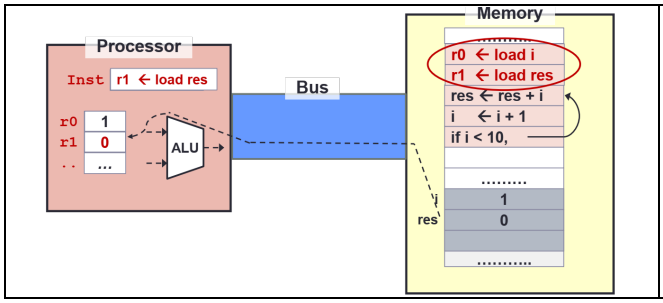
Instruction Set Architecture (ISA)

Assembly Language

Machine Code	Assembly Language
Instructions in binary e.g.: `1000 1100 1010 0000` → Add two numbers	Human readable e.g.: `add A, B` → Add two numbers
Hard and tedious to code	Easier to write than machine code, symbolic version of machine code
`1000 1100 1010 0000` ← ASSEMBLER ← `add A, B`	
May also be written in hexadecimal for a more human-readable format	May provide '**pseudo-instructions**' as syntactic sugar
	When considering performance, only real instructions are counted

Structure:

- Both **instruction** and **data** are stored in **memory**. Transferred into the **processor** (perform computation) during execution.
- To **avoid frequent access** of memory. Provide **temporary storage** for values in the **processor** (known as **registers**)



Moving data from memory into a register – load
Moving data from a register into memory – store
After move data into registers:
- Arithmetic operations can now work directly on registers only (much faster!)

Then move back the values from register to their "home" in memory

General Purpose Registers: 32 [32-bits(4-bytes) registers] in MIPS

- 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:** A register can hold any 32-bit number
  - The number has <u>no implicit data type</u> and is interpreted <u>according to the instruction that uses it</u>
  - Unlike program variables!
  - Machine/Assembly instruction assumes the data stored in the register is of the correct type

MIPS Assembly Language:

- Each line of assembly code contains at most 1 instruction

The **major types** of assembly instruction:

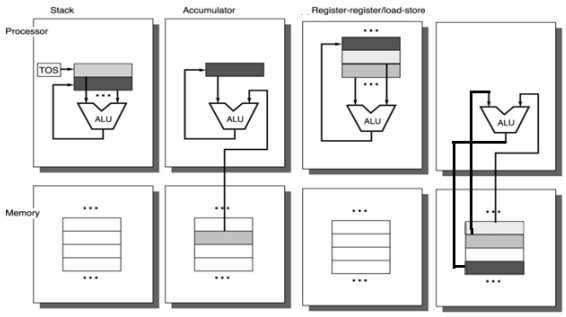
- Memory: Move values between memory and registers
- Calculation: Arithmetic and other operations [load/store]
- Control flow: Change the sequential execution [branch and jump]

RISC vs CISC

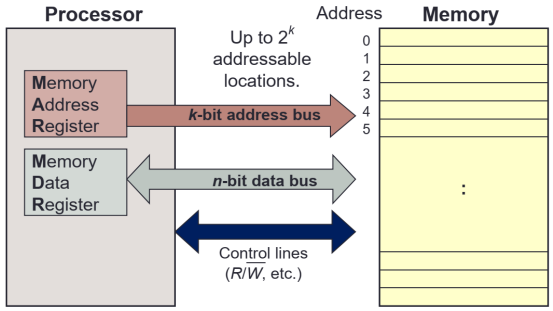
	RISC	CISC
Full Name	Reduced Instruction Set Computer	Complex Instruction Set Computer
Example	MIPS, ARM	x86-32 (IA32)
Instruction	Keep the instruction set small and simple, makes it easier to build/optimise hardware	Single instruction performs complex operation
Operation	Burden on software to combine simpler operations to implement high-level language statements	Complex implementation, no room for hardware optimization
Storage Architecture	Register-Register (Load/Store) design	a mixture of Register-Register and Register-Memory

Storage Architecture

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	



Memory Address and Content



Given k-bit address, the address space is of size 2^k ($0 \sim 2^k-1$)

Each memory transfer consists of one word of n bits

Endianness: The relative ordering of the bytes in a multiple-byte word stored in memory

Big-endian:	Little-endian:
Most significant byte stored in lowest address. **Example:** IBM 360/370, Motorola 68000, ***MIPS*** (Silicon Graphics), SPARC.	Least significant byte stored in lowest address. **Example:** Intel 80x86, DEC VAX, DEC Alpha.
Example: **0xDE AD BE EF** Stored as: 0 DE, 1 AD, 2 BE, 3 EF	Example: **0xDE AD BE EF** Stored as: 0 EF, 1 BE, 2 AD, 3 DE

NOTE: The endian-ness of MIPS is actually implementation specific.

Addressing Modes (Only 3 in MIPS)

Mode	Character	Example
Register	Operand is in a register	add $t1, $t2, $t3
Immediate	Operand is specified in the instruction directly	addi $t1, $t2, 98
Displacement	Operand is in memory with address calculated as **Base + Offset**	lw $t1, 20($t2)

Addressing Mode	Example	Meaning
Register	`Add R4, R3`	`R4 ← R4 + R3`
Immediate	`Add R4, #3`	`R4 ← R4 + 3`
Displacement	`Add R4, 100(R1)`	`R4 ← R4 + M[100+R1]`
Register Indirect	`Add R4, (R1)`	`R4 ← R4 + M[R1]`
Indexed / Base	`Add R3, (R1 + R2)`	`R3 ← R3 + M[R1 + R2]`
Direct or Absolute	`Add R1, (1001)`	`R1 ← R1 + M[1001]`
Memory Indirect	`Add R1, @R3`	`R1 ← R1 + M[M[R3]]`
Auto-Increment	`Add R1, (R2)+`	`R1 ← R1+M[R2]; R2 ← R2+d`
Auto-Decrement	`Add R1, -(R2)`	`R2 ← R2-d; R1 ← R1+M[R2]`
Scaled	`Add R1, 100(R2)[R3]`	`R1 ← R1 + M[100+R2+R3×d]`

Instruction

	Variable-length instructions	Fixed-length instructions
Example	Intel 80x86(1-17 bytes) Digital VAX (1-54 bytes)	Used in most RISC MIPS, PowerPC (4 bytes)
Fetch and Decode	Require multi-step fetch and decode	Allow for easy fetch and decode
Instruction	Allow for a more flexible (but complex) and compact instruction set	Instruction bits are scarce.

opcode: unique code to specify the desired operation

operands: zero or more additional information needed for the operation.

The operation designates the **type** and **size** of the operands

Type	Size/bits
Character	8
Half-word	16
word	32
single-precision floating point	32
double-precision floating point	64

Expectations from any new 32-bit architecture: Support for 8-, 16- and 32-bit integer and 32-bit and 64-bit floating point operations. A 64-bit architecture would need to support 64-bit integers as well.

Encoding

	opcode	address	address
Type-A	4 bits	4 bits	4 bits
	opcode	funct?	address
Type-B	4 bits	4 bits	4 bits

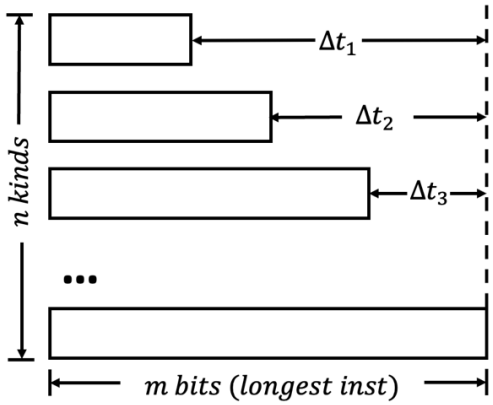
# of instructions

Maximum: $1 + (2^6 - 1) \times 2^5 = 1 + 63 \times 32 = 2017$

Minimum: $1 \times 2^5 + 2^6 - 1$ *(1 is used for type-B)* = 95

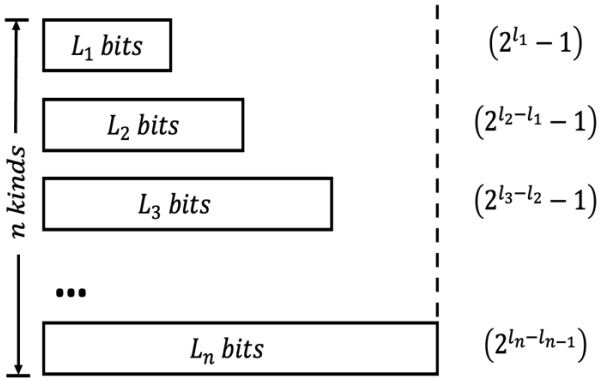
General Solution

Find Maximum # instruction



$$N_{max} = 2^m - \sum_i 2^{\Delta t_i} + (n-1)$$

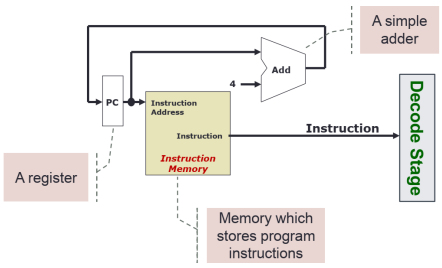
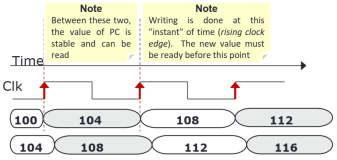
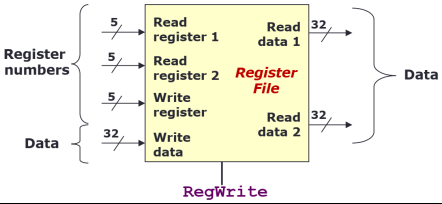
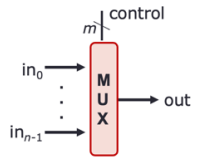
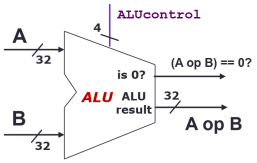
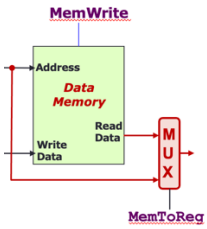
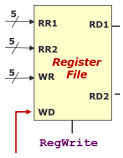
Find Minimum # instruction



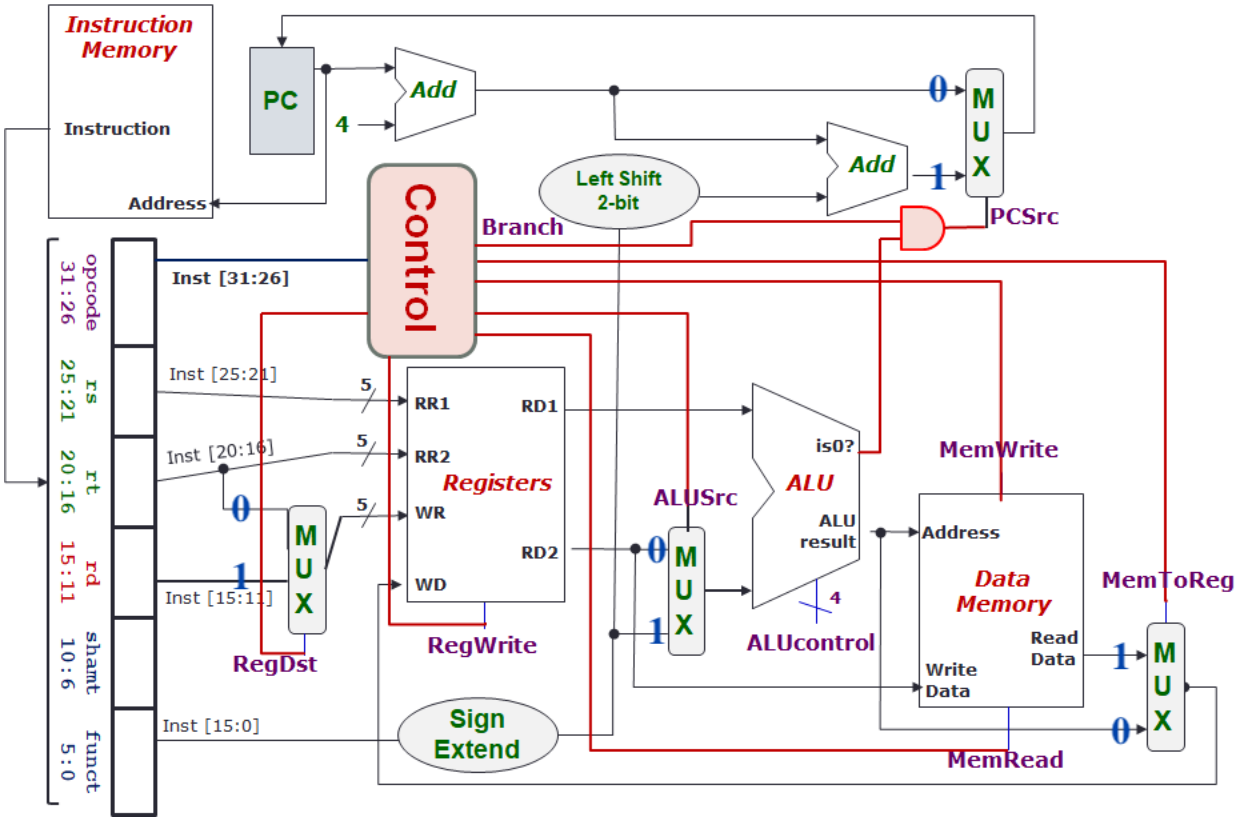
$$N_{min} = (2^{l_1} - 1) + (2^{l_2 - l_1} - 1) + \cdots$$
$$+ (2^{l_{n-1} - l_{n-2}} - 1) + (2^{l_n - l_{n-1}})$$

MIPS Processor

MIPS Instruction Execution Summary

Fetch Stage		1 2	1. Use the Program Counter (PC) to **fetch the instruction** from memory (first 1/2 of clock period) 2. **Increment** the PC by 4 to get the address of the next instruction (update at next rising clock edge) 
		3	Output to the next stage (Decode)
Decode Stage			**Register File:** Collection of registers. register number → Register File (32 32-bit Registers) $rs → Read Register 1 → Read Data 1 $rt → Read Register 2 → Read Data 2
			**Multiplexer [RegDst and ALUSrc]** Function: Selects one input from multiple input lines Inputs: n lines of same width Control: m bits where $n = 2^m$ Output: Select i^{th} input line if control = i (one of inputs)
ALU Stage			Branch instruction: we need to perform two Calculations The **1-bit** "isZero" signal: handle equal/not equal check **Branch Target Address:** ▪ Need PC (from Fetch Stage) ▪ Need Offset (from Decode Stage)
Memory Stage			Only the **load** and **store** instructions: Use memory address calculated by **ALU Stage** **Output:** Data read from memory (**RD**) for **load inst** Other instructions: Result from ALU Stage will pass through to be used in Register Write stage
Register Write Stage			**Exceptions:** store, branch and jump [No result to write] Result Write stage has no additional element: ▪ Basically just route the correct result into register file ▪ Read result from memory stage to Write Data and Write Register from decode stage.

Datapath & Control



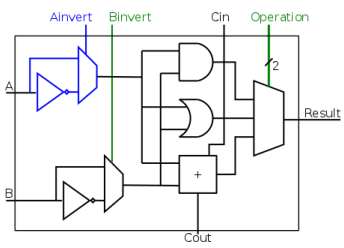
Control Signal generated based on the instruction to be executed

	Stage	0	1
RegDst	Decode/Fetch	Write Register = Inst[20:16] **$rt** [R-format]	Write Register = Inst[15:11] **$rd**
RegWrite	Decode/Fetch	No register write	New value will be written
ALUSrc	ALU	ALUop2 = Read Data 2	ALUop2 = sign_extend(Inst[15:0])
MemRead	Memory	Not performing memory read access	Read memory using ALU result as Address
MemWrite	Memory	Not performing memory write operation	Write to memory at Address (from ALU result) with data from Read Data 2
MemToReg	RegWrite	Register Write Data = ALU result	Register Write Data = Memory Read Data
PCSrc [Branch && isZero]	Memory/ RegWrite	$PC' = $PC+4	$PC' = ($PC+4)+(immediate×4) = ($PC+4)+SignExt(**Inst[15:0]**) << 2
Branch		The operation is not a branch operation	The operation is a branch operation (*i.e.,* beq)

Control Signals Summary Table

	RegDst	RegWrite	ALUSrc	ALUcontrol/op	MemRead	MemWrite	MemToReg	Branch
R	1	1	0	?/?	0	0	0	0
lw	0	1	1	0010/00	1	0	1	0
sw	X	0	1	0010/00	0	1	X	0
beq	X	0	0	0110/01	0	0	X	1

ALU control in bitwise operation



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

Two-level Implementation

Step 1 Generate 2-bit ALUop signal from 6-bit opcode

Step 2 Generate ALUcontrol signal from ALUop and **optionally** 6-bit Funct field

ALU control truth table

	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	0010
sw	0	0	X	X	X	X	X	X	0010
beq	0	1	X	X	X	X	X	X	0110
add	1	0	1	0	0	0	0	0	0010
sub	1	0	1	0	0	0	1	0	0110
and	1	0	1	0	0	1	0	0	0000
or	1	0	1	0	0	1	0	1	0001
slt	1	0	1	0	1	0	1	0	0111

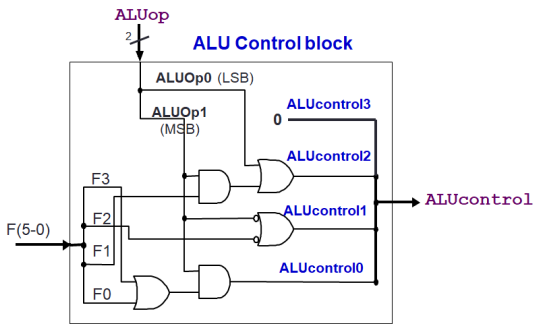
ALU control Formula

ALUcontrol3 = 0

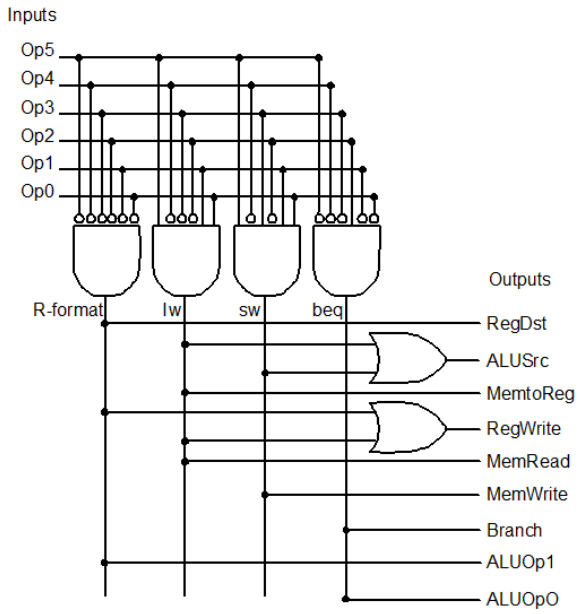
ALUcontrol2 = ALUop0 || (ALUop1 && F1)

ALUcontrol1 = !ALUop1 + !F2

ALUcontrol0 = F0 && F3 && ALUop1



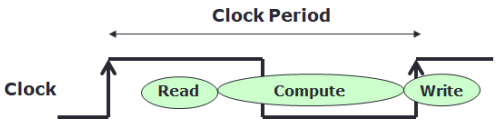
Control Unit: use Op0 to Op5 to mean the bit 0 to bit 5 of the opcode.



Ctrl	Signal	Selector
0	`RegDst`	*R-Format*
1	`ALUSrc`	`lw` OR `sw`
2	`MemToReg`	`lw`
3	`RegWrite`	*R-Format* OR `lw`
4	`MemRead`	`lw`
5	`MemWrite`	`sw`
6	`Branch`	`beq`
7	`ALUop1`	*R-Format*
8	`ALUop0`	`beq`

Single Cycle

A naive implementation is to have all of the execution above within a single clock period. This means that we have to split a single clock cycle into 3 parts. One possibilities is as follows to prevent reading storage element when it is being written.



Problem: clock speed has to accommodate the slowest instruction. Consider the following example:

Solution:

(a) **Multicycle**: break up the instruction into execution steps. The simplest one is really to just break it up to the same execution steps as the stage. So we have 5 steps for one instruction. What this means is that each instruction takes up to 5 execution steps where each execution step is 1 clock cycle.

Advantage: each clock cycle is smaller (i.e., faster) but each instruction takes more clock cycle to execute. This may be advantageous if each instructions can take variable number of clock cycles to complete.

(b) **Pipelining:** Similar to multicycle, but we take further steps to optimise. Consider what happens when we are currently executing the ALU stage. Note that the instruction memory is now idle! That is a waste of perfectly good component.