CSCI 210: Computer Architecture
Lecture 8: Computer Representation of MIPS Instructions

Stephen Checkoway

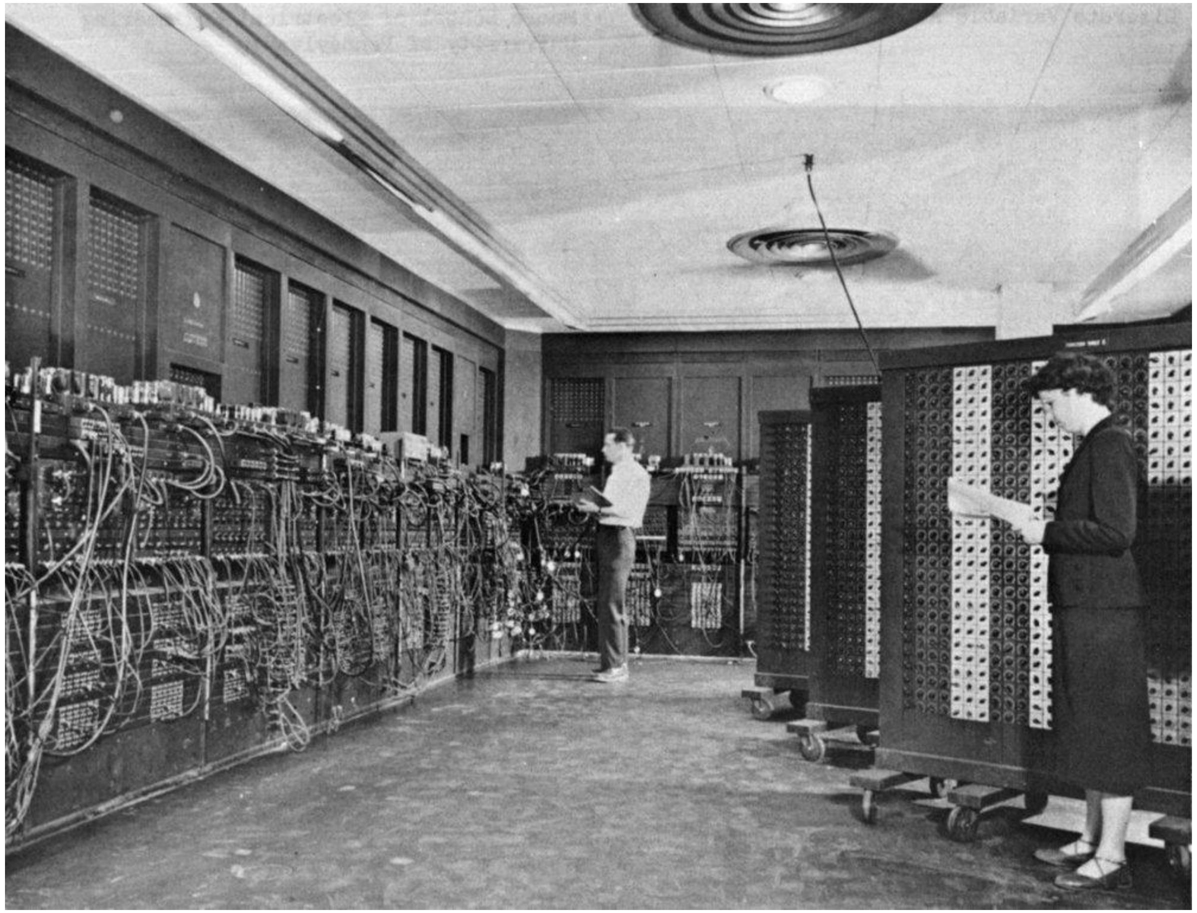
Oberlin College

Slides from Cynthia Taylor

Announcements

- Problem Set 2 due Friday
- Lab 1 due Sunday

CS History: ENIAC



U.S. Army photo of ENIAC

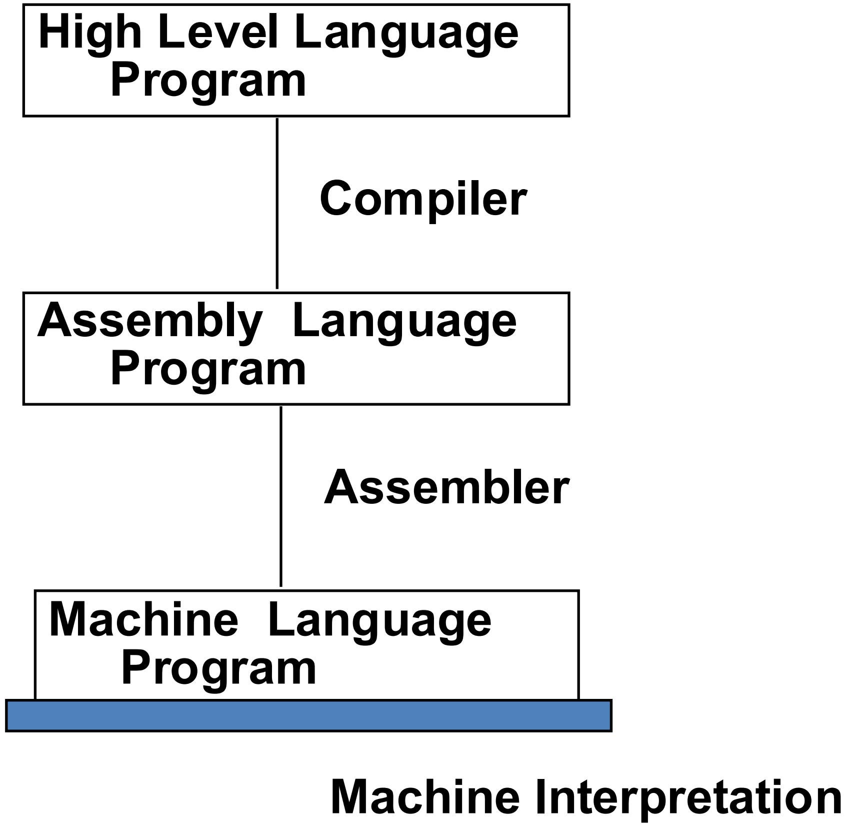
- Electronic Numerical Integrator And Computer
- First programmable, electronic, general-purpose computer
- Created by the US Army in 1945
- Designed to compute ballistic tables during WWII
- Originally didn't have storage
- Decimal, not binary!

CS History: ENIAC

- Programmers were Kay McNulty, Jean Bartok, Betty Snyder, Marlyn Meltzer, Fran Bilas, and Ruth Lichterman.
- Selected from a group of 200 women employed hand calculating equations for the army
- Programmed by connecting components with cables and setting switches
- Kay McNulty developed the use of subroutines
- Betty Snyder and Jean Bartok went on to help develop the first commercial computers

U.S. Army photo

How to Speak Computer




```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw  $15,   0($2)
lw  $16,   4($2)
sw  $16,   0($2)
sw  $15,   4($2)
```

```
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
```

Two Key Principles of Machine Design

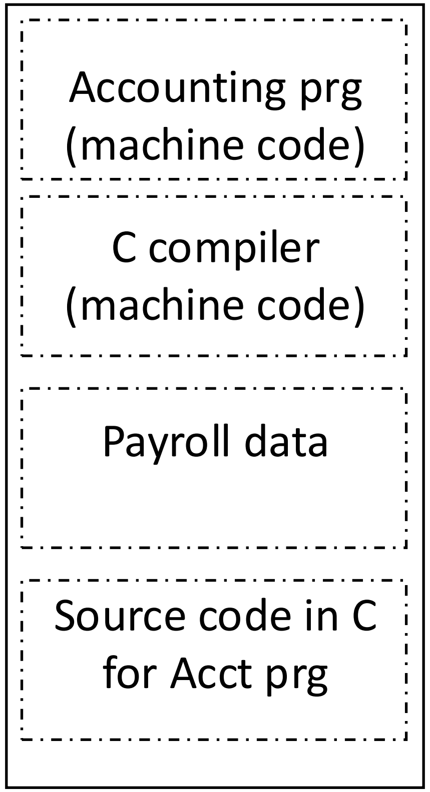
1. Instructions are represented as numbers and, as such, are indistinguishable from data

2. Programs are stored in alterable memory (that can be read or written to) just like data

Stored-program concept

- Programs can be shipped as files of binary numbers – binary compatibility
- Computers can inherit ready-made software provided they are compatible with an existing ISA and OS – leads industry to align around a small number of ISAs

Memory



What happens if someone writes new machine code in the memory where your program is stored, overwriting your program?

A. The program will crash.

B. The old instructions will run.

C. The new instructions will run.

D. None of the above

Recall: Instruction Set Architecture

- Definition of how to access the hardware from software
- Supported instructions, registers, etc . . .

Key ISA decisions

- operations
  - how many?
  - which ones
- operands
  - how many?
  - location
  - types
- instruction format
  - size
  - how many formats?

destination operand *operation*

y = x + b

source operands

add   r1, r2, r5

how does the computer know what
0001 0100 1101 1111
means?

RISC versus CISC (Historically)

- Complex Instruction Set Computing
  - Larger instruction set
  - More complicated instructions built into hardware
  - Variable number of clock cycles per instruction

- Reduced Instruction Set Computing
  - Small, highly optimized set of instructions
  - Memory accesses are specific instructions
  - One instruction per clock cycle (only the very first RISCs!)

A = A*B

RISC (MIPS-esque)

```
lw    $t0, 0(A)
lw    $t1, 0(B)
mul   $s1, $t0, $t1
sw    $s1, 0(A)
```

CISC

```
mul   B, A
```

Which of these is faster?

RISC

```
lw    $t0, 0(A)
lw    $t1, 0(B)
mul   $s1, $t0, $t1
sw    $s1, 0(A)
```

CISC

```
mul   B, A
```

RISC vs CISC

RISC

- More work for compiler/assembly programmer
- More RAM used to store instructions
- Less complex hardware

CISC

- Less work for compiler/assembly programmer
- Fewer instructions to store
- More complex hardware

So . . . Which System “Won”?

- Most processors are RISC
- BUT the x86 (Intel) is CISC
- x86 breaks down CISC assembly into multiple, RISC-like, machine language instructions
- Distinction between RISC and CISC is less clear
  - Some RISC instruction sets have more instructions than some CISC sets

The computer figures out what format an instruction is from

A. Codes embedded in the instruction itself.

B. A special register that is loaded with the instruction.

C. It tries each format and sees which one forms a valid instruction.

D. None of the above

Instruction Formats What does each bit mean?

- Having many different instruction formats...
  - complicates decoding
  - uses more instruction bits (to specify the format)

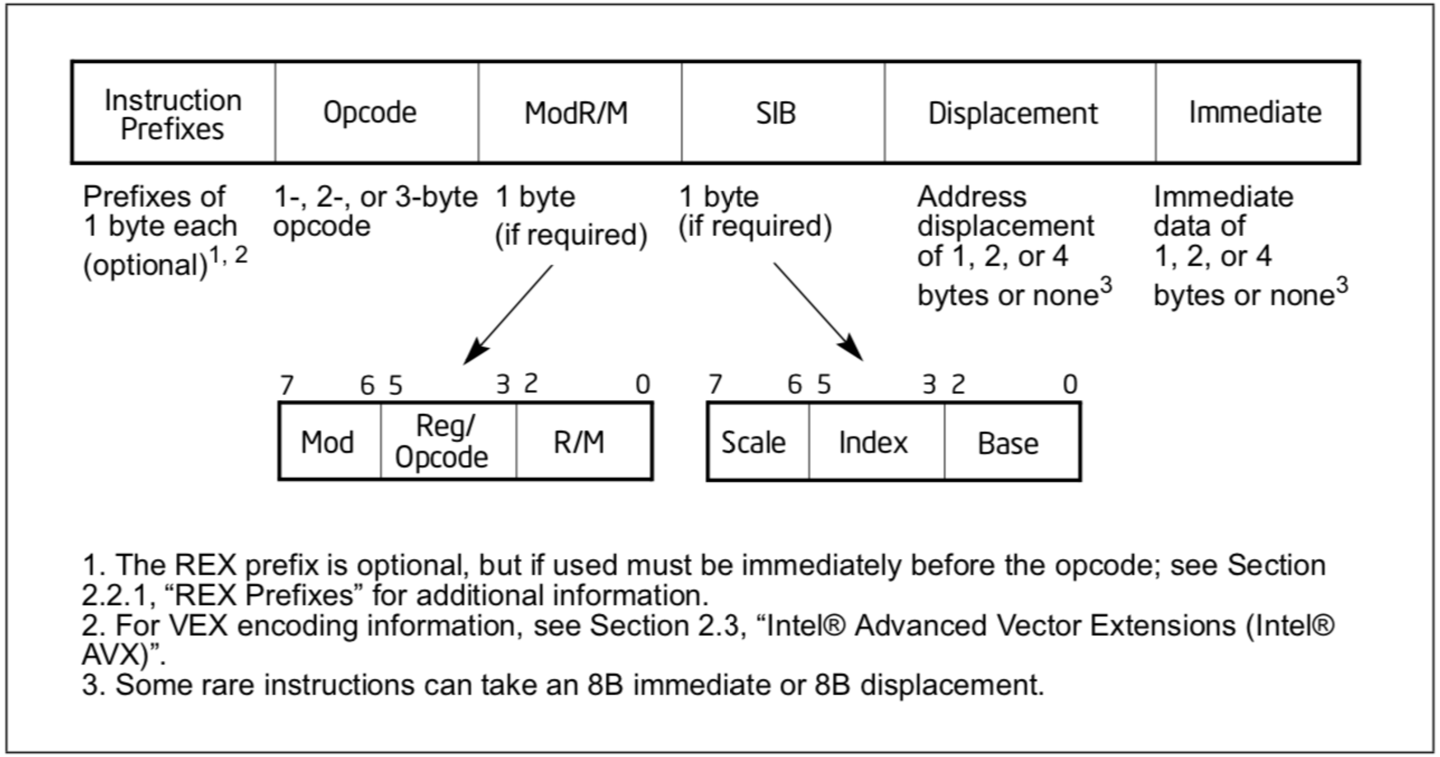


Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

x86-64 example

Encoding	Instruction
01 d8	add eax, ebx
48 01 d8	add rax, rbx
48 03 03	add rax, qword ptr [rbx]
48 03 04 8b	add rax, qword ptr [rbx + 4*rcx]
48 03 44 8b 18	add rax, qword ptr [rbx + 4*rcx + 0x18]

REX prefix specifying 64-bit registers

Opcode specifying the instruction

ModR/M specifying the operands (including reg vs. mem)

SIB specifying the scale, index register, and base register

Displacement (offset)

Representing Instructions

- MIPS instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R-type	**opcode**	**rs**	**rt**	**rd**	**sa**	**funct**
I-type	**opcode**	**rs**	**rt**	**immediate**		
J-type	**opcode**	**target**				

MIPS Instruction Fields for R-type

- MIPS fields are given names to make them easier to refer to

op	rs	rt	rd	shamt	funct

op	6-bits	opcode that specifies the operation
rs	5-bits	register file address of the first source operand
rt	5-bits	register file address of the second source operand
rd	5-bits	register file address of the result’s destination
shamt	5-bits	shift amount (for shift instructions)
funct	6-bits	function code augmenting the opcode

MIPS Arithmetic Instructions Format

$$t0 = s1 - s2$$

```
sub $t0, $s1, $s2
```

opcode	rs	rt	rd	sa	funct
0	17	18	8	0	0x22

R-format Example

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

```
add $t0, $s1, $s2
```

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	$0 / 20_{hex}$
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		$0 / 21_{hex}$

NAME	NUMBER	USE
$zero	0	The Constant Value 0
$at	1	Assembler Temporary
$v0-$v1	2-3	Values for Function Results and Expression Evaluation
$a0-$a3	4-7	Arguments
$t0-$t7	8-15	Temporaries
$s0-$s7	16-23	Saved Temporaries
$t8-$t9	24-25	Temporaries
$k0-$k1	26-27	Reserved for OS Kernel
$gp	28	Global Pointer
$sp	29	Stack Pointer
$fp	30	Frame Pointer
$ra	31	Return Address

Convert this MIPS machine instruction to assembly:

`000000 01110 10001 10010 00000 100010`

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Selection	Instruction
A	`add $s2, $t7, $s4`
B	`add $s1, $t6, $s3`
C	`sub $t6, $s1, $s2`
D	`sub $s2, $t6, $s1`
E	None of the above

MIPS I-format Instructions

op	rs	rt	constant or offset
6 bits	5 bits	5 bits	16 bits

- Immediate arithmetic and load/store instructions
  - rt: destination (for arithmetic or lw) or source register number (for sw)
  - Constant: -2^{15} to $+2^{15}-1$ (or 0 to $2^{16}-1$ for some instructions)
  - Offset: offset added to base address in rs for lw/sw

Machine Language – I Format

op	rs	rt	constant or offset
6 bits	5 bits	5 bits	16 bits

- Load/Store Instruction Format:

```
lw $t0, 24($s3)
```

Load Linked	ll	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	nor	R	R[rd] = ~ (R[rs] | R[rt])		$0 / 27_{hex}$

NAME	NUMBER	USE
$zero	0	The Constant Value 0
$at	1	Assembler Temporary
$v0-$v1	2-3	Values for Function Results and Expression Evaluation
$a0-$a3	4-7	Arguments
$t0-$t7	8-15	Temporaries
$s0-$s7	16-23	Saved Temporaries
$t8-$t9	24-25	Temporaries
$k0-$k1	26-27	Reserved for OS Kernel
$gp	28	Global Pointer
$sp	29	Stack Pointer
$fp	30	Frame Pointer
$ra	31	Return Address

Machine Language – I Format

op	rs	rt	constant or offset
6 bits	5 bits	5 bits	16 bits

- Immediate Addition Instruction Format:

```
addi $t0, $s3, 26
```

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	$0 / 20_{hex}$
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		$0 / 21_{hex}$

NAME	NUMBER	USE
$zero	0	The Constant Value 0
$at	1	Assembler Temporary
$v0-$v1	2-3	Values for Function Results and Expression Evaluation
$a0-$a3	4-7	Arguments
$t0-$t7	8-15	Temporaries
$s0-$s7	16-23	Saved Temporaries
$t8-$t9	24-25	Temporaries
$k0-$k1	26-27	Reserved for OS Kernel
$gp	28	Global Pointer
$sp	29	Stack Pointer
$fp	30	Frame Pointer
$ra	31	Return Address

Convert this MIPS assembly instruction to machine code

```
sw $t0, 32($s6)
```

Selection	Instruction
A	010101 11011 00100 0000 0000 0010 0000
B	101011 01000 10110 0000 0000 0010 0000
C	101011 10110 01000 0000 0000 0010 0000
D	000000 00010 00000 1010 1110 1100 1000
E	None of the above

Sign-extend vs. zero-extend

op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

- The immediate field of an I-format instruction is either sign-extended or zero-extended
  - sign extension: the sign bit (bit 15) is copied into bits 31–16
  - zero extension: 0 is placed into bits 31–16
- Opcode determines which occurs

Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		$0 / 21_{hex}$
And	and	R	R[rd] = R[rs] & R[rt]		$0 / 24_{hex}$
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}

Questions about Machine Instructions?

Reading

- Next lecture: Bitwise Operations
  - Section 2.7

- Problem Set 2 due Friday

- Lab 1 due Sunday