Program Translation

Program translation uses a collection of tools to perform the translation:

- **Compiler**: Translates high level language programs into a lower level language often called object code.
- **Assembler**: Translates assembly language instructions into object code.
- **Linker**: Combines collections of object code into a single executable machine language program.

Platform Dependence

- **Platform Independent**:
  - A program with instructions that can be executed without modification on several different types of hardware.
  - Examples:
    - User level instructions -- Level 6
    - Many HLL programs (C/C++, Pascal, COBOL, BASIC) -- Level 5
    - Java?

- **Platform Dependent**:
  - A program with instructions that can only be executed on a particular type of hardware.
  - Examples:
    - Assembly language programs -- Level 4
    - HLL programs that use platform specific libraries -- Level 3 (e.g. windowing / graphics).
    - Machine language programs -- Level 2
    - Java?
Platform Dependence

Platform Independent

Platform Dependent

Mac OS X C++ Compiler

WinXP C++ Compiler

Linux C++ Compiler

Mac OS X Executable Program

WinXP Executable Program

Linux Executable Program

Java Programs

Mac OS X Java Compiler

WinXP Java Compiler

Linux Java Compiler

Java Byte Code Program

Mac OS X Java Virtual Machine

WinXP Java Virtual Machine

Linux Java Virtual Machine

Program Translation

C/C++/Java: | Assembly Language | Machine Language
------------+--------------------+----------------------------------
int X=880;  | X: .word 880       |
|                    | 10010011000011110000000000000000 |
|                    | 10011000100011110000000000000000 |
|                    | 01000001111100010000000000000000 |
| ADD R2 R1 #5      | 10110011000100100000000000000101 |
| STORE R2 X        | 10010011000011110000000000100000 |
|                    | 10011001000011110000000000000000 |
|                    | 01000010111100100000000000000000 |
| HALT               | 11111111111111111111111111111111 |

Machine Architecture

Bus Types

Point-to-Point Bus:

Multipoint Bus:
Typical Multipoint Bus Structure

- Multipoint Bus Structure:
  - **Data Bus**: Used to transmit information between devices.
  - **Address Bus**: Used to select the information that is to be transmitted (e.g., memory address).
  - **Control Bus**: Used to issue commands to devices (e.g., read/write) and receive status signals from devices.

Bus Device Types

- **Bus Master(s)**: Devices that can issue commands that initiate the transfer of data across the bus.
  - For example, the CPU is a bus master and can use the control bus to issue a command to main memory to read the value of a variable.
  - Allowing multiple bus masters can improve system performance:
    - E.g., Hard Disk Drives and Direct Memory Access (DMA)

- **Slave Devices**: Devices that can only respond to commands from other devices.
  - For example, main memory is a slave device. It can only respond to commands from other devices asking it to read or write particular memory locations.

Bus Arbitration

- Allowing multiple bus masters can improve system performance but it also introduces the need for a way of allowing devices to take turns controlling the bus:
  - **Bus Arbitration Schemes**:
    - Daisy Chain (i.e., Priority)
    - Centralized Parallel
    - Distributed Self-Selection
    - Distributed Collision Detection

Bus Cycles

- Bus cycles control the flow of information (data commanding/addresses) across the bus.
  - One piece of information moves across the bus during each cycle.

Types of Bus Cycles

- Synchronous: The length of the bus cycle is controlled by a clock and thus every bus cycle is the same length.
- Asynchronous: The length of a bus cycle is determined dynamically by the devices that are communicating.
  - Handshaking (Request / Ready / Acknowledge)

Memory Addresses

- Conceptually main memory is a collection of cells each of which can hold one byte or word of data.
  - Each cell has a unique address that is used to read or write the data in that cell.

<table>
<thead>
<tr>
<th>Address</th>
<th>Binary</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address 0</td>
<td>000</td>
<td>0000</td>
</tr>
<tr>
<td>Address 1</td>
<td>001</td>
<td>0001</td>
</tr>
<tr>
<td>Address 2</td>
<td>010</td>
<td>0002</td>
</tr>
<tr>
<td>Address 3</td>
<td>011</td>
<td>0003</td>
</tr>
<tr>
<td>Address n-2</td>
<td>101</td>
<td>0101</td>
</tr>
<tr>
<td>Address n-1</td>
<td>110</td>
<td>0110</td>
</tr>
</tbody>
</table>

Types of Memory Addressing

- Machines are built to use one of two types of memory addressing:

  - **Byte Addressable Memory**: Each memory address refers to a memory cell that holds a single byte of information.
  - **Word Addressable Memory**: Each memory address refers to a memory cell that holds a full word of information (e.g., 4 bytes).
Byte vs Word Addressable Memory

- **Byte Addressable**: Machine with 32 bit word.
- **Word Addressable**: Machine with 32 bit word.

Alignment and Byte Addressing

- A byte addressable machine may use either aligned or unaligned access:
  - **Aligned access**: Each piece of data must begin on a natural boundary.
    - byte – can be stored anywhere.
    - short – must have address % 2 = 0.
    - int/float – must have address % 4 = 0.
    - long/double – must have address % 8 = 0.
  - **Unaligned access**: data may start at any address regardless of its size.

Memory Addresses

- Within the computer memory addresses are specified using unsigned binary representation. E.g: 1001 1100 0010 00101
- The number of bits available for the memory addresses determines the size of the memory.
- Examples:
  - If a computer uses 16 bits for memory addresses how many memory addresses can we have?
  - If I want to build a byte addressable computer that can access 4GB of RAM how large do my memory addresses need to be?

Constructing Memory

- The computer’s memory is formed by a number of modules, each of which is built from a number of chips.

RAM Chips

- RAM chips are classified using their length and width.
  - **Length**: Number of chip addresses.
  - **Width**: Number of bits stored in each location.
    - Examples:
      - 1K x 8: 2^10 chip addresses each holding 1 byte.
      - 4M x 16: 2^20 chip addresses each holding 2 bytes.
  - A RAM chip is capable of reading or writing only a single location at a time.

Memory Modules

- Memory modules are constructed by combining a number of RAM chips.
  - Also, classified by length and width.
    - Examples:
      - 1M x 64: 2^20 module addresses x 64 bits = 8MB
      - 16K x 16: 2^16 module addresses x 16 bits = 32KB
  - How could we construct a 1Mx64 module from 512Kx16 chips?
### Memory Addresses and Modules

- 1Mx64 module using 512Kx16 chips.

- **Question**: How many memory addresses are on this module if the machine is byte addressable machine? How many bits are needed for the address?

### Uses Multiple Modules

- Often the RAM of a computer will be comprised of multiple memory modules.

- **Two approaches to addressing with multiple modules:**
  - **High order interleaving**
  - **Low order interleaving**

### High Order Interleaving Example

- Byte addressable memory with 8, 4 address x 8 bit modules (32 bytes total)
  - 5 bit memory addresses
    - 3 bit module select
    - 2 bit module address

### Low Order Interleaving Example

- Byte addressable memory with 8, 4 address x 8 bit modules (32 bytes total)
  - 5 bit memory addresses
    - 3 bit module select
    - 2 bit module address

### Interrupts

- **Interrupt**: An event that alters the normal flow of execution in the system.

- **I/O Requests**
  - Disk read complete
  - Network data received
  - Mouse click
  - Key typed
  - etc…
Assembly Language Programming

A thought before we begin…

"He who hasn't hacked assembly language as a youth has no heart. He who does as an adult has no brain."  
John Moore

Opcodes and Operands

Assembly language instructions have 2 parts:

- **Opcode**: indicates the operation to be performed.
  
  - `LOAD`, `STORE`, `ADD`, `SUB` etc…

- **Operands**: follow the opcode and indicate the data on which the opcode will operate.
  
  - `ADD R0 R1 #75`
  - `STORE R0 B`
  - `LOAD R1 A`
  - `LOAD R2 #ARR`
  - `LOAD R3 R2 +8`

A First Example

```c
// HLL
int A=10;
int B=20;
int C;

C = A + B;
```

Assembly Language

- Why learn to program in Assembly Language?
  - Helps in learning about machine architecture
  - Code optimization
    - Speed
      - 90/10 rule
      - Real Time Systems / Games / Operating Systems
    - Size
      - Embedded Systems

Our Machine’s Architecture

- 16 Registers (R0-R15)
  - 32 bits each
  - R0-R11 General purpose
  - R12-R15 Reserved

- ≈ $2^{32}$ bytes of memory
  - Smaller in practice
  - Byte addressable
  - Unaligned access

- Memory Mapped I/O
  - `STDOUT`
  - `STDIN`

A First Example

```c
// Assembly Language
A: .word 10
B: .word 20
C: .word 0

C = A + B;
```

```
// Program code starts here.
LOAD R1 A   * R1 ← MM[A]
LOAD R2 B   * R2 ← MM[B]
ADD R3 R1 R2 * R3 ← R1 + R2
STORE R3 C  * MM[C] ← R3
HALT
```
Assembling a Program

- **Assembler**: Translates an assembly language program into a machine language executable program.
  - Using Our Assembler:
    ```
    java -jar Assembler.jar <assembly> <executable>
    
    <assembly>: Filename of assembly language program to assemble.
    <executable>: Filename for executable program to produce.
    
    Example:
    java -jar Assembler.jar example1.asm example1.exe
    
    Assembles the assembly language program in the file example1.asm into machine language and stores the result in the file example1.exe.
    ```

Running a Program

```java -jar Machine.jar```

Labels

- A label is a mnemonic name for a memory address.
  - Used for global variables:
    ```
    A: .word 10 * MM[A] ← 10
    B: .word 20 * MM[B] ← 20
    C: .word * MM[C]
    ```
  - Used for branch targets:
    ```
    LOOP: ADD R3 R2 R1
          JUMP LOOP
    ```

Assembler Directives

- An assembler directive is an instruction to the assembler and is not translated into machine a machine language instruction.
  - Used for memory allocation:
    ```
    A: .word 10 * MM[A] ← 10
    X: .byte 1 2 3 4 5
    ARR: .space 1000 * reserve 1000 bytes
    ```
  - Used to include sub-routines defined in other files:
    ```
    .include mult.asm
    ```
  - Used to pause execution for debugging:
    ```
    .break
    ```

Input / Output

- Our machine performs Input/Output using memory mapped I/O.
  - Our Assembler provides predefined labels:
    ```
    STDIN: Performing a LOAD from this memory address reads a value typed at the keyboard.
    **LOAD R1 STDIN**
    * All values being read must be entered in our machine’s input text field before running the program.
    
    STDOUT: Performing a STORE to this memory address writes a value to the screen.
    **STORE R4 STDOUT**
    ```

Input/Output Example

- Read two values from the keyboard, compute twice their sum and write the result to the screen.
  ```
  // HLL
  int X; X: .word
  int Y; Y: .word
  int Z; Z: .word
  
  Read X; LOAD R0 STDIN
  STORE R0 X
  
  Read Y; LOAD R1 STDIN
  STORE R1 Y
  
  Z = 2*(X+Y); ADD R2 R0 R1
                 ADD R3 R2 R2
                 STORE R2 Z
  
  Print Z; STORE R2 STDOUT
  ```
If Computer Languages were Cars

- C is a racing car that goes incredibly fast but breaks down every fifty miles.
- C++ is a souped-up racing car with dozens of extra features that only breaks down every 250 miles, but when it does, nobody can figure out what went wrong.
- Java is a family station wagon. It's easy to drive, it's not too fast, and you can't hurt yourself.
- Lisp looks like a car, but with enough tweaking you can turn it into a pretty effective airplane or submarine.
- Perl is supposed to be a pretty cool car, but even if you can figure out how to drive a perl car, you won't be able to drive anyone else's.
- Ruby is a car that was formed when the Perl, Python and Smalltalk cars were involved in a three-way collision.
- Assembly Language is a bare engine; you have to build the car yourself and manually supply it with gas while it's running, but if you're careful it can go like a bat out of hell.

Addressing Modes

- **Addressing Mode**: The addressing mode of an instruction determines how the operands of an instruction are interpreted.
  - Immediate
  - Direct
  - Immediate Label
  - Indirect

  Each opcode will allow one or more addressing modes.
  - Not all opcodes will allow all addressing modes.

Direct Addressing Mode

- **Direct Addressing**: The operand is a label indicating a memory address. The value stored at that memory address is used by the instruction.

  - **Direct mode LOAD**
    - Assembly: `LOAD R1 A`
    - Meaning: `R1 ← MM[A]`

  - **Direct mode STORE**
    - Assembly: `STORE R4 B`
    - Meaning: `MM[B] ← R4`

Immediate Addressing Mode

- **Immediate Addressing**: The value of the operand is the value that is used by the instruction.

  - Immediate mode is indicated by the use of a # before the operand.

  - **Examples**:
    - **Immediate Mode LOAD**
      - Assembly: `LOAD R1 #17`
      - Meaning: `R1 ← 17`
    - **Immediate Mode ADD**
      - Assembly: `ADD R2 R1 #5`
      - Meaning: `R2 ← R1 + 5`

Some Assembly Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD R R R</td>
<td>ADD R1 R2 R3</td>
<td>R1 ← R2 + R3</td>
<td>Immediate Mode</td>
</tr>
<tr>
<td>SUB R R R</td>
<td>SUB R1 R2 R3</td>
<td>R1 ← R2 - R3</td>
<td>Immediate Mode</td>
</tr>
<tr>
<td>ADD R R #</td>
<td>ADD R1 R2 #231</td>
<td>R1 ← R2 + 231</td>
<td>Immediate Mode</td>
</tr>
<tr>
<td>SUB R R #</td>
<td>SUB R1 R2 #0xFF</td>
<td>R1 ← R2 - 0xFF</td>
<td>Immediate Mode</td>
</tr>
<tr>
<td>AND R R R</td>
<td>AND R1 R2 R3</td>
<td>R1 ← R2 &amp; R3</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td>AND R R #</td>
<td>AND R1 R2 #723</td>
<td>R1 ← R2 &amp; 723</td>
<td>Immediate Mode</td>
</tr>
<tr>
<td>OR R R R</td>
<td>OR R1 R2 R3</td>
<td>R1 ← R2</td>
<td>Bitwise OR</td>
</tr>
<tr>
<td>OR R R #</td>
<td>OR R1 R2 #0xAB</td>
<td>R1 ← R2</td>
<td>Immediate Mode</td>
</tr>
<tr>
<td>MOV R R</td>
<td>MOV R1 R2</td>
<td>R1 ← R2</td>
<td></td>
</tr>
<tr>
<td>NOT R R</td>
<td>NOT R1 R2</td>
<td>R1 ← ~R2</td>
<td>Bitwise NOT</td>
</tr>
<tr>
<td>SHL R R</td>
<td>SHL R1 R2</td>
<td>R1 ← R2 &lt;&lt; 1</td>
<td>LSb</td>
</tr>
<tr>
<td>SHR R R</td>
<td>SHR R1 R2</td>
<td>R1 ← R2 &gt;&gt;&gt; 1</td>
<td>MSb</td>
</tr>
<tr>
<td>LOAD R R</td>
<td>LOAD R1 R2</td>
<td>R1 ← MM[R2]</td>
<td>Direct Mode</td>
</tr>
<tr>
<td>STORE R R</td>
<td>STORE R1 R2</td>
<td>MM[R1] ← R2</td>
<td>Direct Mode</td>
</tr>
</tbody>
</table>

Another Example

- Write an assembly program that computes the same result as the following HLL program:

  ```
  int A;
  int B;
  int C;
  Read A;
  Read B;
  C = 4*(A+10) - (2*B-20);
  Print C;
  ```

  ```assembly
  MOV R1 A
  MOV R2 B
  ADD R1 R2
  MOV R3 R1
  ADD R1 R3
  SUB R1 R3
  MOV R4 R1
  OR R1 R5
  MOV R1 R2
  SUB R1 R2
  MOV R1 R3
  OR R1 R5
  MOV R6 R1
  POP R1
  ADD R1 R6
  ```
A Question?

Given what we know so far, is it possible to translate the following HLL code into our assembly language?

```hll
if (x > 5) {
    x = x + 1;
} else {
    x = x - 1;
}
```

If so, how? If not, why not and what would we need to make it possible?

Our Branching Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>JUMP L</td>
<td>JUMP JLOC</td>
<td>PC = JLOC</td>
</tr>
<tr>
<td>BNEG R L</td>
<td>BNEG R1 BLOC</td>
<td>IF R1 &lt; 0 THEN PC = BLOC</td>
</tr>
<tr>
<td>BPOS R L</td>
<td>BPOS R1 BLOC</td>
<td>IF R1 &gt; 0 THEN PC = BLOC</td>
</tr>
<tr>
<td>BZERO R L</td>
<td>BZERO R1 BLOC</td>
<td>IF R1 == 0 THEN PC = BLOC</td>
</tr>
<tr>
<td>BNZERO R L</td>
<td>BNZERO R1 BLOC</td>
<td>IF R1 != 0 THEN PC = BLOC</td>
</tr>
<tr>
<td>BODD R L</td>
<td>BODD R1 BLOC</td>
<td>IF R1 % 2 != 0 THEN PC = BLOC</td>
</tr>
<tr>
<td>BEVEN R L</td>
<td>BEVEN R1 BLOC</td>
<td>IF R1 % 2 == 0 THEN PC = BLOC</td>
</tr>
<tr>
<td>BEQ R R L</td>
<td>BEQ R1 R2 BLOC</td>
<td>IF R1 == R2 THEN PC = BLOC</td>
</tr>
<tr>
<td>BNEQ R R L</td>
<td>BNEQ R1 R2 BLOC</td>
<td>IF R1 != R2 THEN PC = BLOC</td>
</tr>
<tr>
<td>BGEQ R R L</td>
<td>BGEQ R1 R2 BLOC</td>
<td>IF R1 &gt;= R2 THEN PC = BLOC</td>
</tr>
<tr>
<td>BLEQ R R L</td>
<td>BLEQ R1 R2 BLOC</td>
<td>IF R1 &lt;= R2 THEN PC = BLOC</td>
</tr>
<tr>
<td>BGT R R L</td>
<td>BGT R1 R2 BLOC</td>
<td>IF R1 &gt; R2 THEN PC = BLOC</td>
</tr>
<tr>
<td>BLT R R L</td>
<td>BLT R1 R2 BLOC</td>
<td>IF R1 &lt; R2 THEN PC = BLOC</td>
</tr>
</tbody>
</table>

Branching Example

```assembly
//HLL
int x;
if (x > 5) {
    x = x + 1;
} else {
    x = x - 1;
}
```

```assembly
X: .word...
LOAD R0 #5
BLEQ R1 R0 LINE1  * R1<5?
ADD R1 R1 #1  * R1+1
JUMP LINE2
LINE1: SUB R1 R1 #1  * R1-1
LINE2: * Continue code here.
```

Looping Example

```hll
int i;
int x=0;
for (i=1;i<10;i++)
x = x + i;
```

```assembly
//HLL
int i;
int x=0;
for (i=1;i<10;i++)
x = x + i;
```

```assembly
I: .word...
X: .word 0
LOAD R0 #1  * R0 is 1
LOAD R1 #10  * R1 is 10
LOAD R2 X  * R2 is x
START: BGEQ R0 R1 END  * i>=10?
ADD R2 R2 R0  * x=x+1
ADD R0 R0 #1  * i++
JUMP START
END: STORE R0 I
STORE R2 X
...HALT
```