Chapter2. Instruction:Language of the Computer

wnwlcks123 · 방금 전

CS

CS

목록 보기 4/4

== 3주차 5,6 강의 ==

Instruction Set

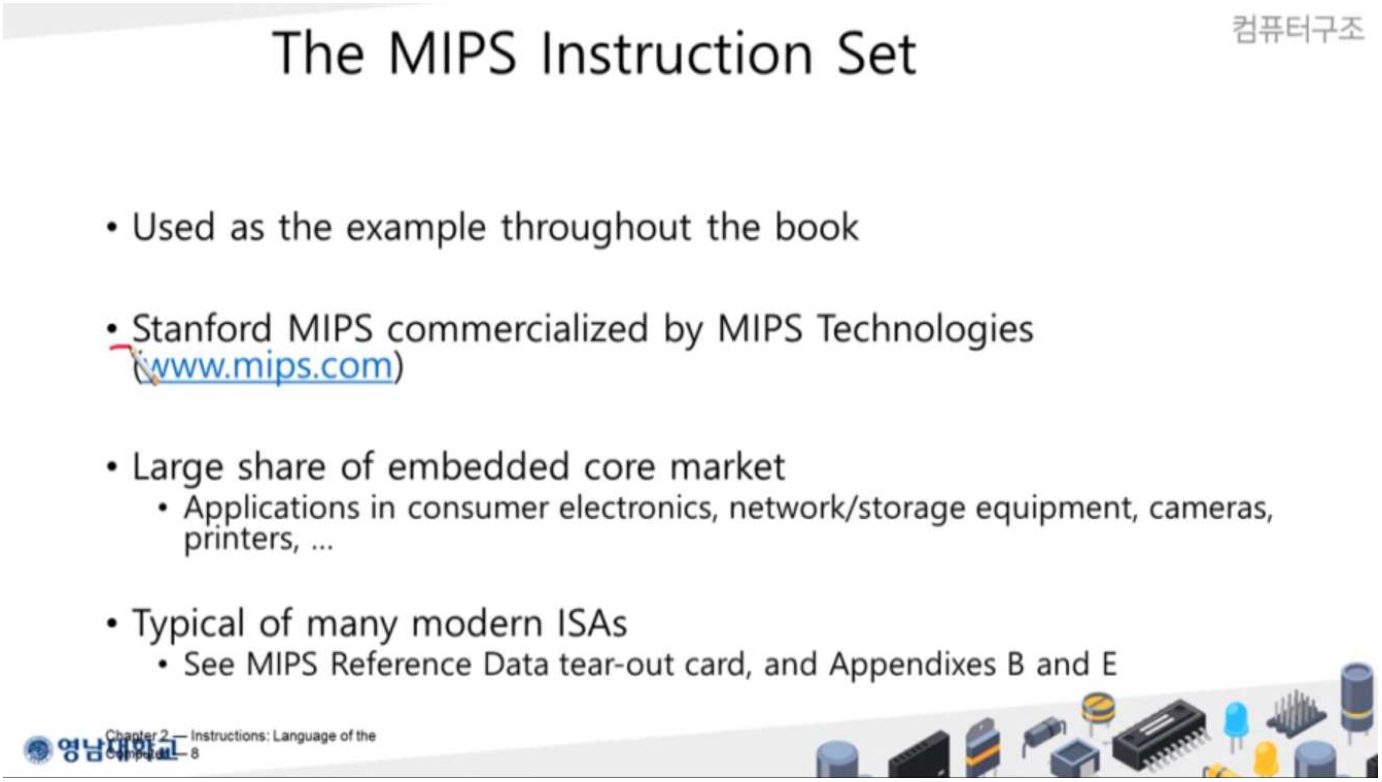
- The repertoire of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets

- 컴퓨터에서 지원하는 명령어들의 집합
- 프로세스들마다 다른 iss 를 가지고 있다.
- 초창기 컴퓨터들은 simple is 를 가지고 있었다.
- 현대의 많은 프로세서들은 simple is 로 가고 있다. ex.MIPS

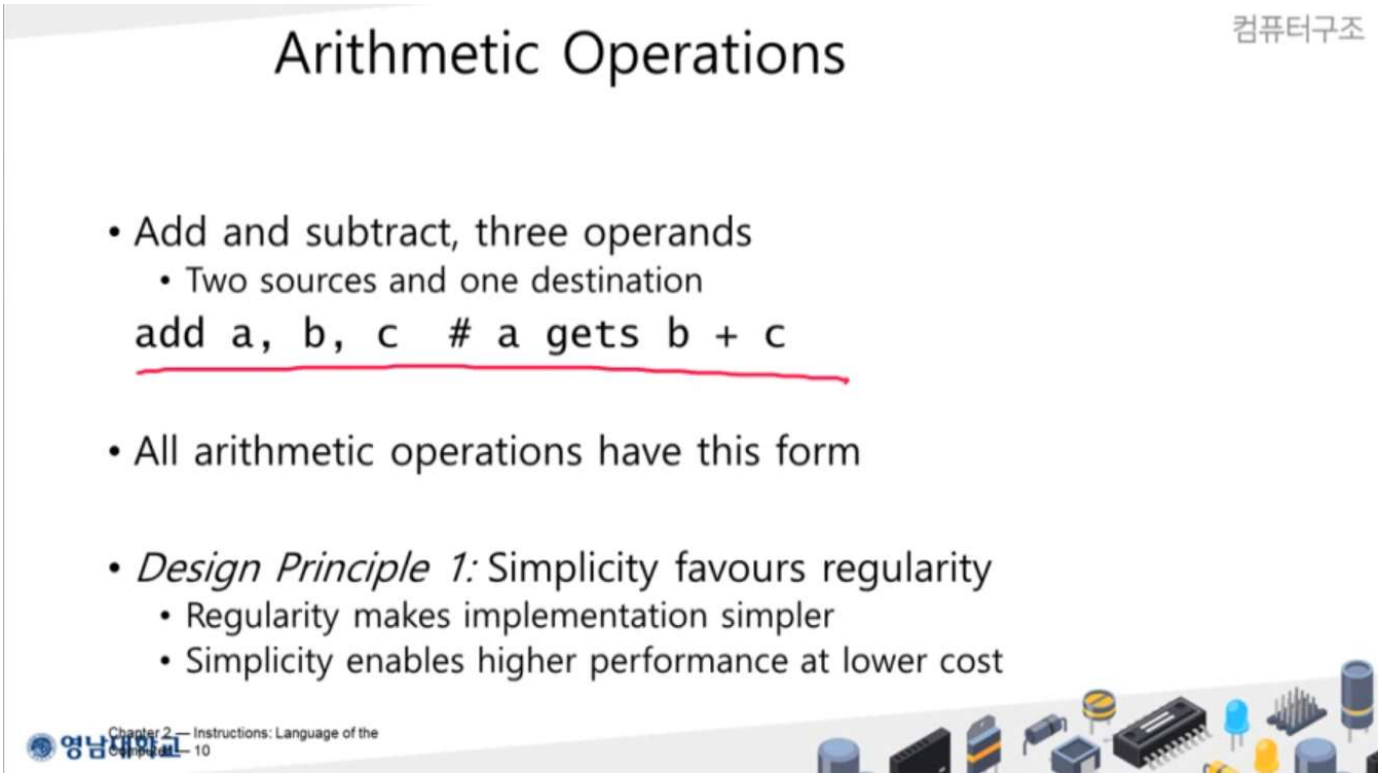
Instruction Set Architecture (ISA)

- ISA, or simply architecture – the abstract interface between the hardware and the lowest level software that encompasses all the information necessary to write a machine language program, including instructions, registers, memory access, I/O, ...
  - Enables implementations of varying cost and performance to run identical software
- The combination of the basic instruction set (the ISA) and the operating system interface is called the application binary interface (ABI)
  - ABI – The user portion of the instruction set plus the operating system interfaces used by application programmers. Defines a standard for binary portability across computers.

- ABI : 다른 컴퓨터에서도 구동될수있게 ??



- Embedded 시스템에 많이 사용된다.
- MIPS가 최신의 ISA의 전형적인 예



- 간단하게 만들기위해선 정규성,규칙이 주어져야함
- 간단하게 만들어야 저비용으로 고성능을 만들기 쉽다.

Arithmetic Example

- C code:

```
f = (g + h) - (i + j);
```

- Compiled MIPS code:

```
add t0, g, h   # temp t0 = g + h
add t1, i, j   # temp t1 = i + j
sub f, t0, t1  # f = t0 - t1
```


- 위와같은 일은 컴파일러가 한다.

Register Operands

- Arithmetic instructions use register operands
- MIPS has a 32 × 32-bit register file
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a "word"
- Assembler names
  - $t0, $t1, ..., $t9 for temporary values
  - $s0, $s1, ..., $s7 for saved variables
- *Design Principle 2:* Smaller is faster
  - c.f. main memory: millions of locations

- Arithmetic instructions는 레지스터간의 연산이다.

Register Operand Example

- C code:

```
f = (g + h) - (i + j);
```

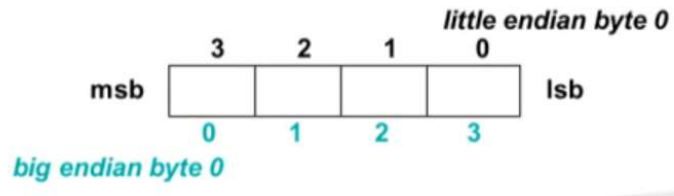
  - f, …, j in $s0, …, $s4

- Compiled MIPS code:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

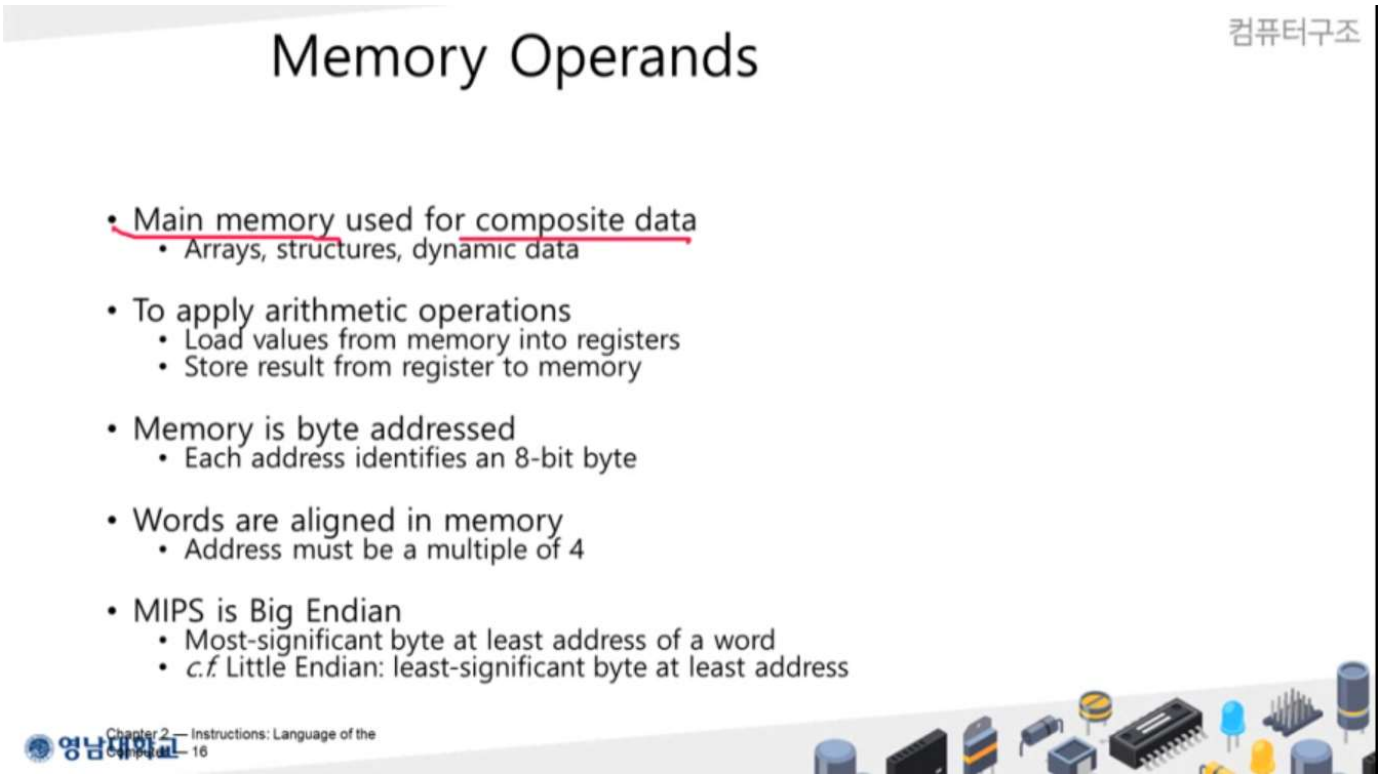

Byte Addresses

- Since 8-bit bytes are so useful, most architectures address individual bytes in memory
  - Alignment restriction - the memory address of a word must be on natural word boundaries (a multiple of 4 in MIPS-32)
- Big Endian: leftmost byte is word address
  IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- Little Endian: rightmost byte is word address
  Intel 80x86, DEC Vax, DEC Alpha (Windows NT)



- 대부분의 프로세서들은 바이트가 기본 단위이다.
- MIPS는 모든 명령어가 32bit이다.

- 데이터를 메모리에 어떻게 저장할것이냐의 이슈에서 나온것 (Big Endian과 Little Endian으로 나뉨)
- 가장 오른쪽에 높은값 부터 넣는것.
- 가장 오른쪽에 낮은값 부터 넣는것.



- 메인메모리는 주로 데이타들
- arithmetic operations 적용하려면 : 메모리에있는것을 레지스터에 올리고, 사용할때는 레지스터에있는것을 메모리로 저장한다 : Load/Store Architecture
- Load/Store Architecture : 메모리에 접근하는 명령어들의 집합, 이 명령어를 사용해야만 메모리에 접근 가능.
- 메모리의 기본 단위는 byte
- 메모리의 있는 모든 주소는 4byte로 되어있다.
- MIPS는 Big Endian이다.
- 책) 자주 사용하지 않는 변수를 메모리에 넣는 것을 레지스터를 spiling 한다고 말한다.

메모리 연산 예제

Memory Operand Example 1

- C code:

```
g = h + A[8];
```

  - g in $s1, h in $s2, base address of A in $s3
- Compiled MIPS code:
  - Index 8 requires offset of 32
    - 4 bytes per word

```
lw  $t0, 32($s3)    # load word
add $s1, $s2, $t0
```

offset

base register

Memory Operand Example 2

- C code:

```
A[12] = h + A[8];
```

  - h in $s2, base address of A in $s3
- Compiled MIPS code:
  - Index 8 requires offset of 32

```
lw  $t0, 32($s3)    # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)    # store word
```

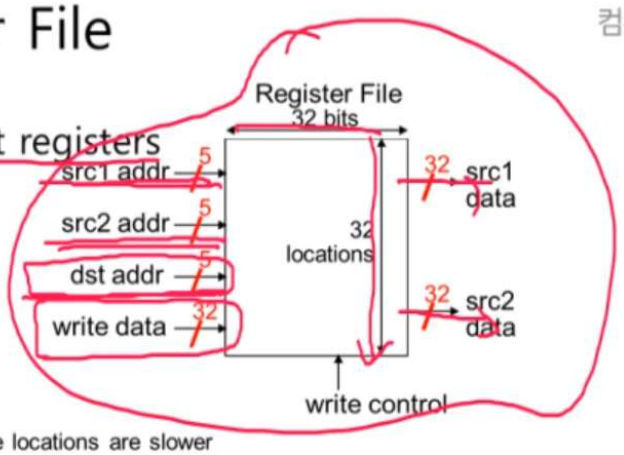
Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!

- 레지스터는 메모리에 접근하는것보다 훨씬 빠르다.
- 메모리에있는 데이타를 연산하기위해선 로드 스토어 명령어를 사용해야한다.
- 가능하면 레지스터를 최대한 활용하는것이 좋다.
- 자주사용하지않는 데이터는 메모리에 둔다.

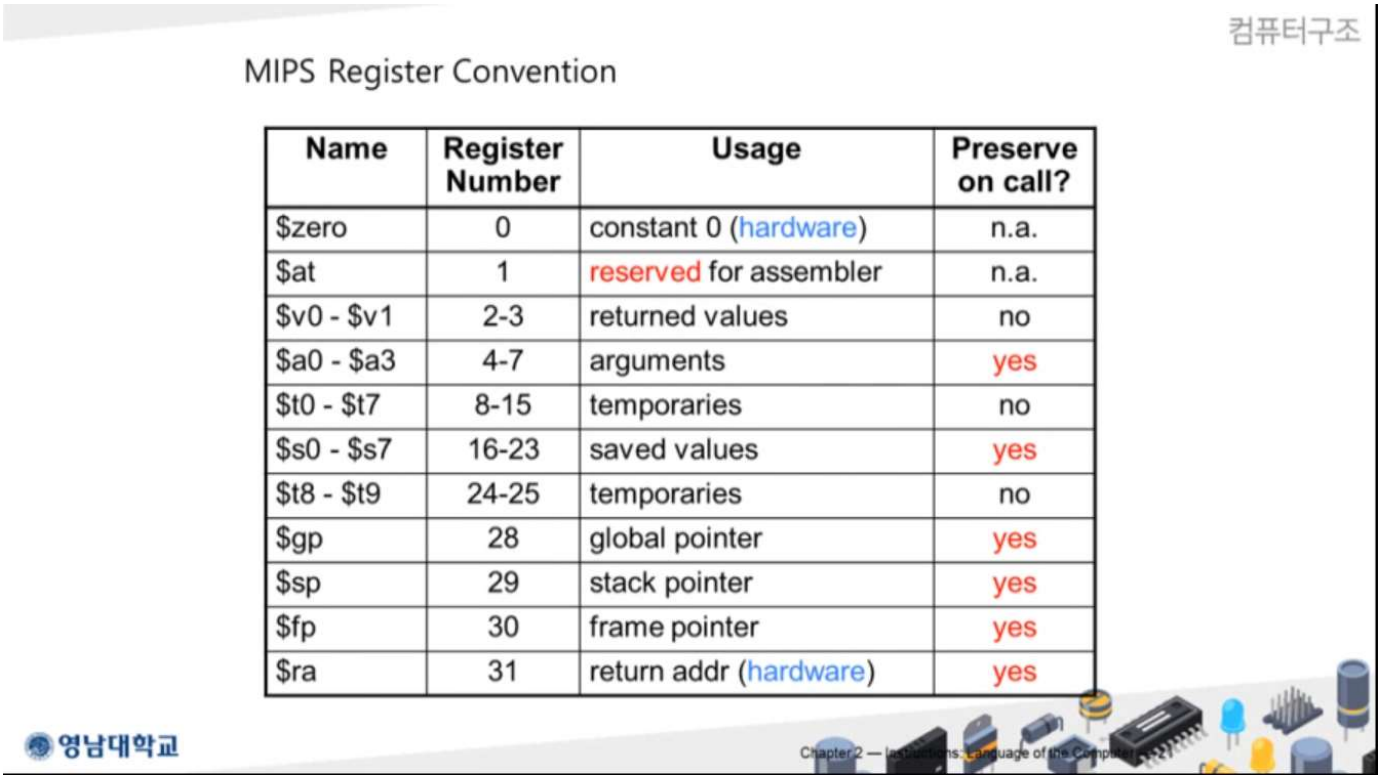
MIPS Register File

- Holds thirty-two 32-bit registers
  - Two read ports and
  - One write port



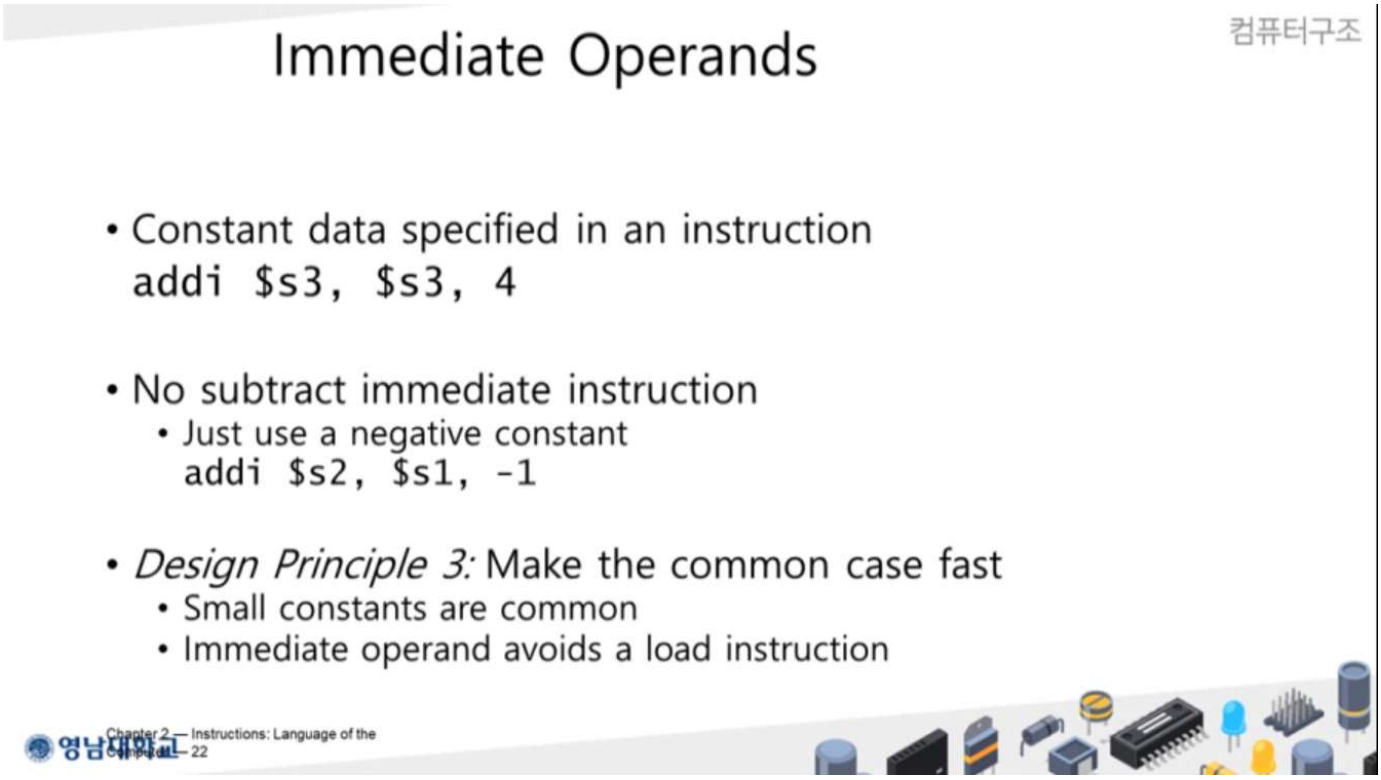
- Registers are
  - Faster than main memory
    - But register files with more locations are slower
    - (e.g., a 64 word file could be as much as 50% slower than a 32 word file)
    - Read/write port increase impacts speed quadratically
  - Easier for a compiler to use
    - e.g., (A*B) – (C*D) – (E*F) can do multiplies in any order vs. stack
  - Can hold variables so that
    - code density improves (since register are named with fewer bits than a memory location)

- 가능하면 레지스터에 많은 변수값을 가지고 있는게 성능향상에 도움이 된다.
- 하지만 레지스터를 키우면 느려질수 있다. 그러므로 자주 사용하지 않는 변수들은 메모리에 직접 올리는것이 효율적일 수 있다.

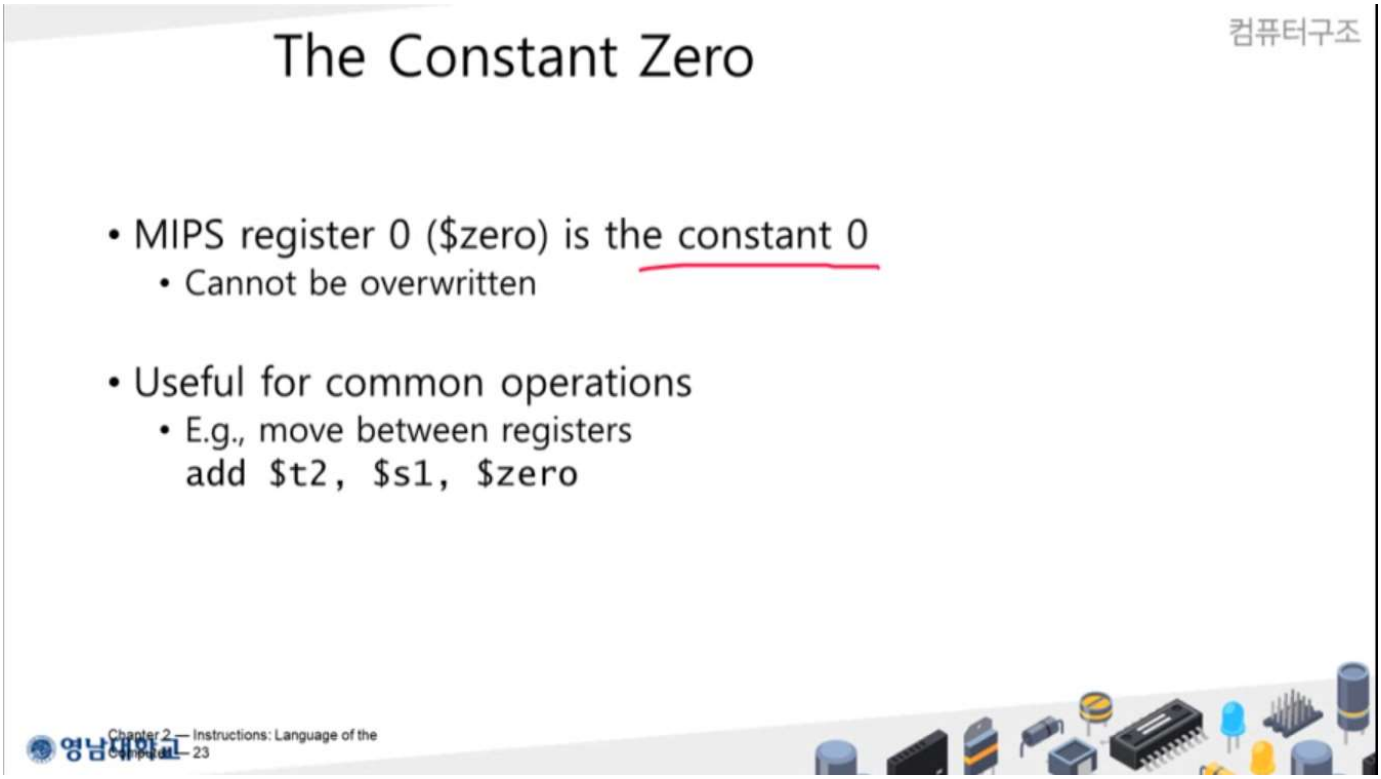


Name	Register Number	Usage	Preserve on call?
$zero	0	constant 0 (hardware)	n.a.
$at	1	reserved for assembler	n.a.
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	yes
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	yes
$t8 - $t9	24-25	temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return addr (hardware)	yes

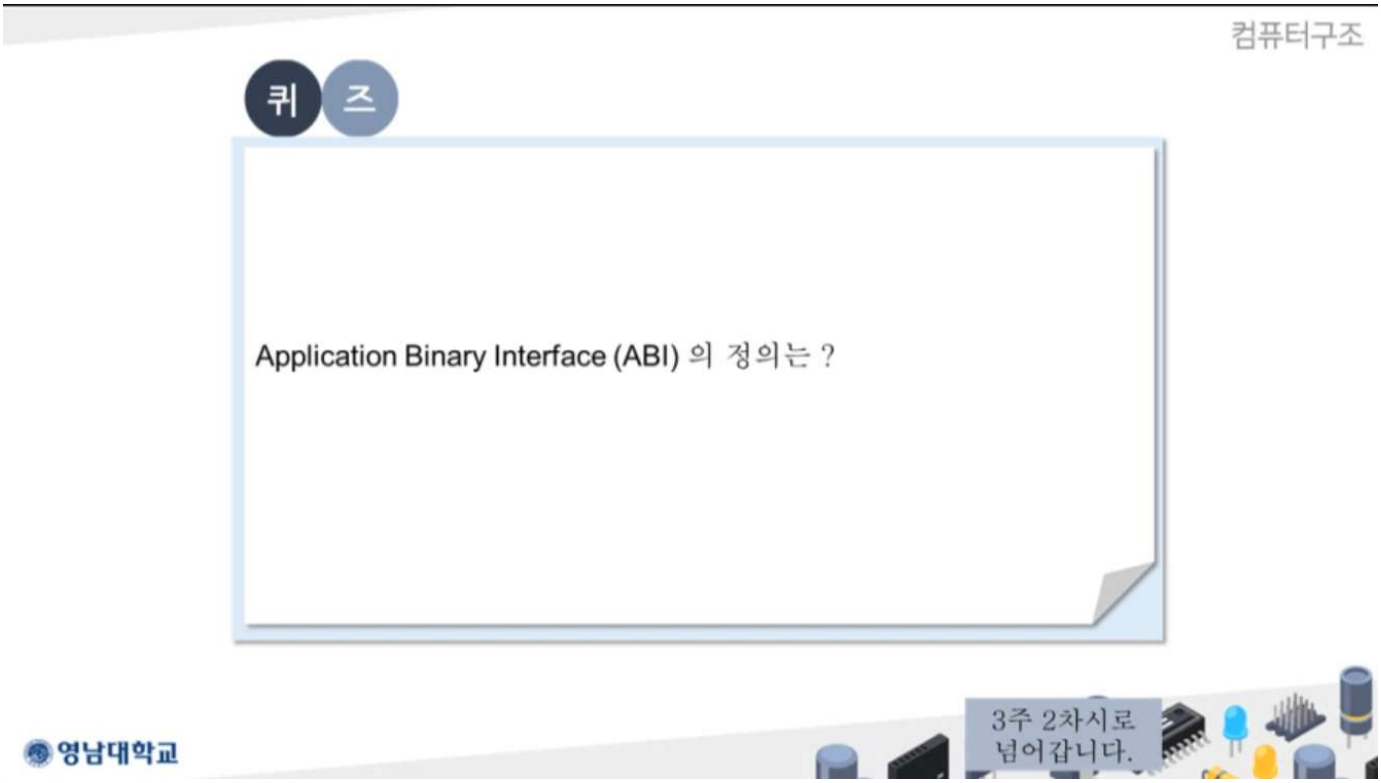
- 레지스터의 종류와 사용처 참고



- 오퍼랜드 중에 하나가 상수인것을 immediate 라고 한다.



- 제로레지스터를 만든이유 : move와 copy같은 명령어들을 만들지 않아도 단순히 빈레지스터(제로레지스터)에 넣기만 하면 되기때문에 간단해서 사용한다.



- 응용 프로그램과 운영 체제 또는 응용 프로그램과 해당 라이브러리, 마지막으로 응용 프로그램의 구성요소 간에서 사용되는 낮은 수준의 인터페이스이다.

Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: 0 to $+2^n - 1$
- Example
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bits
  - 0 to +4,294,967,295

2s-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
    $= -2,147,483,648 + 2,147,483,644 = -4_{10}$
- Using 32 bits
  - –2,147,483,648 to +2,147,483,647

2s-Complement Signed Integers

- Bit 31 is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers
- $-(-2^{n-1})$ can't be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111

Signed Negation

- Complement and add 1
  - Complement means 1 → 0, 0 → 1

$$x + \bar{x} = 1111\ldots111_2 = -1$$

$$\bar{x} + 1 = -x$$

- Example: negate +2
  - +2 = 0000 0000 … 0010_2
  - –2 = 1111 1111 … 1101_2 + 1
    = 1111 1111 … 1110_2

Sign Extension

- Representing a number using more bits
  - Preserve the numeric value
- In MIPS instruction set
  - `addi`: extend immediate value
  - `lb`, `lh`: extend loaded byte/halfword
  - `beq`, `bne`: extend the displacement
- Replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110

- 비트수가 증가할때 그것을 어떻게 처리할 것인가.

Representing Instructions

- Instructions are encoded in binary
  - Called machine code
- MIPS instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, ...
  - Regularity!
- Register numbers
  - $t0 – $t7 are reg's 8 – 15
  - $t8 – $t9 are reg's 24 – 25
  - $s0 – $s7 are reg's 16 – 23

- instruction은 바이너리 코드로 되어있다.

MIPS R-format Instructions

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Instruction fields
  - op: operation code (opcode)
  - rs: first source register number
  - rt: second source register number
  - rd: destination register number
  - shamt: shift amount (00000 for now)
  - funct: function code (extends opcode)

R-format Example

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

```
add $t0, $s1, $s2
```

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$0000001000110010010000000100000_2 = 02324020_{16}$

Hexadecimal

- Base 16
  - Compact representation of bit strings
  - 4 bits per hex digit

0	0000	4	0100	8	1000	c 12	1100
1	0001	5	0101	9	1001	d 13	1101
2	0010	6	0110	a 10	1010	e 14	1110
3	0011	7	0111	b 11	1011	f 15	1111

- Example: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

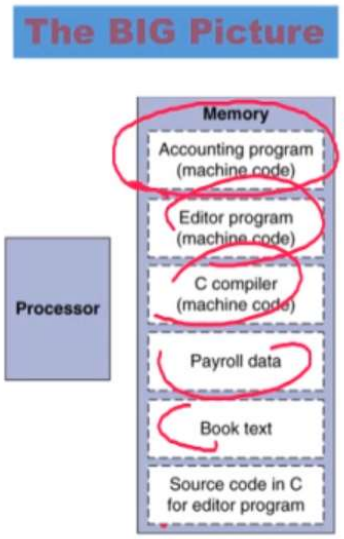
MIPS I-format Instructions

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: -2^{15} to $+2^{15} - 1$
  - Address: offset added to base address in rs

- *Design Principle 4:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

Stored Program Computers



- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs

2.6

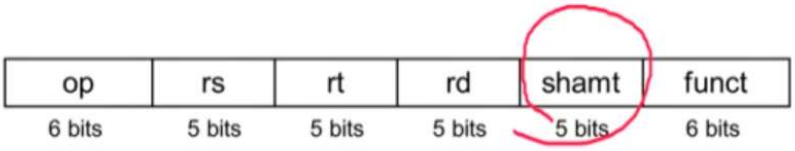
Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	MIPS
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR	|	|	or, ori
Bitwise NOT	~	~	nor

- Useful for extracting and inserting groups of bits in a word

Shift Operations



- shamt: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - `sll` by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - `srl` by *i* bits divides by 2^i (unsigned only)

- Shift left logical 은 2^i 만큼 곱한다. (음양수 다 적용 가능)
- Shift right logical 은 2^i 만큼 나눈다. 빈 비트는 0을 넣는다. (양수에만 적용 가능)

AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

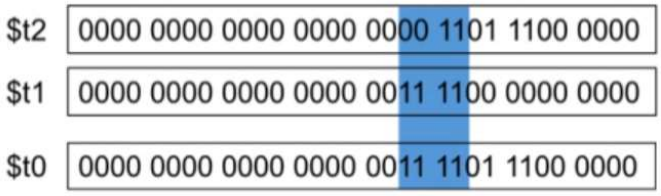
```
and $t0, $t1, $t2
```

- AND연산 : 어떤 특정할 값을 뽑아내는데 사용

OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

```
or $t0, $t1, $t2
```



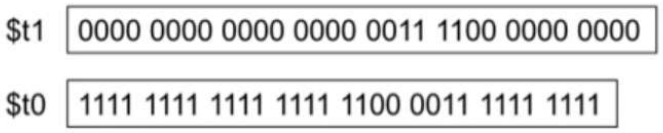
- OR연산 : 어떤 특정한 값을 1로 만들 때 필요.

NOT Operations

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0
- MIPS has NOR 3-operand instruction
  - a NOR b == NOT (a OR b)

```
nor $t0, $t1, $zero
```

Register 0: always read as zero



- MIPS에서는 NOT 연산은 없다. NOR 사용

Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `beq rs, rt, L1`
  - if (rs == rt) branch to instruction labeled L1;
- `bne rs, rt, L1`
  - if (rs != rt) branch to instruction labeled L1;
- `j L1`
  - unconditional jump to instruction labeled L1

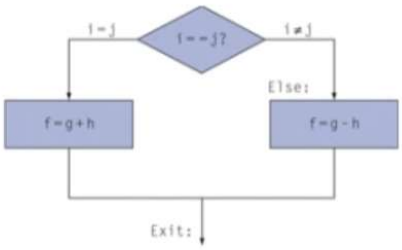
- branch 명령어
- beq rs, rt, l1 = rs 와 rt 가 같으면 l1 명령어를 실행하라
- bne rs, rt, l1 = rs 와 rt 가 다르면 l1 명령어를 실행하라
- j, l1 조건없이 무조건 실행

Compiling If Statements

- C code:

```
if (i==j) f = g+h;
else f = g-h;
```

  - f, g, ... in $s0, $s1, ...



- Compiled MIPS code:

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

Assembler calculates addresses

Compiling Loop Statements

- C code:

```
while (save[i] == k) i += 1;
```

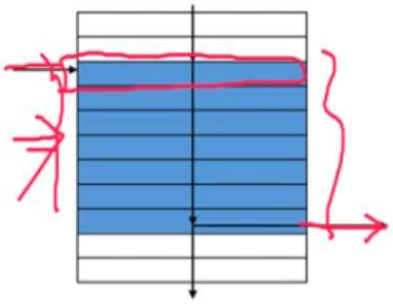
  - i in $s3, k in $s5, address of save in $s6

- Compiled MIPS code:

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```


Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)



- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

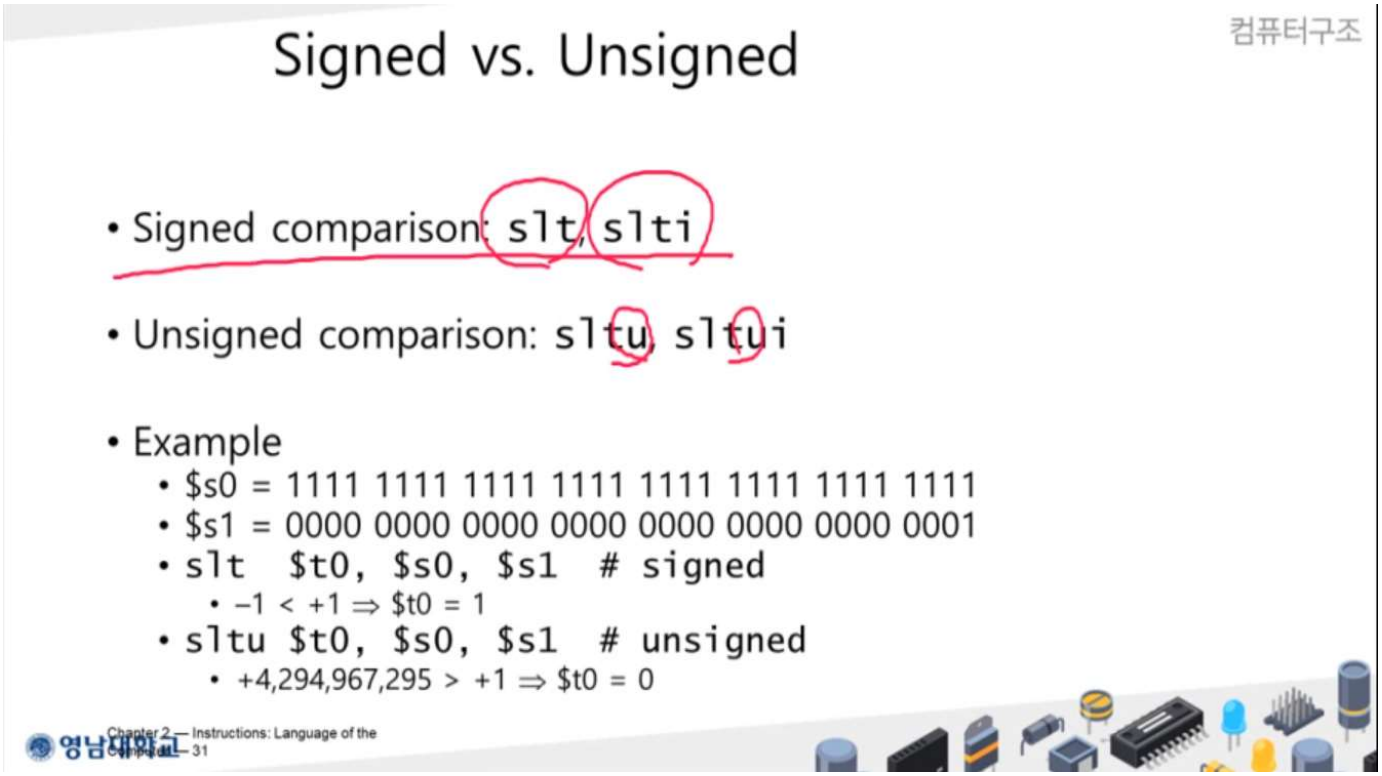
More Conditional Operations

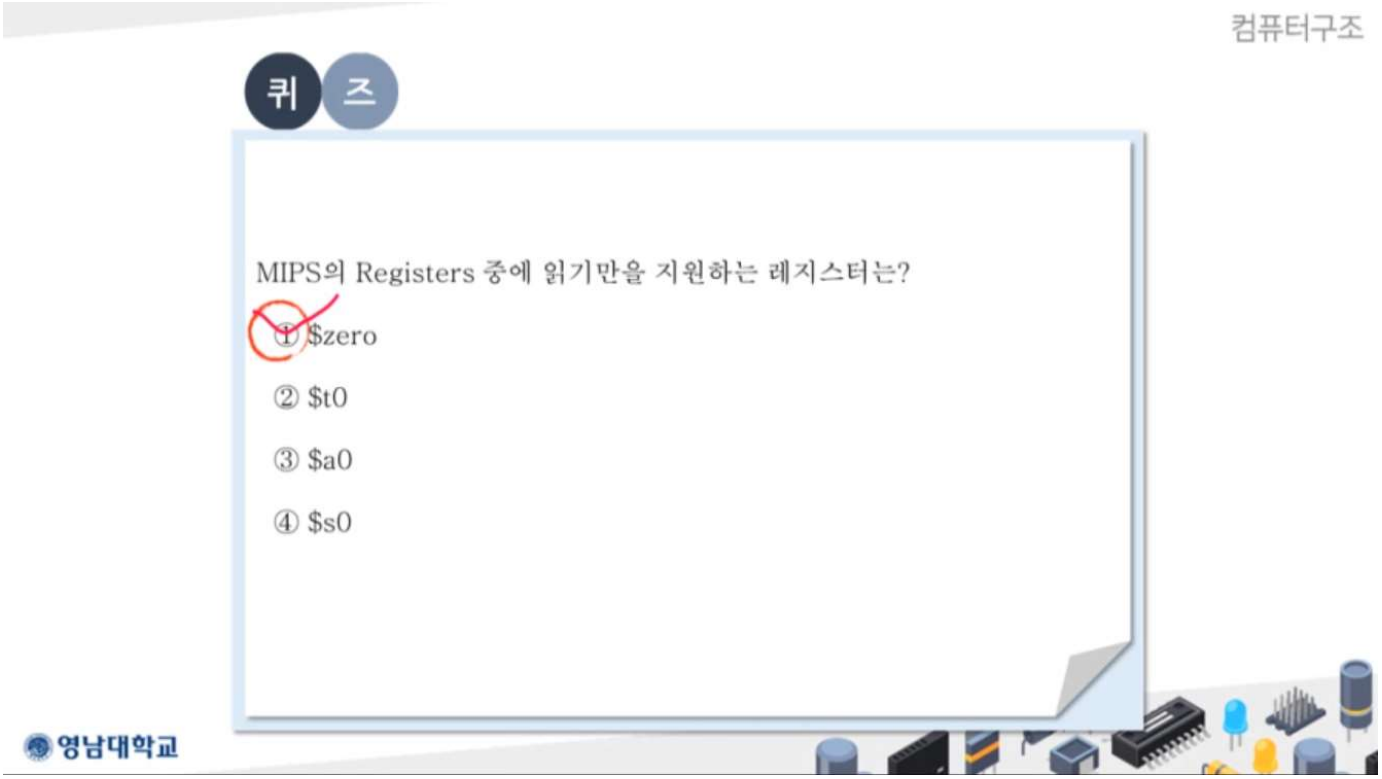
- Set result to 1 if a condition is true
  - Otherwise, set to 0
- `slt rd, rs, rt`
  - if (rs < rt) rd = 1; else rd = 0;
- `slti rt, rs, constant`
  - if (rs < constant) rt = 1; else rt = 0;
- Use in combination with **beq**, **bne**

```
slt $t0, $s1, $s2  # if ($s1 < $s2)
bne $t0, $zero, L  #   branch to L
```


Branch Instruction Design

- Why not **blt**, **bge**, etc?
- Hardware for <, ≥, ... slower than =, ≠
  - Combining with branch involves more work per instruction, requiring a slower clock
  - All instructions penalized!
- **beq** and **bne** are the common case
- This is a good design compromise





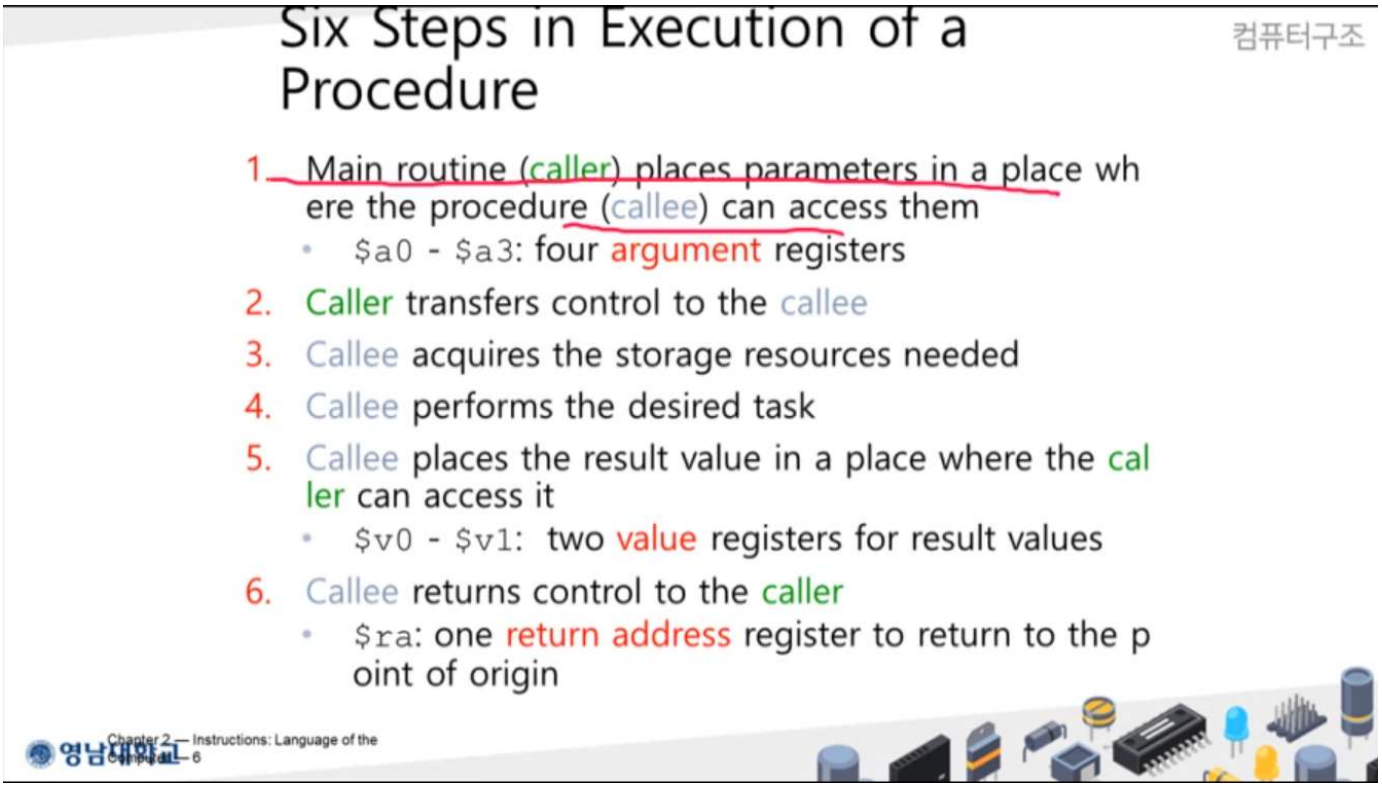
요점정리

- 명령어는 숫자로 표현된다.
- 프로그램은 메모리에 기억되어 있어서 데이터처럼 읽고 쓸 수 있다.
  이것이 내장 프로그램의 개념이다. 이 개념을 발명한 덕택에 컴퓨터가 눈부시게 발전할 수있었다, 메모리에는 편집기가 편집 중인 소스 코드, 컴파일된 기계어 프로그램, 실행

프로그램이 사용하는 텍스트 데이터, 심지어는 기계어를 생성하는 컴파일러 까지도 기억될 수 있다.

- 명령어를 숫자처럼 취급하게 된 결과, 프로그램이 이진수 파일 형태로 판매되게 되었다. 이것이 상업적으로는 만약 기존 명령어 집ㅈ합과 호환성이 있다면 다른 컴퓨터의 소프트웨어를 물려 받을 수 있다는 의미를 갖는다. 이러한 "이진 호환성" 문제 때문에 상업적으로 살아남는 명령어 집합 구조는 극히 소수로 집집약된다.

== 4주차 7,8 강의 ==



6단계의 프로시져 수행단계

1. caller가 callee를 액세스 하는곳에 파라미터(아규먼트를 가져다 놓는다)
2. caller가 callee로 컨트롤을 넘김
3. callee가 실행하기 위해서 필요한 메모리를 할당받는다.
4. callee가 자기가 하게될 일을 수행한다.
5. callee가 자기가 할 일을 수행을 마치게 되면 caller에게 return 밸류를 넘겨준다.
6. callee가 caller에게 컨트롤러를 넘긴다.

- caller : 함수를 부르는 함수.
- callee : 불려지는 함수.

Leaf Procedure Example

- C code:

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

  - Arguments g, ..., j in $a0, ..., $a3
  - f in $s0 (hence, need to save $s0 on stack)
  - Result in $v0

- Leaf Procedure : 함수내에서 다른 함수를 부르지 않는 함수

Leaf Procedure Example

- MIPS code:

```
leaf_example:
    addi $sp, $sp, -4
    sw   $s0, 0($sp)
    add  $t0, $a0, $a1
    add  $t1, $a2, $a3
    sub  $s0, $t0, $t1
    add  $v0, $s0, $zero
    lw   $s0, 0($sp)
    addi $sp, $sp, 4
    jr   $ra
```

Save $s0 on stack

Procedure body

Result

Restore $s0

Return

Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

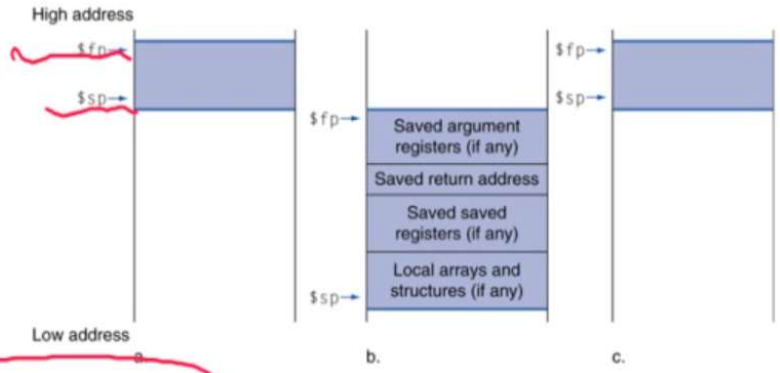
- Non-Leaf Procedure : 함수 내에서 다른 함수를 부르는 함수
- caller는 stack에 필요한 것들을 저장 해 놓는다. (return address값, 인자값 등등)
- 사용한 후에 복원 시켜야 함.

Non-Leaf Procedure Example

- MIPS code:

```
fact:
    addi $sp, $sp, -8      # adjust stack for 2 items
    sw   $ra, 4($sp)       # save return address
    sw   $a0, 0($sp)       # save argument
    slti $t0, $a0, 1       # test for n < 1
    beq  $t0, $zero, L1
    addi $v0, $zero, 1     # if so, result is 1
    addi $sp, $sp, 8       #   pop 2 items from stack
    jr   $ra               #   and return
L1: addi $a0, $a0, -1      # else decrement n
    jal  fact              # recursive call
    lw   $a0, 0($sp)       # restore original n
    lw   $ra, 4($sp)       #   and return address
    addi $sp, $sp, 8       # pop 2 items from stack
    mul  $v0, $a0, $v0     # multiply to get result
    jr   $ra               # and return
```


Local Data on the Stack

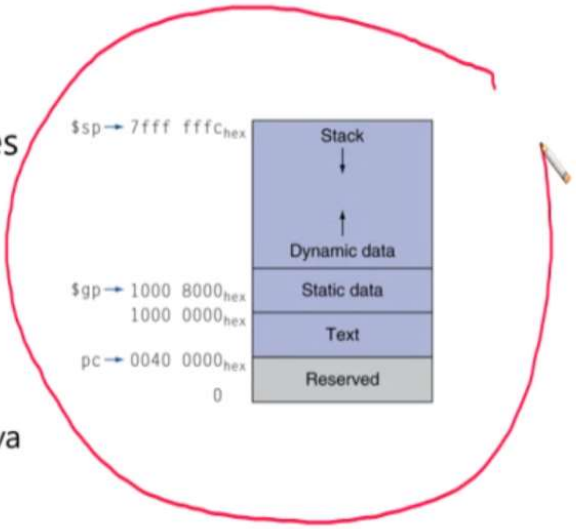


- Local data allocated by callee
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage

- 지역변수는 stack에 할당이 된다.

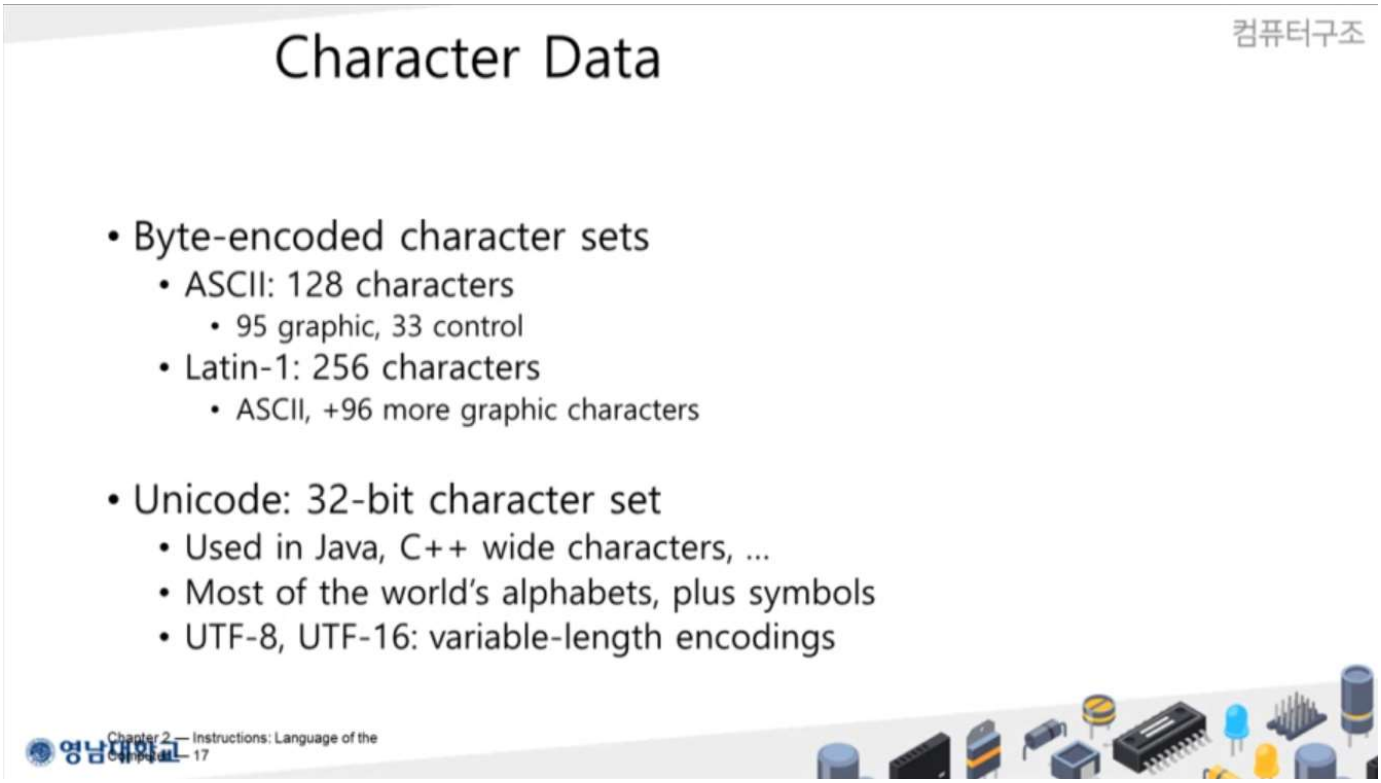
Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage



- text : 프로그램 코드(명령어들)
- static data : 전역변수
- gp(global pointer) : static data를 저장할 수 있는 중간값.

- heap : 객체를 저장할 수 있는곳.
- stack :



- Byte- encoded character
  ASCII : 128 characters
  Latin-1 : 256 characters 영미권에서 사용

- Unicode
  UTF-8
  UTF-16

퀴 즈

"jal ProcedureLabel" 명령어의 의미는?

- 정답 : 프로시저 레이블에 있는곳으로 이 멸령어를 실행한 후에 이 프로시져 레이블에있는 명령어가 다음 실행할 명령어가 되는것이고 동시에 jal 명령어 다음에 있는 명령어에 시작 메모리 주석이 ra 레지스터에 복사하게 된다.

32-bit Constants

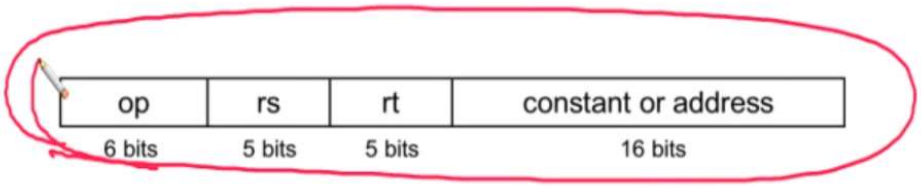
- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant

  `lui rt, constant`
  - Copies 16-bit constant to left 16 bits of rt
  - Clears right 16 bits of rt to 0

```
lui $s0, 61      0000 0000 0111 1101 0000 0000 0000 0000
ori $s0, $s0, 2304  0000 0000 0111 1101 0000 1001 0000 0000
```

Branch Addressing

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward

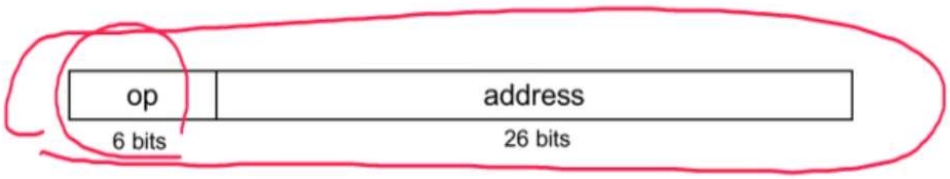


- PC-relative addressing
  - Target address = PC + offset × 4
  - PC already incremented by 4 by this time

- Branch 명령어의 구성 : Opcode, two registers, target address

Jump Addressing

- Jump (`j` and `jal`) targets could be anywhere in text segment
  - Encode full address in instruction



- (Pseudo)Direct jump addressing
  - Target address = $PC_{31...28}$: (address × 4)

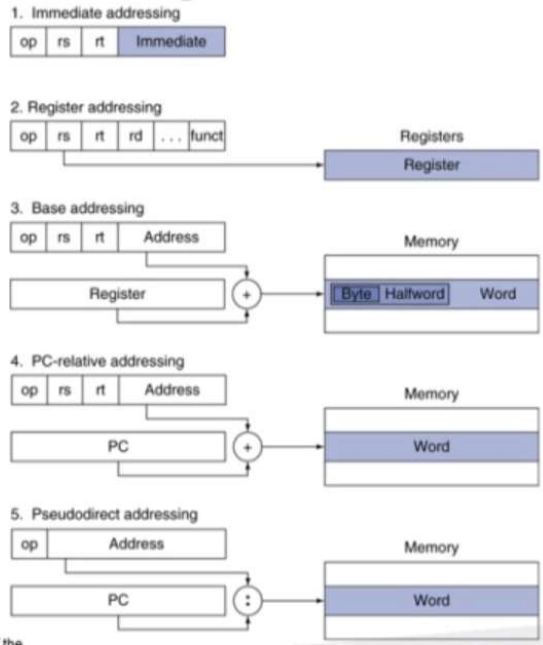
Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000

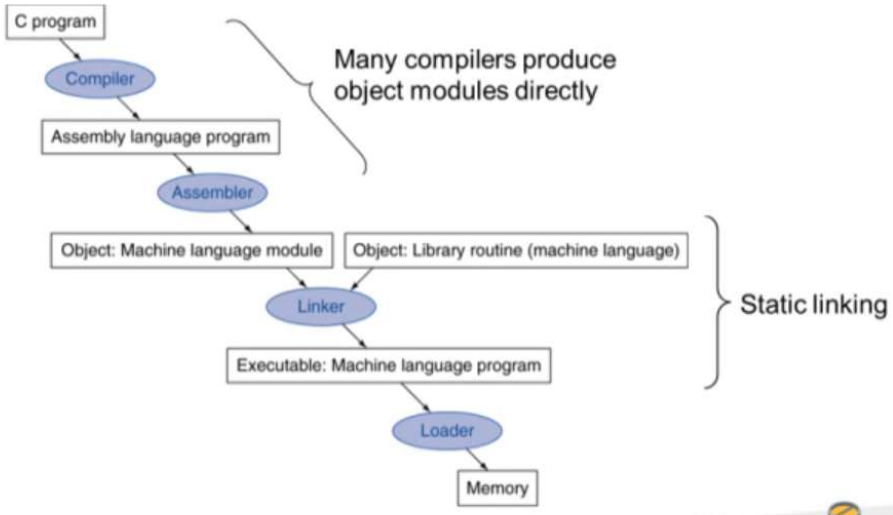
```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024						

Addressing Mode Summary



Translation and Startup



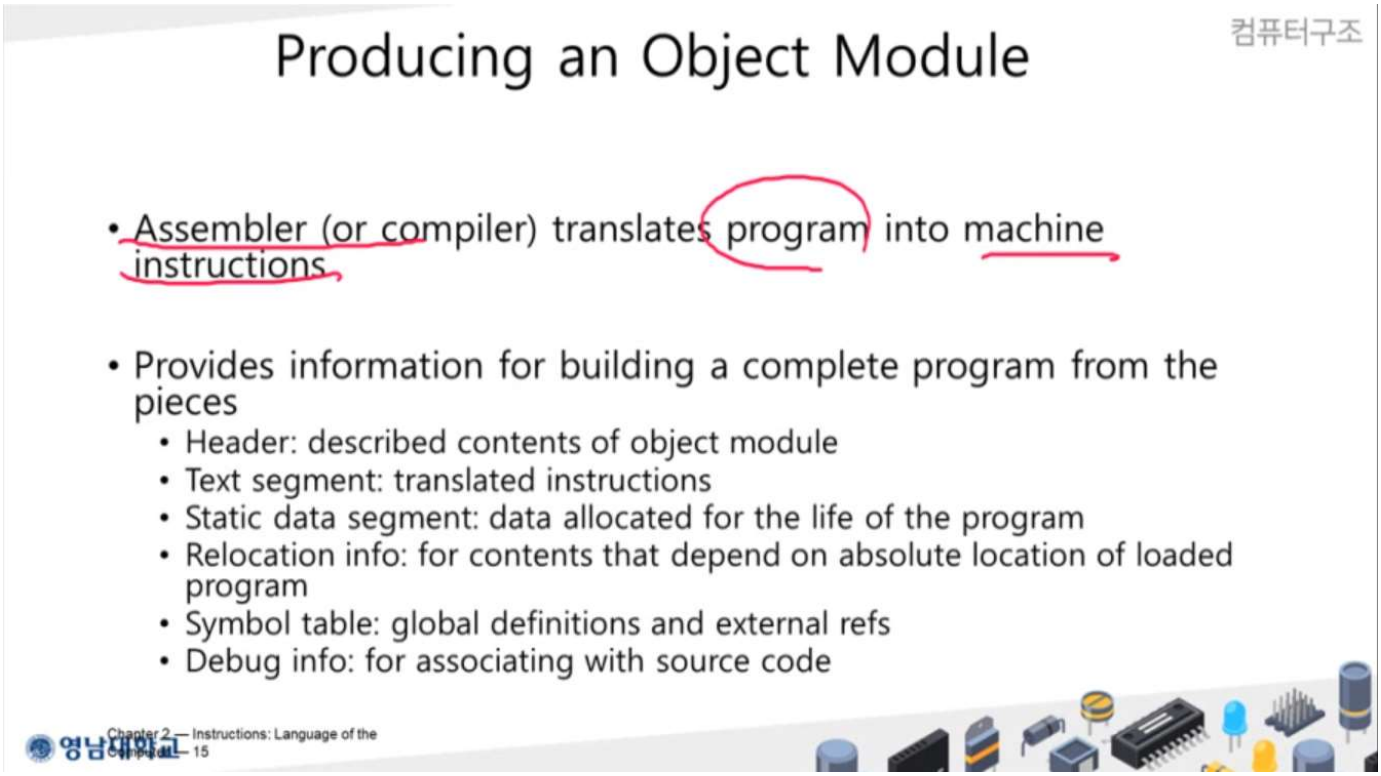
- 프로그래밍의 시작 단계

Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudoinstructions: figments of the assembler's imagination

```
move $t0, $t1  →  add $t0, $zero, $t1
blt $t0, $t1, L  →  slt $at, $t0, $t1
                    bne $at, $zero, L
```

  - $at (register 1): assembler temporary



- Assembler : 프로그램을 머신코드로 바꾼다.

- Header : 이 모듈에 대한 내용을 담고있다.
- Text segment : 명령어들
- Static data segment : 전역변수
- Relocation info : 어떤것들은 시작주소에 의해서 어드레스들이 결정되는것이 있다.
- Symbol table : 함수, 전역변수 등등
- Debug info : 디버깅을 위해서 소스코드에 관련한 정보를 가지고 있다.

Linking Object Modules

- Produces an executable image
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external refs
- Could leave location dependencies for fixing by a relocating loader
  - But with virtual memory, no need to do this
  - Program can be loaded into absolute location in virtual memory space

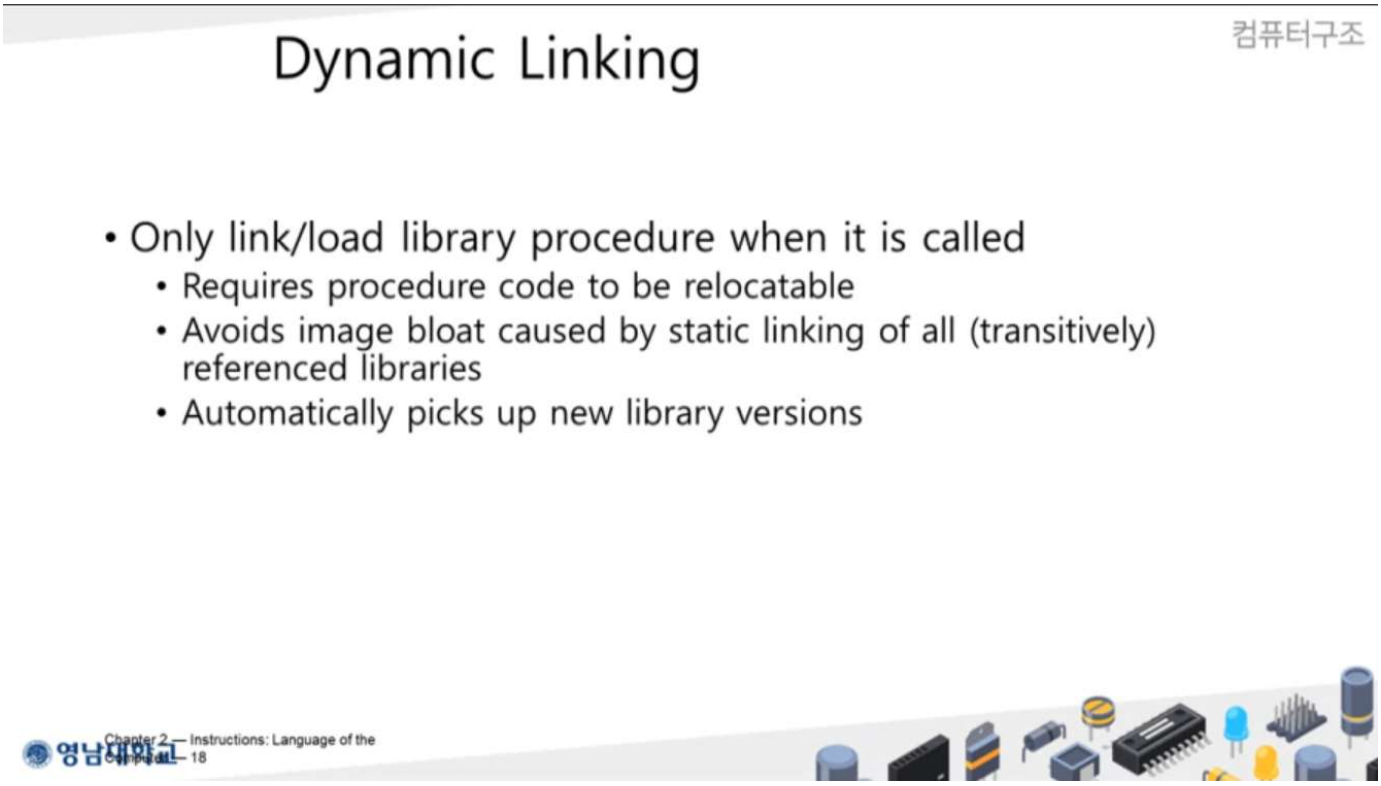
프로그램을 로딩하는 순서

Loading a Program

- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space
  3. Copy text and initialized data into memory
     - Or set page table entries so they can be faulted in
  4. Set up arguments on stack
  5. Initialize registers (including $sp, $fp, $gp)
  6. Jump to startup routine
     - Copies arguments to $a0, ... and calls main
     - When main returns, do exit syscall

1. 헤더를 읽어 각각의 세그먼트의 크기를 읽는다
2. 가상 메모리 장소를 만든다.
3. 텍스트와 데이터를 메모리로 복사. (페이지 테이블 엔트리만 세팅해 준다)

4. 아규먼트를 스택에 할당.
5. 레지스터를 초기화.
6. 처음 시작한 루틴으로 점프.



- Dynamic Linking : Static Linking과 반대 되는 개념
- 라이브러리를 사용한다.
- Static Linking에 비해서 실제 이미지 크기가 커지지 않는다.
- 새로운 라이브러리가 나오면 자동으로 라이브러리가 적용되게 한다 (확장성 용이? 같은뜻)

Lazy Linkage

Indirection table

Stub: Loads routine ID, Jump to linker/loader

Linker/loader code

Dynamically mapped code

a. First call to DLL routine

b. Subsequent calls to DLL routine

퀴즈

MIPS의 Addressing mode가 아닌 것은?

① Immediate Addressing

② Register Addressing

③ Base Addressing

④ PC Addressing

== 4주차 9,10 강의 ==

C Sort Example

- Illustrates use of assembly instructions for a C bubble sort function
- Swap procedure (leaf)

```
void swap(int v[], int k)
{
  int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

  - v in $a0, k in $a1, temp in $t0

- 버블정렬

The Procedure Swap

```
swap: sll $t1, $a1, 2    # $t1 = k * 4
      add $t1, $a0, $t1  # $t1 = v+(k*4)
                         #   (address of v[k])
      lw $t0, 0($t1)     # $t0 (temp) = v[k]
      lw $t2, 4($t1)     # $t2 = v[k+1]
      sw $t2, 0($t1)     # v[k] = $t2 (v[k+1])
      sw $t0, 4($t1)     # v[k+1] = $t0 (temp)
      jr $ra             # return to calling routine
```


The Sort Procedure in C

- Non-leaf (calls swap)

```
void sort (int v[], int n)
{
  int i, j;
  for (i = 0; i < n; i += 1) {
    for (j = i - 1;
         j >= 0 && v[j] > v[j + 1];
         j -= 1) {
      swap(v,j);
    }
  }
}
```

- v in $a0, k in $a1, i in $s0, j in $s1

The Procedure Body

```
        move $s2, $a0          # save $a0 into $s2
        move $s3, $a1          # save $a1 into $s3
        move $s0, $zero        # i = 0
for1tst: slt  $t0, $s0, $s3    # $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
        beq  $t0, $zero, exit1 # go to exit1 if $s0 ≥ $s3 (i ≥ n)
        addi $s1, $s0, -1      # j = i - 1
for2tst: slti $t0, $s1, 0      # $t0 = 1 if $s1 < 0 (j < 0)
        bne  $t0, $zero, exit2 # go to exit2 if $s1 < 0 (j < 0)
        sll  $t1, $s1, 2       # $t1 = j * 4
        add  $t2, $s2, $t1     # $t2 = v + (j * 4)
        lw   $t3, 0($t2)       # $t3 = v[j]
        lw   $t4, 4($t2)       # $t4 = v[j + 1]
        slt  $t0, $t4, $t3     # $t0 = 0 if $t4 ≥ $t3
        beq  $t0, $zero, exit2 # go to exit2 if $t4 ≥ $t3
        move $a0, $s2          # 1st param of swap is v (old $a0)
        move $a1, $s1          # 2nd param of swap is j
        jal  swap              # call swap procedure
        addi $s1, $s1, -1      # j -= 1
        j    for2tst           # jump to test of inner loop
exit2:  addi $s0, $s0, 1       # i += 1
        j    for1tst           # jump to test of outer loop
```

Move params

Outer loop

Inner loop

Pass params & call

Inner loop

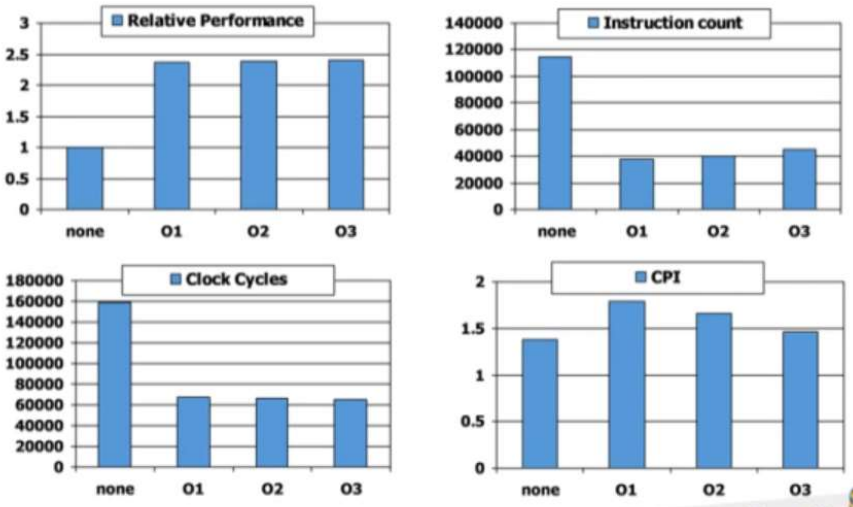
Outer loop

The Full Procedure

```
sort:   addi $sp,$sp, -20      # make room on stack for 5 registers
        sw $ra, 16($sp)        # save $ra on stack
        sw $s3,12($sp)         # save $s3 on stack
        sw $s2, 8($sp)         # save $s2 on stack
        sw $s1, 4($sp)         # save $s1 on stack
        sw $s0, 0($sp)         # save $s0 on stack
        …                      # procedure body
        …
exit1:  lw $s0, 0($sp)         # restore $s0 from stack
        lw $s1, 4($sp)         # restore $s1 from stack
        lw $s2, 8($sp)         # restore $s2 from stack
        lw $s3,12($sp)         # restore $s3 from stack
        lw $ra,16($sp)         # restore $ra from stack
        addi $sp,$sp, 20       # restore stack pointer
        jr $ra
```


Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux



- 컴파일러

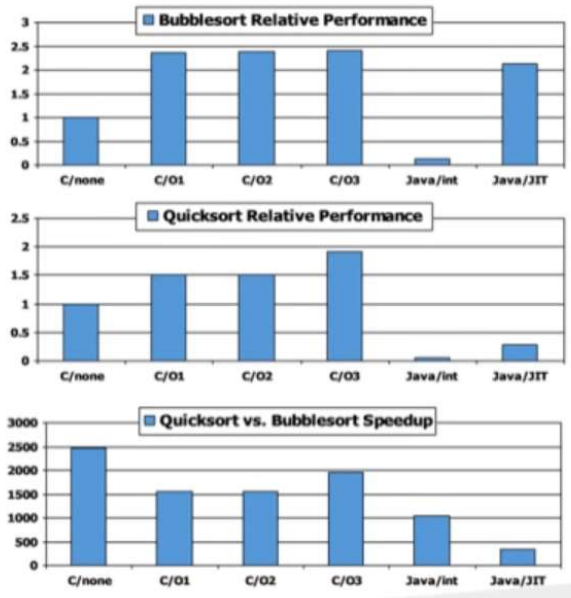
Compiler Benefits

- Comparing performance for bubble (exchange) sort
  - To sort 100,000 words with the array initialized to random values on a Pentium 4 with a 3.06 GHz clock rate, a 533 MHz system bus, with 2 GB of DDR SDRAM, using Linux version 2.4.20

gcc opt	Relative perf ormance	Clock cycl es (M)	Instr coun t (M)	CPI
None	1.00	158,615	114,938	1.38
O1 (medium)	2.37	66,990	37,470	1.79
O2 (full)	2.38	66,521	39,993	1.66
O3 (proc mig)	2.41	65,747	44,993	1.46

- The unoptimized code has the best CPI, the O1 version has the lowest instruction count, but the O3 version is the fastest. Why?

Effect of Language and Algorithm



- 명령어와 CPI는 좋은 성능 지표가 아니니 종합적으로 봐야한다.
- 컴파일러의 성능기준은 알고리즘
- 아무리 컴파일러와 시스템이 좋아도 알고리즘이 제일 중요하다.

Arrays vs. Pointers

- Array indexing involves
  - Multiplying index by element size
  - Adding to array base address
- Pointers correspond directly to memory addresses
  - Can avoid indexing complexity

- Array : 인덱스를 사용한다, 인덱스에 엘리먼트사이즈를 곱하는일을 하고 베이스 어드레스에 더해줘야한다.
- Pointer : 메모리 주소, 인덱스를 사용하지 않아도 된다.

Example: Clearing and Array

```
clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```

```
        move $t0,$zero   # i = 0
loop1: sll $t1,$t0,2     # $t1 = i * 4
       add $t2,$a0,$t1   # $t2 =
                         #   &array[i]
       sw $zero, 0($t2)  # array[i] = 0
       addi $t0,$t0,1    # i = i + 1
       slt $t3,$t0,$a1   # $t3 =
                         #   (i < size)
       bne $t3,$zero,loop1 # if (…)
                           # goto loop1
```

```
clear2(int *array, int size) {
  int *p;
  for (p = &array[0]; p < &array[size];
       p = p + 1)
    *p = 0;
}
```

```
        move $t0,$a0     # p = & array[0]
        sll $t1,$a1,2    # $t1 = size * 4
        add $t2,$a0,$t1  # $t2 =
                         #   &array[size]
loop2:  sw $zero,0($t0)  # Memory[p] = 0
        addi $t0,$t0,4   # p = p + 4
        slt $t3,$t0,$t2  # $t3 =
                         #(p<&array[size])
        bne $t3,$zero,loop2 # if (…)
                            # goto loop2
```


Comparison of Array vs. Ptr

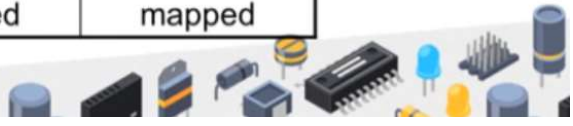
- Multiply "strength reduced" to shift
- Array version requires shift to be inside loop
  - Part of index calculation for incremented i
  - c.f. incrementing pointer
- Compiler can achieve same effect as manual use of pointers
  - Induction variable elimination
  - Better to make program clearer and safer

- Array는 항상 loop 안에서 shift에 관련된 index 계산을 해야한다.
- 최신 컴파일러들은 전문적인 프로그래머가 포인터를 사용해서 프로그램을 작성한 것 만큼의 비슷한 성능을 보여준다.
- Array를 사용하면 프로그램을 이해하기 쉽고, 오류가 발생할 확률이 적다.

ARM & MIPS Similarities

- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

	ARM	MIPS
Date announced	1985	1985
Instruction size	32 bits	32 bits
Address space	32-bit flat	32-bit flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Registers	15 × 32-bit	31 × 32-bit
Input/output	Memory mapped	Memory mapped



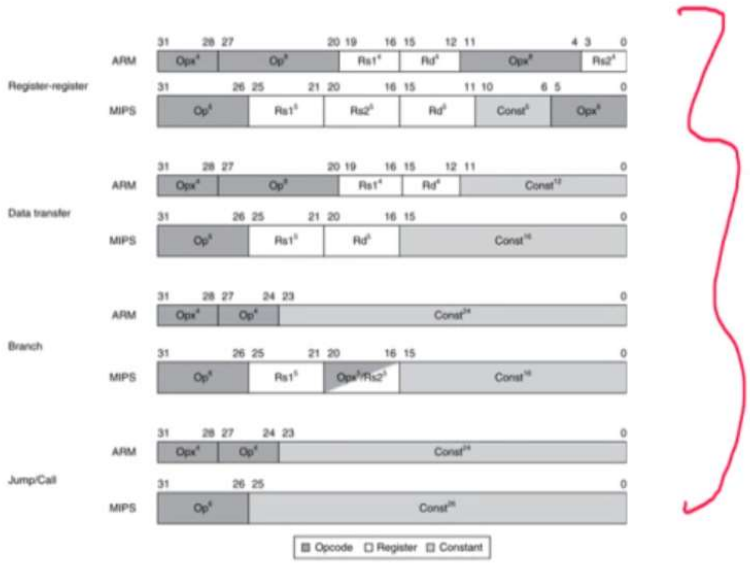
- ARM : MIPS와 비슷함, 요즘 우리가 사용하는 스마트폰은 다 ARM 기반

Compare and Branch in ARM

- Uses condition codes for result of an arithmetic/logical instruction
  - Negative, zero, carry, overflow
  - Compare instructions to set condition codes without keeping the result
- Each instruction can be conditional
  - Top 4 bits of instruction word: condition value
  - Can avoid branches over single instructions

- 명령어를 실행하면 그 결과에 대한 조건을 사용할 수 있다.
- 조건에 따라서 뒤에나오는 명령어들이 실행이될수도 안될수도 있는 컨디션을 제공한다.
- Branch 명령어를 사용하지 않고 Branch 기능을 사용 할 수 있다.

Instruction Encoding



퀴 즈

"lui rt, constant" 명령어의 의미는?

5주 2차시로 넘어갑니다.

The Intel x86 ISA

- Evolution with backward compatibility
  - 8080 (1974): 8-bit microprocessor
    - Accumulator, plus 3 index-register pairs
  - 8086 (1978): 16-bit extension to 8080
    - Complex instruction set (CISC)
  - 8087 (1980): floating-point coprocessor
    - Adds FP instructions and register stack
  - 80286 (1982): 24-bit addresses, MMU
    - Segmented memory mapping and protection
  - 80386 (1985): 32-bit extension (now IA-32)
    - Additional addressing modes and operations
    - Paged memory mapping as well as segments

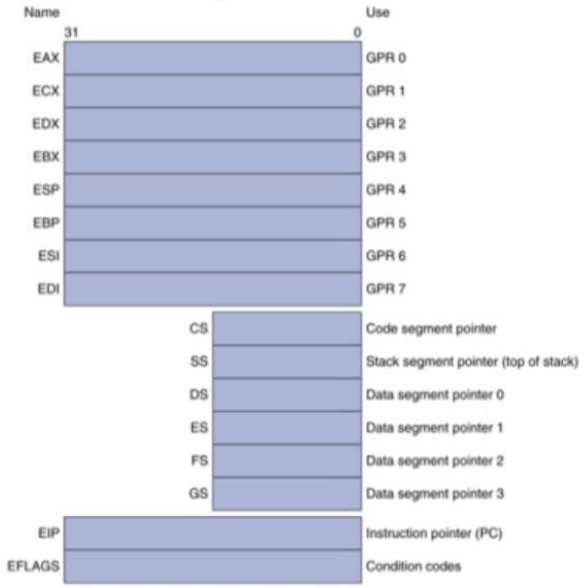
The Intel x86 ISA

- Further evolution...
  - i486 (1989): pipelined, on-chip caches and FPU
    - Compatible competitors: AMD, Cyrix, ...
  - Pentium (1993): superscalar, 64-bit datapath
    - Later versions added MMX (Multi-Media eXtension) instructions
    - The infamous FDIV bug
  - Pentium Pro (1995), Pentium II (1997)
    - New microarchitecture (see Colwell, *The Pentium Chronicles*)
  - Pentium III (1999)
    - Added SSE (Streaming SIMD Extensions) and associated registers
  - Pentium 4 (2001)
    - New microarchitecture
    - Added SSE2 instructions

The Intel x86 ISA

- And further...
  - AMD64 (2003): extended architecture to 64 bits
  - EM64T – Extended Memory 64 Technology (2004)
    - AMD64 adopted by Intel (with refinements)
    - Added SSE3 instructions
  - Intel Core (2006)
    - Added SSE4 instructions, virtual machine support
  - AMD64 (announced 2007): SSE5 instructions
    - Intel declined to follow, instead...
  - Advanced Vector Extension (announced 2008)
    - Longer SSE registers, more instructions
- If Intel didn't extend with compatibility, its competitors would!
  - Technical elegance ≠ market success

Basic x86 Registers



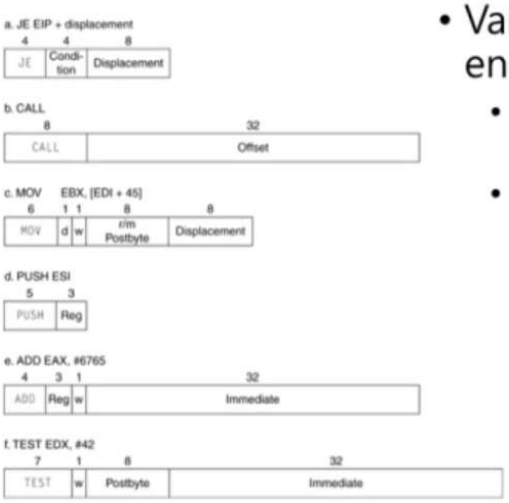
Basic x86 Addressing Modes

- Two operands per instruction

Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

- Memory addressing modes
  - Address in register
  - Address = R_{base} + displacement
  - Address = $R_{base} + 2^{scale} \times R_{index}$ (scale = 0, 1, 2, or 3)
  - Address = $R_{base} + 2^{scale} \times R_{index}$ + displacement

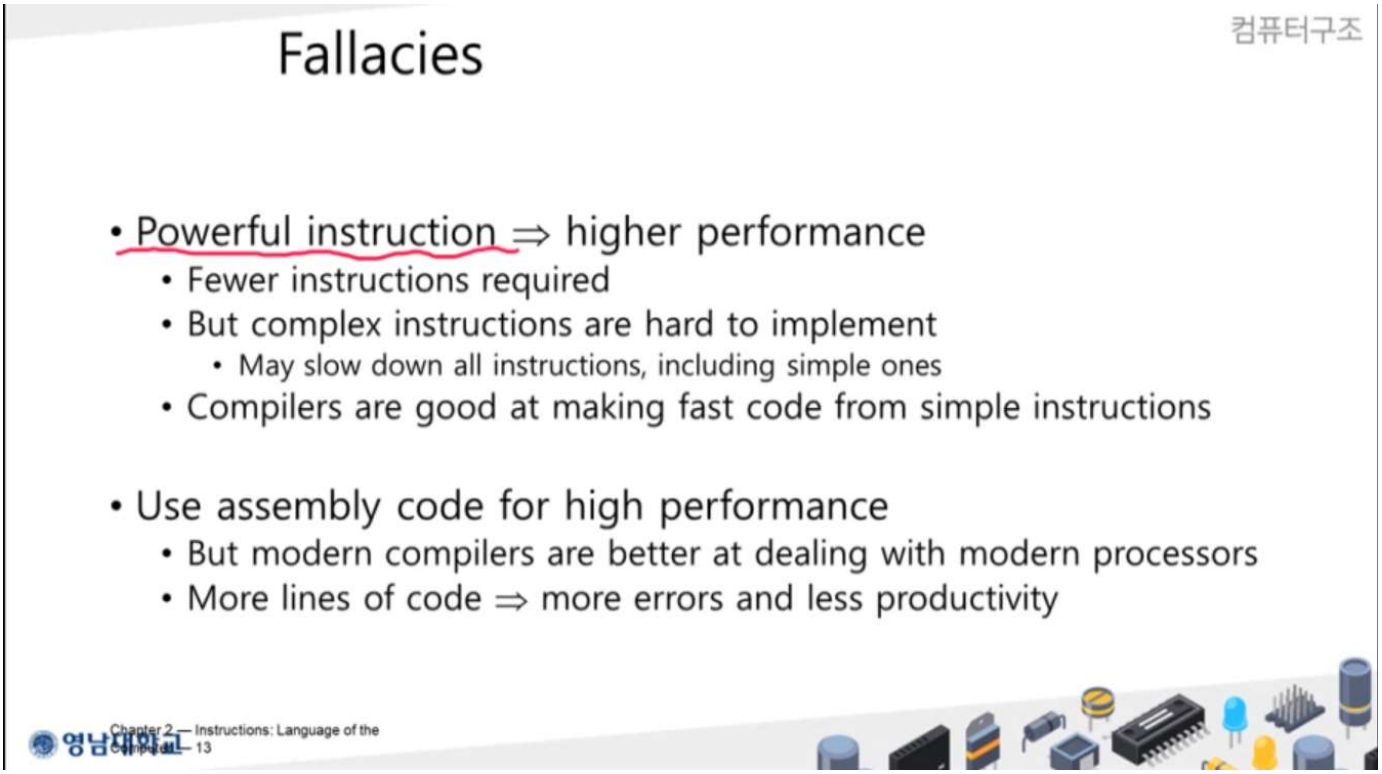
x86 Instruction Encoding



- Variable length encoding
  - Postfix bytes specify addressing mode
  - Prefix bytes modify operation
    - Operand length, repetition, locking, ...

Implementing IA-32

- Complex instruction set makes implementation difficult
  - Hardware translates instructions to simpler microoperations
    - Simple instructions: 1–1
    - Complex instructions: 1–many
  - Microengine similar to RISC
  - Market share makes this economically viable
- Comparable performance to RISC
  - Compilers avoid complex instructions

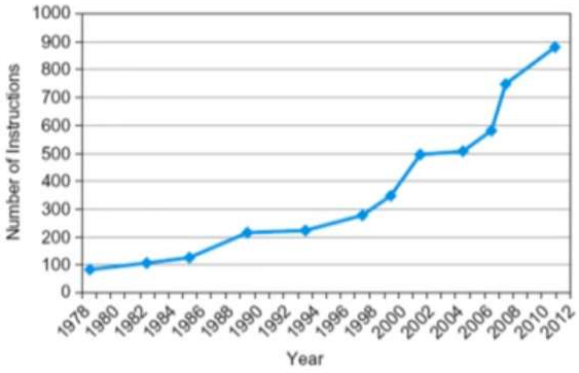


- Powerful instruction : 여러개의 명령어를 수행할 수 있는것. (high performance 예상)
  하지만 그렇지 않다.
  명령어 수를 줄일 수 있다 그렇지만,
  로직이 많아진다 : clack이 느려진다. 모든 명령어들이 느리게 동작한다.

- Assembly 코드를 사용하면 (high performance 예상)
  최적화 가능
  코드의 라인의 수가 많아진다. (에러 발생의 확률이 높아진다.)

Fallacies

- Backward compatibility ⇒ instruction set doesn't change
  - But they do accrete more instructions



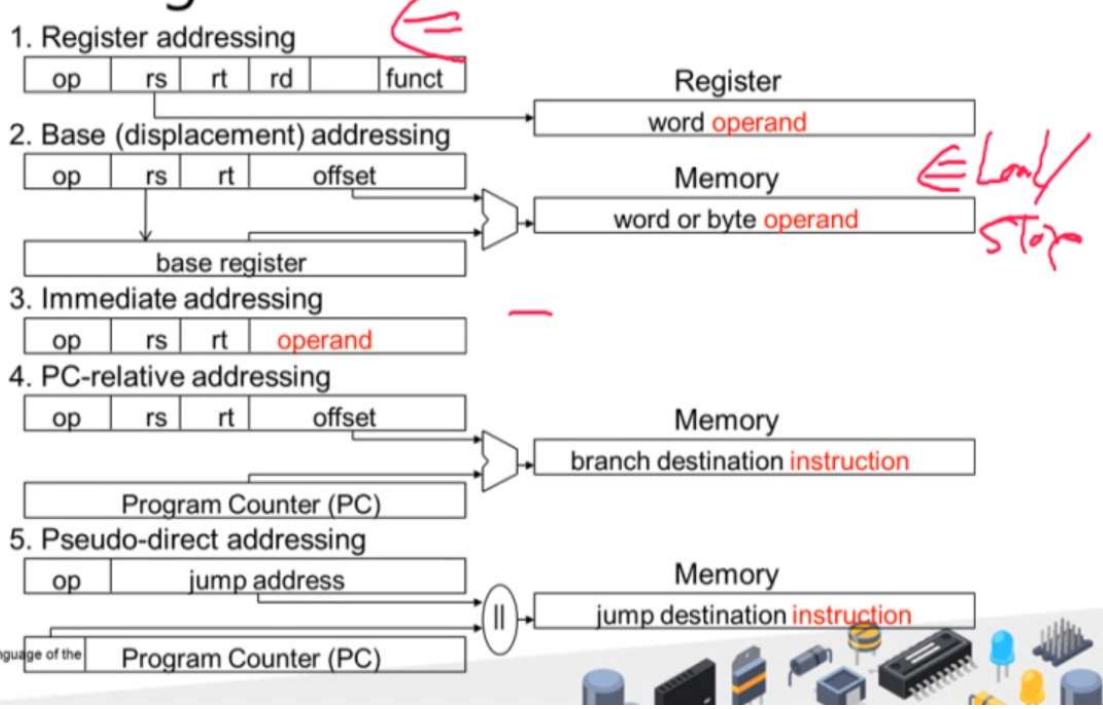
x86 instruction set

- Backward compatibility : 버전이 없그레이 돼도 이전버전의 명령어들을 사용 할 수 있는것.

Pitfalls

- Sequential words are not at sequential addresses
  - Increment by 4, not by 1!
- Keeping a pointer to an automatic variable after procedure returns
  - e.g., passing pointer back via an argument
  - Pointer becomes invalid when stack popped

Addressing Modes Illustrated



Concluding Remarks

- Design principles
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises
- Layers of software/hardware
  - Compiler, assembler, hardware
- MIPS: typical of RISC ISAs
  - c.f. x86

Concluding Remarks

- Measure MIPS instruction executions in benchmark programs
  - Consider making the common case fast
  - Consider compromises

Instruction class	MIPS examples	SPEC2006 Int	SPEC2006 FP
Arithmetic	add, sub, addi	16%	48%
Data transfer	lw, sw, lb, lbu, lh, lhu, sb, lui	35%	36%
Logical	and, or, nor, andi, ori, sll, srl	12%	4%
Cond. Branch	beq, bne, slt, slti, sltiu	34%	8%
Jump	j, jr, jal	2%	0%

퀴 즈

다음 중 design principles이 아닌 것은?

① Simplicity favors regularity

② Bigger is faster

③ Make the common case fast

④ Good design demands good compromises

도로로

필승

이전 포스트
Chapter4. The Processor

0개의 댓글

댓글을 작성하세요

댓글 작성