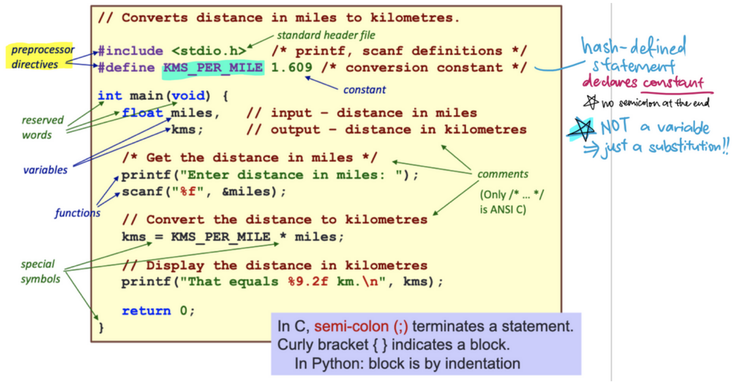
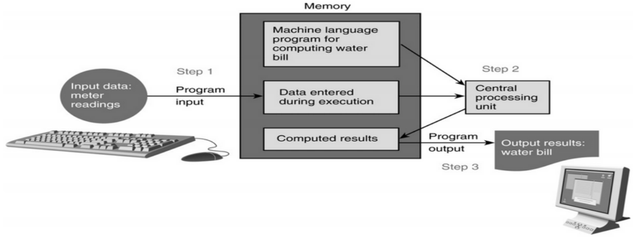
01. C Syntax

- uninitialised variables will contain random values



Programme Structure

- A basic C program has 4 main parts:
  - Preprocessor directives:
    - eg: #include <stdio.h>, #include <math.h>, #define PI 3.142
  - Input: through stdin (using scanf), or file input
  - Compute: through arithmetic operations and assignment statements
  - Output: through stdout (using printf), or file output



Assignment Statement

- returns the value of the variable assigned

```
a = 12;          // will return 12
z = (a = 12);    // hence this is possible
z = a = 12;      // same thing
```

Operations

- binary operators: `+ - * / %`
  - left associative ⇒ evaluates left to right
  - e.g. 45 / 15 / 2 ⇒ 3 / 2 ⇒ 1
  - `%` is remainer (not modulo) ⇒ `-10%4 = -2`
- unary operators: `+ -`
  - right associative → it 'belongs' to the item on its right
  - -6/3 is the same as (-6)/3 NOT -(6/3)

Associativity & Precedence

INCREASING PRECEDENCE ↑

Operator Type	Operator	Associativity
Primary expression operators	`( )` *expr++ expr--*	Left to right
Unary operators	`* & + -` *++expr --expr (typecast)*	Right to left
Binary operators	`* / %`	Left to right
	`+ -`	
Assignment operators	`= += -= *= /= %=`	Right to left

switch/case

- **fall-through behaviour** → if you remove all the `break`s, when the value matches a case, every case afterwards will be run

```
switch (<variable or expression>) {
  case value1:
    /* ... */
    break;

  case value2:
    /* ... */
    break;

  default:
    /* executes if <var/exp> not equals value1 or value2 */
    break;
}
```

loops

```
while ( condition )
{
   // loop body
}
```

```
do
{
   // loop body
} while ( condition );
```

```
for ( initialization; condition; update )
{
   // loop body
}
```

Initialization: initialize the **loop variable**

Condition: repeat loop while the condition on **loop variable** is true

Update: change value of **loop variable**

02. C Syntax (types & pointers)

Typing

- **strongly-typed** → every variable has to be declared with a data type
- **weakly-typed** → type depends on how the variable is used

Truth Values

- FALSE values: `false` or `0` or `null`
- TRUE values: everything else
  - `true` will be printed as `1`

Data Types

- `int`: 4 bytes, -2^{31} to $2^{31}-1$
- `float` / `double`: 4 bytes / 8 bytes
- `char`: 1 byte

Mixed-Type Arithmetic

```c
int m = 10/4;      // m = 2
int n = 10/4.0;    // n = 2
float p = 10/4;    // p = 2.0, same as float p = (float)(10/4)
float q = 10/4.0;  // q = 2.5
```

Type Casting

```c
/* syntax: (type) expression */
int ii = 5; float ff = 15.34
float a = (float) ii / 2;    // a = 2.5
float b = (float) (ii / 2);  // b = 2.0, floor division
int c = (int) ff / ii;       // c = 3
```

syntax

```c
#include <stdio.h>
#define MAX 10

// function prototype:
int readArray(int [], int);

int main(void) {
  int array[MAX];
  readArray(array, MAX);
  return 0;
}

// function definition:
int readArray(int arr[], int n) {
  printf("Enter up to %d integers.\n", n);
  int i;
  for (i = 0; i < n; i++) {
    scanf("%d", &arr[i]);
  }
}
```

Placeholder	Variable Type	Function Use
%c	char	printf / scanf
%d	int	printf / scanf
%f	float or double	printf
%f	float	scanf
%lf	double	scanf
%e	float or double	printf (for scientific notation)

format specifiers

- `%5d` - integer in a width of 5, right justified
- `%8.3f` - real number in a width of 8, with 3 dp, right justified

💡 for `scanf`: use format specifier without indicating width, dp, etc

- `%p` specifier: prints address (for pointers)

pointers

- **pointer** → a variable that contains the address of another variable
  - symbol table keeps track of the variable's address

```c
int main(void) {
  int a = 3, *b;  // b is a pointer to an int
  b = &a;         // b points to the address of a
  *b = 5;         // set a through b, a=5
}
```

- `&` - address operator
  - `&x` → address of the memory cell where the value of `x` is stored
  - gets the address of a variable
- `*` - declares a pointer
  - `type *pointer_name` (e.g. `int *x`)
- `*` - dereferencing (access variable through pointer)
  - `*x = 32`
  - following through the pointer to get the value
- format specifier for pointers: `%p`
- incrementing a pointer: brackets needed - `(*p1)++;`
  - without brackets: increments pointer to next address (depending on size of the data type) aka += sizeof(*p1)

```c
double a, *b;
b = &a;  // legal
double c, d;
*d = &c;  // legal
double e, f;
f = &e;  // ILLEGAL!
```

03. Number Systems

Data Representation

- 1 byte = 8 bits
- word = multiple of a byte (e.g. 1 byte, 2 bytes, 4 bytes)
  - 64-bit machine ⇒ 1 word is 8 bytes
- N bits can represent up to 2^N values
- to represent M values: $\lceil \log_2 M \rceil$ bits required

Weighted Number systems

- **weighted** number system → has a **base (radix)**
  - base/radix R has weights in powers of R

Prefixes in C

- prefix `0` for octal (e.g. `032` = $(32)_8$)
- prefix `0x` for hexadecimal (e.g. `0x32` = $(32)_{16}$)
- prefix `0b` for binary

Conversion

- decimal to binary
  - whole numbers: repeated **division** by 2, LSB → MSB
  - fractions: repeated **multiplication** by 2, MSB → LSB
- decimal to base-R:
  - for whole numbers: repeated **division** by R
  - for fractions: repeated **multiplication** by R
- binary → octal: partition in groups of 3
- octal → binary: convert each digit into 3-bit binary
- binary → hexadecimal: partition in groups of 4
- hexadecimal → binary: convert each digit into 4-bit binary

ASCII

- American Standard Code for Information Interchange
- 7 bits plus 1 parity bit (for error checking) ⇒ $2^7 = 128$
- in C: `char` datatype is 1 byte = 8 bit integer
  - corresponds to ASCII - can typecast int/char
  - e.g. convert uppercase char to lowercase: `c = c + 'a' - 'A'`

Negative Numbers

- **unsigned** numbers: only non-negative values
- **signed** numbers: include all values (positive and negative)

📌 for negating non-whole numbers: same as whole numbers (ignore the decimal point, then put it back)

Overflow

- positive + positive = negative, OR
- negative + negative = positive

1s addition:

- if there is a carry out, add 1 to the result (wrap around)

```
  -2     1101
+ -5   + 1010
----   -------
  -7    10111
----   +    1
       -------
        1000
       -------
```

No overflow

```
  -3     1100
+ -7   + 1000
----   -------
 -10    10100
----   +    1
       -------
        0101
       -------
```

Overflow!

2s addition:

- ignore the carry out

Sign-and-Magnitude

- MSB represents the sign (0 is positive)
- **range (8-bit)**: -127_{10} to $+127_{10}$
  - 2 zeroes: `00000000` ($+0_{10}$) and `10000000` (-0_{10})
- **negating a number**: reverse the first bit
- **issues**
  1. there are two zeroes (which may be useful for limits!)
  2. not good for performing arithmetic due to the zero in front

1s Complement

- negated value of x, $-x = 2^n - x - 1$
- **negating a number**: invert the bits
- **range (8-bit):** -127_{10} to $+127_{10}$
  - 2 zeroes: `00000000` ($+0_{10}$) and `11111111` (-0_{10})
  - **range (n-bits):** -2^{n-1} to 2^{n-1}

2s Complement

- = 1s complement + 1
- negated value of x, $-x = 2^n - x$
- **negating a number**
  - invert the bits, then add 1
- **range (8-bit):** -128_{10} to $+127_{10}$
  - zero: `00000000` $= +0_{10}$
  - **range (n-bits):** -2^{n-1} to $2^{n-1} - 1$

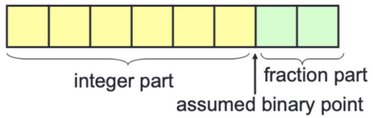
Excess Representation

- `00..00` $= -2^n$
- `10..00` $= 0$
- to express n in Excess-M representation: $n + M$

04. Number Representations

Fixed-point representation

- reserve a certain number of bits for the whole number part and the fraction part



  - If 2s complement is used, we can represent values like:
    $011010.11_{2s} = 26.75_{10}$
    $111110.11_{2s} = -000001.01_{2} = -1.25_{10}$

- **issues**: limited range

Floating-point representation

- IEEE 754 floating-point representation
  - exponent is excess-127

single-precision: 1-bit sign / 8-bit exponent / 23-bit mantissa

- 3 components:
  1. sign
  2. exponent
  3. **mantissa** (fraction)
- mantissa is **normalised** with an implicit leading bit 1
  - to maximise the numbers to be stored
  - normalise it to $1.xxxx * 2^n$ ⇒ the rightmost bit is always 1 ⇒ no need to store it
- comparison to fixed-point representation
  - **advantage**: much better range and accuracy
  - **disadvantage**: more complex representation

05. Arrays, Strings & Structs

Arrays

- a **homogenous** collection of data → data is all of the same type
- declaration: `arr = elementType[size]`
  - `arr` refers to `&arr[0]`
- an array name is a fixed (**constant**) pointer
  - points to the first element in the array
  - cannot be reassigned - `arr1 = arr2` is illegal!

```c
// an array can ONLY be initialised at the time of declaratio
int evens[5] = {2, 4, 6, 8, 10};
// if you initialise values, no need to declare length of arr
int odds[] = {1, 3, 5};
// uninitialised values will be zero value
int some[5] = {1, 2, 3};  // some = [1, 2, 3, 0, 0]

int numbers[3];
printf("Enter 3 integers:");
for (i = 0; i < 3; i++) {
  scanf("%d", &numbers[i]);
}
```

in function prototypes

```c
// parameter names are optional
int sumArray(int [], int);          // valid
int sumArray(int arr[], int size);  // valid
int sumArray(int *, int);  // pointer is valid too

// size can be specified but will be ignored
int sumArray(int arr[8], int size);

// function definition
int sumArray(int *arr, int size) {  ...  }
int sumArray(int arr[8], int size) {  ...  }  // size ignored
```

Strings

- array of characters terminated with a null character `\0`
  - ASCII value of 0
- string functions: `#include <string.h>`

```c
char my_str[] = "hello";
char my_str[] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

I/O

- in
  - `fgets(str, size, stdin)` - reads (size - 1) chars, or until newline
  - `scanf("%s", str)` - reads until whitespace
- out
  - `puts(str)` - terminates with newline
  - `printf("%s\n", str)` - prints until it encounters `\0` in `str`

string functions

- `strlen(s)` → returns number of characters in s up to `\0`
- `strcmp(s1, s2)` → compares the ASCII values of corresponding characters
  - returns s1 - s2
  - negative number / positive number / 0 if they are equal
- `strncmp(s1, s2, n)` → strcmp for first n characters of s1 and s2
- `strcpy(s1, s2)` → copy s2 into s1
  - cannot directly assign `s1 = "Hello"`
  - can copy: `strcpy(s1, "Hello")`
- `strncpy(s1, s2, n)` → copy first n characters of s2 into s1

Structs

- allow grouping of heterogenous data
- passed by value into functions
  - unless: passing array of structs to a function
  - array members of structs are deeply copied
- can be reassigned

```c
// declare BEFORE function prototypes
typedef struct {
  int length, width;
  float height;
} box_t;

typedef struct {
  int id;
  box_t smaller_box;
} nested_box_t;

// initialising struct variables
box_t mybox = {2, 3, 5.1};
nested_box_t big_box = {0, {4, 3, 6.7}};

// accessing members
box.length = 1;
big_box.smaller_box.width = 2;
```

create new types called `box_t` and `nested_box_t`

💡 no memory is allocated to a type

arrow operator: `->`

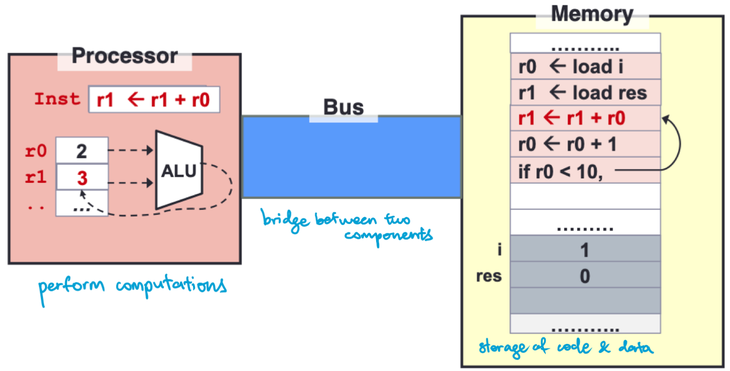
- `(*player_ptr).name` is equivalent to `player_ptr->name`

💡 `*player_ptr.name` means `*(player_ptr.name)` (dot has higher precedence)

06. MIPS + ISA

Instruction Set Architecture

- **ISA** → abstraction of the interface between the hardware and low-level software
  - software: to be translated into the instruction set
  - hardware: implements the instruction set
- **stored-memory concept** → both **instructions** and **data** are stored in memory
- load-store model → limit memory operations and relies on registers for storage during execution
- C –compiler→ Assembly –assembler→ Machine code



- major types of assembly instruction:
  - **memory**: move values between memory and registers
  - **calculation**: arithmetic and other operations
  - **control flow**: change the sequential execution (sequence in which instructions are executed)

Registers

Name	Register number	Usage
`$zero`	0	Constant value 0
`$v0-$v1`	2-3	Values for results and expression evaluation (pass function values out)
`$a0-$a3`	4-7	Arguments (to pass params to functions)
`$t0-$t7`	8-15	Temporaries
`$s0-$s7`	16-23	Program variables

Name	Register number	Usage
`$t8-$t9`	24-25	More temporaries
`$gp`	28	Global pointer
`$sp`	29	Stack pointer
`$fp`	30	Frame pointer
`$ra`	31	Return address

(try not to touch)

$at (register 1) is reserved for the assembler.
$k0-$k1 (registers 26-27) are reserved for the operation system.

07. MIPS Assembly Language

instructions

Operation	Opcode in MIPS	Immediate Version (if applicable)
Addition	`add $s0, $s1, $s2`	`addi $s0, $s1, C16`$_{2s}$ `C16`$_{2s}$ is [-2^{15} to 2^{15}-1]
Subtraction	`sub $s0, $s1, $s2`	
Shift left logical	`sll $s0, $s1, C5` `C5` is [0 to 2^5-1]	
Shift right logical	`srl $s0, $s1, C5`	
AND bitwise	`and $s0, $s1, $s2`	`andi $s0, $s1, C16` `C16` is a 16-bit pattern
OR bitwise	`or $s0, $s1, $s2`	`ori $s0, $s1, C16`
NOR bitwise	`nor $s0, $s1, $s2`	
XOR bitwise	`xor $s0, $s1, $s2`	`xori $s0, $s1, C16`

- `add $s0, $s1, $zero` synonymous with `move $s0, $s1`
- to get a **"NOT" operation**: `nor $t0, $t0, $zero`
- `lui` → load upper immediate (sets the upper 16 bits of the register)

> 💡 registers have no type
> ⇒ contains either **data values** or **memory address**

loading large constants

1. use `lui` to set the upper 16 bits (e.g. `lui $t0, 0xAAAA`)
   - lower bits filled with zeroes
2. use `ori` to set the lower-order bits (e.g. `ori $t0, $t0, 0xF0F0`)
   - lower bits will be set

memory instructions

- `lw target, disp(src)` → load contents of Mem[src+disp] to target
- `sw src, disp(target)` → store contents of src to Mem[targ+disp]
- `lb` / `sb` → Load/Store byte (doesn't need word-align)

control flow

- `bne` → branch if Not Equal (`bne $t0, $t1, label`)
- `beq` → branch if Equal (`beq $t0, $t1, label`)
- `j` → jump unconditionally (`beq $t0, $t1, label`)
- `slt` → set to 1 on less than, else 0 (`slt dest, src1, src2`)

instruction formats

R-format (Register format: `op $r1, $r2, $r3`)
- Instructions which use 2 source registers and 1 destination register
- `e.g. add, sub, and, or, nor, slt, etc`
- **Special cases:** `srl, sll`, etc.

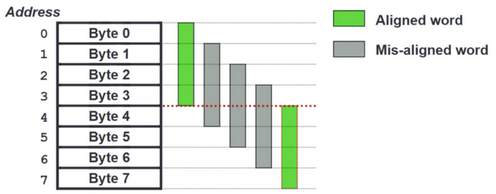
I-format (Immediate format: `op $r1, $r2, Immd`)
- Instructions which use 1 source register, 1 immediate value and 1 destination register
- `e.g. addi, andi, ori, slti, lw, sw, beq, bne`, etc.

J-format (Jump format: `op Immd`)
- `j` instruction uses only one immediate value

memory organisation

- each location has an **address** → an index into the array
  - for a k-bit address, the address space is of size 2^k
  - largest address possible: $2^k - 1$
    - bc we start from 0
- **byte addressing** → one byte (8 bits) in every location/address
  - more than one byte → **word addressing**
- load-store architectures can only load data at word boundaries (divisible by n bytes)

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



MIPS

- load-store register architecture
  - 32 registers, each 32-bit (4 bytes) long
  - each word contains 4 bytes
  - memory addresses are 32-bit long
- 2^{30} memory **words** ($2^{32}/4$)
  - accessed only by data transfer instructions (aka **memory instructions**)
- MIPS uses **byte addresses** ⇒ consecutive words (word boundaries) differ by 4
  - e.g. `Mem[0], Mem[4], ...`

08. MIPS Instruction Encoding

Instruction Formats

MIPS_Reference_Data_page1.pdf 89.7KB

R-Format

6	5	5	5	5	6
opcode	rs	rt	rd	shamt	funct

each field is a 5/6-bit **unsigned** integer
opcode always = 0, shamt set to 0 for all non-shift instructions
rs set to 0 for sll/srl

I-Format

6	5	5	16
opcode	rs	rt	immediate

immediate is a **signed integer 2s complement** (up to 2^{16} values)

J-Format

6 bits	26 bits
opcode	target address

- MIPS will take the 4 MSBs from PC+4 (next instruction after the jump instruction)
- omit 2 LSB since instruction addresses are word-aligned
- maximum jump range ⇒ $2^{26+2+4} = 2^{32}$

PC-Relative Addressing

- **Program Counter** (PC) → special register that keeps address of the instruction being executed in the processor
- target address = PC + 16-bit `immediate` field
  - can branch $\pm 2^{15}$ words $= \pm 2^{17}$ bytes from the PC
  - interpret `immediate` as the number of words since instructions are word-aligned ⇒ larger range!
- next branch calculation:
  - if branch is not taken: PC+4
  - if branch is taken: (PC+4) + (immediate x 4)

Summary

MIPS assembly language

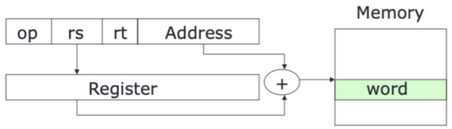
Category	Instruction	Example	Meaning	Comments
	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
Arithmetic	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Used to add constants
	load word	`lw $s1, 100($s2)`	`$s1 = Memory[$s2 + 100]`	Word from memory to register
	store word	`sw $s1, 100($s2)`	`Memory[$s2 + 100] = $s1`	Word from register to memory
Data transfer	load byte	`lb $s1, 100($s2)`	`$s1 = Memory[$s2 + 100]`	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	`Memory[$s2 + 100] = $s1`	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$s1 = 100 * 2^16	Loads constant in upper 16 bits
	branch on equal	`beq $s1, $s2, 25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
Conditional branch	branch on not equal	`bne $s1, $s2, 25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1, $s2, $s3`	if (`$s2 < $s3`) `$s1` = 1; else `$s1` = 0	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if (`$s2 < 100`) `$s1` = 1; else `$s1` = 0	Compare less than constant
	jump	`j 2500`	go to 10000	Jump to target address
Uncondi-tional jump	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

Addressing Modes

- **register addressing** → operands are registers

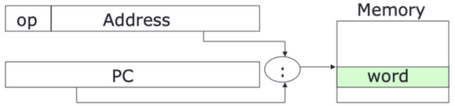
- **immediate addressing** → operand is a constant encoded in the instruction itself (e.g. `ori`, `addi`, `andi`, `slti`)

- **base addressing** (aka **displacement addressing**) → operand is at the memory location whose address is the sum of a register and a constant in the instruction
  - e.g. `lw`, `sw` - base address (specified by register) + displacement



- **PC-relative addressing** → address is the sum of PC and constant in the instruction (e.g. `beq`, `bne`)
  - branch address is relative to PC+4

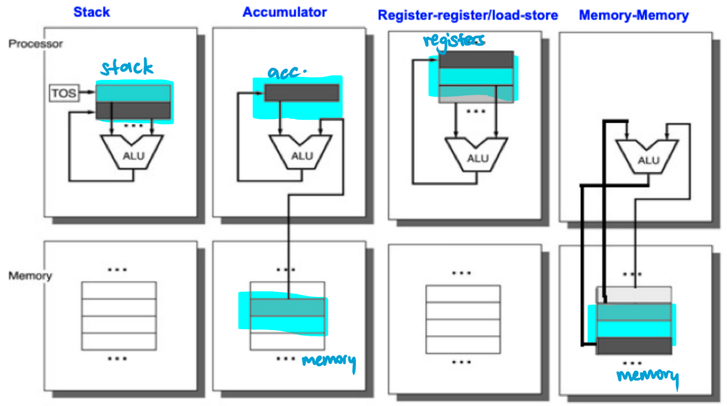
- **pseudo-direct addressing** → 26-bit of instruction concatenated with the 4 MSBs of PC (e.g. `j`)



09. Instruction Set Architecture

1. Data Storage

Storage Architectures



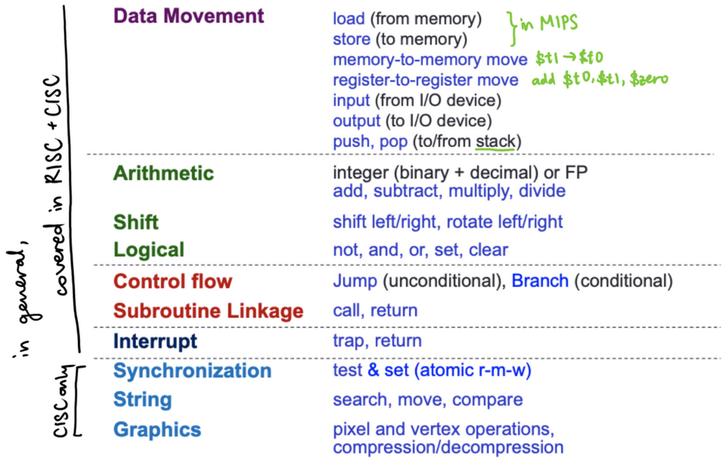
ISA	Instructions	Explanation
Stack	**push** *@src* **pop** *@dest* **add**	Load value in *@src* onto top of stack. Transfer value at top of stack to *@dest*. Remove top two values in stack, add them, and load the sum onto top of stack.
Accumulator	**load** *@src* **add** *@src* **store** *@dest*	Load value in *@src* into accumulator. Add value in *@src* and value in accumulator, and put sum back into accumulator. Store the value in accumulator into *@dest*.
Memory-Memory	**add** *@dest, @src1, @src2*	Add values in *@src1* and *@src2*, and put the sum into *@dest*.
Register-Register	**load** *$reg, @src* **add** *$dest, $src1, $src2* **store** *$reg, @dest*	Load value in *@src* into *$reg*. Add values in *$src1* and *$src2*, and put sum into *$dest*. Store value in *$reg* into *@dest*.

2. Memory Addressing Modes

- **endianness** → the relative ordering of bytes in a multiple-byte word stored in memory
  - **big** endian → MSB stored in lowest address
  - **small** endian → LSB stored in lowest address
- **addressing mode** → ways to specify an operand in an assembly
- 3 addressing modes in MIPS
  1. **register** → operand is in a register (e.g. `add $t1, $t2, $t3`)
  2. **immediate** → operand is specified directly in the instruction (e.g. `addi $t1, $t2, 98`)
  3. **displacement** → operand is in memory with address calculated as base + offset (e.g. `lw $t1, 20($t2)`)
     - a form of immediate mode instruction

3. Operations in the Instruction Set

- every instruction set should have a set of standard operations



4. Instruction Formats

instruction length

- fixed-length instructions
  - ✓ easy fetch and decode
  - ✓ simplified pipelining and parallelism
  - ✓ instruction bits are scarce
- variable-length instructions
  - × require multiple steps to fetch and decode instructions
  - ✓ more flexible

instruction fields

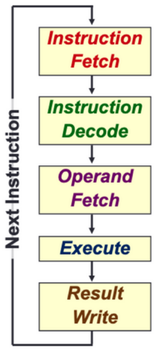
- type and size of operands (i.e. how to divide up the instructions)
- instruction costs of:
  - **opcode** → unique code to specify the desired operation
    - designates the type and size of operands
  - **operands** → zero or more additional information needed for the instruction
- 32-bit architecture should support
  - 8-, 16-, 32-bit integer operations
  - 32- and 64-bit floating point operations

5. Instruction Encoding

- choice of variable/fixed/hybrid encoding
- **expanding opcode scheme** → opcode variable lengths for different instructions ⇒ maximise instruction bits
  - use unused bits to define opcode ⇒ larger instruction set

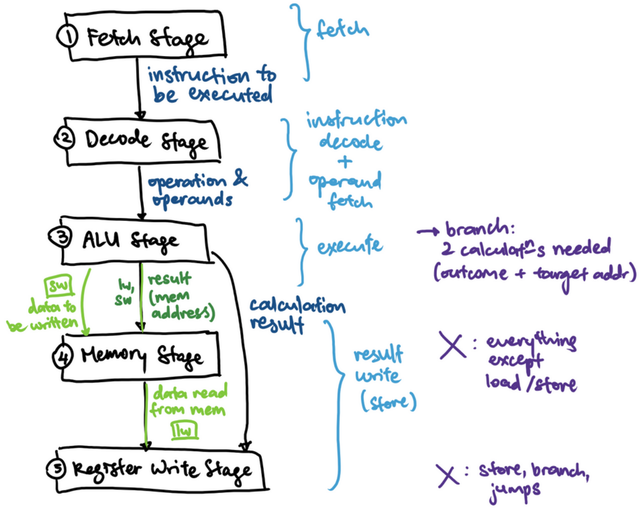
10. Datapath

Instruction Execution Cycle



1. **Fetch:**
   - Get instruction from memory
   - Address is in **P**rogram **C**ounter (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

in MIPS



overview

- programmer writes program in high-level language (e.g. C)
- compiler translates to assembly language (MIPS)
- assembler translates to machine code (binaries)
- processor executes machine code (binaries)

two main components of a Processor

1. **datapath** → collection of components that process data
   - takes in data from operands, processes it, writes the data back
     - performs the arithmetic, logical, memory operations
2. **control** → tells datapath, memory and I/O devices what to do according to program instructions
   - generates control signals

1. Fetch Stage

- use PC to fetch instruction from memory
- increment PC by 4 to get the next instruction (using an Adder)
- output (to Decode): instruction to be executed

2. Decode Stage

- gathers data from the instruction fields
  1. read **opcode** and determine the instruction type and field lengths
  2. read data from all necessary registers
- output (to ALU): operation and the necessary operands

3. ALU (execution) Stage

- output (to memory stage): calculation result

4. Memory Stage

- *only* `load` and `store` instructions needed to perform operations in this stage
  - uses memory address calculated by ALU stage (input)
- all other instructions are idle in this stage
  - result from ALU stage will pass through this stage to be used in Register Write stage
- **inputs**:
  - computation result to be used as memory address
  - register value to be written to memory (only `sw`)
- **outputs** (to Register Write stage): result to be stored (only `lw`)

5. Register Write Stage

- write the result of some computation into a register
  - do nothing: stores / branches / jumps
- input:
  - destination register number
  - computation result (from either memory or ALU)

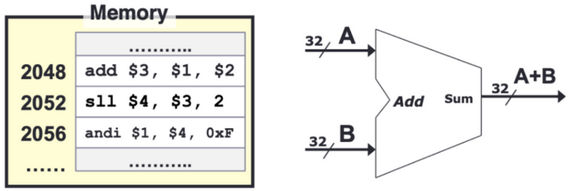
11. Elements of the Datapath

Instruction Memory [1]

- stores instructions (sequential circuit)
- supplies instructions given an address
  - input: instruction address M
  - outputs: contents of address M (binary pattern of instructions)

Adder [1]

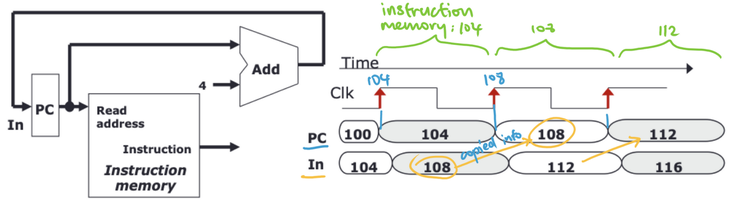
- combinational logic to implement addition of 2 numbers



Memory | Adder

Clock [1]

- a square wave used by the processor
  - times operations inside the processor (e.g. reading & updating PC)
- allows us to read and update the PC at the same time
  - PC is read during the first half of the clock period
  - PC is updated only at the rising edge

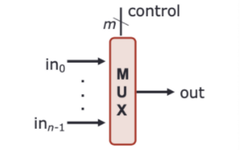


Register File [2]

- collection of 32 registers (each 32 bits wide)
  - can be read by specifying register number
- **read** at most 2 registers per instruction
- **write** at most one register per instruction
- `RegWrite` → control signal to indicate writing of register

Multiplexer

- selects one input from multiple input lines
  - inputs: *n* lines of same width
  - outputs: select i^{th} input line if control = i
  - control: m bits where $n = 2^m$



Control=0 → select in_0 to out
Control=3 → select in_3 to out

Register File | Multiplexer

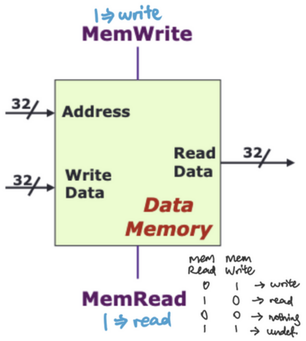
ALU [3]

- combinational logic to implement arithmetic and logical operations
- inputs: two 32-bit numbers
- outputs:
  - result of arithmetic/logical operation
  - 1-bit signal to indicate whether result is zero
- control (`ALUcontrol`): 4-bit to decide the operation
  - set using opcode + funct field
- 2 calculations needed for branch instructions (branch outcome + branch target address)

Data Memory [4]

- storage element for the data of a program
- inputs: memory address
  - & data to be written (for store instructions)
- outputs: data read from memory (for load instructions)
- control: `MemRead` and `MemWrite` controls
  - only one can be asserted at any point in time

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



ALU | Data Memory

12. Control

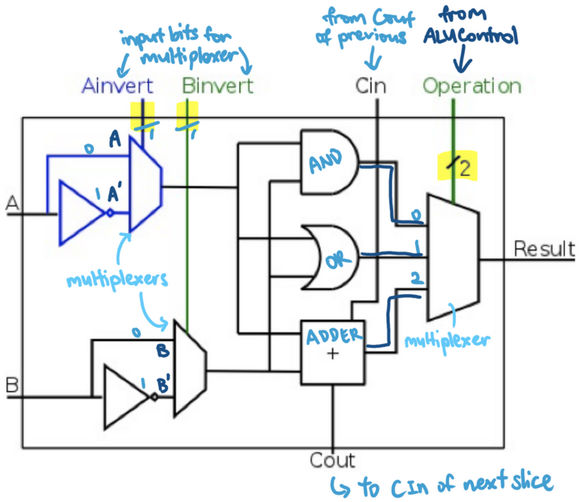
Control Signals

(can all be generated using opcode directly)

- `RegDst` *@ Decode/Operand Fetch*
  - 0/1 → write register = `Inst[20:16]` / `Inst[15:11]`
- `RegWrite` *@ Decode/Operand Fetch*
  - 0/1 → No register write / WD written to WR
- `ALUSrc` *@ ALU* (determines first input)
  - 0 → `Operand2 = Register Read Data 2`
  - 1 → `Operand2 = SignExt(Inst[15:0])` (sign ext immediate)
- `MemRead` *@ Memory*
  - 0/1 → no read / reads memory using *Address* (returned in *RD*)
- `MemWrite` *@ Memory*
  - 0/1 → no write / writes Register RD 2 into mem[*Address*]
- `MemToReg` *@ RegWrite*
  - 0/1 → register write data = ALU result / memory read data
- `PCSrc` *@ Memory/RegWrite*
  - 0/1 → next PC = PC + 4 / PC = SignExt(Inst[15:0]) << 2 + (PC + 4)
  - PCSrc = set to 1 if *Branch* AND *is0* are both 1
    - aka (isBranchInstruction AND branchIsTaken)

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

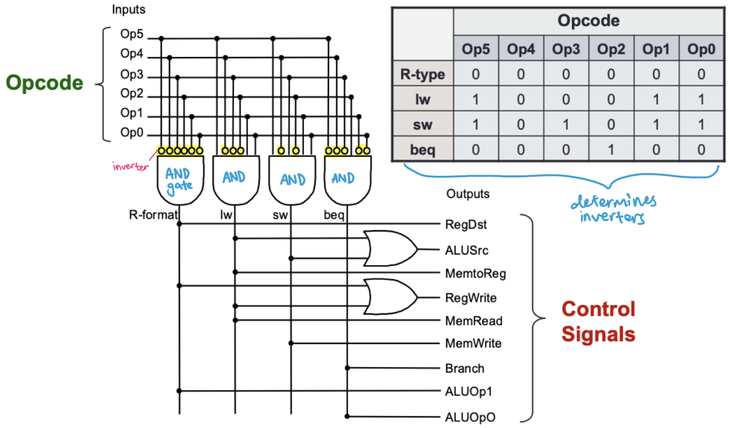
ALU



ALU operation controlled by 2-bit ALUControl

Controller Design

- determines Control Signals from opcode



	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

Multilevel Decoding

- to determine ALUControl signal
  - depends on 12 variables (6-bit opcode + 6-bit funct)
- reduce the number of cases, then generate the full output
  - reduce the size of the main controller - simplify design process
- **how it works**
  1. use opcode to generate 2-bit `ALUop` signal
  2. use `ALUop` signal and funct (for R-type) to generate 4-bit `ALUcontrol` signal

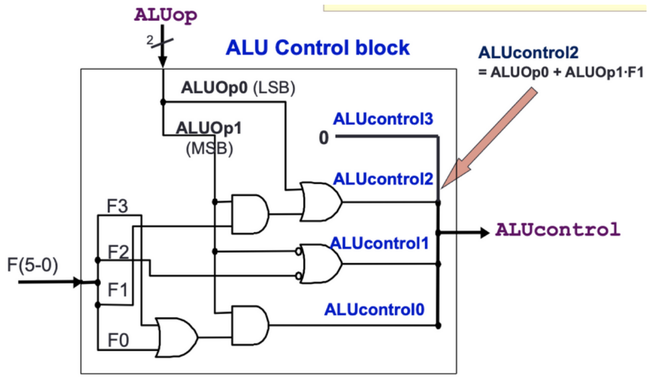
Opcode	ALUop	Instruction Operation	Funct field	ALU action	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

(funct: determines; not used; ALU action: output)

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



ALU Control Block

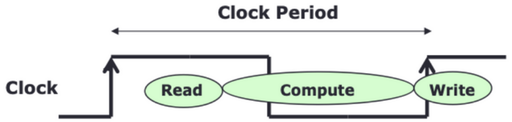
13. Instruction Execution

Instruction Execution

- coordinating the stages together: fetch, decode, memory, write etc
- 2 ways:
  - single cycle implementation
  - multicycle implementation

Single Cycle Implementation

- how it works
  1. read contents of one or more storage elements
  2. perform computation through some combinational logic
  3. write results to one or more storage elements (register/memory)
- all performed within a clock period
  - avoids reading a storage element when it's being written



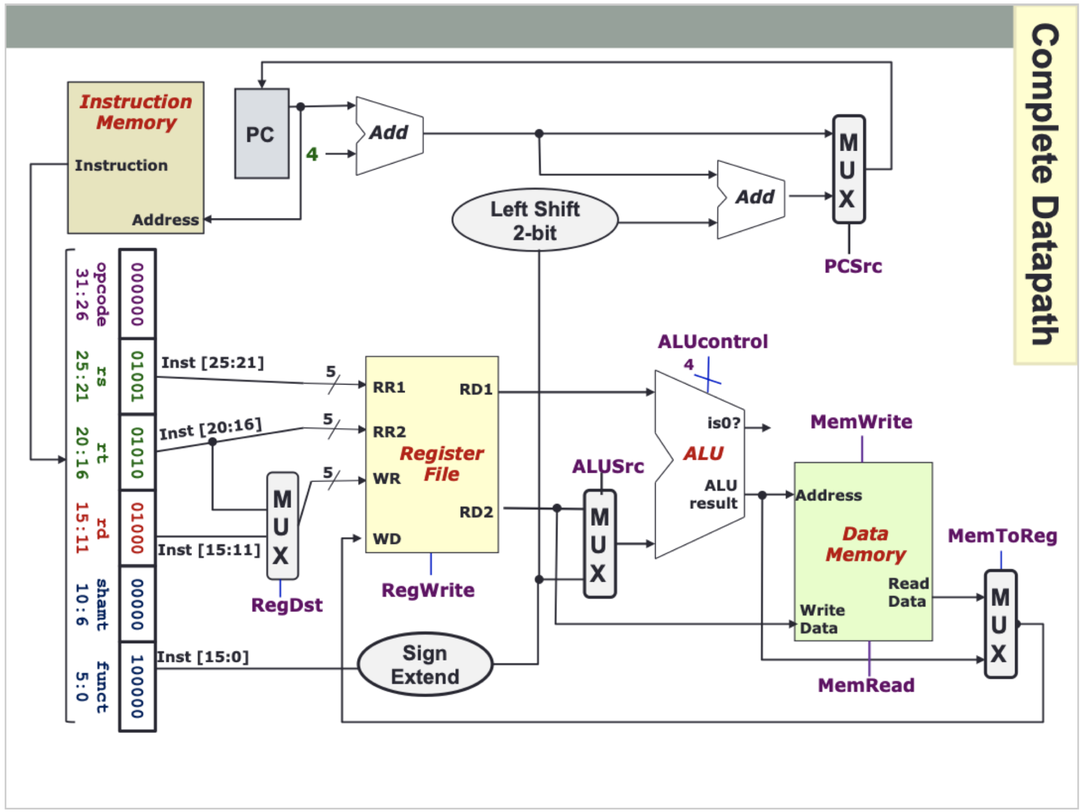
- time taken depends on slowest instruction
- **disadvantage**
  - clock cycle must be long enough to accommodate the slowest instruction ⇒ all instructions will take the same time as the slowest instruction

Multicycle Implementation

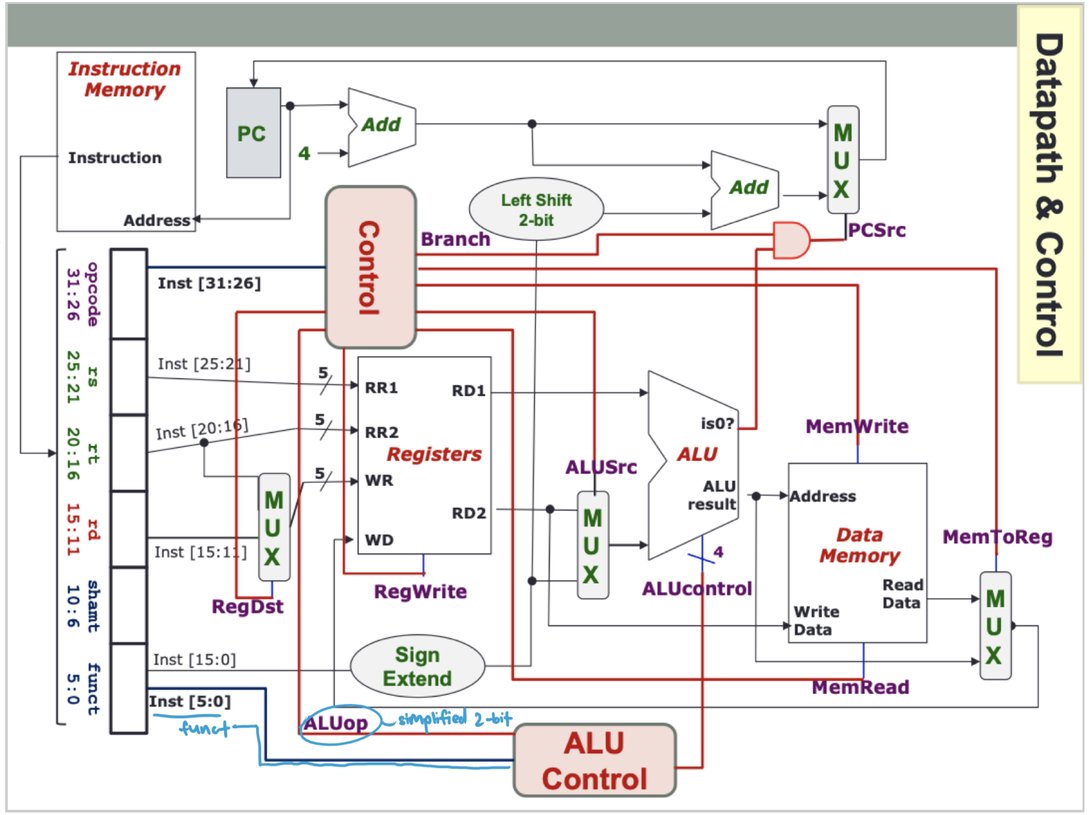
- how it works: break up the instruction into execution steps
  1. instruction fetch
  2. instruction decode and register read
  3. ALU operation
  4. memory read/write
  5. register write
  - each execution step takes one clock cycle
- time taken depends on number of steps
  - cycle time is determined by the slowest step
- **disadvantage**
  - may not necessarily be faster - depends on mix of instructions

Complete Datapath Diagram

▼ datapath



▼ datapath & control



	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 (writing to address (NOT register))	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Control Flow Determination

Control Signals

(can all be generated using opcode directly)

- `RegDst`
  - 0 → write register = `Inst[20:16]`
  - 1 → write register = `Inst[15:11]`
- `RegWrite`
  - 0 → No register write
  - 1 → New value will be written (WD written to WR)
- `ALUSrc` - determines first input of ALU
  - 0 → `Operand2 = Register Read Data 2`
  - 1 → `Operand2 = SignExt(Inst[15:0])` (sign ext immediate)
- `MemRead`
  - 0 → not performing memory read access
  - 1 → read memory using *Address* (returned in *Read Data*)
- `MemWrite`
  - 0 → not performing memory write operation
  - 1 → write Register Read Data 2 into memory[*Address*]
- `MemToReg` - chooses what to be written back into register file
  - 0 → register write data = ALU result
  - 1 → register write data = memory read data
- `PCSrc`
  - 0 → next PC = PC + 4
  - 1 → next PC = SignExt(Inst[15:0]) << 2 + (PC + 4)
  - what it does
    - PCSrc = set to 1 if *Branch* AND *is0* are both 1
      - aka (isBranchInstruction AND branchIsTaken)

	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

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

Generation of 2-bit **ALUop** signal will be discussed later

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