Assembly code and the machine model

David Marchant
Based on slides by Troels Henriksen

2022-09-07

Followups from last lecture

- `int main()` -> `int main(void)`
- but `int main(int argc, char* argv[])` is more common anyway...
- make and c are different
- make uses a makefile to automate commands
- c is compiled into runnable code via gcc/clang/other

Agenda

Machine execution

- Computer architectures
- Registers and instructions
- From C to RISC-V

Memory

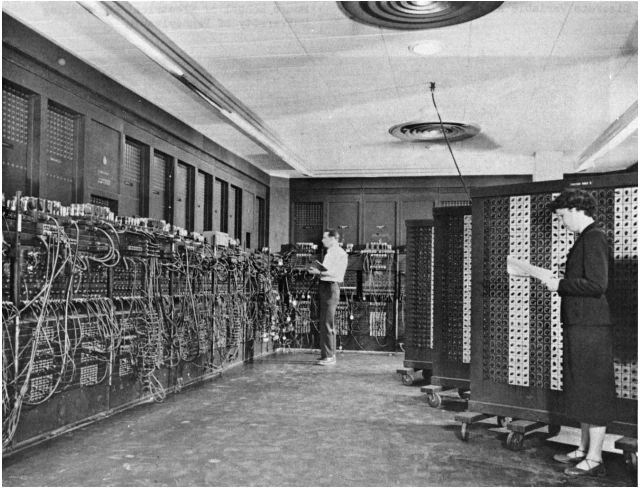
- Byte-addressable memory
- In RISC-V

Control flow

- Conditionals
- Loops

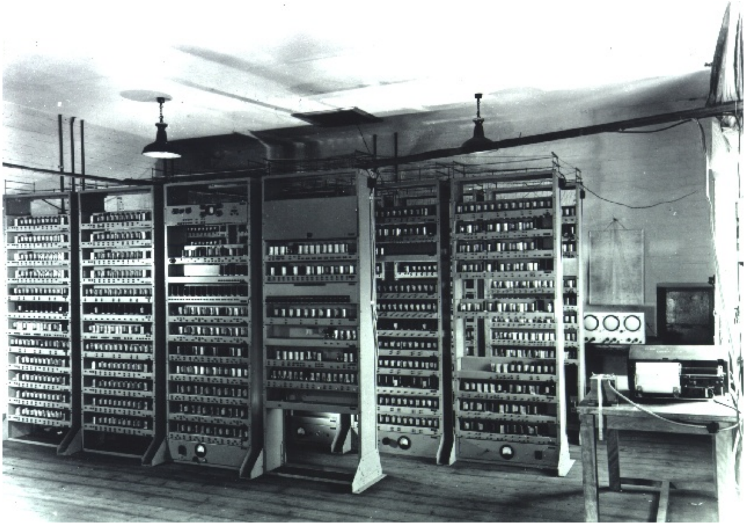
Programmable computers

Early computers were programmed by physically moving wires and switches.



ENIAC, 1947-1955, first programmable, electronic, general-purpose, digital computer.

Stored-program computers



EDSAC, 1949-1958, one of the first stored-program computers.

Machine code

Machine Code

A machine-readable sequence of *instructions* that cause the computer to change its state.

- A primitive programming language directly implemented by hardware.
- We use a textual surface syntax called *assembly code*.
- An *assembler* turns human-readable assembly code into actual machine code.

Types of machine code

- Early machine code languages were intended for human programming and had many conveniences and even high-level features.
- Each *architecture* has its own machine code; new machines often had new ones with more features.

PDP-7, 1965, a highly succesful and cheap minicomputer.

Types of machine code, continued

- Today the vast majority of programmers use *compilers* to generate machine code.
- Fairly few general-purpose architectures in use, most quite similar:

x86: Intel and AMD processors, used in most PCs and servers.

ARM: Previously mostly for low-power and mobile, now also used in Apple's newer laptops and encroaching on servers.

POWER: IBMs architecture; still in use for some high-end servers.

RISC-V: Open architecture, very new, currently mostly used for embedded purposes, but growing.

Long tail: MIPS, SPARC, Alpha, z/Architecture, GPUs, DSPs, ...

Types of machine code, continued

- Today the vast majority of programmers use *compilers* to generate machine code.
- Fairly few general-purpose architectures in use, most quite similar:

x86: Intel and AMD processors, used in most PCs and servers.

ARM: Previously mostly for low-power and mobile, now also used in Apple's newer laptops and encroaching on servers.

POWER: IBMs architecture; still in use for some high-end servers.

RISC-V: Open architecture, very new, currently mostly used for embedded purposes, but growing.

Long tail: MIPS, SPARC, Alpha, z/Architecture, GPUs, DSPs, ...

You will be taught RISC-V/32

- Concepts generalise to all other general-purpose architectures.
- ...but they might be uglier elsewhere.

Registers and instructions

The two main concepts are **registers** and **instructions**

```
add x1, x2, x3
```

- `add` is the instruction type.
- `x1`, `x2`, `x3` are names of registers.
- Add the values in the registers `x2` and `x3` and put the result in `x1`.
- **A register is basically a variable implemented in hardware.**
- **An instruction is basically a simple function implemented in hardware.**
- RISC-V/32 exposes a *fixed* number of registers with a *fixed* size (32 bits).
- ...and a fixed number of instructions.

Registers as variables

We can see the RISC-V machine as having 32 32-bit integer variables.

```
int32_t x0, x1, x2, x3, x4, x5, x6, x7 x8, x9, x10, x11,
x12, x13, x14, x15 x16, x17, x18, x19, x20, x21, x22, x23
x23, x24, x25, x26, x27, x28, x29, x30, x31;
```

Each instruction then changes these variables.

Instruction	**Meaning**
add x_i, x_j, x_k	$x_i = x_j + x_k$
sub x_i, x_j, x_k	$x_i = x_j - x_k$
addi x_i, x_j, v	$x_i = x_j + v$
and x_i, x_j, x_k	$x_i = x_j \mathbin{\&} x_k$
andi x_i, x_j, v	$x_i = x_j \mathbin{\&} v$

Playing with a RISC-V interpreter

`https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/`

Machine execution

- Computer architectures
- Registers and instructions
- From C to RISC-V

Memory

- Byte-addressable memory
- In RISC-V

Control flow

- Conditionals
- Loops

From C to RISC-V

A compiler must decide in which registers to store variables.

```
int a, b, c, d, e;
```

C variable:	a	b	c	d	e
RISC-V register:	x1	x2	x3	x4	x5

(More variables than registers? We'll talk about that later.)

From C to RISC-V

A compiler must decide in which registers to store variables.

```
int a, b, c, d, e;
```

C variable:	a	b	c	d	e
RISC-V register:	x1	x2	x3	x4	x5

(More variables than registers? We'll talk about that later.)

Assembly instructions are very simple, so high-level languages must break up complex expressions, which often requires extra registers.

```
e = (a + b) - (c + d);
```

From C to RISC-V

A compiler must decide in which registers to store variables.

```
int a, b, c, d, e;
```

C variable:	a	b	c	d	e
RISC-V register:	x1	x2	x3	x4	x5

(More variables than registers? We'll talk about that later.)

Assembly instructions are very simple, so high-level languages must break up complex expressions, which often requires extra registers.

```
e = (a + b) - (c + d);
```

```
add x6, x1, x2      # a + b
add x7, x3, x4      # c + d
sub x5, x6, x7      # (a + b) - (c + d)
```

Register names

- The RISC-V registers have names and designated uses.

Register	Name	Intended use
`x0`	`zero`	The constant value 0 (writes ignored)
`x1`	`ra`	Return address
`x2`	`sp`	Stack pointer
`x4`	`gp`	Global pointer
`x5-x7`	`t0-t2`	Temporaries
`x8-x9`	`s0-s1`	Saved registers
`x10-x11`	`a0-a1`	Arguments/return values
`x12-x17`	`a2-a7`	Function arguments
`x18-x27`	`s2-s11`	Saved registers
`x28-x31`	`t3-t6`	Temporaries

- With a few exceptions, these uses are just **conventions**, not enforced by machine.
- We'll see why the conventions are useful in the next lecture.

Pseudo instructions

Pseudo instruction

An instruction allowed in the assembly syntax, but translated into one or more other instructions by the assembler.

- Think of them as shortcuts.
- Unfortunately, the browser RISC-V interpreter does not allow most pseudoinstructions.

mv x_i, x_j	Move	

Pseudo instructions

Pseudo instruction

An instruction allowed in the assembly syntax, but translated into one or more other instructions by the assembler.

- Think of them as shortcuts.
- Unfortunately, the browser RISC-V interpreter does not allow most pseudoinstructions.

`mv` x_i, x_j	Move	`addi` x_i, x_j, `0`
`li` x_i, k	Load immediate	

Pseudo instructions

Pseudo instruction

An instruction allowed in the assembly syntax, but translated into one or more other instructions by the assembler.

- Think of them as shortcuts.
- Unfortunately, the browser RISC-V interpreter does not allow most pseudoinstructions.

`mv` x_i, x_j	Move	`addi` x_i, x_j, `0`
`li` x_i, k	Load immediate	`addi` x_i, `zero`, k
`neg` x_i	Negate	

Pseudo instructions

Pseudo instruction

An instruction allowed in the assembly syntax, but translated into one or more other instructions by the assembler.

- Think of them as shortcuts.
- Unfortunately, the browser RISC-V interpreter does not allow most pseudoinstructions.

`mv` $\mathtt{x}_i$, $\mathtt{x}_j$	Move	`addi` $\mathtt{x}_i$, $\mathtt{x}_j$, `0`
`li` $\mathtt{x}_i$, k	Load immediate	`addi` x_i, `zero`, k
`neg` $\mathtt{x}_i$	Negate	`sub` $\mathtt{x}_i$, `zero`, $\mathtt{x}_j$
`nop`	No operation	

Pseudo instructions

Pseudo instruction

An instruction allowed in the assembly syntax, but translated into one or more other instructions by the assembler.

- Think of them as shortcuts.
- Unfortunately, the browser RISC-V interpreter does not allow most pseudoinstructions.

`mv` $\texttt{x}_i$, $\texttt{x}_j$	Move	`addi` $\texttt{x}_i$, $\texttt{x}_j$, `0`
`li` $\texttt{x}_i$, k	Load immediate	`addi` x_i, `zero`, k
`neg` $\texttt{x}_i$	Negate	`sub` $\texttt{x}_i$, `zero`, $\texttt{x}_j$
`nop`	No operation	`add zero, zero, zero`

Why are they not implemented directly in hardware?

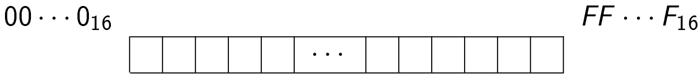
Compiler Explorer

`https://godbolt.org/`

Remember to set the compiler to "RISC-V rv32gc clang" (or some other RISC-V).

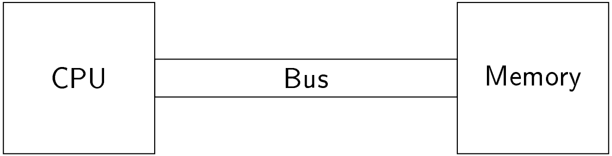
Byte-oriented memory organisation

- Registers are for scratch space; data is primarily stored in *memory*.



- **Programs refer to data by address**
  - Conceptually, envision as large array of bytes.
    - It's not really, but it works as a semantic model.
  - An address is like an index into that array.
    - A *pointer* stores an address.
    - **Addresses are ultimately just unsigned integers.**
- **System provides private address space to each *process*.**
  - You'll learn how in a few weeks; just trust me for now.

The Von Neumann Bottleneck



- **Reading:** CPU sends address to memory.
  - Memory responds with contents at address.
- **Writing:** CPU sends address and data to memory.
  - Memory overwrites location with new contents.
- **The bus is slow!**

The distance between computation and storage is the main performance obstacle in most programs.

Machine words

Any given computer has a "word size".

- "Native" size of integer-valued data.
  - But especially of memory addresses.
- 32-bit machines used to be the norm and are still found (e.g. **RISC-V/32**).
  - 2^{32} different addresses, meaning $4\,GiB$ can be addressed.
- 64-bit machines are most common.
  - 2^{64} different addresses, meaning $18\,EiB$ can be addressed.
  - $18.4 \cdot 10^{18}$ bytes.
  - Current machines only use lower 48 bits of address.
- Machines also support other data formats.
  - Fractions or multiples of word size.
  - Always integral number of types.
  - Smaller types (e.g. 16-bit integers) take less space in memory, but are (usually) not faster than the "native" words.
  - But bigger types (e.g. 128-bit integers) are slower.

Word-oriented memory organisation

- **Addresses specify byte locations**
  - Address of first byte in word.
  - Addresses of successive words differ by 4 (32 bit) or 8 (64 bit).
  - *Addresses always refer to a byte* even when addressing larger types.
- **We can take the address of any variable in a C program**
  - `&x` gives us the address of `x`.
  - If `x` has type `T`, then `&x` has type `T*`.

Byte ordering

- **So, how are the bytes within a multi-byte word ordered in memory?**
  - Most significant byte at lowest address, or least significant byte at lowest address?
- **Conventions**
  - Big endian: SPARC, POWER, Internet protocols.
    - Most significant byte has lowest address (“comes first”).
  - Little endian: x86, ARM (mostly), RISC-V.
  - Most significant byte has highest address (“comes last”).

Byte ordering example

- **Example**
  - Variable has 4-byte value of `0x01234567`.
  - Address `&x` is `0x100`.
    - No matter what, the address of an object is always the address of the *first* byte in the object (counting from lowest addresses).

Big endian

0x0fe	0x0ff	0x100	0x101	0x102	0x103	0x104	0x105
		01	23	45	67		

Little endian

0x0fe	0x0ff	0x100	0x101	0x102	0x103	0x104	0x105
		67	45	23	01		

Byte ordering example

- **Example**
  - Variable has 4-byte value of `0x01234567`.
  - Address `&x` is `0x100`.
    - No matter what, the address of an object is always the address of the *first* byte in the object (counting from lowest addresses).

Big endian

0x0fe	0x0ff	0x100	0x101	0x102	0x103	0x104	0x105
		01	23	45	67		

Little endian

0x0fe	0x0ff	0x100	0x101	0x102	0x103	0x104	0x105
		67	45	23	01		

Important note

This difference is *not visible* unless you start decomposing integers as bytes with memory operations. Bit-shifting etc. always acts as expected.

An M-byte memory is conceptually an array

```
byte Memory[M];
```

An M-byte memory is conceptually an array

```
byte Memory[M];
```

Main difference between registers and memory

Memory is *dynamically addressable*, while the registers we operate on are statically encoded in instructions.

- In a **stored program computer**, *instructions* are also just data stored in memory (4 bytes per instruction in RISC-V).
- In a **von Neumann** computer, instructions are stored in the *same memory* as all other data.
  - Allows for self-modifying code (don't do this).
- **Harvard architectures** store instructions in a separate (usually read-only) memory.

Memory transfer instructions

Note how the *dynamic contents* of a register influences *which* memory address we access.

Instruction	Meaning
`lb` $\texttt{x}_i$, $v(\texttt{x}_j)$	$\texttt{x}_i$ = `Memory`[$\texttt{x}_j + v$]
`sb` $\texttt{x}_i$, $v(\texttt{x}_j)$	`Memory`[$\texttt{x}_j + v$] = $\texttt{x}_i$
`lw` $\texttt{x}_i$, $v(\texttt{x}_j)$	$\texttt{x}_i$ = `Memory`[$\texttt{x}_j + v$]
`sw` $\texttt{x}_i$, $v(\texttt{x}_j)$	`Memory`[$\texttt{x}_j + v$] = $\texttt{x}_i$

Program Counter

The *program counter* (PC) is a special register that contains the address of the current instruction in memory.

Address	**Contents (in assembly syntax)**
`0x100`	`add x6, x1, x2`
`0x104`	`add x7, x3, x4`
`0x108`	`sub x5, x6, x7`

- Every *non-jump* instruction implicitly increments PC by 4.

Program Counter

The *program counter* (PC) is a special register that contains the address of the current instruction in memory.

	Address	Contents (in assembly syntax)
→	`0x100`	`add x6, x1, x2`
	`0x104`	`add x7, x3, x4`
	`0x108`	`sub x5, x6, x7`

- Every *non-jump* instruction implicitly increments PC by 4.

Program Counter

The *program counter* (PC) is a special register that contains the address of the current instruction in memory.

	Address	Contents (in assembly syntax)
	`0x100`	`add x6, x1, x2`
→	`0x104`	`add x7, x3, x4`
	`0x108`	`sub x5, x6, x7`

- Every *non-jump* instruction implicitly increments PC by 4.

Program Counter

The *program counter* (PC) is a special register that contains the address of the current instruction in memory.

	Address	Contents (in assembly syntax)
	`0x100`	`add x6, x1, x2`
	`0x104`	`add x7, x3, x4`
$\rightarrow$	`0x108`	`sub x5, x6, x7`

- Every *non-jump* instruction implicitly increments PC by 4.

Control flow

What makes computers interesting is their ability to make decisions based on data.

- Machine code does not support structured control flow (`if`, `while`, `for`, etc).
- Instead we have *conditional jumps*.

Control flow

What makes computers interesting is their ability to make decisions based on data.

- Machine code does not support structured control flow (`if`, `while`, `for`, etc).
- Instead we have *conditional jumps*.

```
if (x1 < x2) {
  x3 = x1;
} else {
  x3 = x2;
}
```

Control flow

What makes computers interesting is their ability to make decisions based on data.

- Machine code does not support structured control flow (`if`, `while`, `for`, etc).
- Instead we have *conditional jumps*.

```
if (x1 < x2) {
  x3 = x1;
} else {
  x3 = x2;
}
```

```
blt x1, x2, L0
addi x3, x2, 0
jal x0, L1
L0:
addi x3, x1, 0
L1:
```

Instruction	Meaning
blt x_i, x_j, k	if (x_i < x_j) PC += k
jal x_i, k	x_i = PC + 4; PC += k

(Assembler automatically inserts right offset k when we use a label.)

Machine execution

- Computer architectures
- Registers and instructions
- From C to RISC-V

Memory

- Byte-addressable memory
- In RISC-V

Control flow

- Conditionals
- Loops

Loops are implemented by jumping *backwards*

- Suppose we want to implement $c = a \times b$. Assume:
  - c is in register `a0`.
  - a is in register `a1`.
  - b is in register `a2`.
- We'll implement it by *repeated addition* of a (b times).

Loops are implemented by jumping *backwards*

- Suppose we want to implement $c = a \times b$. Assume:
  - c is in register `a0`.
  - a is in register `a1`.
  - b is in register `a2`.
- We'll implement it by *repeated addition* of a (b times).

```
addi a0, a1, 0      # initialise a0 = a1
```

Loops are implemented by jumping *backwards*

- Suppose we want to implement $c = a \times b$. Assume:
  - c is in register `a0`.
  - a is in register `a1`.
  - b is in register `a2`.
- We'll implement it by *repeated addition* of a (b times).

```
addi a0, a1, 0       # initialise a0 = a1
LOOP:                # loop label
```

Loops are implemented by jumping *backwards*

- Suppose we want to implement $c = a \times b$. Assume:
  - c is in register `a0`.
  - a is in register `a1`.
  - b is in register `a2`.
- We'll implement it by *repeated addition* of a (b times).

```
addi a0, a1, 0      # initialise a0 = a1
LOOP:               # loop label
beq a2, zero, END   # jump to end if no iterations left
```

Loops are implemented by jumping *backwards*

- Suppose we want to implement $c = a \times b$. Assume:
  - c is in register `a0`.
  - a is in register `a1`.
  - b is in register `a2`.
- We'll implement it by *repeated addition* of a (b times).

```
addi a0, a1, 0      # initialise a0 = a1
LOOP:               # loop label
beq a2, zero, END   # jump to end if no iterations left
addi a2, a2, -1     # decrement b
```

Loops are implemented by jumping *backwards*

- Suppose we want to implement $c = a \times b$. Assume:
  - c is in register `a0`.
  - a is in register `a1`.
  - b is in register `a2`.
- We'll implement it by *repeated addition* of a (b times).

```
addi a0, a1, 0      # initialise a0 = a1
LOOP:               # loop label
beq a2, zero, END   # jump to end if no iterations left
addi a2, a2, -1     # decrement b
add a0, a0, a1      # add a to c
```

Loops are implemented by jumping *backwards*

- Suppose we want to implement $c = a \times b$. Assume:
  - c is in register `a0`.
  - a is in register `a1`.
  - b is in register `a2`.
- We'll implement it by *repeated addition* of a (b times).

```
addi a0, a1, 0      # initialise a0 = a1
LOOP:               # loop label
beq a2, zero, END   # jump to end if no iterations left
addi a2, a2, -1     # decrement b
add a0, a0, a1      # add a to c
jal zero, LOOP      # try again
```

Loops are implemented by jumping *backwards*

- Suppose we want to implement $c = a \times b$. Assume:
  - c is in register `a0`.
  - a is in register `a1`.
  - b is in register `a2`.
- We'll implement it by *repeated addition* of a (b times).

```
addi a0, a1, 0      # initialise a0 = a1
LOOP:               # loop label
beq a2, zero, END   # jump to end if no iterations left
addi a2, a2, -1     # decrement b
add a0, a0, a1      # add a to c
jal zero, LOOP      # try again
END:                # loop end
```

Loops are implemented by jumping *backwards*

- Suppose we want to implement $c = a \times b$. Assume:
  - c is in register `a0`.
  - a is in register `a1`.
  - b is in register `a2`.
- We'll implement it by *repeated addition* of a (b times).

```
addi a0, a1, 0     # initialise a0 = a1
LOOP:              # loop label
beq a2, zero, END  # jump to end if no iterations left
addi a2, a2, -1    # decrement b
add a0, a0, a1     # add a to c
jal zero, LOOP     # try again
END:               # loop end
```

...is this correct for all a and b?

Fibonacci Numbers

```
int n = 10;
int a = 1;
int b = 1;
int i = 0;
while (n != 0) {
  out[i] = b;
  i = i + 1;
  int tmp = a + b;
  a = b;
  b = tmp;
  n = n - 1;
}
```

Fibonacci Numbers

```c
int n = 10;
int a = 1;
int b = 1;
int i = 0;
while (n != 0) {
  out[i] = b;
  i = i + 1;
  int tmp = a + b;
  a = b;
  b = tmp;
  n = n - 1;
}
```

```asm
addi a0, zero, 10
addi t0, zero, 1
addi t1, zero, 1
addi t2, zero, 0
LOOP:
beq a0, zero, DONE
addi a0, a0, -1
sw t1, 0(t2)
addi t2, t2, 4
add t3, t0, t1
add t0, zero, t1
add t1, zero, t3
jal zero, LOOP
DONE:
```

Takeaways

- Instructions operate on data stored in **registers**.
- Load/store instructions ferry data between registers and byte-addressed **memory**.
- Branch/jump instructions move the **instruction pointer**.
- Assembly is somewhat tedious but fundamentally **simple**.
- **Hint:** when you have to write an assembly program, consider first writing it in C(-ish) syntax with a single "instruction" per statement, and translate from there.