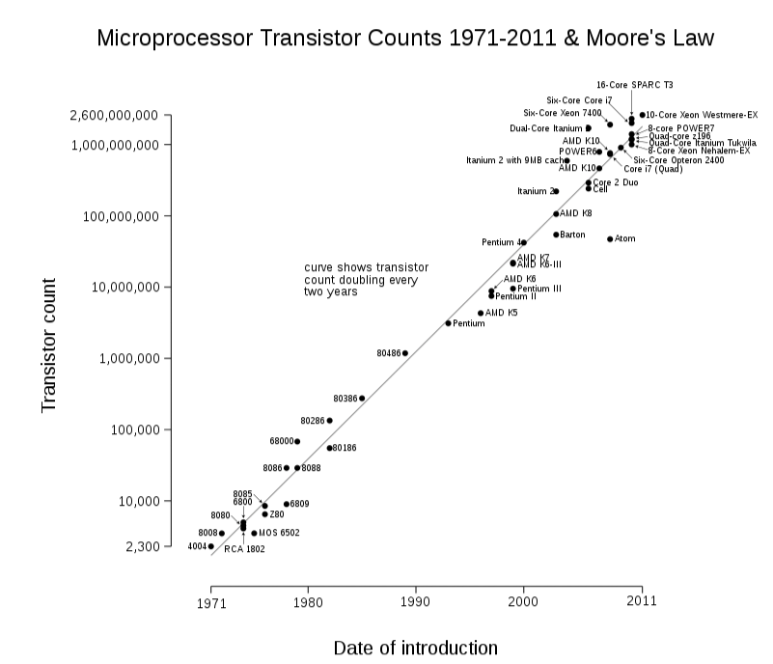
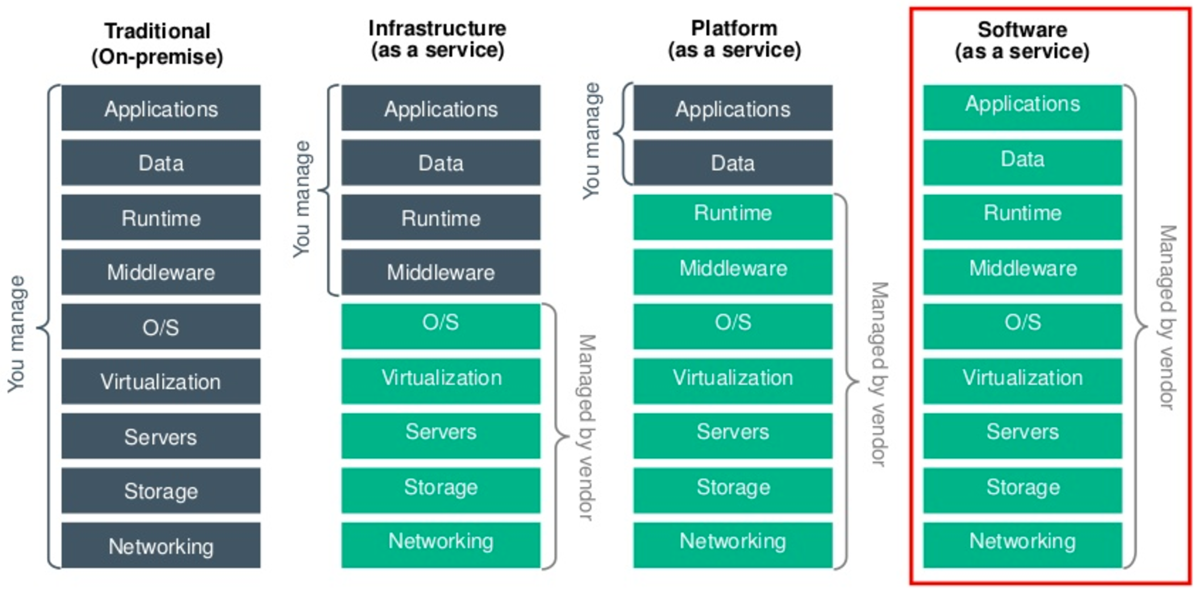
**CH1. Abstraction in Computer**

Moore’s Law: 반도체 집적회로의 성능이 2년마다 2배로 증가한다는 법칙 (empirical)

**Computer**

* Personal: general purpose
* Server: network based, high capacity, reliability, performance
* Super-Computer
* Embedded Computer

**Post-PC**

* Personal Mobile Device
* Cloud Computing
  + WSC: Warehouse Scale Computer, gigantic data center
  + SaaS: Software as a Service

**Factors in performance**

* Algorithm: determines the number of operation executed
* Programming language, compiler, architecture:
* Processor and memory system:
* I/O system (OS)

**8 Great Ideas**

* Abstraction to simplify design
* Make the common case fast
* Parallelism
* Pipelining
* Prediction
* Hierarchy of memories
* Dependability via redundancy

**Below your program**

1. Application software: written in HLL
2. System software

* Compiler: HLL code to machine code
* OS: manage memory and storage

1. Hardware: Processor (CPU), memory, I/O controllers

**Program**: a set of instruction for computer to execute

* Computer is finite. So, computer can execute a limited set of primitive instructions
* By combining primitive instructions, complex and huge programs is created.

**Program code**

1. High-level language: Human-readable, productivity, portability
2. Assembly language: Textual representation of instructions
3. Machine language: Binary digits, hardware representation

**Components of a computer**

1. Input
2. Output: display (LCD monitor, touchscreen)
3. memory,
4. data-path, control: In Processor (CPU)
   1. Datapath: performs arithmetic operation
   2. Control: controls datapath, memory, I/O

\* cache: small fast SRAM memory for immediate access to data

**Abstractions**

1. Storage
   1. Volatile: Main memory
   2. Non-Volatile: Magnetic disk, Flash memory, Optical disk
2. Network
   1. LAN: Ethernet
   2. WAN: Internet
   3. Wireless network: Wi-Fi, Bluetooth

=> advantage: Communication, resource sharing, nonlocal access

**Technology**

Processor (CPU)

* 1. Ingredient: Silicon (Semi-conductor)
  2. Making process

스크린샷이(가) 표시된 사진

자동 생성된 설명

* + 1. Silicon ingot: diameter: 8~12 inches, length: 12~24 inches
    2. convert to wafer by slicing (under 0.1 inches width)
    3. Since It is impossible to make innocent wafer, make into the pieces (dies or chips)
       1. Yield(수율): proportion of working dies per wafer

=> Cost is nonlinear to area and defect rate

To reduce the cost

1. Decrease die area
2. Manufacturing process should be improved

**Performance**

1. Response time: How long it takes to do a task
2. Throughput: Total work done per unit time

Ex) A: 10s, B: 15s then

Measuring execution time

- Elapsed time: Total response time: Processing + I/O + O/S overhead + ….

- CPU time: Only processing time (= CPU performance)

o CPU time: Time to execute program

o System CPU time: CPU time + time to ready to execute program

x Clock: determinant that hardware event occurs.

\* clock cycle (or tick): time interval of clock

\* clock period: duration of clock cycle

\* clock frequency (=clock speed): cycles per second (inverse of clock period)

스크린샷이(가) 표시된 사진

자동 생성된 설명

x CPU Times

x Instruction Count

Cf> Extension (when instruction counts are different)

Weighted Average CPI

Ex) If there are 5types of instruction:

Load (5cycles), Store (4cycles), R-type (4cycles), Branch (3cycles), Jump (3cycles)

And if a program has:

50% load, 15% store, 25% R-type, 8% branch, 2% jump

**Power**

=> For this reason, Multiprocessor appear

Multiprocessor needs:

1. Reduce overhead of communication & pipelining

2. load balancing

**Pitfall**

1. Amdahl’s law:

Ex) Suppose a program needs to execute 100s using 80s at multiply. If we want to execute 5 times faster, How much improved multiply time?

20s = 80s/n + (100-80) ⬄ 0 = 80/n

It is impossible to be 5 times faster when 80% of time used in multiply.

2. Low power at idle

3. architecture between performance and energy efficiency is indifferent

4. Using parts of performance as a measure of total performance

There is a case that nevertheless MIPS value is greater than others, computer speed is actually slower than others

* Only Execution time is measure

**CH2. Instruction: Language of the computer**

Instruction: the words of computer’s language

Instruction set: the vocabulary of commands understood by a given architecture

**Hardware Operation**

규칙 1. 간단하게 하기 위해선 규칙적인 것이 좋다.

o Arithmetic operands은 항상 3개이다.

o 한 줄에 하나의 명령어만 사용할 수 있다.

Ex) Add: **add a, b, c** => a = b + c

C code: f = (g+h) – (i+j)

MIPS code:

add t, g, h #t = g+h

add u, i, j #u = i+j

sub v, t, u #v = t-u

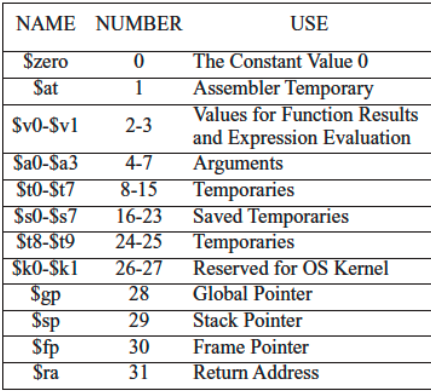
EX2) C code: a = b+c+d

MIPS code:

add t, b, c #t = b+c

add a, t, d #a = t(b+c) + d

**Register Operation** \* register: faster and smaller than main memory



규칙 2. 작은 것이 더 빠르다.

Ex) f = (g + h) – (i + j) 에서 컴파일러가 변수 f, g, h, ,i, j를 레지스터 $s0, $s1, $s2, $s3, $s4에 각각 할당했다고 하자. 컴파일 된 MIPS 코드를 보여라.

**Memory operation**

배열, 구조체 같은 복잡하고 크기가 큰 자료 구조 => 메모리에 저장

- **lw**: load word, 메모리에 저장된 값을 레지스터로 복사하는 명령

Cf> Continuous address

Ex) C code: g = h + A[10] // g, h, A is $s1, $s2, $s3

MIPS code: lw $t0, 10($s3) // 10은 배열의 변위(Offset), ($s3)은 배열의 base register

스크린샷이(가) 표시된 사진

자동 생성된 설명 add $s1, $s2, $t0 // g = h + A[10]

프로그램에서 8비트로 구성된 1byte를 사용하므로 대부분 컴퓨터는 바이트 단위로 주소를 지정한다. Word 주소는 Word를 구성하는 4byte 주소 중 하나를 사용한다. 따라서 연속된 Word (such as a[0], a[1])는 주소가 4byte씩 차이가 난다.

위의 예제는 lw $t0, 10($s3) 에서 lw $t0, 4\*10($s3)로 수정되어야 한다.

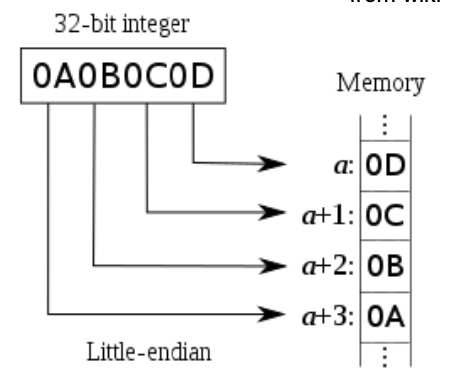
Cf> Big endian, Little endian

컴퓨터가 어떤 바이트 주소를 워드 주소로 사용하는지에 따라 나뉜다.

\* Big endian: 제일 왼쪽 최상위 byte주소를 word주소로 쓰는 경우

\* Little endian: 제일 오른쪽 최하위 byte 주소를 word 주소로 쓰는 경우

우리가 공부하는 MIPS의 경우에는 Big endian 방식을 채택한다.

스크린샷이(가) 표시된 사진

자동 생성된 설명

**- sw**: store word, 레지스터에 저장된 값을 메모리로 복사하는 명령

Ex) C code: A[12] = h + A[10] // g, A is $s1, $s2

MIPS code: lw $t0, 40($s2) // $t0 = A[10]

add $t0, $s1, $t0 // $t1 = h + A[10]

sw $t0, 48($s2) // A[12] = $t1

cf> Register spilling : 자주 사용하지 않는 변수를 메모리에 넣는 일

**Immediate operation**

-> 빠른 연산과 적은 에너지

규칙 3. Make the common case fast

- **addi**: add immediate, 레지스터에 상수를 더한다.

Ex) addi $s1, $s1, 4 // $s1 += 4

=> 뺄셈의 경우에는 음수를 더한다. (subi는 없음)

**Cf> Signed & Unsigned number**

MSB, LSB

\* MSB: most significant bit (가장 왼쪽 비트)

 \* LSB: least significant bit (가장 오른쪽 비트)

**-** UnsignedN-bit number =

o range: 0 to (in 32bits, 0 to )

ex) 0000 0000 0000 0000 0000 0000 0000

= 0 + 0 + 0 + … + =

- Signed N-bit number =  **(2’s completement)**

o range (in 32bits, )

ex) 1111 1111 1111 1111 1111 1111 1111

= =

=> MSB가 0이면 양수, 1이면 음수이다.

- Signed negation

Ex) +2 -> -2

0000 0000 … reverse => 1111 1111 …

And +1 => 1111 1111 …

**Sign extension**

더 큰 바이트로 확장하는 것, 현재 word의 msb의 값으로 새로운 비트를 모두 채운다.

Ex) +2: 0000 0010 => 0000 0000 0000 0010

-2: 1111 1110 => 1111 1111 1111 1110

- lb: load byte, 바이트를 부호 있는 수로 간주하고 남은 24비트를 부호확장해서 채운다.

- lbu: load byte unsigned, 바이트를 부호 없는 수로 간주하고 채운다.

**Representing Instruction**

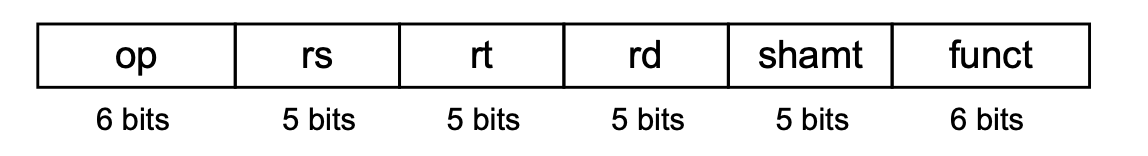
- Machine code

o Instructions are encoded in binary

o 32-bit word

field

**R-format Instructions**



\* op: operation code

\* rs: first source register number

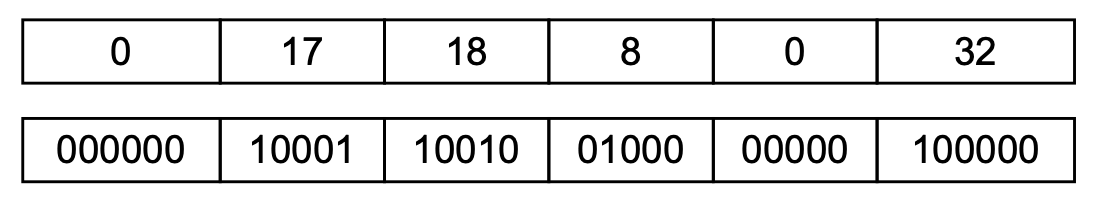
\* rt: second source register number

\* rd: destination source register number, 여기에 연산결과가 기억된다.

\* shamt: shift amount (for shift instruction)

\* funct: function code (extends opcode)

Ex) add $t0, $s1, $s2



위에 Register operation 참고

00000100011001001000000000100000 =

Cf> Hexadecimal

쇼지, 벽이(가) 표시된 사진

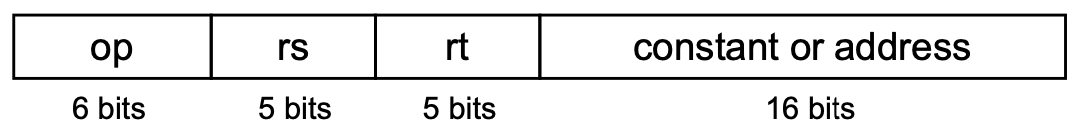
자동 생성된 설명

Ex) e c a 8 6 4 2 0

1110 1100 1010 1000 0110 0100 0010 0000

I-format Instruction

원칙4. 좋은 설계에는 적당한 절충이 필요하다.



If op is sw(35) or lw(43), then rs is base address and 16bits will be address.

하늘, 실외이(가) 표시된 사진

자동 생성된 설명

Ex) lw $t0, 32($s3), sw$t0, 48($s3)

스크린샷이(가) 표시된 사진

자동 생성된 설명

Ex2) A[300]= h + A[300]을 MIPS code로 작성하고 machine code로 변환하여라

**Logical operation**

shift

- **sll**: shift left logical (=multiply 2), shift left and fill with 0 bits

Ex) sll $t2, $s0, 4 // $t2 = $s0 << 4bit

스크린샷이(가) 표시된 사진

자동 생성된 설명

if $s0 = 0000 0000 … 0000 = 9, then $t2 = 0000 0000 … 1001 = 144

- **srl**: shift right logical

Ex) srl $t2, $s0, 4 // $t2 = $s0 >> 4bits

스크린샷이(가) 표시된 사진

자동 생성된 설명

2

if $s0 = 0000 0000 … 1001 = 144, then $t2 = 0000 0000 … 0000 = 9

And / Or

- **and**: select some bits, clear others to 0

일부 패턴을 감추는 역할을 하기 때문에 MASK라고도 부른다.

Ex) and $t0, $t1, $t2

스크린샷이(가) 표시된 사진

자동 생성된 설명

- **or**: set some bits to 1, leave others unchanged

Ex) $t0, $t1, $t2

스크린샷이(가) 표시된 사진

자동 생성된 설명

- **not**

Not을 구현하기 위해서는 3개의 피연산자 형식을 유지하기 위해 NOR으로 구현되었다.

A NOR b == NOT (a or b)

A NOR b == NOT (a or 0) = NOT a

Ex) nor $t0, $t1, $zero

스크린샷이(가) 표시된 사진

자동 생성된 설명

**Conditional branch**

- **beq**: branch if equal

Ex) beq rs, rt, L1 // if (rs == rt) then branch to instruction label L1

- **bne**: branch if not equal

Ex) bne rs, rt, L1 // if (rs != rt) then branch to instruction label L1

- **slt**: set less than cf> **sltu**

Ex) slt $t0, $s0, $s1 // if ($s0 < $s1) then $t0 = 1. Otherwise 0

- **slti**: set less than immediate cf> **sltui**

Ex) slt $t0, $s0, a // if ($s0 < a) then $t0 = 1. Otherwise 0

- **j**: jump

Ex) j L1 // unconditional jump to instruction label L1

**If implement**

Ex) C code: if (i==j) f = g+h;

else f = g-h;

i, j, f, g, h is $s0, $s1, $s2, $s3, $s4

MIPS:

bne $s0, $s1, ELSE

add $s2, $s3, $s4

j EXIT

ELSE: sub $s2, $s3, $s4

EXIT:

**Loop implement**

Ex) C code: while (save[i] == k) i += 1;

i = $s3, k = $s5, base address of *save* is $s6

MIPS:

LOOP:

sll $t1, $s3, 2 // i \* 4

add $t1, $t1, $s6 // base address + i \*4

lw $t0, 0($t1)

bne $t0, $s5, EXIT

addi $s3, $s3, 1

j LOOP

Ex2) C code: for (int i=0; i<k; i++) save[i] = i+2;

i = $s3, k = $s5, base address of *save* is $s6

MIPS:

add $s3, $s3, 0

LOOP:

slt $t0, $s3, $s5

bne $t0, 1, EXIT

sll $t1, $s3, 2

add $t1, $s3, $s6

lw $t2, 0($t1)

addi $t2, $s3, 2

sw $t2, 0($t1)

addi $s3, $s3, 1

j LOOP

EXIT:

x < y (max index of array such ) -> 0

means

sltu $t0, $s1, $t2

beq $t0, $zero, IndexOutOfBounds

**Procedure calling**

Step

1. 매개 변수들을 레지스터에 넣는다.

2. 프로시저로 제어권을 넘긴다.

3. 프로시저를 수행하기 위한 저장공간을 할당받는다.

4. 프로시저의 연산을 수행한다.

5. 결과 값을 호출자를 위한 레지스터에 넣는다.

6. 호출을 한 시점(Instruction)으로 돌아간다.

$a0 ~ $a3: 매개변수를 넘기기 위한 레지스터 (4~7번)

$v0 ~ $v1: 반환값을 반환하기 위한 레지스터 (2~3번)

$t0 ~ $t9: 임시 레지스터, callee의해 덮어 쓰여질 수 있다.

$s0 ~ $s7: 비휘발성 레지스터, callee에 의해만 저장되고 복원되어야 한다.

$gp: global pointer for static data (28번)

$sp: stack pointer (29번)

$fp: frame pointer (30번)

$ra: return address, 프로시저가 어디로 돌아갈지 기억하고 있는 레지스터 (31번)

- **jal**: jump and link, 지정된 주소로 점프하면서 동시에 다음 명령어의 주소를 $ra를 저장

Ex) jal ProcedureLabel

- **jr**: jump register, 레지스터에 저장된 주소로 무조건 점프한다.

Cf> Program Counter (PC): 현재 수행 중인 명령어의 주소를 기억하는 레지스터

jal X 명령을 이용해서 procedure X(callee)로 점프하고 $ra 레지스터의 PC+4 값을 저장한다.( jal = $ra <- PC + 4) callee는 계산을 끝낸 후 결과를 $v0 or $v1에 넣고 jr $ra명령을 실행하여 복귀한다.

Ex) C code: int leaf\_example(int g, h, i, j) {

int f;

f = (g+h) – (i+j);

return f;

} // g, h, i, j: $a0~$a3, f is $s0, result is $v0

MIPS code:

LEAF: addi $sp, $sp, -4 // int f, 4바이트 공간을 할당해준다.

sw $s0, 0($sp) // f에 공간 할당 후, 쓰레기값 넣기

add $t0, $a0, $a1

add $t1, $a2, $a3

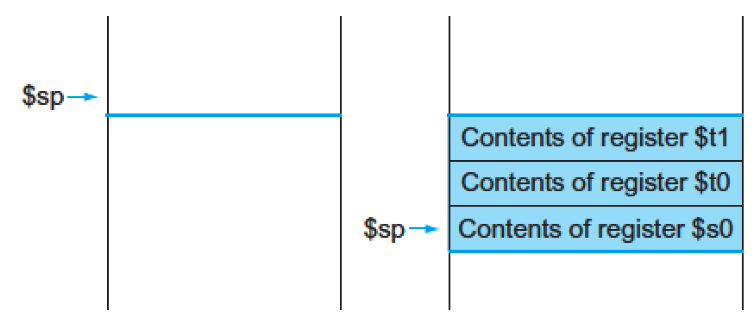
sub $s0, $t0, $t1

add $v0, $s0, 0

lw $s0, 0($sp) // $s0 값을 원래 초기값(쓰레기 값)

addi $sp, $sp, 4 // 다시 변수 f 공간 반납

jr $ra

cf> memory allocation with stack pointer

**Recursive procedure**

값이 보존되어야 할 모든 레지스터를 스택에 넣는다. 복귀한 후에 메모리에서 값을 꺼내 레지스터를 원상 복구하고 이에 맞춰 스택포인터를 다시 조정한다.

o Caller pushes

1. $a0 ~ $a3: 매개변수를 저장하는 레지스터

2. $t0 ~ $t9: call 이후에 필요하다

o Callee pushes

1. $ra: return address

2. $s0 ~ $s7: callee가 사용하는 모든 레지스터

Ex) C code:

int fact (int n) {

if (n<1) return 1;

else return n\*fact(n-1); // Argument in n $a0, result in $v0

MIPS code:

FACT:

addi $sp, $sp, -8 // n, return address 공간 할당

sw $ra, 4($sp) // 이번 프로시저의 return address 저장

sw $a0, 0($sp) // 이번 프로시저의 n값 저장

slti $t0, $a0, 1 // n < 1

beq $t0, $zero, L1 // if (n > 1)

addi $v0, $zero, 1 // else n = 1

addi $sp, 8 // pop 2 items from stack

jr $ra // jal로 돌아감, 정확히는 jal다음 (PC+4)

L1:

addi $a0, -1 // n--

jal FACT // FACT로 점프 및 PC+4를 $ra에 저장

lw $a0, 0($sp) // 저장된 n 값 지우면서 $a0에 저장

lw $ra, 4($sp) // 저장된 return address $ra에 저장

addi $sp, $sp, 8 // 다 지우면서 올라가기

mul $v0, $a0, $v0 // n\*fact(n-1)

jr $ra // jal FACT 의 다음 instruction으로 이동

스크린샷이(가) 표시된 사진

자동 생성된 설명

텍스트, 지도이(가) 표시된 사진

자동 생성된 설명

**global pointer**

모든 프로시저 외부에서 선언된 변수는 정적 변수로 간주되고 나머지는 자동 변수로 간주된다.

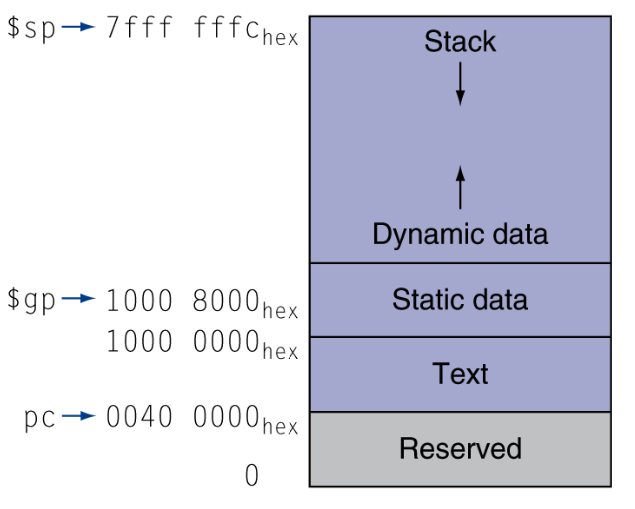
**Frame pointer**

프로시저 프레임의 첫 번째 워드를 가리키는 포인터, 스택 포인터의 값이 프로시저 내에서 바뀔 수도 있으므로 메모리 내 지역 변수에 대한 변위는 기준이 달라질 수 있다. 그 때 사용하는 것이 frame pointer 이다.

스크린샷이(가) 표시된 사진

자동 생성된 설명

**Memory Layout**

o Text: program code (machine code)

o Static data: global variable

o Dynamic data: heap

ex) malloc in C, new in java

o Stack: automatic variable

Cf> more than 4 arguments

MIPS: 나머지 인수를 프레임 포인터 바로 위 스택에 넣는다.

**Byte / Halfword operation**

- **lb, lh**: load byte, load halfword

Ex) lb rt, offset(rs) // rt로 32비트 부호확장을 수행한다.

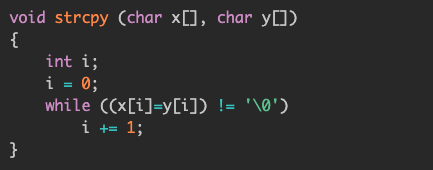
- **lbu, lhu**: load byte unsigned, load halfword unsigned

Ex) lbu rt, offset(rs) // rt로 0으로 채우는 32비트 확장을 수행 (unsinged)

- **sb, sh**: store byte, store halfword

Ex) sb rt, offset // 가장 오른쪽에 있는 바이트 혹은 halfword를 offset(rs)에 저장한다.

Cf> string copy



MIPS code:

32-bit constants

- lui: load upper immediate

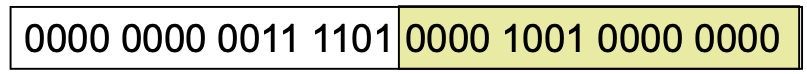
Ex) lui $s0, 61 // 상위 4바이트를 61로 채운다.

클립아트이(가) 표시된 사진

자동 생성된 설명

- ori: or immediate

Ex) ori $s0, $s0, 2304 // 하위 4바이트에 원래 숫자 + 2304를 더한다.



**Branch addressing**

**- PC-relative addressing (beq, bne)**

스크린샷이(가) 표시된 사진

자동 생성된 설명

모든 주소가 16비트 필드에 들어가기엔 너무 작아서 어떤 레지스터를 지정해 그 값을 분기 주소와 더하도록 하는 것이다.

* 대부분의 분기의 목적지는 분기 명령 근처에 위치한다.

PC = 레지스터 + 분기주소

o Target address: PC + offset x 4: PC를 기준으로 범위 안에서 분기할 수 있다.

주의!

목적 주소를 알아내려 할때 PC는 이미 4만큼 증가했다.

Target address: PC + 4 + offset x 4

**Jump addressing**

**- (Pseudo) direct jump addressing** (instruction j or jal)

스크린샷이(가) 표시된 사진

자동 생성된 설명

J-format

주소를 저장할 때, 오른쪽으로 두 번 Shift한 값을 저장하기 때문에 28비트의 주소를 기억한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

위와 같이 상위 4바이트 내에서 이동 가능하다.

**Register addressing**: R-format instruction,

레지스터 안의 32비트를 이용해 주소 전체를 이동할 수 있다.



**Base addressing: R-format instruction**

스크린샷이(가) 표시된 사진

자동 생성된 설명

**Immediate addressing**



Cf> Branching far away

만약, 분기의 목적지가 16비트 Offset으로 표현하기에 너무 멀리 있다면, 코드가 재작성된다.

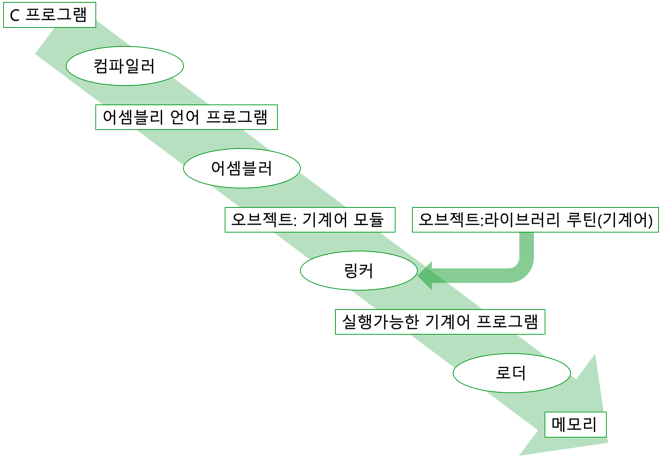
Ex) beq $s0, $s1, L1 에서 L1이 너무 멀리 있다면



처럼 재작성한다.

**Translation and Startup**

텍스트이(가) 표시된 사진

자동 생성된 설명

- 컴파일러

사람이 짠 high-level language를 assembly language로 바꾼다.

- 어셈블러

대부분의 어셈블러 명령어는 기계 명령어와 일대일 대응이 되는데, 어셈블러는 그것을 좀 더 다듬는다. 예를 들어 move $t0, $t1 → add $t0, $zero, $t1. 간단한 변경 뿐 아니라 $at 레지스터(assembler temporary)를 이용하여 코드 자체를 바꾸기도 한다.

또한 어셈블러는 어셈블리 프로그램을 기계어로 바꾼다.

o object file header: 목적 파일을 구성하는 각 부분의 크기와 위치를 서술

o text segment: 기계어 코드

o static data segment: 프로그램 수명 동안 할당되는 데이터가 들어 있다.

Ex) global variables, string, constant

o relocation info: 프로그램이 메모리에 적재될 때 절대 주소의 의존하는 명령어와 워드를 표시

o symbol table: 아직 정의되지 않고 남아 있는 레이블을 저장

o debugging info: 각 모듈이 어떻게 번역되었는가에 대한 간단한 설명

- 링커

따로 어셈블된 기계어 프로그램을 하나로 연결해주는 역할

1. 코드와 데이터 모듈을 메모리에 심벌 형태로 올려놓는다.

2. 데이터와 명령어 레이블의 주소를 결정

3. 외부 및 내부 참조 해결

=> 링커가 모듈을 메모리에 적재할 때 모두 절대 실제 위치에 해당하는 값으로 재설정 해야함

**Ch3 Arithmetic for computers**

- **Addition, Subtraction**

Subtraction is implemented by adding 2’s complement.

Ex) 0001 – 1100 = 0001 + 0100 = 0101

o overflow

|  |  |  |  |
| --- | --- | --- | --- |
|  | addition | subtraction | example |
| (+), (+) | O, (when msb = 1) | X | 011 + 001 = 100 |
| (+), (-) | X | O, (when msb = 1) |  |
| (-), (-) | O, (when msb = 0) | X |  |
| (-), (+) | X | O, (when msb = 0) |  |

Overflow exception

\* **add, addi, sub**: occur exception Ex) ada, portran

Overflow가 발생하면, 예외 처리기를 작동시킨다.,

PC를 EPC 레지스터에 저장하고 미리 정의된 예외 처리기 주소로 점프한다.

처리를 마친 후, 연산을 계속하기 위해 mfc0 명령을 이용해 EPC 값을 되돌린다.

Ex) 부호 있는 덧셈

addu $t0, $t1, $t2 // $t0 = sum, but don’t trap

xor $t3, $t1, $t2 //

slt $t3, $t3, $zero // check if signs differ

bne $t3, $zero, NO\_OVERFLOW // if sign is differ, then NO\_overflow

xor $t3, $t0, $t1 // check sum’s and $t1’s sign

slt $t3, $t3, $zero // if they differ

bne $t3, $zero, Overflow // then Overflow

Ex) 부호 없는 덧셈

addu $t0, $t1, $t2 // $t0 = sum

nor $t3, $t1, $zero // $t3 = !$t1 (

sltu $t3, $t3, $t2 // ( < $t2

// ⬄

bne $t3, $zero, OVERFLOW // if $t3 == 1, overflow

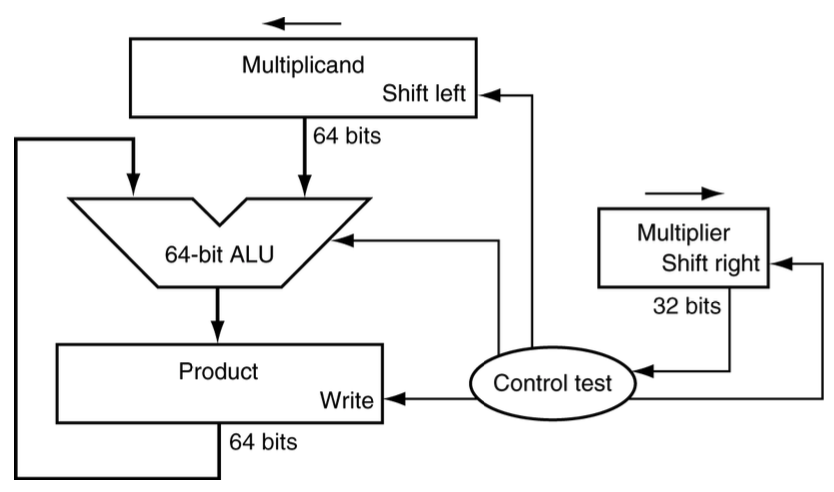
\* **addu, addiu, subu**: does not occur exception ex) C, java

**- multiplication**

Ex)

스크린샷이(가) 표시된 사진

자동 생성된 설명



* Control test: 0인지 1인지 구분하여 ALU에 값을 넣어준다.
  + 0: multiplicand 연산을 생략하고 product에 값을 넣어준다.
  + 1: Multiplicand 연산 후 ALU에서 기존 product 값과 더해서 Product에 넣어준다.
* 64-bit ALU: 덧셈을 수행하는 부분

Algorithm procedure

1. Multiplier 에서 lsb를 control test 에 넣고 0, 1 판별 후 shift right 연산 수행

2. multiplicand에 shift left 연산을 수행한다.

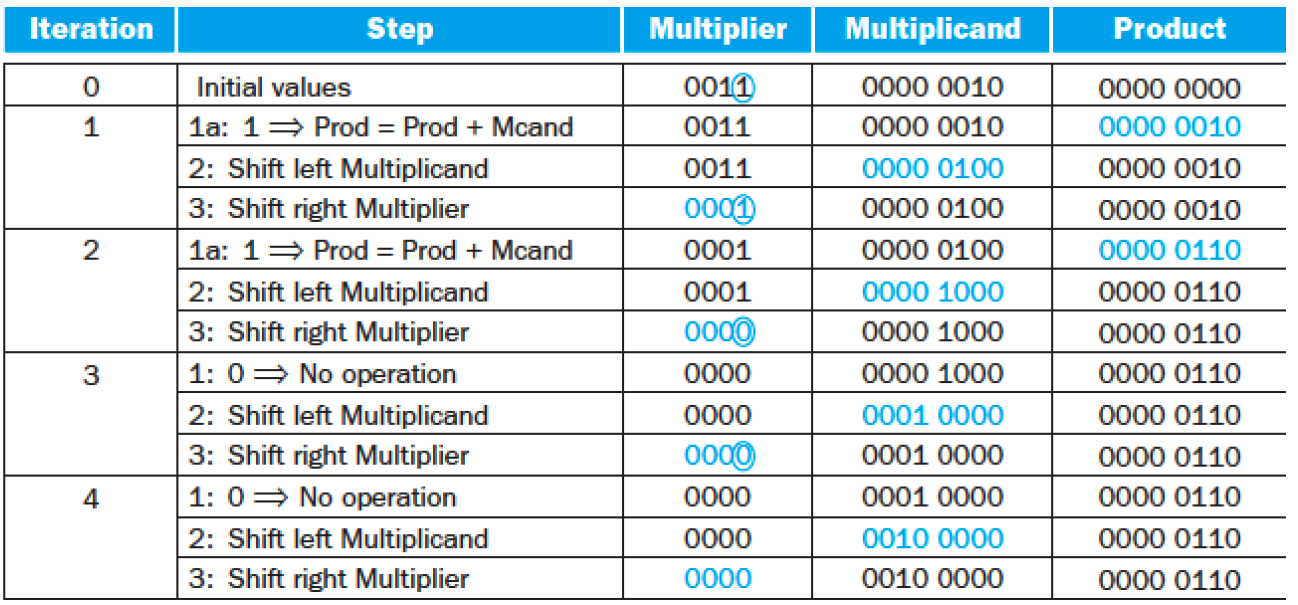
3. 1이면 multiplicand와 연산을 수행해서 ALU에 진입하고 0이면 그냥 ALU로 들어간다.

4. 이전 Product와 새로 들어온 값을 ALU를 통해 더한다.

스크린샷이(가) 표시된 사진

자동 생성된 설명

Example)



**Optimize Multiplier**

스크린샷이(가) 표시된 사진

자동 생성된 설명

* Product의 하위 32비트에 Multiplier를 먼저 저장해둔다.
* Control test, ALU에서의 연산을 상위 32비트에 저장한다.

**Faster Multiplier**

1. adder를 많이 쓴다 -> 비싸다.

2. pipelining

Product can value at 64bits, but MIPS register is typically 32bits

=> HI, LO register

o HI: upper 32bits

o LO: lower 32bits

**multiplier instrcution**

**mult rs, rt / multu rs, rt**

: rs와 rt의 값을 곱해서 64비트 결과값을 HI와 LO를 사용하여 저장한다.

**mfhi rd / mflo rd**

HI or LO의 값을 rd로 옮긴다.

**mul rd, rs, rt**

결과값의 하위 32bits만 rd에 저장한다.

- Division

스크린샷이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명스크린샷이(가) 표시된 사진

자동 생성된 설명

1. divisor(1000)를 remainder(1001010)에서 뺀다.

2. 만약 결과가 0보다 크거나 같으면 결과 값은 다시 remainder에 저장하고 몫 레지스터의 맨 오른쪽 비트에 1을 저장한다.

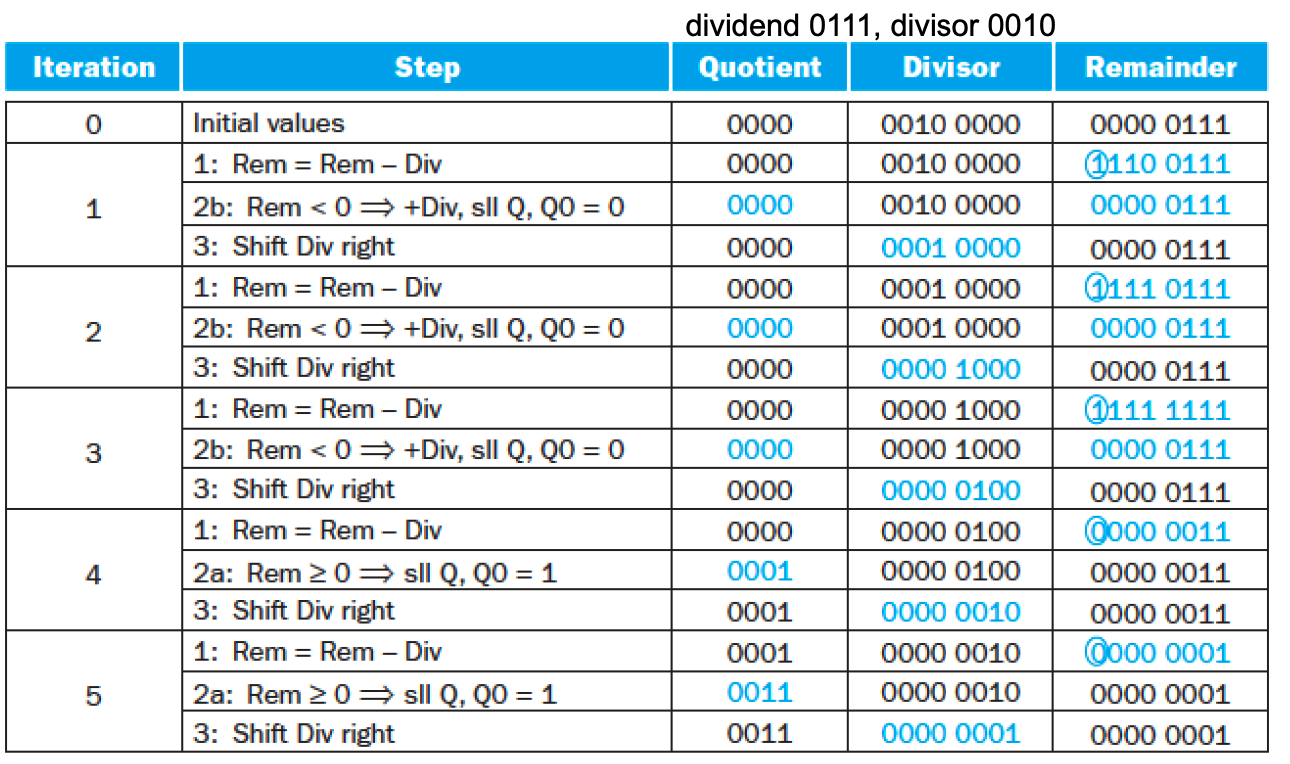
Ex) 1001 – 1000 > 0, remainder = 1, quotient = 1,

3. 만약 결과가 0보다 작으면 빼기 전 상태로 remainder를 돌려놓고, 몫 레지스터의 맨 오른쪽 비트에 0을 저장한다.

Ex) 0 – 1000 < 0, remainder = 1, quotient = 10 => 100 => 1001, remainder: 10

4. Divisor 레지스터에 right shift 연산을 수행한다.

Example



Optimized Divider

텍스트이(가) 표시된 사진

자동 생성된 설명

곱셈 연산기와 똑같다.

Divide operation

- div rs, rt / divu rs, ru

:

- Floating point

해야 되나 싶으면 하고 해도 되나 싶으면 하지마라