CS2100 Summary Sheet

Data Representation

Byte: 8 bits
Nibble: 4 bits
Word: Multiple of bytes depending on computer architecture
(MIPS is 4 bytes == 32 bits)

$\lceil \log_2 M \rceil$ bits required to represent M values
N bits can represent up to 2^N values

Base R to Decimal Conversion

$2A.8_{16} = 2\times16^1 + 10\times16^0 + 8\times16^{-1}$
$= 32 + 10 + 0.5 = 42.5_{10}$

Decimal to Binary Conversion

Repeated Division-By-2 (For whole numbers)
Use successive division by 2 until quotient is zero.
First remainder is LSB, and last is the MSB.

2	43	
2	21 rem 1	← LSB
2	10 rem 1	
2	5 rem 0	
2	2 rem 1	
2	1 rem 0	
	0 rem 1	← MSB

Example: $(43)_{10} = (101011)_2$

Repeated Multiplication-By-2 (For decimal fractions)
Use repeated multiplication by 2 until fractional product is zero.

	Carry	
0.3125×2=0.625	0	←MSB
0.625×2=1.25	1	
0.25×2=0.50	0	
0.5×2=1.00	1	←LSB

Example: $(0.3125)_{10} = (.0101)_2$

Decimal to base-R Conversion

Whole Numbers: Repeated division by R
Fraction: Repeated multiplication by R

Base R1 to R2 Conversion

Base R1 -> Base 10 -> Base R2

Example: Convert $(01231)_4$ to base 6
Decimal $= 0 + 1+4^3 + 2+4^2 + 3+4^1 + 1x4^0 = 109_{10}$
Repeated-Division by 6
109
18 R1
3 R0
0 R3 (MSB)
$(301)_6$

Example: Convert $(0.FE)_{16}$ to base 4
Method 1 (Grouping):
1111 1110
11 11 11 10
3332

Method 2 (Repeated multiplication):
$(.FE)_{16} = (.9921875)_{10} = (.3332)_4$
0.9921875 x 4 = 3.96875 C3
0.96875 x 4 = 3.875 C3
0.875 x 4 = 3.5 C3
0.5 x 4 = 2.0 C2

Binary/Hex/Octal Conversion

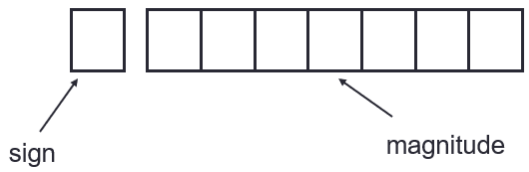
- Binary → Octal: partition in groups of 3
  - $(10\ 111\ 011\ 001\ .\ 101\ 110)_2$ = **$(2731.56)_8$**
- Octal → Binary: reverse
  - $(2731.56)_8$ = **$(10\ 111\ 011\ 001\ .\ 101\ 110)_2$**
- Binary → Hexadecimal: partition in groups of 4
  - $(101\ 1101\ 1001\ .\ 1011\ 1000)_2$ = **$(5D9.B8)_{16}$**
- Hexadecimal → Binary: reverse
  - $(5D9.B8)_{16}$ = **$(101\ 1101\ 1001\ .\ 1011\ 1000)_2$**

Negative Numbers

Unsigned numbers: Only non-negative values
Signed numbers: Include all values (positive and negative)

- Sign-And-Mangitude
- 1s Complement
- 2s Complement

Sign-and-Magnitude



Sign is represented by a 'sign bit' (MSB)

- 0 for +ve
- 1 for -ve

Negation

Just <u>invert the sign bit</u>
0010 0001 = $+33_{10}$
1000 0001 = -33_{10}

Largest value:	$2^{n-1} - 1$	0111 1111 = +127
Smallest value:	$-(2^{n-1} - 1)$	1111 1111 = -127
Zeroes:	+0	0000 0000 = +0
	-0	1000 0000 = -0
Range:	$\pm (2^n - 1)$	-127 to +127 (suboptimal)

Problems

- **Redundant Bit**: Representing small numbers such as 0 would still require 8 bits instead of 1 bit
- **Limit**: Representing large positive/negative numbers may be impossible due to the limited number of bits
- **Redundant 0s**: There are two possible zeroes: +0 and -0

1s Complement

Given a number **x** which can be expressed as an n-bit binary number, its negated value can be obtained in 1s-complement using: $-x = 2^n - x - 1$
Example: With an 8 bit number 0000 1100 (12_{10}), its negated value is:
- $0000\ 1100_2 = 2^8 - 12 - 1$
$= 243$
$= 1111\ 0011_{1s}$ (-12_{10})

Negation

<u>Invert all the bits</u>
$14_{10} = (0000\ 1110)_2 = (0000\ 1110)_{1s}$
$-14_{10} = -(0000\ 1110)_2 = (1111\ 0001)_{1s}$

Largest value:	$2^{n-1} - 1$	0111 1111 = +127
Smallest value:	$-(2^{n-1} - 1)$	1000 0000 = -127
Zeroes:	**+0**	0000 0000 = +0
	-0	1111 1111 = -0
Range:	$\pm (2^n - 1)$	-127 to +127 (suboptimal)

2s Complement

Given a number **x** which can be expressed as an n-bit binary number, its negated value can be obtained in 2s-complement using: $-x = 2^n - x$
Example: With an 8 bit number 0000 1100 (12_{10}), its negated value is:
- $0000\ 1100_2 = 2^8 - 12$
$= 244$
$= 1111\ 0100_{2s}$ (-12_{10})

Negation

<u>Invert all the bits, then add 1</u>
$14_{10} = (0000\ 1110)_2 = (0000\ 1110)_{2s}$
$-14_{10} = -(0000\ 1110)_2 = (1111\ 0010)_{2s}$

Largest value:	$2^{n-1} - 1$	0111 1111 = +127
Smallest value:	$-(2^{n-1} - 1)$	1000 0000 = -128
Zero:	**+0**	0000 0000 = +0
Range:	-2^{n-1} to $2^{n-1} - 1$	-128 to +127 (suboptimal)

Complement on Fractions

Negate 0101.01 in 1s-complement: 1010.10
Negate 0101.01 in 2s-complement: 1010.11

2s Complement on Addition/Subtraction

Addition of integers, A + B:

1. Perform binary addition on the two numbers
2. Ignore the carry out of the MSB
3. Check for overflow. Overflow occurs if the 'carry in' and 'carry out' of the MSB are different, or if result is opposite sign of A and B

Algorithm for subtraction, A – B:

1. Take 2s complement of B
2. Add 2s complement of B to A

Detecting overflow:

- *positive* add *positive* → *negative*
- *negative* add *negative* → *positive*

Note: +ve add -ve, and -ve add +ve will never produce overflow

Example: 4-bit 2s complement system
$0101_{2s} + 0110_{2s} = 1011_{2s}$
$5_{10} + 6_{10} = -5_{10}$ (overflow!)

$1001_{2s} + 1101_{2s} = \underline{1}0110_{2s}$ (discard end-carry) $= 0110_{2s}$
$-7_{10} + -3_{10} = 6_{10}$ (overflow!)

N Complements

We generalize $(r-1)$'s-complement (also called radix diminished complement) to include fraction as follows:

$$(r-1)\text{'s complement of } N = r^n - r^{-m} - N$$

where n is the number of integer digits and m the number of fractional digits. (If there are no fractional digits, then $m = 0$ and the formula becomes $r^n - 1 - N$ as given in class.)

Excess Representation

Excess-8 Representation	Value
0000	-8
0001	-7
0010	-6
0011	-5
0100	-4
0101	-3
0110	-2
0111	-1

Excess-8 Representation	Value
1000	0
1001	1
1010	2
1011	3
1100	4
1101	5
1110	6
1111	7

IEEE 754 Floating-Point Representation

$$-6.5_{10} = -110.1_2 = -1.\boxed{101}_2 \times 2^2$$

Exponent = 2 + 127 = 129 = 10000001_2

1	10000001	10100000000000000000000
sign	exponent (excess-127)	mantissa

Single-precision (32 bits): 1-bit sign, 8-bit exponent with bias 127, 23-bit mantissa
Double-precision (64 bits): 1-bit sign, 11-bit exponent with bias 1023, 52-bit mantissa

Sign Bit: 0 for positive, 1 for negative
Mantissa: Normalised with an implicit leading bit 1
110.1_2 -> normalized -> $1.\mathbf{101}_2 \times 2^2$ -> only 101 is stored in mantissa
0.00101101 -> $1.\mathbf{01101}_2 \times 2^{-3}$ -> only 01101 is stored in mantissa
Exponent: N + 127

MIPS Introduction

You write programs in high-level programming languages,

A + B

Compiler translates this into assembly language statement

Add A,B

Assembler translates this statement into **machine language instructions** that the processor can execute

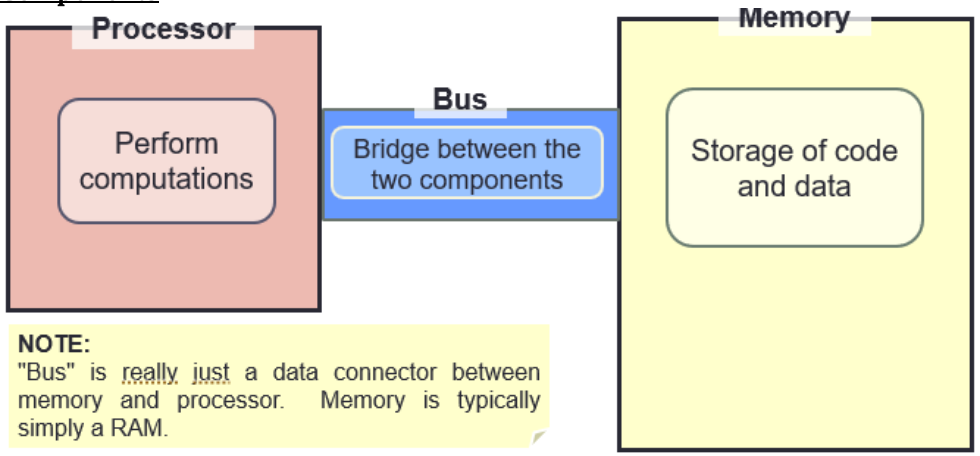
1000 1100 1010 0000

Instruction Set Architecure (ISA)

An abstraction on the interface between the hardware and the low-level software.

- Includes everything programmers need to know to make the machine code work correctly
- Allows computer designers to talk about functions independently from the hardware that performs them
- Allows many implementations of varying cost and performance to run identical software

Components



Stored-memory concept: Both instruction and data are stored in memory

Load-store model: Limit memory operations and relies and on registers for storage during execution

General Purpose Registers

- Fast memories in the processor
  - Data are transferred from memory to registers for faster processing
- Limited in number
  - A typical architecture has 16 to 32 registers
  - Compiler associates variables in program with registers
- Registers have no data type
  - Unlike program variables
  - Machine/Assembly instruction assumes the data stored in the register is of the correct type

MIPS Instructions

Operation	Opcode
Addition	`add $rd, $rs, $rt`
Subtraction	`sub $rd, $rs, $rt`
Shift Left Logical	`sll $rd, $rt, C5`
Shift Right Logical	`srl $rd, $rt, C5`
Bitwise AND	`and $rd, $rs, $rt`
Bitwise OR	`or $rd, $rs, $rt`
Bitwise XOR	`xor $rd, $rs, $rt`
Bitwise NOR	`nor $rd, $rs, $rt`
Addition	`addi $rt, $rs,` $C16_{2s}$
Bitwise AND	`andi $rt, $rs, C16`
Bitwise OR	`ori $rt, $rs, C16`
Bitwise XOR	`xori $rt, $rs, C16`
Load	`lw $rt, offset($rs)`
Store	`sw $rt, offset($rs)`
Branch on Equal	`beq $rs, $rt, label`
Branch on Not Equal	`bnq $rs, $rt, label`

C5 is [0 to 2^5-1] **$C16_{2s}$** is [-2^{15} to 2^{15}-1] **C16** is a 16-bit pattern

Note: C16 are NOT sign-extended,

$C16_{2s}$ are sign-extended, otherwise addi will not work properly as the processor can only work with 32-bits

Load Upper Immediate (lui)

`lui $t0, C16`$_{2s}$

Implementing NOT

- `nor $t0, $t0, $zero`
- `xor $t0, $t0, $t2 ($t2 contain all 1s)`

Memory Organisation

Each location of the memory has an address. Given a k-bit address, the address space is of size 2^k.

The memory map on the right contains one byte every address – called byte addressing.

Address	Content
0	8 bits
1	8 bits
2	8 bits
3	8 bits
4	8 bits
5	8 bits

Using distinct memory address, we can access:

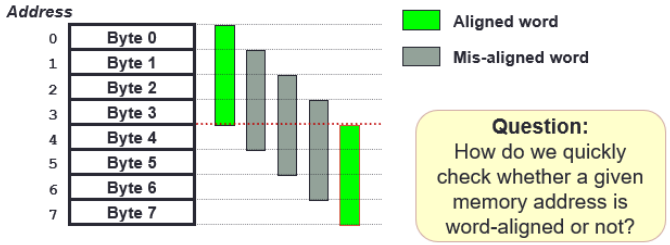
- A single byte (byte addressable) or
- A single word (word addressable)

Word is: usually 2^n bytes. Also commonly coincides with the register size, integer size, and instruction size in most architectures.

Word Alignment

- Words are aligned in memory if they begin at a byte address that is a multiple of the number of bytes in a word.

Example: If a word consists of 4 bytes, then:



MIPS Memory Instructions

MIPS is a load-store register architecture

- 32 registers, each 32-bit long (4 bytes)
- Each word contains 32 bits (4 bytes)
- Memory Addresses are 32-bit long (4 bytes)

Name	Examples	Comments
32 registers	$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at	Fast processor storage for data. In MIPS, data must be in registers to perform arithmetic.
2^{30} memory words	`Mem[0]`, `Mem[4]`, ..., `Mem[4294967292]`	Accessed only by data transfer instructions. MIPS uses **byte addresses**, so consecutive words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

$$WordCount = \frac{2^{32}\ bytes}{4\ bytes/word} = 2^{30}\ memory\ words$$

Address vs Value

Registers do NOT have types

- A register can hold any 32-bit number:
  - The number has no implicit data type and is interpreted according to the instruction that uses it
  - add $t2, $t1, $t0
    - t1 and t0 should contain data values
  - lw $t2, 0($t0)
    - t0 should contain a memory address

Byte vs Word

Consecutive word addresses in machines with byte-addressing do not differ by 1

Common Error:

Assume that the address of the next word can be found by incrementing the address in a register by 1 instead of by the word size in bytes

Decision Making Instructions

Conditional (branch)

```
bne $t0, $t1, label
beq $t0, $t1, label
```

Unconditional (jump)

```
j label
```

- A label is an "anchor" in the assembly code to indicate point of interest, usually as branch target.
- Labels are NOT instructions
- `j label`, is technically equivalent to, `beq $zero, $zero, L1.`

IF Statement

C Statement to translate	Variables Mapping
`if (i == j)` `  f = g + h;`	f → $s0 g → $s1 h → $s2 i → $s3 j → $s4

```
      beq $s3, $s4, L1
      j   Exit
L1:   add $s0, $s1, $s2
Exit:
```

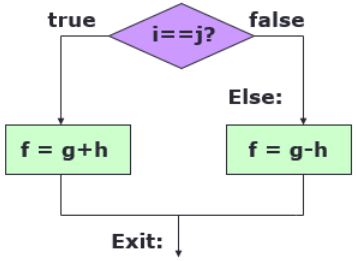
```
      bne $s3, $s4, Exit
      add $s0, $s1, $s2
Exit:
```

IF/ELSE Statement

C Statement to translate	Variables Mapping

`if (i == j)`
`  f = g + h;`
`else`
`  f = g - h;`
	f → $s0
	g → $s1
	h → $s2
	i → $s3
	j → $s4

```
      bne $s3, $s4, Else
      add $s0, $s1, $s2
      j   Exit
Else: sub $s0, $s1, $s2
Exit:
```



- Question: Rewrite with `beq`?

Loops

```
while (j == k)
  i = i + 1;
```

```
Loop:  if (j != k)
          goto Exit;
       i = i+1;
       goto Loop;
Exit:
```

C Statement to translate	Variables Mapping

`Loop:  if (j != k)`
`  goto Exit;`
`  i = i+1;`
`  goto Loop;`
`Exit:`
	i → $s3
	j → $s4
	k → $s5

NOTE:
This shows the process clearly:
1. Convert from while to if(…) goto
2. Convert from there to MIPS

- What is the corresponding MIPS code?

```
Loop: bne  $s4, $s5, Exit   # if (j!= k) Exit
      addi $s3, $s3, 1
      j    Loop             # repeat loop
Exit:
```

FOR Loops

C Statement to translate	Variables Mapping
`for ( i=0; i<10; i++)` `  a = a + 5;`	i → $s0 a → $s2

```
       add  $s0, $zero, $zero
       addi $s1, $zero, 10
Loop:  beq  $s0, $s1, Exit
       addi $s2, $s2, 5
       addi $s0, $s0, 1
       j    Loop
Exit:
```

NOTE:
Alternatively, if you know how to compile while-loop, then you can translate the for-loop into:

```
i = 0;
while (i < 10) {
  a = a + 5;
  i++;
}
```

Inequalities

To build a "blt $s1, $s2, L" instruction,

```
slt $t0, $s1, $s2
bne $t0, $zero, L
```

==

```
if ($s1 < $s2)
    goto L;
```

Arrays and Loop

Address of A[] → $t0
Result → $t8
&A[i] → $t1

```
        addi $t8, $zero, 0
        addi $t1, $t0, 0
        addi $t2, $t0, 160
loop:   bge  $t1, $t2, end
        lw   $t3, 0($t1)
        bne  $t3, $zero, skip
        addi $t8, $t8, 1
skip:   addi $t1, $t1, 4
        j loop
end:
```

Comments

NOTE:
Consider the code using pointer arith:

```
result = 0;
ptr = A;
end = &A[40];
while ( ptr < end ) {
    if ( *ptr == 0 )
        result++;
    ptr++;
}
```

The resulting MIPS looks like the code on the left.

MIPS Instruction Formats

- Each MIPS instruction has a fixed-length of 32bits

R-Format (Register format: op $r1, $r2, $r3)

MIPS instruction

```
arith  $rd, $rs, $rt
```

NOTE:
opcode is always 0
shamt is always 0
arith is arithmetic operation

opcode	rs	rt	rd	shamt	funct
0	rs	rt	rd	0	XX

MIPS instruction

```
shift  $rd, $rt, shamt
```

NOTE:
opcode is always 0
rs is always 0
shift is shift operation

opcode	rs	rt	rd	shamt	funct
0	0	rt	rd	shamt	XX

I-Format (Immediate Format: op $r1, $r2, Immd)

MIPS instruction

```
arith  $rt, $rs, C16_2s
```

NOTE:
$C16_{2s}$ is in 2s complement
arith is arithmetic operation

opcode	rs	rt	immediate
XX	rs	rt	$C16_{2s}$

MIPS instruction

```
ld/st  $rt, C16_2s($rs)
```

NOTE:
$C16_{2s}$ is in 2s complement
ld/st is a load or store operation

opcode	rs	rt	immediate
XX	rs	rt	$C16_{2s}$

MIPS instruction

```
logic  $rt, $rs, C16
```

NOTE:
C16 is raw binary (*no negative*)
logic is logical operation (*bitwise operation*)

opcode	rs	rt	immediate
XX	rs	rt	C16

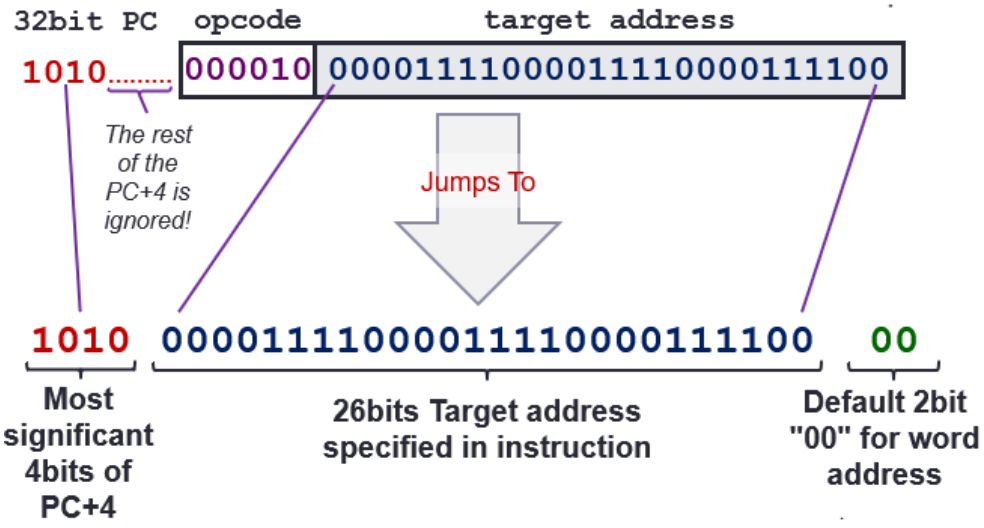
MIPS instruction

```
branch $rs, $rt, label
```

NOTE:
branch is a branch operation
label is converted to number first (*PC-relative addressing*)

opcode	rs	rt	immediate
XX	rs	rt	label

J-Format (Jump format: op Immd)

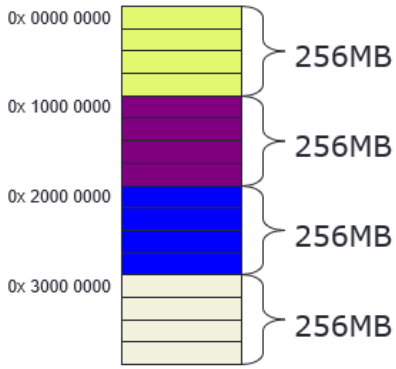


- We can only specify 26 bits of a 32-bit address
- Optimisation: Jumps will only jump to word-aligned addresses, so last 2 bits are always 00.
- Now we can specify 28 bits of 32 bit address
- MIPS choose to take the 4 most significant bits from PC+4.
- The maximum jump range **is 256MB boundary**.

If you are at the top of the boundary, you ***cannot*** jump up.

If you are at the bottom of the boundary, you ***cannot*** jump down.

Can you figure out the address of the *top* and the *bottom*? *(discuss in forum)*



- We can only jump within our **block** due to the use of the first 4-bits from the PC.

Addressing Modes

Register Addressing: Operand is a register
- add, sub, and, or, xor, sll, srl ...

Immediate Addressing: Operand is a constant
- addi, andi, ori, xori, slti

Base addressing (displacement addressing): operand is at the memory location whose address is sum of a register and a constant in the instruction
- lw, sw

PC-relative addressing: address is sum of PC and constant in the instruction
- beq, bne

Pseudo-direct addressing: 26-bit of instruction concatenated with upper 4bits of PC
- j

Instruction Set Architecture (ISA)

Complex Instruction Set Computer (CISC)
- Example: x86-32 (IA32)
- Single instruction performs complex operation
- Smaller program size as memory was premium
- Complex implementation, no room for hardware optimization

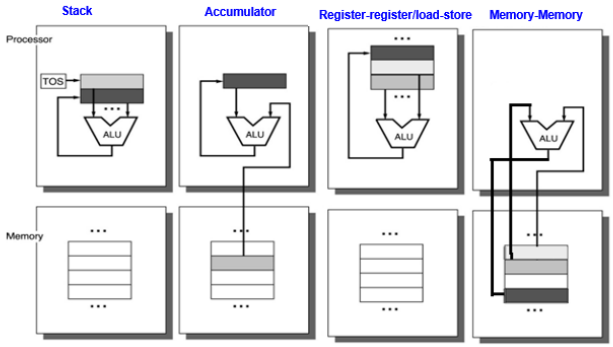
Reduced Instruction Set Comptuer (RISC)
- Example: MIPS, ARM
- Keep the instruction set small and simple, easier to build/optmise hardware
- Burden on software to combine simpler operations to implement high-level language statements

Concept #1: Data Storage

von Neumann Architecture: Data (operands) are stored in memory

Stack	Accumulator	Register (load-store)	Memory-Memory
Push A	Load A	Load R1,A	Add C, A, B
Push B	Add B	Load R2,B	
Add	Store C	Add R3,R1,R2	
Pop C		Store R3,C	

C = A+B



- **Stack Architecture**
  - Operands are implicitly on top the stack
- **Accumulator Architecture**
  - One operand is implicitly in the accumulator (a special register)
- **General-purpose Register Architecture**
  - Only explicit operands.
  - **Register-Memory Architecture** (one operand in memory)
  - **Register-Register (or load-store) Architecture**
    - Example: MIPS
- **Memory-Memory Architecture**
  - All operands in memory

For modern processors:

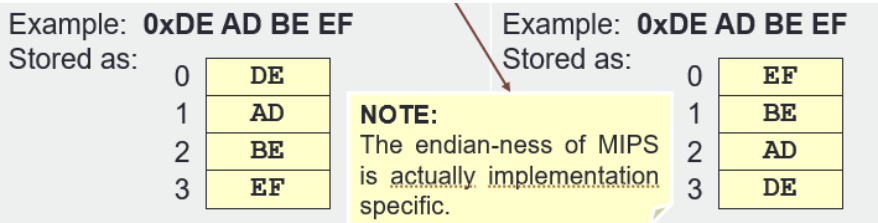
- General Purpose Register (GPR) is the most common choice for storage design
- RISC computers typically use Register-Register design (E.g. MIPS, ARM)
- CISC Computers use a mixture of Register-Register and Register-Memory

Concept #2: Memory Address and Content

- Given k-bit address, the address space is of size 2^k.
- Each memory transfer consists of one word of n bits.

Memory Content: Endianness

- **Big-Endian:** Most significant byte stored in lowest address
- **Little-Endian:** Least significant byte stored in lowest address



y

MIPS has 3 addressing modes:

- **Register**
  - Operand is in a register *(e.g. add $t1, $t2, $t3)*
- **Immediate**
  - Operand is specified in the instruction directly *(e.g. addi $t1, $t2, 98)*
- **Displacement**
  - Operand is in memory with address calculated as base + offset *(e.g. lw $t1, 20($t2))*

Concept #3: Operations in Instruction Set

Data Movement	load, store, memory-to-memory move, register-to-register move, input, output, push, pop
Arithmetic	integer or FP add, subtract, multiply divide
Shift Logical	shift left/right, rotate left,right, not, and, or, set ,clear
Control Flow	jump, branch
Subroutine Linkage	call, return
Interrupt	trap, return
Synchronisation	test & set
String	search, move, compare
Graphics	pixel and vertex operations, compression/decompression

Amdahl's Law – make the common cases fast. (Load, Conditional branch, Compare, Store)

Concept #4: Instruction Formats

Instruction Length

Variable-length instructions

- Instructions vary in bytes
- Require multi-step fetch and decode
- Allow for a more flexible and compact instruction set

Fixed-length instructions

- Used in most RISC
- MIPS: Instructions are fixed 4 bytes long
- Allow for easy fetch and decode
- Simplify pipelining and parallelism
- Instruction bits are scarce

Hybrid Instructions

- Mix of variable and fixed length instructions

Instruction Fields

Instruction consists of

- **Opcode**: Unique code to specify the desired operation
- **Operarands**: Zero or more additional information needed for the operation

The operation designates the type and size of the operands. Typical type and sizes: Character (8 bits), half-word (16 bits), word (32 bits), single-precision FP (1 word), double-precision FP (2 word).

Concept #5: Encoding the Instruction Set

Issues:

- Code size, speed/performance, design complexity

Things to be decided:

- Number of registers
- Number of addressing modes
- Number of operands in an instruction

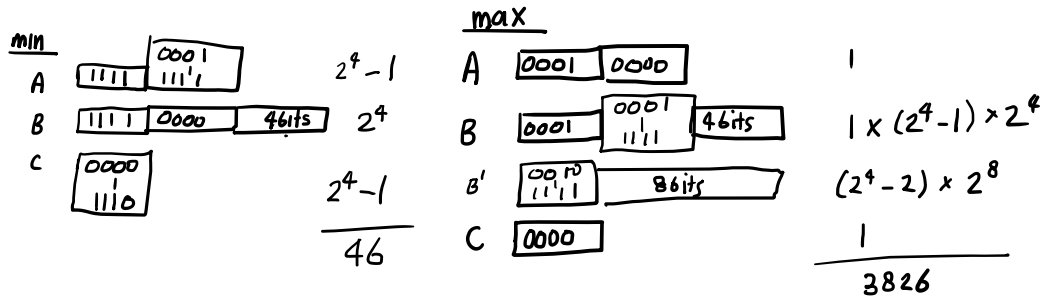
The different competing forces:

- Have many registers and addressing modes
- Reduce code size
- Have instruction length that is easy to handle (fixed-length are easier to handle)

A 8 bits
B 12 bits
C 4 bits

$max = 2^{12} - 2^4 - 2^8 + 2 = 3826$

$min = (2^4-1) + (2^4-1) + 2^4 = 46$



Datapath

Collection of components that process data. Performs the arithmetic, logical and memory operations.

- **Arithmetic and Logical operations**
  - add, sub, and, or, addi, andi, ori, slt
- **Data transfer instructions**
  - lw, sw
- **Branches**
  - beq, bne

Note
andi and ori is not supported in this current processor design because we always do "sign extension" on immediate value

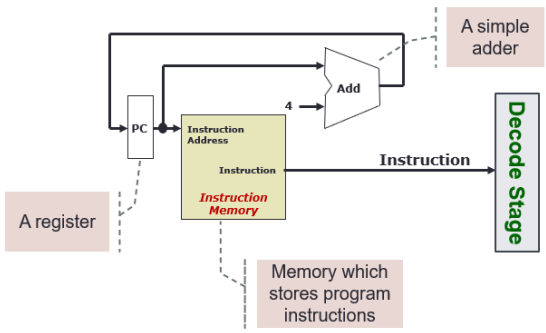
Note: sll and srl can be done by multiplication (which be done by add with loop). J can be done by beq $zero, $zero, label if we ignore the difference related to 256MB of blocks.

Instruction Execution Cycle (Basic)

1. Fetch
   - Get instruction from memory
   - Address is in Program Counter (PC) Register
2. Decode
   - Find out the operation required
3. Operand Fetch
   - Get operand(s) needed for operation
4. Execute
   - Perform the required operation
5. Result Write (Store)
   - Store the result of the operation

	add $rd, $rs, $rt	lw $rt, ofst($rs)	beq $rs, $rt, ofst
Fetch	*standard*	*standard*	*standard*
Decode Operand Fetch	○ Read [**$rs**] as *opr1* ○ Read [**$rt**] as *opr2*	○ Read [**$rs**] as *opr1* ○ Use ***ofst*** as *opr2*	○ Read [**$rs**] as *opr1* ○ Read [**$rt**] as *opr2*
ALU	*Result = opr1 + opr2*	*MemAddr = opr1 + opr2*	*Taken = (opr1 == opr2)?* *Target =* (**PC**+4) + **ofst×4**
Memory Access		Use *MemAddr* to read from memory	
Result Write	*Result* stored in **$rd**	*Memory* data stored in **$rt**	if (*Taken*) **PC** = *Target*

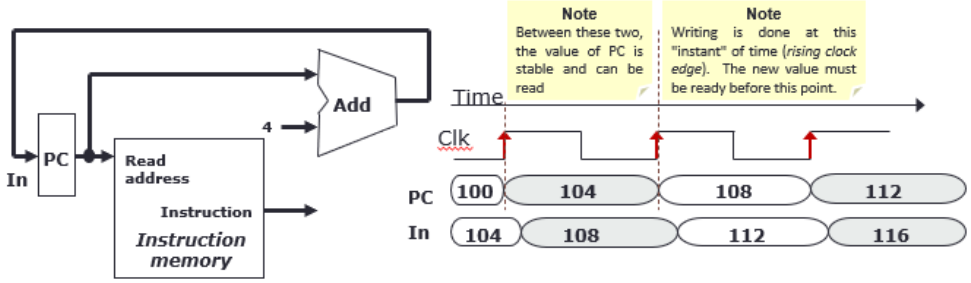
Fetch Stage



- Use the Program Counter (PC) to fetch instruction from memory
  - PC is implemented as special register in the processor
- Increment the PC by 4 to get the address of the next instruction
  - Note exception when branch/jump is executed
- Output to the next stage (decode)

Clocking

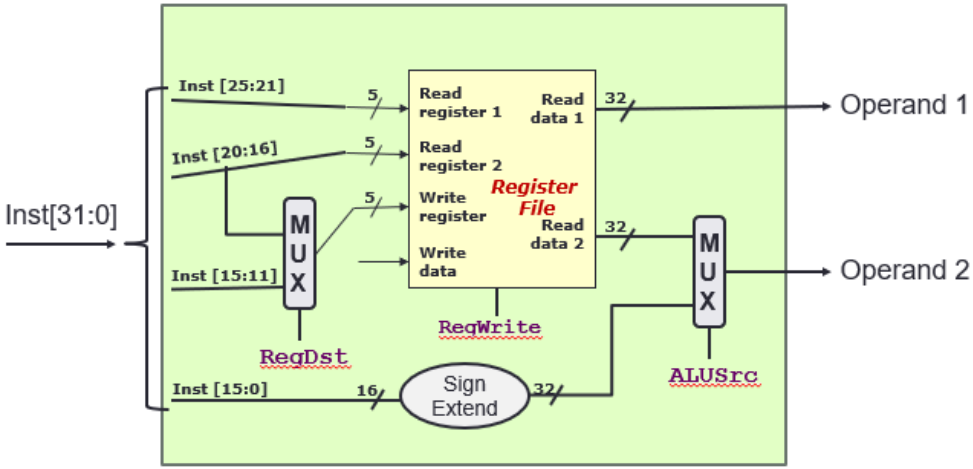
- PC is read during the first half of the clock period and it is updated with PC+4 at the **next rising clock edge**



Decode Stage

- Gather data from the instruction fields
  - Read the opcode to determine instruction type and field lengths
  - Read data from all necessary registers
- Input from previous stage (fetch): instruction to be executed
- Output to next stage (ALU): operation and necessary operands

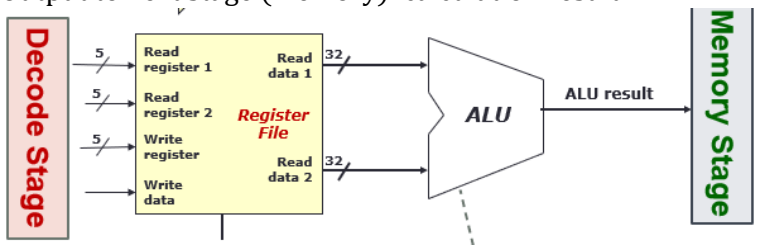
Decode Stage Summary



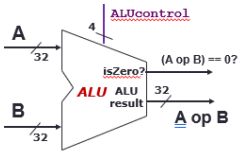
ALU Stage

- Arithmetic-Logic Unit
- Also called the Execution stage
- Performs the real work for most instructions here (calculations)

Input from previous stage (decode): operation and operands
Output to next stage (memory): calculation result

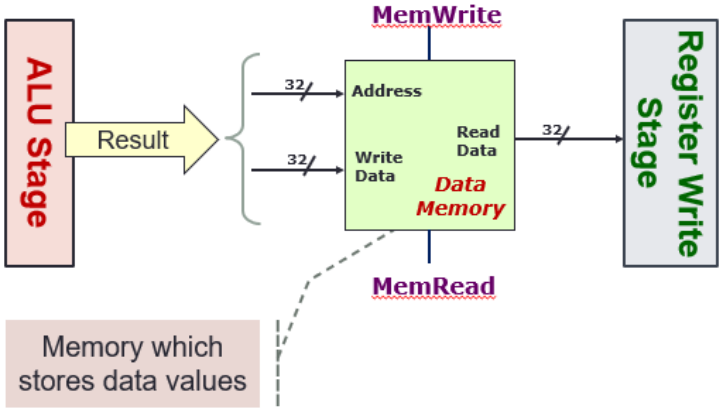


- **ALU (Arithmetic Logic Unit)**
  - Combinational logic to implement arithmetic and logical operations
- **Inputs:**
  - Two 32-bit numbers
- **Control:**
  - 4-bit to decide the particular operation
- **Outputs:**
  - Result of arithmetic/logical operation
  - A 1-bit signal to indicate whether result is zero



ALUcontrol	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR

Memory Stage



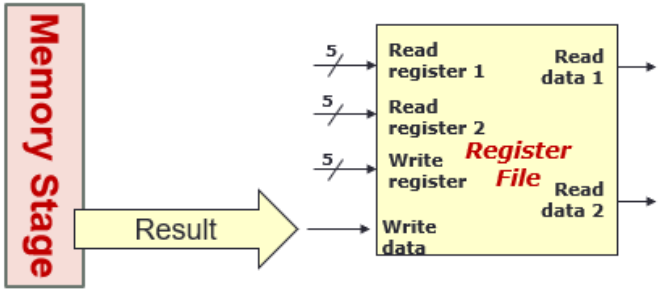
- Only the load and store instructions need to perform operation in this stage
  - Use memory address calculated by ALU stage
  - Read from or write to data memory
- All other instructions remain idle
  - Result from ALU stage will pass through to be used in Register Write stage if applicable

Input from previous stage (ALU): computation result to be used as memory address
Output to next stage (Register Write): result to be stored

Register Write Stage

- Most instructions write the result of some computation into a register
- Exceptions are stores, branches, jumps

Input from previous stage (memory): computation either from mem or ALU

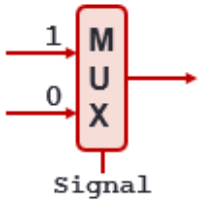


Control

Control Signal	Execution Stage	Purpose
RegDst	Decode/Operand Fetch	Select the destination register number
RegWrite	Decode/Operand Fetch RegWrite	Enable writing of register
ALUSrc	ALU	Select the 2nd operand for ALU
ALUcontrol	ALU	Select the operation to be performed
MemRead / MemWrite	Memory	Enable reading/writing of data memory
MemToReg	RegWrite	Select the result to be written back to register file
PCSrc	Memory/RegWrite	Select the next PC value

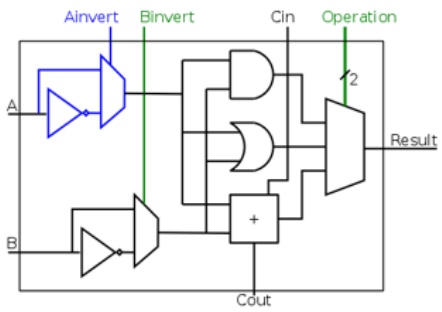
- The control signals are generated based on the instruction to be executed
  - Opcode -> Instruction Format
  - Example:
    - R-format -> RegDst = 1
  - R-Type instruction has additional information
    - 6bit funct field
- Idea: Design a circuit to generate these signals based on opcode and funct

NOTE: Input of MUX of MemToReg is flipped.



ALUControl

ALUcontrol			Function
Ainvert	Binvert	Operation	
0	0	00	AND
0	0	01	OR
0	0	10	add
0	1	10	subtract
0	1	11	slt
1	1	00	NOR

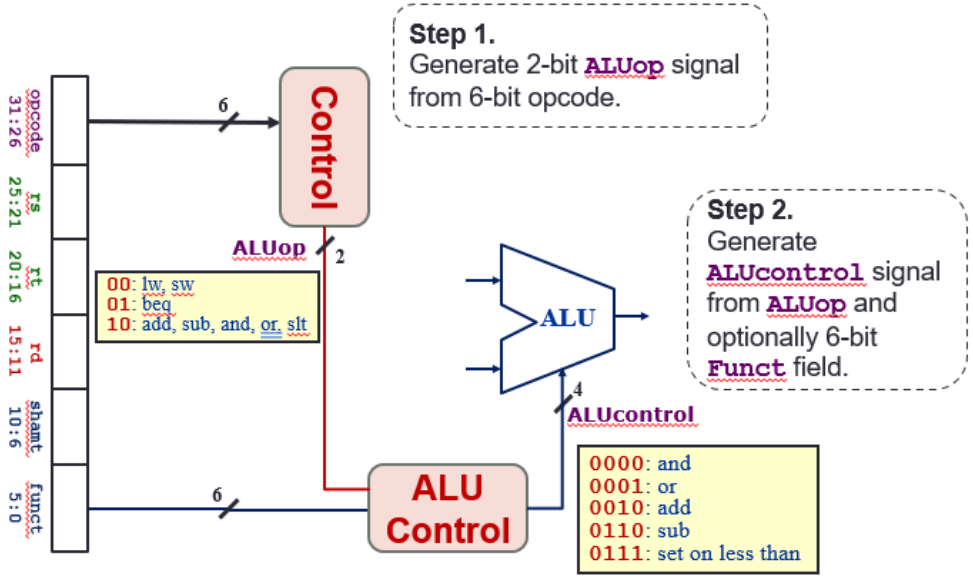


Intermediate Signal: ALUop

- Use Opcode to generate a 2-bit ALUop signal

Instruction type	ALUop
lw / sw	00
beq	01
R-type	10

- Since both lw and sw are performing ADDITION, they can have same ALUop. Beq is performing SUBTRACTION, so it will need a different ALUop. Lastly, R-type cannot be differentiated by opcode, so we defer to funct instead.



Opcode	ALUop	Instruction Operation	Funct field	ALU action	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	10 0000	add	0010
R-type	10	subtract	10 0010	subtract	0110
R-type	10	AND	10 0100	AND	0000
R-type	10	OR	10 0101	OR	0001
R-type	10	set on less than	10 1010	set on less than	0111

Instruction Type	ALUop
lw / sw	00
beq	01
R-type	10

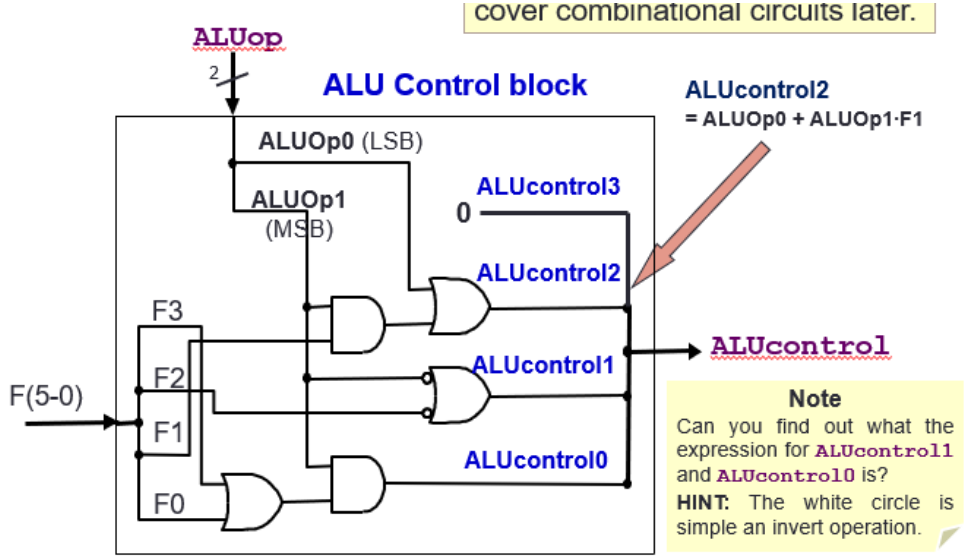
ALUcontrol	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR

Generation of 2-bit ALUop signal will be discussed later

	ALUop		Funct Field (F[5:0] == Inst[5:0])						ALU control
	MSB	LSB	F5	F4	F3	F2	F1	F0	
lw	0	0	X	X	X	X	X	X	0 0 1 0
sw	0	0	X	X	X	X	X	X	0 0 1 0
beq	~~0~~ X	1	X	X	X	X	X	X	0 1 1 0
add	1	~~0~~ X	~~1~~ X	~~0~~ X	0	0	0	0	0 0 1 0
sub	1	~~0~~ X	~~1~~ X	~~0~~ X	0	0	1	0	0 1 1 0
and	1	~~0~~ X	~~1~~ X	~~0~~ X	0	1	0	0	0 0 0 0
or	1	~~0~~ X	~~1~~ X	~~0~~ X	0	1	0	1	0 0 0 1
slt	1	~~0~~ X	~~1~~ X	~~0~~ X	1	0	1	0	0 1 1 1

- ALUcontrol3 = 0
- ALUcontrol2 = ALUop0 + ALUop1 . F1

cover combinational circuits later.

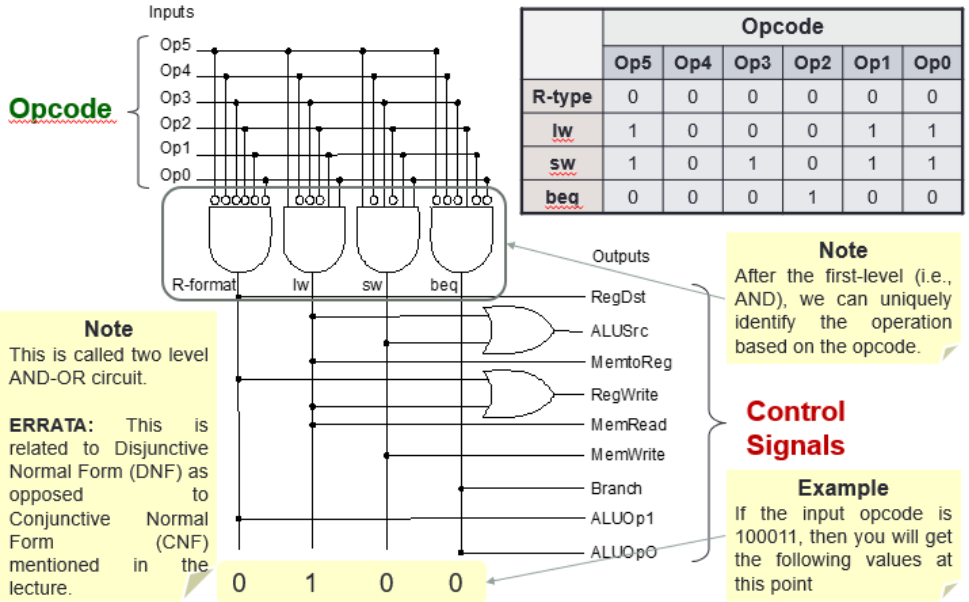


Control Design: Outputs

	RegDst	ALUSrc	MemTo Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop	
								op1	op0
R-type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

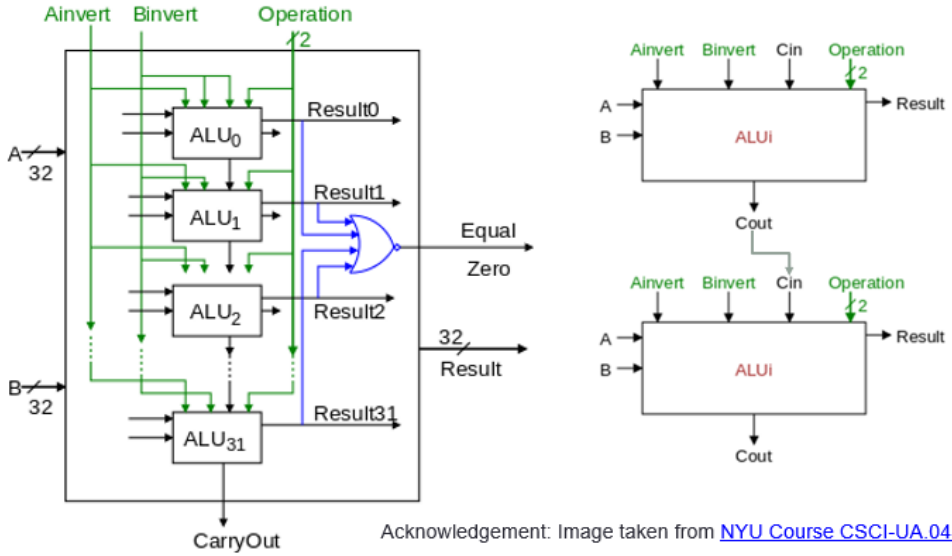
Control Design: Inputs

	Opcode (Op[5:0] == Inst[31:26])						
	Op5	Op4	Op3	Op2	Op1	Op0	Value in Hexadecimal
R-type	0	0	0	0	0	0	0
lw	1	0	0	0	1	1	23
sw	1	0	1	0	1	1	2B
beq	0	0	0	1	0	0	4



	Opcode					
	Op5	Op4	Op3	Op2	Op1	Op0
R-type	0	0	0	0	0	0
lw	1	0	0	0	1	1
sw	1	0	1	0	1	1
beq	0	0	0	1	0	0

Constructing 32bits from 1bit

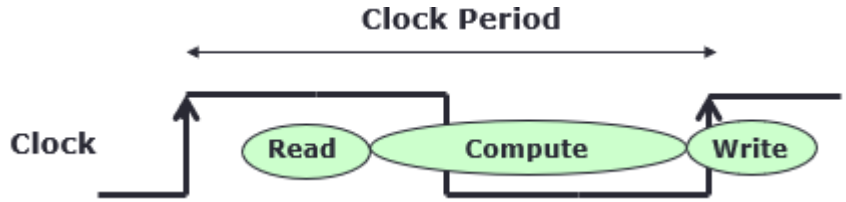


Acknowledgement: Image taken from NYU Course CSCI-UA.0436

Big Picture: Instruction Execution

- Read contents of one or more storage elements (register/memory)
- Perform computation through some combinational logic
- Write results to one or more storage elements (register/memory)

All these performed within a clock period



Single Cycle Implementation: Shortcoming

- All instructions take as much time as the slowest one
- Long cycle time for each instruction

Solution #1: Multicycle Implementation

- Break up the instruction into execution steps:
  - Instruction fetch
  - Instruction decode and register read
  - ALU operation
  - Memory read/write
  - Register write
- Each execution step takes one clock cycle
  - Cycle time is much shorter, ie. Clock frequency is much higher
- Instructions take variable number of clock cycles to complete execution

Solution #2: Pipelining

- Break up instructions into execution steps one per clock cycle
- Allow different instructions to be in different execution steps simultaneously

Decimal Base-10	Binary Base-2	Octal Base-8	Hexa Decimal Base-16
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[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	[NEGATIVE 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	[ENG OF 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	[SUBSTITUTE]	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	[DEL]

Positive Power of 2

Exp	Val	Exp	Val	Exp	Val	Exp	Val
2^0	1	2^8	256	2^{16}	65,536	2^{24}	16,777,216
2^1	2	2^9	512	2^{17}	131,072	2^{25}	33,554,432
2^2	4	2^{10}	1,024	2^{18}	262,144	2^{26}	67,108,864
2^3	8	2^{11}	2,048	2^{19}	524,288	2^{27}	134,217,728
2^4	16	2^{12}	4,096	2^{20}	1,048,576	2^{28}	268,435,456
2^5	32	2^{13}	8,192	2^{21}	2,097,152	2^{29}	536,870,912
2^6	64	2^{14}	16,384	2^{22}	4,194,304	2^{30}	1,073,741,824
2^7	128	2^{15}	32,768	2^{23}	8,388,608	2^{31}	2,147,483,648

Negative Power of 2

Exp	Val	Exp	Val
2^{-1}	0.5	2^{-9}	0.001953125
2^{-2}	0.25	2^{-10}	0.0009765625
2^{-3}	0.125	2^{-11}	0.00048828125
2^{-4}	0.0625	2^{-12}	0.000244140625
2^{-5}	0.03125	2^{-13}	0.0001220703125
2^{-6}	0.015625	2^{-14}	0.00006103515625
2^{-7}	0.0078125	2^{-15}	0.000030517578125
2^{-8}	0.00390625	2^{-16}	0.0000152587890625

Datapath & Control