIC instruction (legacy mode)
- $(b,p)$
  - See the following slide
  - Index addressing can’t be used with indirect or immediate addressing
SIC/XE Machine Architecture

• Relative addressing modes
  – To produce re-locatable programs
  – Base-relative addressing
    • Usually for accessing local variables in procedures
    • What’s the difference from indexed addressing?
  – PC-relative addressing
    • Usually for targets of branches

<table>
<thead>
<tr>
<th>Mode</th>
<th>Indication</th>
<th>Target Address Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base relative</td>
<td>b=1, p=0</td>
<td>TA=(B) + disp (0≤disp≤4095)</td>
</tr>
<tr>
<td>Program-counter relative</td>
<td>b=0, p=1</td>
<td>TA=(PC) + disp (-2048≤disp≤2047)</td>
</tr>
</tbody>
</table>
SIC/XE Machine Architecture

- Format 4
  - $e=1$
  - $(b,p)=(0,0)$
  - $(n,i)≠(0,0)$
  - $x$: free (can be direct or indexed)

- Format 3: Invalid cases
  - $x=1$ and $(n,i) = ((1,0)\text{ or } (0,1))$
  - $(b,p)=(1,1)$

- Note: $n,i$ are not parts of an opcode!!
TA+(B)+(X)
• Decoding procedure (referential)
  – Opcode: format 1? format 2?
  – SIC (n=0,i=0) ?SIC/XE? (otherwise)
  – (n,i)
  – x
  – e
  – (b,p)

See the last chart in Appendix A!!
SIC/XE Machine Architecture

- Instruction sets
  - Load/store new registers
    - LDB, STB
  - Floating-point arithmetic
    - ADDF, SUBF, MULF, DIVF
  - Register-to-register operations
    - ADDR, SUBR, MULR, DIVR
  - Supervisor call
    - SVC
(word) \text{ALPHA} \leftrightarrow 5, (byte) \text{C1} \leftrightarrow 'Z'

(a)

\begin{align*}
\text{LDA} & \quad \text{FIVE} \\
\text{STA} & \quad \text{ALPHA} \\
\text{LDCH} & \quad \text{CHARZ} \\
\text{STCH} & \quad \text{C1}
\end{align*}

\text{LOAD CONSTANT 5 INTO REGISTER A} \\
\text{STORE IN ALPHA} \\
\text{LOAD CHARACTER 'Z' INTO REGISTER A}

(b)

\begin{align*}
\text{LDA} & \quad \#5 \\
\text{STA} & \quad \text{ALPHA} \\
\text{LDA} & \quad \#90 \\
\text{STCH} & \quad \text{C1}
\end{align*}

\text{LOAD VALUE 5 INTO REGISTER A} \\
\text{STORE IN ALPHA} \\
\text{LOAD ASCII CODE FOR 'Z' INTO REG A} \\
\text{STORE IN CHARACTER VARIABLE C1}

(a)

- \text{ALPHA} \quad \text{RESW} \quad 1
- \text{FIVE} \quad \text{WORD} \quad 5
- \text{CHARZ} \quad \text{BYTE} \quad \text{'Z'}
- \text{C1} \quad \text{RESB} \quad 1

\text{ONE-WORD VARIABLE} \\
\text{ONE-WORD CONSTANT} \\
\text{ONE-BYTE CONSTANT} \\
\text{ONE-BYTE VARIABLE}

(b)

- \text{ALPHA} \quad \text{RESW} \quad 1
- \text{C1} \quad \text{RESB} \quad 1

\text{ONE-WORD VARIABLE} \\
\text{ONE-BYTE VARIABLE}
How do register-to-register operations improve performance?

\[ BETA \leftarrow (\text{ALPHA} + \text{INCR} - 1) \]
\[ DELTA \leftarrow (\text{GAMMA} + \text{INCR} - 1) \]

\[ A \leftarrow (S) + (A) \]
TIX foo:
X ← (X) + 1
CMP X, foo

for (i = 0; i < 11; i++)
STR2[i] = STR1[i];

There is no immediate version for TIX
Why ADDR S,X not indexed addressing?

For(i=0;i<100;i++)
   GAMMA[i]=ALPHA[i]+BETA[i];
CALL READ SUBROUTINE

SUBROUTINE TO READ 100-BYTE RECORD

READ
LDX #0
LDT #100

RLOOP
TD INDEV
JEQ RLOOP
RD INDEV
STCH RECORD,X

TIXR T
JLT RLOOP
RSUB

INDEV BYTE X’F1’
RECORD RESB 100

INPUT DEVICE NUMBER
100-BYTE BUFFER FOR INPUT RECORD
Case Studies

• CISC vs. RISC
  – x86 architecture and Power PC
• Items that we are interested in:
  – Memory architectures
  – Registers
  – Data formats
  – Instruction formats
  – Addressing modes
  – Instruction sets
  – Input and output
IA-32 Architecture

- **Memory**
  - 1 double word=2 words, 1 word=2 bytes, 1 byte=8 bits
  - MMU handles memory accesses in a two-level fashion
    - Logical address
      - **Segments** are used with the intention of separating memory for different purposes
      - **Offsets** inside segments are used to address the target address
    - Linear address
      - Up to 4GB, to support virtual memory
      - A target address is composed by a page number and page offset
      - Page could be in RAM or external storage

---

![Selector](Selector)

Selector

Offset

Segmentation Mechanism

Linear Address

Paging Mechanism

Physical Address
IA-32 Architecture

- Memory: segmentation

1. Selector picks up a segment descriptor
2. Base+offset is the target address
IA-32 Architecture

• Memory: paging
# IA-32 Architecture (Legacy Registers)

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>Base</th>
<th>Counter</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>AH</td>
<td>BH</td>
<td>CH</td>
<td>DH</td>
</tr>
<tr>
<td>AL</td>
<td>BL</td>
<td>CL</td>
<td>DL</td>
</tr>
<tr>
<td>AX</td>
<td>BX</td>
<td>CX</td>
<td>DX</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code Segment</th>
<th>Data Segment</th>
<th>Stack Segment</th>
<th>Extra Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
<td>DS</td>
<td>SS</td>
<td>ES</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction Pointer</th>
<th>Stack Pointer</th>
<th>Base Pointer</th>
<th>Source Index</th>
<th>Destination Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP</td>
<td>SP</td>
<td>BP</td>
<td>SI</td>
<td>DI</td>
</tr>
</tbody>
</table>
IA-32 Architecture

• Registers
Consider the Instruction: \texttt{mov ax, bx}

The Assembler translates this into: \texttt{8B C3}

<table>
<thead>
<tr>
<th>low addr</th>
<th>opcode</th>
<th>d</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>mod</td>
<td>reg</td>
<td>r/m</td>
<td></td>
</tr>
<tr>
<td>optional</td>
<td>optional</td>
<td>Low Displacement or Immediate</td>
<td></td>
</tr>
<tr>
<td>optional</td>
<td>High Displacement or Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>optional</td>
<td>Low Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>High Immediate</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>high addr</th>
<th>optional</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>High Immediate</td>
</tr>
</tbody>
</table>

| opcode is: | 100010 |
| d is:      | 1      |
| w is:      | 1      |
| mod is:    | 11     |
| reg is:    | 000    |
| r/m is:    | 011    |

\texttt{mov} is: destination is register
\texttt{d is:} 1: destination size = 1 word
\texttt{w is:} 1: this indicates that r/m specifies a register
\texttt{mod is:} 11: destination register is \texttt{ax}
\texttt{reg is:} 000: source register is \texttt{bx}
IA-32 Architecture

• Instruction format
  – Instruction prefix
    • REP MOVESW
      • Repeat move string in terms of words, the number of words to move is in CX, from source address DS:(E)SI to destination address ES:(E)DI.
  – Complicated instruction format
    • Slow decoding
    • One instruction perform vast jobs
Direct Addressing

mov [7000h], ax

ds:7000h ← ax

mov es:[7000h], ax

es:7000h ← ax

prefix byte
- longer instruction
- more fetch time

opcode | mod r/m | displacement

effective address
Register Indirect Addressing

mov al, [bp] ; al gets 8 bits at SS:BP
mov ah, [bx] ; ah gets 8 bits at DS:BX
mov ax, [di] ; ax gets 16 bits at DS:DI
mov eax, [si] ; eax gets 32 bits at DS:SI
Based Indirect Addressing

```plaintext
mov al, [bp+2] ; al gets 8 bits at SS:BP+2
mov ah, [bx-4] ; ah gets 8 bits at DS:BX-4
```

BP: base register for stack seg
BX: base register for data seg

**Diagram**

- **Opcode**
- **mod r/m**
- **Displacement**

```
BP
BX
```

The diagram shows the indirect addressing based on the base registers BP and BX, with the displacement used to calculate the effective address.
Indexed Indirect Addressing

```
mov    ax,    [di+1000h]    ;ax gets 16 bits at DS:SI+1000h
mov    eax,   [si+300h]     ;eax gets 32 bits at DS:SI+300h
```
Based Indexed Indirect Addressing

mov ax, [bp+di] ; ax gets 16 bits at SS:BP+DI
mov ax, [di+bp] ; ax gets 16 bits at DS:BP+DI
mov eax, [bx+si+10h] ; eax gets 32 bits at DS:BX+SI+10h
mov cx, LIST[bp+si-7] ; cx gets 16 bits at SS:BP+SI-7
Power PC Architecture

- POWER
  - Performance Optimization with Enhanced RISC

- Basically PPC is a RISC uP
  - A very small amount of instructions
    - PPC: 200+ (purely RISC uP’s usually have 100+)
    - IA-32: 400+
    - However PPC supports some complex instruction
      - Multiply and add
PPC Architecture

• Memory
  – 1 quadword = 2 double words, 1 double word = 2 words, 1 word = 4 bytes, 1 byte = 8 bits
    • Wide data bus, for the fetch of multiple instructions at the same time
    • Efficient instruction re-ordering, pipelining, parallelizing
  – Virtual address space \(2^{64}\) bytes
    • Divided into fixed-length segments, 256MB each
    • Segments are divided into pages, 4KB each
PPC Architecture

• Registers
  – There are 32 general-purpose registers GPR0~GPR31
    • Reduce the number of load/store instructions needed
    • Improve ILP (instruction level parallelism)
    • Introduce more wires between registers and ALU though
PPC Architecture

• Instruction format and instruction set
  – Only load, store, branch instructions take memory address as the operand
  – 200+ instructions supported
  – All instructions are 32 bits long
    • For efficient pipelining
    • Very fast instruction decoding
PPC Architecture

• Load/Store
  – Register indirect: \( TA=(\text{reg}) \)
  – Register indirect with index: \( TA=(\text{reg1})+(\text{reg2}) \)
  – Register indirect with immediate index: \( TA=(\text{reg})+\text{disp} \)

• Branches
  – Absolute: \( TA=\text{address} \)
  – Relative: \( TA=\text{PC}+\text{disp} \)
  – Link register or count register indirect: \( TA=(\text{LR}) \) or \( TA=(\text{CR}) \)