Big Picture: Where are we going?

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

```
addi r5, r0, 10
addi r5, r5, 15
```

$r0 = 0$

$r5 = r0 + 10$

$r5 = r15 + 15$
Big Picture: Where are we going?

C
compiler

MIPS
assembly

assembler

machine
code

loader

CPU

Circuits

Gates

Transistors

Silicon

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

addi r5, r0, 10
muli r5, r5, 2
addi r5, r5, 15

001000000000010100000000000000001010
00000000000001010010000100000000
00100000101001010000000000001111

High Level Languages

Instruction Set Architecture (ISA)
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 (MIPS)
- MIPS assembly instructions (.s file)

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

```
li  $2,1
lw  $3,28($fp)
slt $2,$3,$2
```
sum.s (abridged)

.globl n
.data
.type n, @object
n: .word 100
.rdata
$str0$: .ascii "Sum 1 to %d is %d\n"
.text
.globl main
.type main, @function
main: addiu $sp,$sp,-48
sw $31,44($sp)
sw $fp,40($sp)
move $fp,$sp
sw $4,48($fp)
sw $5,52($fp)
la $2,n
lw $2,0($2)
sw $2,28($fp)
sw $0,32($fp)
li $2,1
sw $2,24($fp)
$L2: lw $2,24($fp)
lw $3,28($fp)
slt $2,$3,$2
bne $2,$0,$L3
lw $3,32($fp)
lw $2,24($fp)
addu $2,$3,$2
sw $2,32($fp)
lw $2,24($fp)
addiu $2,$2,1
sw $2,24($fp)
b $L2
$L3: la $4,$str0
lw $5,28($fp)
lw $6,32($fp)
jal printf
move $sp,$fp
lw $31,44($sp)
lw $fp,40($sp)
addiu $sp,$sp,48
li $2,1
sw $2,24($fp)
j $31
Assembler

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

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

```
addi r5, r0, 10
muli r5, r5, 2
addi r5, r5, 15
```

```
001000000000010101000000000000001010
0000000000000101010101010000100000
00100000101001010100000000000001111
```
MIPS Assembly Instructions

Arithmetic/Logical
- ADD, ADDU, SUB, SUBU, AND, OR, XOR, NOR, SLT, SLTU
- ADDI, ADDIU, ANDI, ORI, XORI, LUI, SLL, SRL, SLLV, SRLV, SRAV, SLTI, SLTIU
- MULT, DIV, MFLO, MTLO, MFHI, MTHI

Memory Access
- LW, LH, LB, LHU, LBU, LWL, LWR
- SW, SH, SB, SWL, SWR

Control flow
- BEQ, BNE, BLEZ, BLTZ, BGEZ, BGTZ
- J, JR, JAL, JALR, BEQL, BNEL, BLEZL, BGTZL

Special
- LL, SC, SYSCALL, BREAK, SYNC, COPROC
Pseudo-Instructions

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 r0, r0, 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 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 r1,r2,r3
ori r2, r4, 3
...
```

```
“cornell cs”
13
25
```

```
add r1,r2,r3
ori r2, r4, 3
...
```
Assembling Programs

- Assembly files consist of a mix of
  - instructions
  - pseudo-instructions
  - assembler (data/layout) directives
    (Assembler lays out binary values in memory based on directives)

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

.text
.ent main
main: la $4, Larray
li $5, 15
...
li $4, 0
jal exit
.end main
.data
Larray:
.long 51, 491, 3991
Assembling Programs

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

```
00100000001
00100000010
00010000100
...
```

```
10100010000
10110000011
00100010101
...
```

```
00100000001
00100000010
00010000100
...
```

```
10100010000
10110000011
00100010101
...
```

- Data Memory
- Program Memory
- CPU
- ALU
- Control
- Registers
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
Symbols and References

math.c

```c
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;
}
```

(extern == defined in another file)

**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
Handling forward references

Example:

\[
\begin{align*}
\text{bne } & \$1, \$2, \text{ L} & \quad \text{Looking for L} \\
\text{sll } & \$0, \$0, 0 \\
\text{L: addiu } & \$2, \$3, 0x2 & \quad \text{Found L}
\end{align*}
\]

The assembler will change this to

\[
\begin{align*}
\text{bne } & \$1, \$2, +1 \\
\text{sll } & \$0, \$0, 0 \\
\text{addiu } & \$2, \$3, 0x2
\end{align*}
\]

Final machine code

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

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

<table>
<thead>
<tr>
<th>SYMBOL TABLE:</th>
<th>segment</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>l df</td>
<td><em>ABS</em> 00000000 math.c</td>
</tr>
<tr>
<td>00000000</td>
<td>l d</td>
<td>.text 00000000 .text</td>
</tr>
<tr>
<td>00000000</td>
<td>l d</td>
<td>.data 00000000 .data</td>
</tr>
<tr>
<td>00000000</td>
<td>l d</td>
<td>.bss 00000000 .bss</td>
</tr>
<tr>
<td>00000008</td>
<td>l O</td>
<td>.data 00000004 randomval</td>
</tr>
<tr>
<td>00000060</td>
<td>l F</td>
<td>.text 00000028 is_prime</td>
</tr>
<tr>
<td>00000000</td>
<td>l d</td>
<td>.rodata 00000000 .rodata</td>
</tr>
<tr>
<td>00000000</td>
<td>l d</td>
<td>.comment 00000000 .comment</td>
</tr>
<tr>
<td>00000000</td>
<td>g O</td>
<td>.data 00000004 pi</td>
</tr>
<tr>
<td>00000004</td>
<td>g O</td>
<td>.data 00000004 e</td>
</tr>
<tr>
<td>00000000</td>
<td>g F</td>
<td>.text 00000028 get_n</td>
</tr>
<tr>
<td>00000028</td>
<td>g F</td>
<td>.text 00000038 square</td>
</tr>
<tr>
<td>00000088</td>
<td>g F</td>
<td>.text 0000004c pick_prime</td>
</tr>
<tr>
<td>00000000</td>
<td></td>
<td><em>UND</em> 00000000 usrid</td>
</tr>
<tr>
<td>00000000</td>
<td></td>
<td><em>UND</em> 00000000 printf</td>
</tr>
</tbody>
</table>
Separate Compilation & Assembly

Compiler → Source files → Assembly files → Obj files → Linker → Executable program

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 MIPS simulator
Static Libraries

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

Q: Every program contains the entire library?!?
Linker Example: Resolving an External Fn Call

main.o

... 0C000000
40 21035000
44 1b80050C
48 8C040000
50 21047002
... 0C000000
... 00 T main
00 D usrid
*UND* printf
*UND* pi
*UND* get_n
40,JAL, printf
... 54,JAL, get_n

math.o

... 21032040
28 0C000000
2C 1b301402
30 3C040000
34 34040000
...
20 T get_n
00 D pi
*UND* printf
*UND* usrid
28,JAL, printf

printf.o

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

sum.exe

... 21032040
0C40023C
1b301402
3C041000
34040004
...
0040 0100
0C40023C
21035000
1b80050C
8C040000
21047002
...
00 T main
00 D pi
*UND* printf
*UND* usrid
28,JAL, printf

printf

... 10201000
21040330
22500102
...
1000 0000
... global variables
go here (later)

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 MIPS assembly, pseudo-instructions, directives, etc.)

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

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

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