Big Picture: Where are we going?

C
compiler

RISC-V
assembly

assembly

machine
code

CPU

Circuits

Gates

Transistors

Silicon

int x = 10;
x = 2 * x + 15;
addi x5, x0, 10
muli x5, x5, 2
addi x5, x5, 15
x0 = 0
x5 = x0 + 10
x5 = x5<<1 #x5 = x5 * 2
x5 = x15 + 15

op = r-type               x5    shamt=1     x5      func=sll
00000000101000000000101010010011
00000000001000101000001010000000
00000000111100101000001010010011
10                              r0                  r5 op = addi
15                               r5                 r5 op = addi

A B

32

RF

32
Big Picture: Where are we going?

```
int x = 10;
x = 2 * x + 15;
```

```
addi x5, x0, 10
muli x5, x5, 2
addi x5, x5, 15
```

```
0000000000000000101001000100100000010100010011
0000000000000100010100000100001100000012
00000000000011001010000101000101010010011
```

High Level Languages

Instruction Set Architecture (ISA)
When most people say “compile” they mean the entire process: compile + assemble + link

“It’s alive!”

Executing in Memory process

When most people say "compile" they mean the entire process: compile + assemble + link
#include <stdio.h>

int n = 100;
int main (int argc, char* argv[ ]) {
    int i;
    int m = n;
    int sum = 0;

    for (i = 1; i <= m; i++) {
        sum += i;
    }
    printf ("Sum 1 to %d is %d\n", n, sum);
}
Compiler

**Input:** Code File (.c)
- Source code
- `#includes`, function declarations & definitions, global variables, *etc.*

**Output:** Assembly File (RISC-V)
- RISC-V assembly instructions (.s file)

```c
for (i = 1; i <= m; i++) {
    sum += i;
}
```

```assembly
li  x2,1
lw  x3,fp,28
slt x2,x3,x2
```
.globl n
.data
.n: .word 100
.rdata
$str0$: .string "Sum 1 to %d is %d\n"
.text
.globl main
.type main, @function
main:  addiu $sp,$sp,-48
       sw $ra,44($sp)
       sw $fp,40($sp)
       move $fp,$sp
       sw $a0,-36($fp)
       sw $a1,-40($fp)
       la $a5,n
       lw $a5,0($a5)
       sw $a5,-28($fp)
       sw $0,-24($fp)
       li $a5,1
       sw $a5,-20($fp)
       sw $a4,-24($fp)
       lw $a5,-20($fp)
       addu $a5,$a4,$a5
       sw $a5,-24($fp)
       lw $a5,-20($fp)
       addi $a5,$a5,1
       sw $a5,-20($fp)
       j $L2
       $L3$:  la $4,$str0
       lw $a1,-28($fp)
       lw $a2,-24($fp)
       jal printf
       li $a0,0
       mv $sp,$fp
       lw $ra,44($sp)
       lw $fp,40($sp)
       addiu $sp,$sp,48
       jr $ra

$L2$:  lw $a4,-20($fp)
        lw $a5,-28($fp)
        blt $a5,$a4,$L3
        lw $a4,-24($fp)
        lw $a5,-20($fp)
        addu $a5,$a4,$a5
        sw $a5,-24($fp)
        lw $a5,-20($fp)
        addi $a5,$a5,1
        sw $a5,-20($fp)
        j $L2
Assembler

**Input:** Assembly File (.s)
- assembly instructions, pseudo-instructions
- program data (strings, variables), layout directives

**Output:** Object File in binary machine code
RISC-V instructions in executable form
(.o file in Unix, .obj in Windows)

```
addi r5, r0, 10
muli r5, r5, 2
addi r5, r5, 15
```
RISC-V Assembly Instructions

Arithmetic/Logical
- ADD, SUB, AND, OR, XOR, SLT, SLTU
- ADDI, ANDI, ORI, XORI, LUI, SLL, SRL, SLTI, SLTIU
- MUL, DIV

Memory Access
- LW, LH, LB, LHU, LBU,
- SW, SH, SB

Control flow
- BEQ, BNE, BLE, BLT, BGE
- JAL, JALR

Special
- LR, SC, SCALL, SBREAK
Assembly shorthand, technically not machine instructions, but easily converted into 1+ instructions that are

<table>
<thead>
<tr>
<th>Pseudo-Insns</th>
<th>Actual Insns</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOP</td>
<td>SLL x0, x0, 0</td>
<td># do nothing</td>
</tr>
<tr>
<td>MOVE reg, reg</td>
<td>ADD r2, r0, r1</td>
<td># copy between regs</td>
</tr>
<tr>
<td>LI reg, 0x45678</td>
<td>LUI reg, 0x4</td>
<td>#load immediate</td>
</tr>
<tr>
<td></td>
<td>ORI reg, reg, 0x5678</td>
<td></td>
</tr>
<tr>
<td>LA reg, label</td>
<td></td>
<td># load address (32 bits)</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td># unconditional conditional branch</td>
</tr>
<tr>
<td>BLT reg, reg, label</td>
<td>SLT r1, rA, rB</td>
<td># branch less than</td>
</tr>
<tr>
<td></td>
<td>BNE r1, r0, label</td>
<td></td>
</tr>
</tbody>
</table>

+ a few more…
Program Layout

- Programs consist of **segments** used for different purposes
  - **Text**: holds instructions
  - **Data**: holds statically allocated program data such as variables, strings, etc.

```
add x1,x2,x3
ori x2, x4, 3
```

```
“cornell cs”
13
25
```

```
...”
Assembling Programs

- Assembly files consist of a mix of
  - instructions
  - pseudo-instructions
  - assembler (data/layout) directives

- Assembled to an Object File
  - Header
  - Text Segment
  - Data Segment
  - Relocation Information
  - Symbol Table
  - Debugging Information
Assembling Programs

- Assembly using a (modified) Harvard architecture
- Need segments since data and program stored together in memory
Takeaway

• Assembly is a low-level task
  • Need to assemble assembly language into machine code binary. Requires
    - Assembly language instructions
    - *pseudo-instructions*
    - And Specify layout and data using *assembler directives*

• Today, we use a modified Harvard Architecture (Von Neumann architecture) that mixes data and instructions in memory
  … but kept in separate *segments*
  … and has separate caches
int pi = 3;
int e = 2;
static int randomval = 7;

extern int usrid;
extern int printf(char *str, ...);

int square(int x) { ... }
static int is_prime(int x) { ... }
int pick_prime() { ... }
int get_n() {
    return usrid;
}

(math.c

Global labels: Externally visible “exported” symbols
- Can be referenced from other object files
- Exported functions, global variables
  - Examples: pi, e, userid, printf, pick_prime, pick_random

Local labels: Internally visible only symbols
- Only used within this object file
- static functions, static variables, loop labels, ...
  - Examples: randomval, is_prime

(extern == defined in another file)
Handling forward references

Example:

\[
\begin{align*}
\text{bne } & x1, x2, \ L & \quad \text{Looking for } L \\
\text{sll } & x0, x0, 0 \\
\text{L: } & \text{addi } x2, x3, 0x2 & \quad \text{Found } L
\end{align*}
\]

The assembler will change this to

\[
\begin{align*}
\text{bne } & x1, x2, +1 \\
\text{sll } & x0, x0, 0 \\
& \text{addi } x2, x3, 0x2
\end{align*}
\]

Final machine code

\[
\begin{align*}
0x14220001 \ & \# \text{ bne } & \quad \text{actually: } 000101\ldots \\
0x00000000 \ & \# \text{ sll } & \quad 000000\ldots \\
0x24620002 \ & \# \text{ addiu } & \quad 001001\ldots
\end{align*}
\]
Object file

Header
• Size and position of pieces of file

Text Segment
• instructions

Data Segment
• static data (local/global vars, strings, constants)

Debugging Information
• line number → code address map, etc.

Symbol Table
• External (exported) references
• Unresolved (imported) references
Object File Formats

Unix
• a.out
• COFF: Common Object File Format
• ELF: Executable and Linking Format

Windows
• PE: Portable Executable

All support both executable and object files
Objdump disassembly

> mipsel-linux-objdump --disassemble math.o

Disassembly of section .text:

00000000 <get_n>:
   0:  27bdfff8  addiu sp,sp,-8
   4:  afbe0000  sw   s8,0(sp)
   8:  03a0f021  move s8,sp
  c:  3c020000  lui  v0,0x0
 10:  8c420008  lw    v0,8(v0)
 14:  03c0e821  move sp,s8
 18:  8fbe0000  lw    s8,0(sp)
 1c:  27bd0008  addiu sp,sp,8
 20:  03e00008  jr    ra
 24:  00000000  nop

elsewhere in another file: int usrid = 41;
int get_n() {
    return usrid;
}
## Objdump symbols

```bash
> mipsel-linux-objdump --syms math.o
```

<table>
<thead>
<tr>
<th>SYMBOL TABLE</th>
<th>segment</th>
<th>size</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000</td>
<td>l df</td>
<td><em>ABS</em></td>
<td>0000000000 math.c</td>
</tr>
<tr>
<td>0000000000</td>
<td>l d</td>
<td>.text</td>
<td>0000000000 .text</td>
</tr>
<tr>
<td>0000000000</td>
<td>l d</td>
<td>.data</td>
<td>0000000000 .data</td>
</tr>
<tr>
<td>0000000000</td>
<td>l d</td>
<td>.bss</td>
<td>0000000000 .bss</td>
</tr>
<tr>
<td>0000000008</td>
<td>l O</td>
<td>.data</td>
<td>0000000004 randomval</td>
</tr>
<tr>
<td>0000000060</td>
<td>l F</td>
<td>.text</td>
<td>0000000028 is_prime</td>
</tr>
<tr>
<td>0000000000</td>
<td>l d</td>
<td>.rodata</td>
<td>0000000000 .rodata</td>
</tr>
<tr>
<td>0000000000</td>
<td>l d</td>
<td>.comment</td>
<td>0000000000 .comment</td>
</tr>
<tr>
<td>0000000000</td>
<td>g O</td>
<td>.data</td>
<td>0000000004 pi</td>
</tr>
<tr>
<td>0000000004</td>
<td>g O</td>
<td>.data</td>
<td>0000000004 e</td>
</tr>
<tr>
<td>0000000000</td>
<td>g F</td>
<td>.text</td>
<td>0000000028 get_n</td>
</tr>
<tr>
<td>0000000028</td>
<td>g F</td>
<td>.text</td>
<td>0000000038 square</td>
</tr>
<tr>
<td>0000000088</td>
<td>g F</td>
<td>.text</td>
<td>000000004c pick_prime</td>
</tr>
<tr>
<td>0000000000</td>
<td><em>UND</em></td>
<td>0000000000 usrid</td>
<td></td>
</tr>
<tr>
<td>0000000000</td>
<td><em>UND</em></td>
<td>0000000000 printf</td>
<td></td>
</tr>
</tbody>
</table>
Separate Compilation & Assembly

Compiler  Assembler  Linker

sum.c  sum.s  sum.o
source files  assembly files  obj files

math.c  math.s  math.o

Executable program
exists on disk
loader
Executing in Memory
process
Linkers

Linker combines object files into an executable file:

- Resolve as-yet-unresolved symbols
- Each has illusion of own address space
  → Relocate each object’s text and data segments
- Record top-level entry point in executable file

End result: a program on disk, ready to execute

E.g. ./sum Linux
   ./sum.exe Windows
   simulate sum Class RISC-V simulator
**Static Libraries**

*Static Library*: Collection of object files (think: like a zip archive)

Q: Every program contains the entire library?!?
Linker Example: Loading a Global Variable

main.o

... 0C000000
40 21035000
44 1b80050C
48 8C040000
50 21047002
54 0C000000
...

00 T main
00 D usrid
*UND* printf
*UND* pi
*UND* get_n
40,JAL, printf
...
54,JAL, get_n

math.o

... 24 21032040
28 0C000000
2C 1b301402
30 3C040000
34 34040000
...

20 T get_n
00 D pi
*UND* printf
*UND* usrid
28,JAL, printf
30,LUI, usrid
34,LA, usrid

sum.exe

... 0040 0100
0C400000
21035000
0040 0200
1b80050C
0040 0300
8C040000
21047002
0040 0400
0C000000
...

00000003
0077616B

LA num: 1000
1000 ORI 0004

math

... 0C400000
21035000
1b301402
3C041000
34040000
...

main

... 0040 0000
0C400000
21032040
0C400023
1b301402
3C041000
34040000
...

printf

... 10201000
21040330
22500102
...

Entry: 0040 0100
text: 0040 0000
data: 1000 0000

pi 00000003
usrid 0077616B

Relocation info

40, JAL, printf
28, JAL, printf
30, LUI, usrid
34, LA, usrid

Symbol table

Entry: 0040 0100
text: 0040 0000
data: 1000 0000
Loaders

*Loader* reads executable from disk into memory

- Initializes registers, stack, arguments to first function
- Jumps to entry-point

Part of the Operating System (OS)
Shared Libraries

Q: Every program contains parts of same library?!
Static and Dynamic Linking

Static linking
- Big executable files (all/most of needed libraries inside)
- Don’t benefit from updates to library
- No load-time linking

Dynamic linking
- Small executable files (just point to shared library)
- Library update benefits all programs that use it
- Load-time cost to do final linking
  - But dll code is probably already in memory
  - And can do the linking incrementally, on-demand
Takeaway

**Compiler** produces assembly files
  (contain RISC-V assembly, pseudo-instructions, directives, etc.)

**Assembler** produces object files
  (contain RISC-V machine code, missing symbols, some layout information, etc.)

**Linker** joins object files into one executable file
  (contains RISC-V machine code, no missing symbols, some layout information)

**Loader** puts program into memory, jumps to 1\(^{st}\) insn, and starts executing a *process*
  (machine code)