Assembling Programs

What is an assembler?

- Expands pseudo-operations into machine instructions
- Translates text assembly language to binary machine code
- Output: object file
  - ".o" files (Unix)
  - ".obj" files (Windows/DOS)

Assembling Programs

```
         .text                         # directive
         .ent main                     # directive
main:    la $4,$array                  # pseudo-op
         li $5,15                      # pseudo-op
         ...
         li $4,0                       # pseudo-op
         jal exit
         .end main                     # directive

         .data                         # directive
$array:  .long 51,491,3991,4,6881,-41  # directive
         .globl exit .text             # directive
```

Handling Forward References

- Two-pass assembly
  - 1: allocate instructions, thus determining addresses
  - 2: assemble instructions knowing all labels
- One-pass or backpatch assembly
  - 1: assemble instructions, put in zero for unknown offsets/addresses, keep track of unfinished instructions
  - Backpatch: when labels appear or at the end of pass 1, fill in the unfinished instructions.

Handling Forward References

Example:

```
    bne $1,$2,L      # branch forward
    sll $0,$0,0      # to label L
L:  addiu $2,$3,$2
```

The assembler will change this to:

```
  bne $1,$2,+1     # branch forward 1 word
  sll $0,$0,0      # relative to the sll
  addiu $2,$3,$2
```

Final machine code:

```
  0x14220001  # bne
  0x00000000  # sll
  0x24620002  # addiu
```

Assembling Programs

Start at address zero (arbitrary).

- Keep track of where the jumps are
- Keep track of references to labels in data
- Keep track of unresolved labels (like “`exit`”)

All this information is saved in the object file.

Try using `mips-sgi-irix5-objdump` on the `.o` files generated for your project.

Object File

- Header
- Code segment (text segment in Unix)
- Data segment
- Relocation information
- Symbol table
- Debugging information

Try using `mips-sgi-irix5-nm` on the `.o` files generated for your project to see the symbol table.

Code Reuse

Standard functions saved in libraries.

- On Unix: `lib`*name*`.a`, `lib`*name*`.so` files
- On Windows: *name*`.lib`, *name*`.dll` files
- Consist of a collection of object files

The linker takes a collection of object files and libraries and generates an executable program.

- On Unix: `ld`
- On Windows: `link`

Linkers

- Static
  - Combine object files and libraries into one executable
  - All symbols are resolved
- Dynamic
  - Generate “partial” executable
  - Add library code at runtime
  - Reduces executable size
  - Libraries can be changed without recompilation
  - One copy of shared library in memory
  - Performance hit

Linkers And Loaders

- Linker
  - resolves all symbols
  - creates final executable
  - stores entry point in executable
- Loader
  - reads executable
  - loads code and data into memory
  - initializes registers, stacks, arguments
  - jumps to start-up routine
  - part of the operating system

ISA Alternatives

- Internal storage: registers, stacks, none
  - registers: choice since 1984
  - stacks: 1960s–70s
  - only memory: not used successfully in 25 years
- Typical operations
  - heavily used ones, little changed since 1970
  - fancy instructions, underused and eliminated
- Operands
  - register-register: all since 1980
  - register-memory: x86, Motorola 680x0, 360
  - memory-memory: VAX

Operations Supported

- Most machines have a base set like the MIPS ISA
- Recently, instructions added for multimedia and graphics applications (Intel MMX, Sun VIS, HP MAX-2)

Some Elaborate Operations:

- arithmetic/logical operations on bytes and halfwords
- string operations: copy, compare
- subroutine call/return
- bit field operations
- data structure support (lists, queues)

Control Flow

- Condition Codes

  Special bits set as a side-effect of arithmetic operations.

```
add r1,r2,r3
bz label
```

- Condition Register

  Evaluate into a register and test its value.

```
cmp r1,r2,r3
bgt r1, label
```

- Compare and Branch

```
bgt r1,r2, label
```

Accessing And Addressing Operands

- Recent architectures are load-store architectures
- Registers are general-purpose
- Substantial differences in different architectures
- Example: VAX
  - any operand can be in a register or memory
  - memory locations can be addressed with many modes

Addressing Modes

Mode	Example	Meaning
register	`add r4,r3`	`r4:=r4+r3`
immediate	`add r4,3`	`r4:=r4+3`
displacement	`add r4,100(r1)`	`r4:=r4+mem[100+r1]`
register indirect	`add r4,(r1)`	`r4:=r4+mem[r1]`
indexed/base	`add r4,(r1+r2)`	`r4:=r4+mem[r1+r2]`
direct/absolute	`add r4,(100)`	`r4:=r4+mem[100]`
memory indirect	`add r4,@(r3)`	`r4:=r4+mem[mem[r3]]`
auto-increment	`add r4,(r3)+`	`r4:=r4+mem[r3]; r3:=r3+d`
auto-decrement	`add r4,-(r3)`	`r3:=r3-d; r4:=r4+mem[r3]`

Instruction Encoding

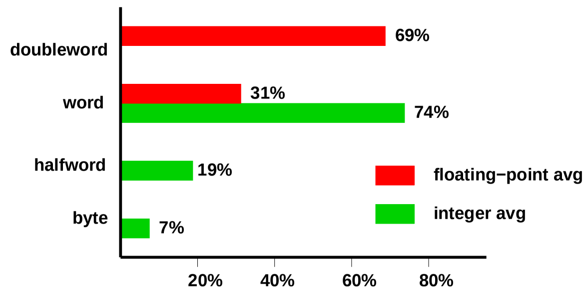
- Fixed
  - Each instruction uses fixed number of bits
  - Example: MIPS, 1 word per instruction
  - Know where next instruction begins without looking at current instruction ⇒ hardware is simpler
- Variable
  - Number of bits used per instruction varies
  - Example: x86 uses 1, 2, 3, ... > 10 bytes
  - Compact code (x86: avg 3 bytes)
  - Hardware more complex

ISA Rationale

- Metrics
  - *design cost: HW and SW*
  - *performance, power, code size*
- Influenced by
  - *program usage: which instructions are frequently used?*
  - *efficient HW implementation strategies*
  - *compiler technology*
- Code efficiency and compilation
  - *orthogonality: avoid special cases*
  - *complex operations are hard to compile to*

Operand Usage

Operand sizes:



⇒ support 8-bit, 16-bit, 32-bit integer, and 32-bit and 64-bit floating-point.

Constant Usage

- Immediate sizes:
  - 50% to 60% fit within 8 bits
  - 75% to 80% fit within 16 bits with sign extension
- Address displacements:
  - 1% of addresses need >16 bits
  - 12-16 bits sufficient
- Conditional branch distance:
  - 35% of integer branches are within -4..+3 ins
  - Virtually none beyond 512 instructions
  - Equality test: most frequent branch case