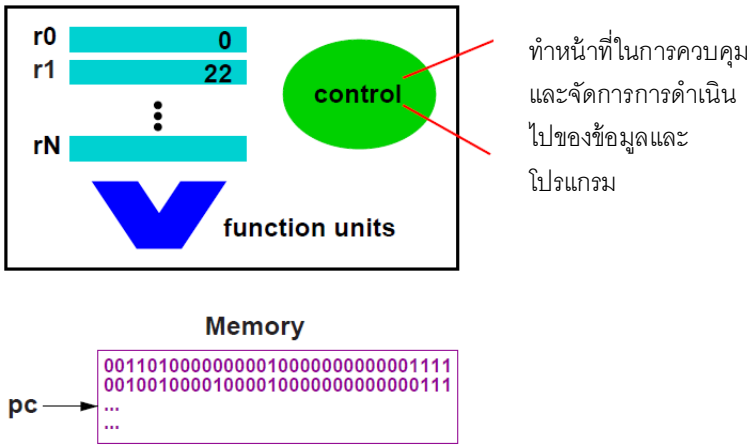
MIPS ISA (MIPS INSTRUCTION SET ARCHITECTURE)

รูปด้านล่างแสดงองค์ประกอบของคอมพิวเตอร์อย่างง่ายที่สุด



MIPS เป็นซีพียูที่ใช้สถาปัตยกรรมแบบ RISC (Reduced Instruction Set Computer)

- ให้มีชุดคำสั่งน้อยๆ เลือกเท่าที่จำเป็น
- ทำให้การสร้าง HW ทำได้ง่ายและเร็ว

MIPS เป็น load/store architecture แบบ 32 bits

- การติดต่อกับ memory ทำผ่านคำสั่ง load/store เท่านั้น

Data type มี 3 แบบ

8 bits (byte) - เทียบเท่า char ในภาษาซี

16 bits (half-word) - เทียบเท่า short ในภาษาซี

32 bits (word) - เทียบเท่า int ในภาษาซี

MIPS memory

ระบบ 32 bits จะอ้าง address memory ได้จาก 0x 0000 0000 ถึง 0x ffff ffff หรือ 2^{32} addresses ในหนึ่งช่อง address จะบรรจุข้อมูลความกว้างขนาด 8 bits หรือ 1 byte ซึ่งระบบหน่วยความจำปัจจุบันจะเป็นลักษณะ byte-oriented memory ทั้งนั้น ดังนั้นระบบ 32 bit address จะอ้างถึงข้อมูลได้ 2^{32} byte หรือ 4 GBytes ($2^2.2^{30}$)

	ส่วนบรรจุข้อมูล (ความกว้าง 8 บิท)
0x ffff ffff	
:	:
:	:
0x 0000 0002	
0x 0000 0001	
0x 0000 0000	

ถ้าขนาดข้อมูลมีขนาดมากกว่า 8 บิท เช่น 32 บิทหรือ 4 bytes ข้อมูลนี้จะคลุมตำแหน่ง address ทั้งหมด 4 ตำแหน่ง เช่น int x = 0x12345678 ถ้า address เริ่มต้นเป็น 0xffff 0000 จะได้ว่า address สุดท้ายเป็น 0xffff 0003 การจัดเก็บข้อมูลที่มากกว่า 8 บิทนี้ สามารถทำได้สองลักษณะ โดยลักษณะแรกคือ Big Endian และอีกลักษณะหนึ่งคือ Little Endian

- Big endian : เอา MSB (Most Significant Byte) ไว้ที่ address ต่ำ
- Little endian : เอา MSB ไว้ที่ address สูง

จากข้อมูล 4 bytes ที่มีค่า 0x12345678 จะได้ว่า MSB คือ 0x12 และ LSB (Least Significant Byte) คือ 0x78 แผนภาพด้านล่างแสดงการเก็บแบบ Little Endian (ซ้าย) และ Big (ขวา) Endian

0x ffff ffff	
:	:
:	:
:	:
0xffff 0003	12
0xffff 0002	34
0xffff 0001	56
0xffff 0000	78
:	:
:	:
:	:
:	:
0x 0000 0002	
0x 0000 0001	
0x 0000 0000	

การเก็บแบบ Little Endian

0x ffff ffff	
:	:
:	:
:	:
0xffff 0003	78
0xffff 0002	56
0xffff 0001	34
0xffff 0000	12
:	:
:	:
:	:
:	:
0x 0000 0002	
0x 0000 0001	
0x 0000 0000	

การเก็บแบบ Big Endian

ซีพียูตระกูล MIPS ใช้ระบบ Big Endian ส่วนซีพียูตระกูล Intel X86 ใช้ระบบ Little Endian

MIPS ISA มี 3 กลุ่มคำสั่ง

- Load/store ติดต่อ/เคลื่อนย้าย ข้อมูลเข้า/ออก memory
- ALU ทำฟังก์ชั้นคณิตศาสตร์และ logic พื้นฐาน

- Control flow การเปลี่ยนคำสั่งในการทำงาน
    - jump หรือ branch

MIPS registers

- 32 ตัว ความกว้าง 32 bits ต่อตัว

  RO, R1, …, R31

  $0, $1, …, $31

- ทุกๆ ตัว เป็น general purpose แต่ยกเว้น R0 ($0) มีค่าเป็นศูนย์เสมอแม้ว่าจะพยายามเขียนค่าอื่นๆเข้าไป!

ชุดคำสั่งกลุ่ม load/store

- Load คือ อ่านค่าจาก memory address หนึ่ง
- Store คือ เขียนค่าไปที่ memory address หนึ่ง
- Load อ่านจาก memory แล้วเอามาเก็บไว้ที่ register
    - Word load, byte load, half-word load
- Store อ่านค่าจาก register ไปเก็บใน memory
    - Word store, half-word store, byte store

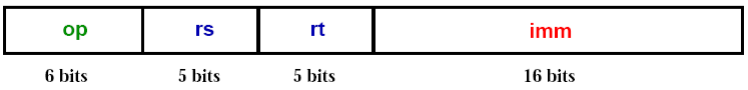
MIPS Load/Store Instructions

```
lb   rt, imm(rs)  # load byte (signed)
lbu  rt, imm(rs)  # load byte (unsigned)

lh   rt, imm(rs)  # load half-word (signed)
lhu  rt, imm(rs)  # load half-word (unsigned)

lw   rt, imm(rs)  # load word

sb   rt, imm(rs)  # store byte
sh   rt, imm(rs)  # store half-word
sw   rt, imm(rs)  # store word
```



หา address ของ memory ที่จะ load/store ยังไง

ตัวอย่างการโหลดข้อมูล word (ขนาด 4 bytes)

lw $7, 4($5)

สมมติว่า $5 = 0xffff0000

Address = 0xffff0000 + 4

0xffff0007	0x78
0xffff0006	0x23
0xffff0005	0x11
0xffff0004	0xaa
0xffff0003	0xbb
0xffff0002	0x76
0xffff0001	0x9e
0xffff0000	0x00

ค่าที่โหลดเข้ามาที่ $7 คือค่า

$ 7 = 0xaa 11 23 78 (MIPS เป็น big endian)

load byte:

lb $7, 4($5)

ได้ $7 = 0xffff ffaa

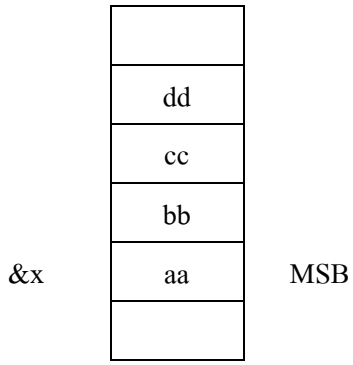
load byte unsigned:

lbu $7, 4($5)

ได้ $7 = 0x000000aa

lb เปรียบเทียบกับ lbu

ให้ตัวแปร x เป็น int ที่อยู่ใน memory และมีค่าเท่ากับ 0xaabbccdd



ถ้าให้ค่า &x เก็บใน $8 ต้องการ load ข้อมูลเข้าที่ $9

lw $9, 0 ($8)

จะได้ว่า $9 = 0xaabbccdd หลังจากการ load word

ถ้าใช้คำสั่ง lb $9, 0 ($8)

$9 = 0xffffffaa (a คือ 1010 บิทซ้ายสุดเป็น 1 ดังนั้นต้องมีการทำ sign extension)

ถ้าใช้คำสั่ง lbu $9, 0 ($8)

$9 = 0x000000aa (แบบ unsigned ไม่สนใจการทำ sign extension)

ถ้า x = 44bbccdd แล้วใช้คำสั่ง lb = $9, 0($8)

$9 = 0x00000044

ถ้าเปลี่ยนใช้คำสั่ง lbu $9, 0 ($8)

$9 = 0x00000044

ให้ x เป็น array of int แปลงส่วนของโค๊ดภาษาซีด้านล่างให้อยู่ในรูป MIPS Assembly

int foo;

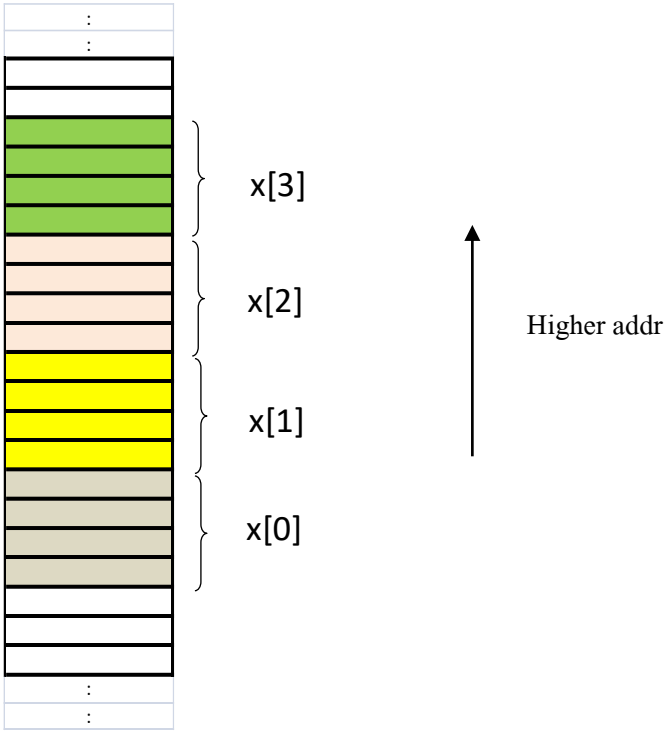
foo = x [2];

x [3] = foo;

ให้ foo เก็บใน $8 และ address เริ่มต้นของ array x เก็บที่ $10

```
lw $8, 8($10)
sw $8, 12($10)
```

แผนภาพการเก็บ array ของ integer (ซึ่งเป็นข้อมูลขนาด 4 bytes) เป็นไปตามด้านล่าง



กลุ่มคำสั่ง ALU (Arithmetic & Logic Instruction)

- ทำ function คณิตศาสตร์และตรรกพื้นฐาน เช่น +, -, and, or
- Operand ของคำสั่ง ALU เป็น register หรือค่าคงที่เท่านั้น
- ค่าของ $0 เป็นศูนย์ตลอด (ไม่ว่าจะพยายามยัดเยียดค่าอื่นให้)

Integer Arithmetic Operations

```
add   rd, rs, rt   # rd = rs + rt
addi  rt, rs, imm  # rt = rs + s_ext(imm)
addiu rt, rs, imm  # rt = rs + s_ext(imm)
addu  rd, rs, rt   # rd = rs + rt
slt   rd, rs, rt   # rd = (rs <s rt)
slti  rt, rs, imm  # rt = (rs <s s_ext(imm))
sltiu rt, rs, imm  # rt = (rs < s_ext(imm))
sltu  rd, rs, rt   # rd = (rs < rt)
sub   rd, rs, rt   # rd = rs - rt
subu  rd, rs, rt   # rd = rs - rt
```


```
int   a, b, c;

a = b - (c+7)

c = (a<0)? 1: 0
```

ให้ $16 เก็บค่าตัวแปร a $17 เก็บค่าตัวแปร b $18 เก็บค่าตัวแปร c

จะแปลงโค้ดด้านบนเป็น MIPS assembly ได้ดังนี้

```
addiu   $8, $18, 7     // temp = c+7
subu    $16, $17, $8   // a = b-temp
slti    $18, $16, 0    // (a<0)? 1:0
```

addiu กับ addi

- สังเกตว่าตัวแปรเป็น int แต่ทำไมชุดคำสั่งที่ออกมาจึงเลือกใช้แบบ unsigned แทนที่จะเป็น signed
- addi มีการ raise exception เวลา overflow addiu เวลาเกิด overflow แล้วปล่อยผ่าน

ภาษา C เลือกใช้แบบ unsigned เพราะ C ไม่คำนึงถึง overflow

Logical Operations

```
and   rd, rs, rt     # rd = rs & rt
andi  rt, rs, imm    # rt = rs & imm
nor   rd, rs, rt     # rd = ~(rs | rt)
lui   rt, imm        # rt = imm << 16
or    rd, rs, rt     # rd = rs | rt
ori   rt, rs, imm    # rt = rs | imm
sll   rd, rt, shamt  # rd = rt << shamt
sllv  rd, rt, rs     # rd = rt << (rs&0x1f)
sra   rd, rt, shamt  # rd = rt >>s shamt
srav  rd, rt, rs     # rd = rt >>s (rs&0x1f)
srl   rd, rt, shamt  # rd = rt >> shamt
srlv  rd, rt, rs     # rd = rt >> (rs&0x1f)
xor   rd, rs, rt     # rd = rs ^ rt
xori  rt, rs, imm    # rt = rs ^ imm
```

ถ้า rt = 0xaabbccdd หลังจากคำสั่ง sra rd, rt, 8

จะได้ rd = 0xffaabbcc

แต่ถ้าใช้คำสั่ง srl rd, rt, 8

จะได้ rd = 0x00aabbcc