Machine Language, Assemblers, and Compilers

Long, long, time ago, I can still remember
How mnemonics used to make me smile...
And I knew that with just the opcode names
that I could play those assembly games
and maybe hack some programs for a while.
But Comp 120 made me shine,
With every lecture that was delivered,
There was bad news at the doorstep,
I couldn't read another program set.
My whole life thus far must have flashed,
the day we built the MIPS data path,
All I know is that my hard disk crashed,
on the day the hardware died.
And I was singing...

MiniMIPS vs PseudoMIPS
Mips R16000

Power4+
Itanium 2

A Computer System

What is a computer system? Where does it end?
Instruction Representation

32-bit (4-byte) ADD instruction:

\[
\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0
\end{array}
\]

op = R-type  Rs  Rt  Rd  func = add

Means, to MIPS,  \( \text{Reg}[3] = \text{Reg}[4] + \text{Reg}[2] \)

But, most of us would prefer to write

\[
\text{add } 3, 4, 2 \quad \text{(ASSEMBLER)}
\]

or, better yet,

\[
a = b + c; \quad \text{(C)}
\]

A Program for Writing Programs

Machine Language:

1's and 0's toggled loaded into memory. (Did anybody ever really do that?)

Assembly Language:

\[
\begin{array}{c}
\text{Symbolic SOURCE text file} \\
\text{ASM PGM} \\
\text{Binary Machine Language}
\end{array}
\]

Assembler:

1. A Symbolic LANGUAGE for representing strings of bits
2. A PROGRAM for translating Assembly Source to binary.
Assembly Source Language

An Assembly SOURCE FILE contains, in symbolic text, values of successive bytes to be loaded into memory... e.g.

```
.data 0x10000000
.byte 1, 2, 3, 4
.byte 5, 6, 7, 8
.word 1, 2, 3, 4
.asciiz "Comp 120"
.align 2
.word 0xfeedbeef
```

Resulting memory dump:

```
[0x10000000] 0x04030201 0x08070605 0x00000001 0x00000002
[0x10000010] 0x00000003 0x00000004 0x706d6f43 0x30323120
[0x10000020] 0x00000000 0xfeedbeef 0x00000000 0x00000000
```

Notice the byte ordering. This MIPS is “little-endian” (The least significant byte of a word or half-word has the lowest address)

Assembler Syntax

- **Assembler DIRECTIVES (Keywords prefixed with a '.')**
  - Control the placement and interpretation of bytes in memory
    - `.data <addr>` Subsequent items are considered data
    - `.text <addr>` Subsequent items are considered instructions
    - `.align N` Skip to next address multiple of $2^N$
  - Allocate Storage
    - `.byte b1, b2, ..., bn` Store a sequence of bytes (8-bits)
    - `.half h1, h2, ..., hn` Store a sequence of half-words (16-bits)
    - `.word w1, w2, ..., wn` Store a sequence of words (32-bits)
    - `.asciiz "string"` Stores a sequence of ASCII encoded bytes
    - `.ascii "string"` Stores a zero-terminated string
    - `.space n` Allocates n successive bytes
  - Define scope
    - `.global sym` Declares symbol to be visible to other files
    - `.extern sym size` Sets size of symbol defined in another file
More Assembler Syntax

- Assembler COMMENTS
  - All text following a ' ' (sharp) to the end of the line is ignored

- Assembler LABELS
  - Labels are symbols that represent memory addresses. Labels take on the values of the address where they are declared. Labels declarations appear at the beginning of a line, and are terminated by a colon. Labels can be established for data items as well as instructions... e.g.

```asm
.data
item: .word 1 # a data word
.text
start: .word 0x00821820 # add $3, $4, $2
        .word 0x00031a00 # sll $3, $3, 8
        .word 0x306300ff # andi $3, $3, 0xff
```

While having an assembler helps, coding like this is still painful. (Don't actually do this!)

---

Even More Assembler Syntax

- Assembler PREDEFINED SYMBOLS
  - Register names and aliases
    - $0-$31, $zero, $v0-$v1, $a0-$a3, $t0-$t9, $s0-$s7, $at, $k0-$k1, $gp, $sp, $fp, $ra

- Assembler MNEMONICS
  - Symbolic representations of individual instructions
    - add, addu, addiu, sub, subu,
    - and, andi, or, ori, xor, xori, or, nor, lui,
    - sll, srl, sra, sraw, srl, srlv,
    - div, divu, mult, multu, mfhi, mflo, mfhi, mflo,
    - slt, sltu, slti, sltiu, beq, bge, bgez, bgtz, blez,
    - bltz, bltzal, bne, j, jal, jalr, jr,
    - lb, lhu, lh, lw, lw, lw, sb, sh, sw, sw, swr, rfe

- pseudo-instructions (some mnemonics are not real instructions)
  - abs, mulf, mul, mulc, neg, negu, not, rem, remu, rol, ror,
    - li, seq, sge, sgeu, sgt, sgtu, sll, sleu, sse, b, beqz, bge,
    - bgeu, bgt, bgtu, ble, bleu, blt, bult, bnez, la, ld, ulh,
    - ulhu, ulw, sd, ush, usw, move, syscall, break, nop
How an Assembler Works

The essential task of an assembler is to convert symbolic text into the binary values that are loaded into successive bytes of memory.

There are two major components to this task.
1) Allocating and initializing data storage
2) Conversion of mnemonics to binary instructions
3) Resolving addresses

An Assembler scans through a file maintaining a set of “location pointers” for next available byte of each segment (.data, .text, .kdata, and .ktext). Let’s see how this works.

Allocating and Initializing Storage

Memory allocation is handled by assembler directives:

```assembly
.data
x:    .word 42
msg:   .asciiz "Hello, World"
.align 2
array: .space 40
foo:   .word 0xdeadbeef
```

When the assembler encounters label declarations while scanning a source file, it creates a SYMBOL table entry to keep track of its location:

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>SEGMENT</th>
<th>Location pointer offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>data</td>
<td>0</td>
</tr>
<tr>
<td>msg</td>
<td>data</td>
<td>4</td>
</tr>
<tr>
<td>array</td>
<td>data</td>
<td>20</td>
</tr>
<tr>
<td>foo</td>
<td>data</td>
<td>60</td>
</tr>
</tbody>
</table>
Encoding Mnemonics

Mnemonics generate binary encoded instructions:

```plaintext
.globl main
.extern bitrev
.text
main:
    li $a0, 0xfeedadad
    jal bitrev
    j $31
```

Note: unresolved addresses are set to 0 in the referencing instruction (see jal above), and set to unknown in the symbol table.

The assembler also computes the offsets to branch targets for you! It uses the location pointer of the next instruction after the branch and the target's address from the symbol table to do this.

Resolving Addresses - 1st Pass

- "Old-style" 2-pass assembler approach
  - In the first pass, data and instructions are encoded and assigned offsets within their segment, while the symbol table is constructed.
  - Unresolved address references are set to 0.

<table>
<thead>
<tr>
<th>Segment Offset</th>
<th>Code</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0x3c010000</td>
<td>lw $8, x</td>
</tr>
<tr>
<td></td>
<td>0x0c280000</td>
<td></td>
</tr>
<tr>
<td>0x0008</td>
<td>0x00004820</td>
<td>add $9,$0,$0</td>
</tr>
<tr>
<td>0x000c</td>
<td>0x310a0001</td>
<td>loop: andi $10,$8,1</td>
</tr>
<tr>
<td>0x0010</td>
<td>0x11400000</td>
<td>beq $10,$0,shift</td>
</tr>
<tr>
<td>0x0014</td>
<td>0x21290001</td>
<td>addi $9,$9,1</td>
</tr>
<tr>
<td>0x0018</td>
<td>0x00084042</td>
<td>shift: srl $8,$8,1</td>
</tr>
<tr>
<td>0x001c</td>
<td>0x15000000</td>
<td>bne $8,$0,loop</td>
</tr>
</tbody>
</table>

Symbol table after Pass 1

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>SEGMENT</th>
<th>Location pointer offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>data</td>
<td>40</td>
</tr>
<tr>
<td>main</td>
<td>text</td>
<td>0</td>
</tr>
<tr>
<td>loop</td>
<td>text</td>
<td>12</td>
</tr>
<tr>
<td>shift</td>
<td>text</td>
<td>24</td>
</tr>
<tr>
<td>bitrev</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
Resolving Addresses – 2nd Pass

• "Old-style" 2-pass assembler approach
  – In the second pass, the appropriate fields of those instructions that reference memory are filled in with the correct values if possible.

```
0x1500fffb
0x00084042
0x21290001
0x11400001
0x310A0001
0x00004820
0x3c010400
0x8c280040
```

Code

```
bne $8,$0, loop
x001c
shft:
srl $8,$8,1
0x0018
addi $9,$9,10
beq $10,$0,shft
0x0014
loop:
andi $10,$8,1
0x000c
add $9,$0,$0
lw $8, x0x0000
```

Instruction Segment Offset

```
0x0000
0x0008
0x000c
0x0010
0x0014
0x0018
0x001c
```

Symbol table after Pass 1

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>SEGMENT</th>
<th>Location pointer offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>data</td>
<td>40</td>
</tr>
<tr>
<td>main</td>
<td>text</td>
<td>0</td>
</tr>
<tr>
<td>loop</td>
<td>text</td>
<td>12</td>
</tr>
<tr>
<td>shift</td>
<td>text</td>
<td>24</td>
</tr>
</tbody>
</table>

Modern Way – 1-Pass Assemblers

Modern assemblers keep more information in their symbol table which allows them to resolve addresses in a single pass.

• Known addresses (backward references) are immediately resolved.
• Unknown addresses (forward references) are "back-filled" once they are resolved.

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>SEGMENT</th>
<th>Location pointer offset</th>
<th>Resolved?</th>
<th>Reference list</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>data</td>
<td>40</td>
<td>Y</td>
<td>NULL</td>
</tr>
<tr>
<td>main</td>
<td>text</td>
<td>0</td>
<td>Y</td>
<td>NULL</td>
</tr>
<tr>
<td>loop</td>
<td>text</td>
<td>12</td>
<td>Y</td>
<td>NULL</td>
</tr>
<tr>
<td>shift</td>
<td>text</td>
<td>T</td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>

0x0010  NULL
The Role of a Linker

Some aspects of address resolution cannot be handled by the assembler alone.

1) References to data or routines in other modules
2) The layout of all segments in memory
3) Support for REUSABLE code modules
4) Support for RELOCATABLE code modules

This final step of resolution is the job of a LINKER.

Static and Dynamic Libraries

LIBRARIES are commonly used routines stored as a concatenation of "Object files". A global symbol table is maintained for the entire library with entry points for each routine.

The routines in LIBRARIES can be referenced by assembly modules. The appropriate routines entry points are resolved by LINKER, and the appropriate code is added to the executable. This sort of linking is called STATIC.

Many programs use common libraries. It is wasteful of both memory and disk space to include the same code in multiple executables. The modern alternative to STATIC linking is to allow the LOADER to resolve the addresses of some libraries routines. This form of linking is called DYNAMIC linking (e.g. .dll).
Assembly Language Example

.globl bitrev # callable by other modules
.text
.align 2
bitrev:
    li $t0, 32 # count = 32
    move $t1, $0 # result = 0
loop:
    sll $t2, $t1, 1 # $t2 = (result << 1)
    and $t3, $a0, 1 # $t3 = (1 & n)
    or $t1, $t2, $t3 # result = $t2 | $t3
    sra $a0, $a0, 1 # n = n >> 1
    sub $t0, $t0, 1 # count = count - 1
    bne $t0, $0, loop # } while (count != 0)
    move $v0, $t1 # return result
    jr $31

Interactive Simulation
Arghh, this is Painful!

Crazy acronyms are only significantly better than bits.

We need a higher level language!

Even with an assembler there is still a lot of low-level book keeping to be concerned with:
- Resource allocation
  (Where variables are in memory or registers)
- Code placement and organization
- Documentation and readability
- Portability

High-Level Languages

Most algorithms are naturally expressed at a high level. Consider the following algorithm for reversing the bits in a 32-bit word:

```c
int bitrev(int n)
{
    int count=32;
    int result = 0;
    do {
        result = (result << 1) | (1 & n);
        n = n>>1;
        count = count-1;
    } while (count != 0);
    return result;
}
```

Here we've expressed our bit reverse algorithm using the high-level semantics of the "C programming language."

Why?
- more readable
  (by both human and machine)
- concise
- unambiguous
- portable
(algorithms frequently outlast their HW platforms)
- compilers are good optimizers!
  (they might know details of the H/W that you don't, and they know lots of tricks for reorganizing code)
How Compilers Work

At the highest level, compilers match simply search for special keywords and replace them with assembly language templates.

The generation of modern compilers is highly automated. This automation is one of the great successes of computer science. Many portable compilers only require that a "backend" of code fragments or "templates" be generated in the assembly language of the target machine.

Compilers are Template Matchers

The basic task of a compiler is to scan a file looking for particular sequences of operations and keywords called templates.

The first major sort of template is an expression. We've already played around converting C expressions to assembly language. A compiler does basically the same thing.

• Input:     • Output:  
  int x, y;     x:     .word 0  
  y = (x-3)*(y+123456)  y:     .word 0  
                       c:     .word 123456  

Once a template is matched, a compiler emits a specific code sequence.
**Data Structures: Arrays**

The C source code

```c
int hist[100];
int score = 2;
...
hist[score] += 1;
```

might translate to:

```asm
.align 2
score: .word 2
hist: .space 400
...
lw $24, score # $24 = score
sll $24, $24, 2 # make word offset
la $24, hist($24) # $24 = &hist[score]
lw $15, ($24) # $15 = hist[score]
la $15, 1($15) # $15 = $15 + 1
sw $15, ($24) # hist[score] += 1
```

**Data Structures: Structs**

```c
struct Point {
    int x, y;
} P1, P2, *p;
...
P1.x = 157;
...
p = &P1;
p->y = 157;
```

might translate to:

```asm
P1: .space 8
P2: .space 8
p: .space 4
...
la $24, P1 # $24 = P1
lw $15, 157 # $15 = 157
sw $15, P1
sw $15, 157 # $15 = 157
sw $15, 157
sw $15, 4($24)
```

Address:
- VARIABLE base address + CONSTANT component offset
- CONSTANT base address + VARIABLE offset computed from index
Conditionals

C code:
if (expr) {
  STUFF
}

MIPS assembly:
  (compile expr into $rx)
  beq $rx, $0, Lendif
  (compile STUFF)
Lendif:

There are little tricks that come into play when compiling conditional code blocks. For instance, the statement:

```c
if (y > 32) {
  x = x + 1;
}
```

compiles to:
```
lw $24, y
ori $15, $0, 32
slt $1, $15, $24
beq $1, $0, Lendif
lw $24, x
addi $24, $24, 1
sw $24, x
```

C code:
if (expr) {
  STUFF1
} else {
  STUFF2
}

MIPS assembly:
  (compile expr into $rx)
  beq $rx, $0, Lelse
  (compile STUFF1)
Lelse:
  (compile STUFF2)
Lendif:

Loops

C code:
while (expr) {
  STUFF
}

MIPS assembly:
  Lwhile:
    (compile expr into $rx)
    beq $rX, $0, Lendw
    (compile STUFF)
    beq $0, $0, Lwhile
    (compile STUFF2)
Lendif:

Alternate MIPS assembly:
  beq $0,$0,Ltest
Lwhile:
  (compile STUFF)
Ltest:
  (compile expr into $rx)
  bne $rX,$0,Lwhile
Lendw:

Compilers spend a lot of time optimizing in and around loops.
- moving all possible computations outside of loops
- unrolling loops to reduce branching overhead
- simplifying expressions that depend on “loop variables”