Computer Architecture

What computer contains and how it works

November 4, 2020

Table of Contents

Table of Contents

Pre-20th Century

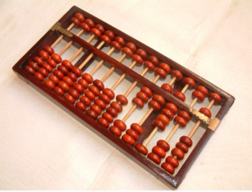


Figure: A Chinese Abacus, 2nd BC

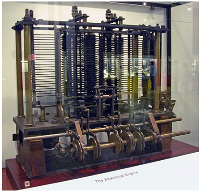


Figure: 1833, Analytical Engine, London Museum

Advent of Digital Computer

Kurt Gödel's 1931, *Incompleteness Theorem*: universal arithmetic-based formal language

Advent of Digital Computer

Kurt Gödel's 1931, *Incompleteness Theorem*: universal arithmetic-based formal language

Alan Turing 1936, *On Computable Numbers*: formal and simple hypothetical devices - Turing Machine

Advent of Digital Computer

Kurt Gödel's 1931, *Incompleteness Theorem*: universal arithmetic-based formal language

Alan Turing 1936, *On Computable Numbers*: formal and simple hypothetical devices - Turing Machine

Von Neumann 1947, *Von Neumann Architecture*: stored-program computers

Advent of Digital Computer

Kurt Gödel's 1931, *Incompleteness Theorem*: universal arithmetic-based formal language

Alan Turing 1936, *On Computable Numbers*: formal and simple hypothetical devices - Turing Machine

Von Neumann 1947, *Von Neumann Architecture*: stored-program computers

Manchester Baby 1948, *First stored-program computer*

First digital computer

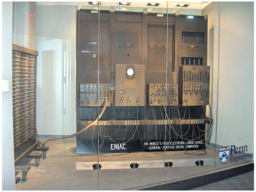


Figure: ENIAC - first electronic computer - 1945

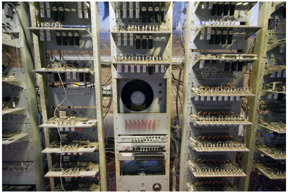


Figure: Manchester Baby, 1948

Simiconductor Industry

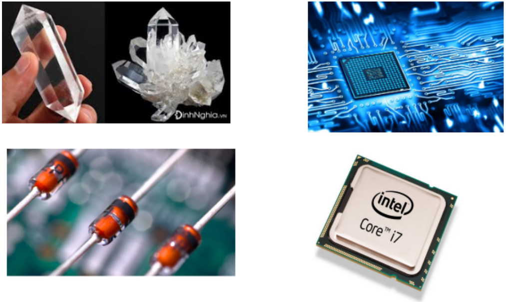


Figure: Silicon Valley

Transistor, CMOS, IC
RAM, ROM
LSIC, Microprocessor,
Wikipedia - Computer Hardware History

Table of Contents

Modern Digital Computer

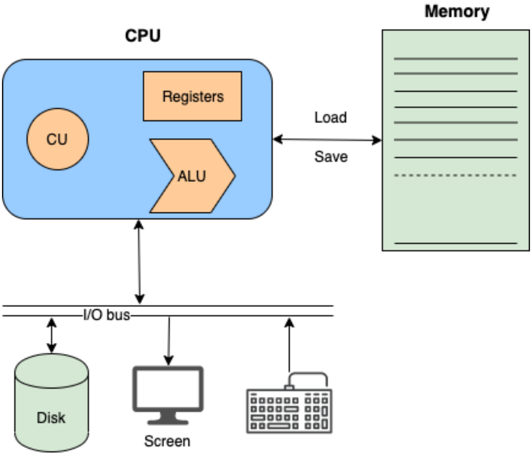


Figure: Modern computer organization

Building Material

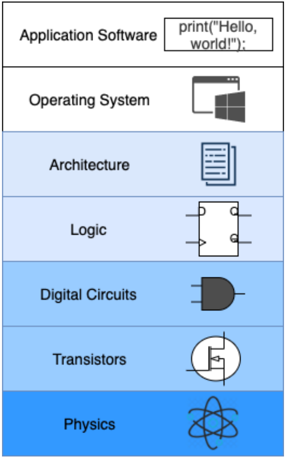
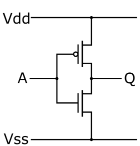
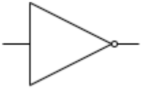
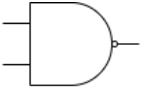


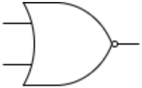
Figure: Digital Abstraction Layers

Logic Circuits

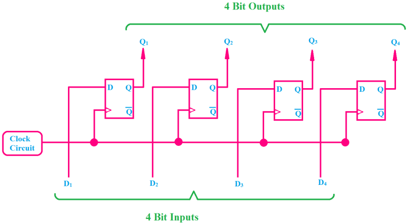



NOT Gate


NAND Gate


NOR Gate

Complex Logic Circuits



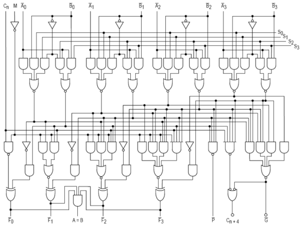
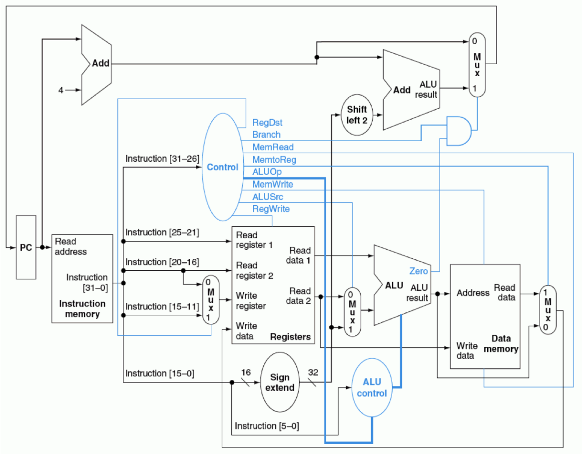


Table of Contents

How CPU work?



Add
4
Add
ALU result
Shift left 2
M u x
0
1
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Zero
ALU
ALU result
Address
Read data
Write data
Data memory
Instruction [15–0]
16
Sign extend
32
ALU control
Instruction [5–0]

Instructions

High-Level Code	Assembly Code
$a = b + c$	add a, b, c
$a = b$ - c	sub a, b, c

Table: Computer Instructions

Instructions

High-Level Code	Assembly Code
a = b + c	add a, b, c
a = b - c	sub a, b, c

Table: Computer Instructions

High-Level Code	Assembly Code
a = b + c - d	add t, b, c
	sub a, t, d

Table: Complex Instructions

Operands: Registers, Memory, and Constants

Instruction operates on `operands`

Operands: Registers, Memory, and Constants

Instruction operates on `operands`

Operands stored in registers or memory

Operands: Registers, Memory, and Constants

Instruction operates on `operands`

Operands stored in registers or memory

Or be constants stored in the instruction itself

Operands: Registers, Memory, and Constants

Instruction operates on `operands`

Operands stored in registers or memory

Or be constants stored in the instruction itself

Memory is large, but slow

Registers

Name	Number	Use
$0	0	the constant value 0
$at	1	assembler temporary
$v0-$v1	2-3	function return value
$a0-$a3	4-7	function arguments
$t0-$t7	8-15	temporary variables
$s0-$s7	16-23	saved variables
$t8-$t9	24-25	temporary variables
$k0-$k1	26-27	OS temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	function return address

Table: MIPS register set

High-Level Code to Assembly

Assume `a-c` held in registers `$s0-$s2` and `f-j` are in `$s3-$s7`

Example

a = b - c;
f = (g + h) - (i + j);

Memory

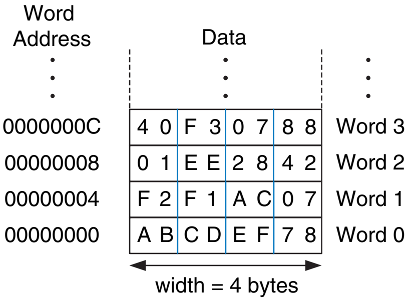


Figure: Byte-addressable memory

Memory Access

Example

\$s3, 1(\$0) # read memory word 1 into \$s3

\$s7, 5(\$0) # write \$s7 to memory word 5

Constans/Immediates

High-Level Code	Assembly Code
a = a + 4	addi \$0, \$0, 4 # \$0 = a, \$s1 = b
a = a - 12	addi \$0, \$0, -12

Table: Immediate Operands

Machine Language

Digital circuits understand only 1's and 0's

Fixed-length instruction: all instructions are encoded with 32 bits

- R-Type *register-type*
- I-Type *immediate-type*
- J-Type *jump-type*

R-Type Instructions

R-type

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

Figure: R-Type Format

Assembly Code / **Field Values**

Assembly Code	op	rs	rt	rd	shamt	funct
`add $s0, $s1, $s2`	0	17	18	16	0	32
`sub $t0, $t3, $t5`	0	11	13	8	0	34
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Machine Code

op	rs	rt	rd	shamt	funct	
000000	10001	10010	10000	00000	100000	(0x02328020)
000000	01011	01101	01000	00000	100010	(0x016D4022)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Figure: R-Type machine code

I-Type Instructions

I-type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Figure: I-type instruction format

Assembly Code	op	rs	rt	imm	op	rs	rt	imm	
`addi $s0, $s1, 5`	8	17	16	5	001000	10001	10000	0000 0000 0000 0101	(0x22300005)
`addi $t0, $s3, -12`	8	19	8	–12	001000	10011	01000	1111 1111 1111 0100	(0x2268FFF4)
`lw $t2, 32($0)`	35	0	10	32	100011	00000	01010	0000 0000 0010 0000	(0x8C0A0020)
`sw $s1, 4($t1)`	43	9	17	4	101011	01001	10001	0000 0000 0000 0100	(0xAD310004)
	6 bits	5 bits	5 bits	16 bits	6 bits	5 bits	5 bits	16 bits	

(Columns: Assembly Code; Field Values: op, rs, rt, imm; Machine Code: op, rs, rt, imm)

Figure: I-type machine code

J-Type Instructions

J-type

op	addr
6 bits	26 bits

Figure: J-type instruction format

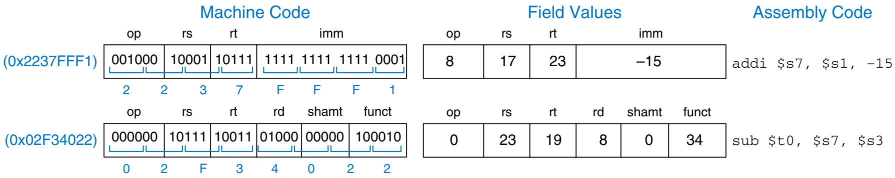


Figure: Machine code to assembly

Stored Program

Assembly Code	Machine Code
`lw $t2, 32($0)`	0x8C0A0020
`add $s0, $s1, $s2`	0x02328020
`addi $t0, $s3, -12`	0x2268FFF4
`sub $t0, $t3, $t5`	0x016D4022

Stored Program

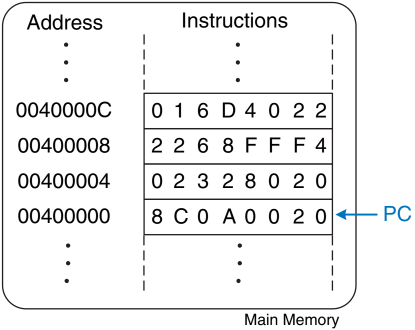


Figure: Stored program in memory

Table of Contents

Arithmetic/Logical Instructions

Source Registers

$s1	1111	1111	1111	1111	0000	0000	0000	0000
$s2	0100	0110	1010	0001	1111	0000	1011	0111

Assembly Code / **Result**

Assembly Code									
`and $s3, $s1, $s2`	$s3	0100	0110	1010	0001	0000	0000	0000	0000
`or  $s4, $s1, $s2`	$s4	1111	1111	1111	1111	1111	0000	1011	0111
`xor $s5, $s1, $s2`	$s5	1011	1001	0101	1110	1111	0000	1011	0111
`nor $s6, $s1, $s2`	$s6	0000	0000	0000	0000	0000	1111	0100	1000

Figure: Logical instructions

Branch

branch if equal (beq) and branch if not equal (bne)

```
addi $s0,$0,4 #$s0=0+4=4
addi $s1,$0,1 #$s1=0+1=1
sll $s1, $s1, 2 #$s1 = 1 << 2 = 4
beq $s0, $s1, target #$s0 == $s1, so branch
sub $s1, $s1, $s0  #not executed
target:
add $s1, $s1, $s0
```

Jump

```
int sum = 0;
for (i = 0; i != 10; i = i + 1) {
 sum = sum + i ;
}
```

Jump

```
int sum = 0;
for (i = 0; i != 10; i = i + 1) {
 sum = sum + i ;
}
```

```
# $s0 = i, $s1 = sum
 add $s1, $0, $0
 addi $s0,$0,0
 addi $t0, $0, 10

for:
 beq $s0, $t0, done
 add $s1, $s1, $s0
 addi $s0, $s0, 1
 j for

done:
```

Function Calls

```
int main() {
 int y;
 y = sum(2, 3);
 ...
}
```

```
int sum(int x, int y) {
  return x + y;
}
```

Function Calls

```
int main() {
 int y;
 y = sum(2, 3);
 ...
}
```

```
int sum(int x, int y) {
  return x + y;
}
```

```
# $s0 = y
main:
 addi $a0, $s0, 2
 addi $a1, $s0, 3
 jal sum
 add $s0, $v0, $0
```

```
# $s0 = result
sum:
 add $v0, $a0, $a1
 jr $ra
```

Futher ...

- Data allocation
- Memory Hierarchy and Caches
- I/O System
- Processor Design
- Digital Circuits