Warm Up Problem

Convert the number -15 into a single byte two's complement notation. What is the corresponding interpretation of this binary encoding as an unsigned integer?

Note: In this course you are expected to use Linux. See the CS 241 common webpage for more information.

CS 241 Lecture 2

Characters and Machine Language

With thanks to Brad Lushman, Troy Vasiga and Kevin Lanctot

Bytes as Characters

- ASCII (American Standard Code for Information Interchange) uses 7 bits to represent characters (see next table)
- Extended ASCII uses the 8th bit (but there are lots of compatibility issues here)
- Note that `'a'` is different than `0xa` (the former is the decimal number 97 in ASCII and the latter is just the number 10 in decimal).
- Unicode extends the above to many different types of character sets.
- We will use ASCII throughout.

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

Highlights

- Characters 0-31 are control characters
- Characters 48-57 are the numbers 0 to 9
- Characters 65-90 are the letters A to Z
- Characters 97-122 are the letters a to z
- Note that 'A' and 'a' are 32 letters away

Bit-Wise Operators

Suppose we have `unsigned char a=5, b=3;`. Note that this is valid as C will interpret 5 as the character corresponding to 00000101 and similar for 3 with the bit string 00000011.

- Bitwise not $\sim$, for example `c = ~a;` gives $c = 11111010$
- Bitwise and &, for example `c = a&b;` gives $c = 00000001$
- Bitwise or $|$, for example `c = a|b;` gives $c = 00000111$
- Bitwise exclusive or ^, for example `c = a ^ b;` gives $c = 00000110$
- Bitwise shift right or left $>>$ and $<<$, for example `c = a >> 2;` gives $c = 00000001$ and `c = a << 3;` gives $c = 00101000$.
- (High level aside: bitshift ambiguity with signed characters - arithmetic left and right shift.)
- These can even be combined with the assignment operator!

Question

With all of these different ways of interpreting a byte, how do we know which one it is by looking at it?

Question

With all of these different ways of interpreting a byte, how do we know which one it is by looking at it?

We can't! We need to remember how the byte was intended to be interpreted when we store the byte in memory!

Computer Programs

What is a Computer Program?

Computer Programs

What is a Computer Program?

- Programs operate on data.
- Program **are** data. This is a **von Neumann architecture**; programs live in the same memory space as the data they operate on
- Programs can manipulate other programs (OSes, viruses, etc.)

Recall: We cannot distinguish without extra information whether or not some data represents instructions and which does not.

Instructions

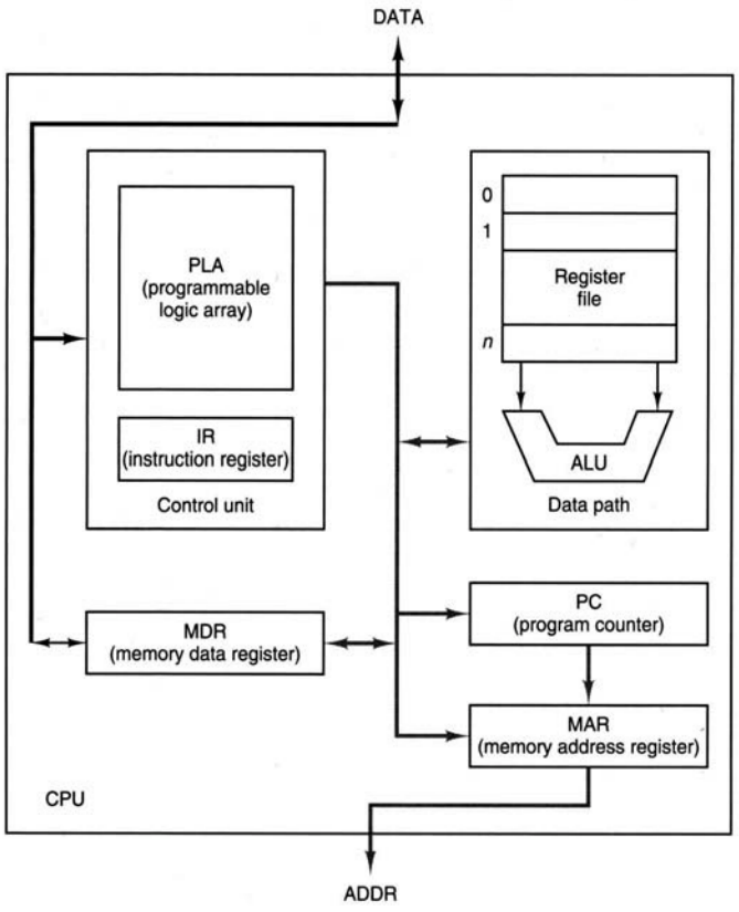
What does an instruction look like?

Instructions

What does an instruction look like?

- Many different processor specific machine languages exist.
- We will use 32-bit MIPS. (There is a 64 bit version but the underlying theory is the same)
- MIPS Machine follows on next two slides

CPU

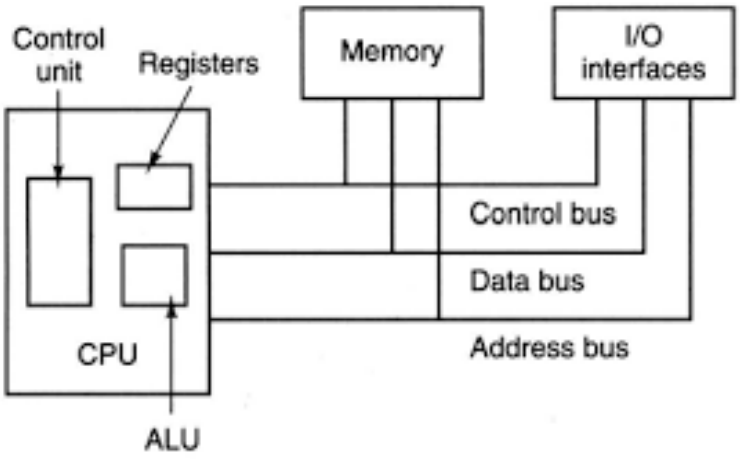


- CPU (Central Processing Unit) - brain of computer
- Registers: \$0, ..., \$31, hi, lo
- Control Unit: Decodes instrs and dispatches
  - PC (Program Counter)
  - IR (Instruction Register)
- Memory: MDR (Memory Data Register) and MAR (Memory Address Register)
- ALU (Arithmetic Logic Unit) - does arithmetic

`http://www2.cs.siu.edu/~cs401/Textbook/ch2.pdf`

"Computer Architecture", Zargham, Prentice Hall, Ch. 2

CPU with Memory



`http://www2.cs.siu.edu/~cs401/Textbook/ch2.pdf`
"Computer Architecture", Zargham, Prentice Hall, Ch. 2

- Bus: short for Latin *omnibus*. Data travels along the bus.
- Memory; many kinds. From fastest to slowest:
  - CPU/registers (fastest)
  - L1 cache
  - L2 cache
  - RAM
  - disk
  - network memory [outside your local machine] (slowest)

More on Registers

- Registers are very fast memory
- MIPS has 32 registers for general use.
- Also the registers `hi` and `lo` (for multiplication and division)
- Some of these registers are special:
  - $0 is always 0
  - $31 is for return addresses
  - $30 is our stack pointer
  - $29 is our frame pointer

High Brow Comment: In most MIPS standards the roles of $29 and $30 above are reversed.

MIPS Instructions

MIPS Reference Sheet

Basic Instruction Formats

	Register	`0000 00ss ssst tttt dddd d000 00ff ffff`	R	`s`, `t`, `d` are interpreted as unsigned
	Immediate	`oooo ooss ssst tttt iiii iiii iiii iiii`	I	`i` is interpreted as two's complement

Instructions

Word	.word i	`iiii iiii iiii iiii iiii iiii iiii iiii`		
Add	add $d, $s, $t	`0000 00ss ssst tttt dddd d000 0010 0000`	R	`$d = $s + $t`
Subtract	sub $d, $s, $t	`0000 00ss ssst tttt dddd d000 0010 0010`	R	`$d = $s - $t`
Multiply	mult $s, $t	`0000 00ss ssst tttt 0000 0000 0001 1000`	R	`hi:lo = $s * $t`
Multiply Unsigned	multu $s, $t	`0000 00ss ssst tttt 0000 0000 0001 1001`	R	`hi:lo = $s * $t`
Divide	div $s, $t	`0000 00ss ssst tttt 0000 0000 0001 1010`	R	`lo = $s / $t; hi = $s % $t`
Divide Unsigned	divu $s, $t	`0000 00ss ssst tttt 0000 0000 0001 1011`	R	`lo = $s / $t; hi = $s % $t`
Move From High/Remainder	mfhi $d	`0000 0000 0000 0000 dddd d000 0001 0000`	R	`$d = hi`
Move From Low/Quotient	mflo $d	`0000 0000 0000 0000 dddd d000 0001 0010`	R	`$d = lo`
Load Immediate And Skip	lis $d	`0000 0000 0000 0000 dddd d000 0001 0100`	R	`$d = MEM[pc]; pc = pc + 4`
Load Word	lw $t, i($s)	`1000 11ss ssst tttt iiii iiii iiii iiii`	I	`$t = MEM [$s + i]`
Store Word	sw $t, i($s)	`1010 11ss ssst tttt iiii iiii iiii iiii`	I	`MEM [$s + i] = $t`
Set Less Than	slt $d, $s, $t	`0000 00ss ssst tttt dddd d000 0010 1010`	R	`$d = 1 if $s < $t; 0 otherwise`
Set Less Than Unsigned	sltu $d, $s, $t	`0000 00ss ssst tttt dddd d000 0010 1011`	R	`$d = 1 if $s < $t; 0 otherwise`
Branch On Equal	beq $s, $t, i	`0001 00ss ssst tttt iiii iiii iiii iiii`	I	`if ($s == $t) pc += i * 4`
Branch On Not Equal	bne $s, $t, i	`0001 01ss ssst tttt iiii iiii iiii iiii`	I	`if ($s != $t) pc += i * 4`
Jump Register	jr $s	`0000 00ss sss0 0000 0000 0000 0000 1000`	R	`pc = $s`
Jump And Link Register	jalr $s	`0000 00ss sss0 0000 0000 0000 0000 1001`	R	`temp = $s; $31 = pc; pc = temp`

When a word is stored to memory location `0xffff000c`, the least-significant byte (eight bits) of the word are sent to the standard output.
Loading a word from memory location `0xffff0004` places the next byte from standard input into the least-significant byte of the destination register.

How MIPS Works

- Recall: Code is just data - it is stored in RAM.
- MIPS takes instructions from RAM and attempts to execute them.
- Recall Problem: MIPS doesn't know what in memory is an instruction *a priori*
- Solution:

How MIPS Works

- Recall: Code is just data - it is stored in RAM.
- MIPS takes instructions from RAM and attempts to execute them.
- Recall Problem: MIPS doesn't know what in memory is an instruction *a priori*
- Solution: Convention is to set memory address 0 in RAM to be an instruction.

How MIPS Works

- Recall: Code is just data - it is stored in RAM.
- MIPS takes instructions from RAM and attempts to execute them.
- Recall Problem: MIPS doesn't know what in memory is an instruction *a priori*
- Solution: Convention is to set memory address 0 in RAM to be an instruction.
- Problem: How does it know what to do next?
- Solution:

How MIPS Works

- Recall: Code is just data - it is stored in RAM.
- MIPS takes instructions from RAM and attempts to execute them.
- Recall Problem: MIPS doesn't know what in memory is an instruction *a priori*
- Solution: Convention is to set memory address 0 in RAM to be an instruction.
- Problem: How does it know what to do next?
- Solution: Have a special register called the **Program Counter** (or `PC` for short) to tell us what instruction to do next.
- Problem: How do we put our program into RAM?
- Solution:

How MIPS Works

- Recall: Code is just data - it is stored in RAM.
- MIPS takes instructions from RAM and attempts to execute them.
- Recall Problem: MIPS doesn't know what in memory is an instruction *a priori*
- Solution: Convention is to set memory address 0 in RAM to be an instruction.
- Problem: How does it know what to do next?
- Solution: Have a special register called the **Program Counter** (or `PC` for short) to tell us what instruction to do next.
- Problem: How do we put our program into RAM?
- Solution: A program called a *loader* puts our program into memory and sets the `PC` to be the first address.

Fetch-Execute Cycle

Algorithm 1 Fetch-Execute Cycle

```
1: PC = 0
2: while true do
3:     IR = MEM[PC]
4:     PC += 4
5:     Decode and execute instruction in IR
6: end while
```

Eventually an instruction will break out of the loop.
This is basically the only program a machine really runs.

First Example

Write a program in MIPS that takes in the values of registers $8 and $9 and stores the result in register $3.

First Example

Write a program in MIPS that takes in the values of registers $8 and $9 and stores the result in register $3.

`add $d, $s, $t: 0000 00ss ssst tttt dddd d000 0010 0000`

Adds the result of registers $s and $t and store these in register $d. Important! The order of $d, $s and $t are shifted in the encoding!

First Example

Write a program in MIPS that takes in the values of registers $8 and $9 and stores the result in register $3.

`add $d, $s, $t`: 0000 00ss ssst tttt dddd d000 0010 0000

Adds the result of registers $s and $t and store these in register $d. Important! The order of $d, $s and $t are shifted in the encoding!

Solution: `add $3, $8, $9`:

0000 0001 0000 1001 0001 1000 0010 0000

Putting Values In Registers

This works well *provided you already had values in registers!*

How to we put values in registers? In CS 241 we give you a non-standard MIPS command:

`lis $d`: `0000 0000 0000 0000 dddd d000 0001 0100`

Load immediate and skip. This places the next value in RAM [an immediate] into $d and increments the program counter by 4 (it skips the next line which is usually not an instruction).

`.word i`: `iiii iiii iiii iiii iiii iiii iiii iiii`.

This is an assembler directive (not a MIPS instruction). This value as a two's complement integer is placed in the correct memory location in RAM as it occurs in the code. Can also use with hexadecimal values using `0xi`.

Example

Write a MIPS program that adds together 11 and 13 and stores the result in register $3..

Example

Write a MIPS program that adds together 11 and 13 and stores the result in register $3..

```
lis $8              0000 0000 0000 0000 0100 0000 0001 0100
.word 11            0000 0000 0000 0000 0000 0000 0000 1011
lis $9              0000 0000 0000 0000 0100 1000 0001 0100
.word 0xd           0000 0000 0000 0000 0000 0000 0000 1101
add $3, $8, $9      0000 0001 0000 1001 0001 1000 0010 0000
```

The code on the left is what we call **Assembly Code**.
The code on the right is what we call **Machine Code**.

Example

Previous example revisited.
What it looks like in RAM with memory locations

```
0x00000000      0000 0000 0000 0000 0100 0000 0001 0100
0x00000004      0000 0000 0000 0000 0000 0000 0000 1011
0x00000008      0000 0000 0000 0000 0100 1000 0001 0100
0x0000000c      0000 0000 0000 0000 0000 0000 0000 1101
0x00000010      0000 0001 0000 1001 0001 1000 0010 0000
```

Incomplete examples

The two examples above are still incomplete. Recall that the fetch-execute cycle has a while loop that we still haven't exited yet. How to we return control back to the loader?

Incomplete examples

The two examples above are still incomplete. Recall that the fetch-execute cycle has a while loop that we still haven't exited yet. How to we return control back to the loader?

`jr $s`: `0000 00ss sss0 0000 0000 0000 0000 1000`

Jump Register. Sets the `pc` to be `$s`.
For us, our return address will typically be in $31 so we will typically call

```
jr $31
```

which is

`0000 0011 1110 0000 0000 0000 0000 1000`

This command returns control to the loader.

Complete Example

Write a MIPS program that adds together 11 and 13 and stores the result in register $3..

Complete Example

Write a MIPS program that adds together 11 and 13 and stores the result in register $3..

```
lis $8              0000 0000 0000 0000 0100 0000 0001 0100
.word 11            0000 0000 0000 0000 0000 0000 0000 1011
lis $9              0000 0000 0000 0000 0100 1000 0001 0100
.word 0xd           0000 0000 0000 0000 0000 0000 0000 1101
add $3, $8, $9      0000 0001 0000 1001 0001 1000 0010 0000
jr $31              0000 0011 1110 0000 0000 0000 0000 1000
```