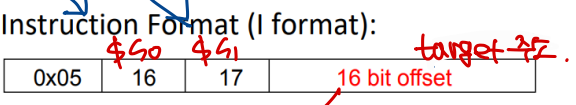
Ch.2-2 Instructions: Language of the Computer

▼ 2.7 판단을 위한 명령어

▼ Conditional branch instructions

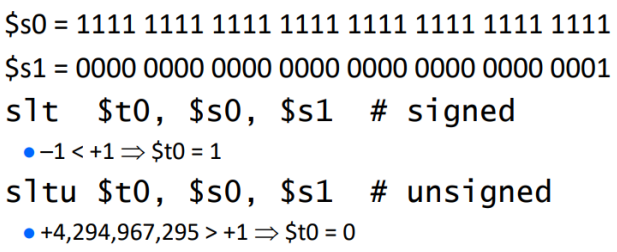
- **branch equal**
  - ***beq*** rs, rt, L1
    - if (rs == rt) → labeled L1로 이동
- **branch nonequal**
  - ***bne*** rs, rt, L1
    - if (rs ≠ rt) → labeled L1로 이동
- ***jump*** → 꼬리에 이어 jump
  - unconditional jump → labeled L1
- example

```
bne $s0, $s1, Lbl #go to Lbl if $s0!=$s1
//Lbl에 가서 bne 실행
beq $s0, $s1, Lbl #go to Lbl if $s0=$s1
```



  - 메모리의 절대적인 주소 공간 → 매번 명령어 address를 loader에게 알려줘야 함 (공간이 부족)
  - 상대적인 주소를 알려줌으로서 해결
    - 앞쪽으로 jump(주소 down), 뒤쪽으로 jump(주소 up)

- **Set on less than** : 두 레지스터의 값을 비교



- ***slt, slti*** : 부호 있는 값에서 사용

```
slt $t0, $s0, $s1 # if $s0 < $s1 then
                  # $t0 = 1 else
                  # $t0 = 0
```

Instruction format (R format):

0	16	17	8		0x24

slt

Alternate versions of `slt`

```
slti $t0, $s0, 25      # if $s0 < 25 then $t0=1 ...
sltu $t0, $s0, $s1     # if $s0 < $s1 then $t0=1 ...
sltiu $t0, $s0, 25     # if $s0 < 25 then $t0=1 ...
```

상수 / unsigned / 상수+unsigned

- ***sltu, sltiu*** : 부호 없는 값에서 사용

- more branch instructions
  - ***slt, beq, bne*** + register $zero → 모든 condition 만들수 있음
    - less than (a<b) → 상수 0을 사용
      - 0보다 크거나 작다로 두 레지스터의 값을 비교할 수 있음

```
slt  $at, $s1,<$s2      #$at set to 1 if
bne  $at, $zero,≠Label  #$s1 < $s2
```

한번에 / 0이 아니면 넘어감

```
①blt $s1, $s2,<Label
```

→ MIPS에서는 비슷한 명령어로 따로 제공하지는 X

    - less than or equal to (a≤b)

```
ble $s1, $s2,≤Label
```

    - greater than (a>b)

```
bgt $s1, $s2,> Label
```

    - grater than or equal to (a≥b)

```
bge $s1, $s2,≥Label
```

⇒ 의사코드처럼 instruction에 포함되어 있음

→ assembler에 의해 인식 : reserved register ($at)가 필요한 이유.

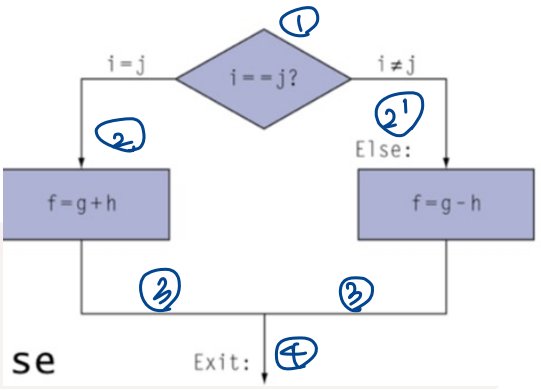
but, 너무 느림, 여러 가지의 branch가 혼합되어 있는 형태

→ 사용하지 않음

⇒ ***bne, beq***가 common case(good design compromise)

slt, bne, beq, zero 조합으로 만들자

▼ Compiling if Statements




```
if (i==j) f = g+h;
else f = g-h;
//f,g ... in $s0, $s1
```

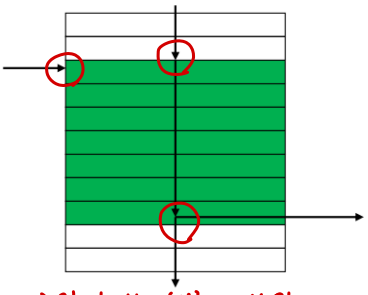
```
      ① bne $s3, $s4, Else  -> s3, s4가 같으면
      ② add $s0, $s1, $s2 -> 더해줌
     ③  j  Exit
② Else: sub $s0, $s1, $s2 -> 다르면 빼줌
④ Exit: ...
//assembler calculates addresses -> 어느 위치로 가야 되는지 알려줌
```

▼ Compiling Loop Statements

```
while (save[i] == k) i += 1;
//      $s6     $s5  $s3
```

```
Loop: sll $t1, $s3, 2 -> address index 증가(4 byte씩 옮겨감) -> shift left
      add $t1, $t1, $s6 -> $s6 + 4i 한 거랑 같음
      lw $t0, 0($t1)
      bne $t0, $s5, Exit 다를 때까지 jump 구문 실행(loop 실행)
      addi $s3, $s3, 1
      j Loop
Exit:
//assembler calculates addresses -> 어느 위치로 가야 되는지 알려줌
```

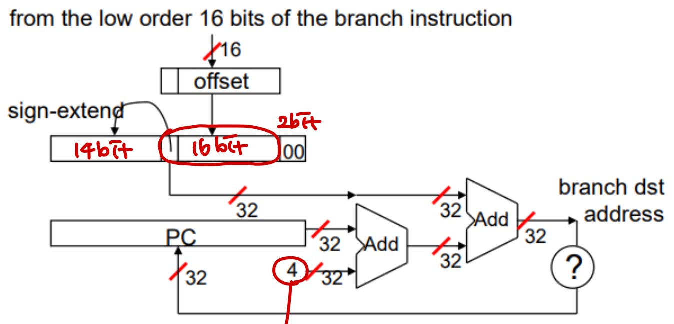
- basic blocks : 명령어 sequence
  - no embedded branches (맨 끝에는 있을 수 있음)
  - no branch targets (맨 앞에는 허용됨)
  - compiler : optimization의 basic block을 identifies
  - advanced processor : basic block의 실행을 가속화



▼ Specifying Branch Destinations

16-bit offset인 register 사용(like in lw and sw)

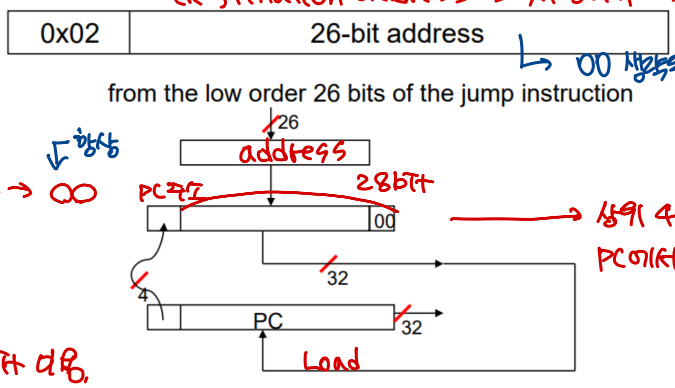
- register : instruction address register (PC) → 현재 실행하고 있는 명령)
    - imstruction에 의해 자동적으로 implied
    - PC : 다음 instruction의 주소를 가지고 있도록 fetch cycle 동안 PC+4로 update
- branch instruction : (-2^15 to 2^15-1) 만큼의 branch distance 한계
    - -2^15 to 2^15-1 ⇒ word address 공간을 표현
    - 하지만 대부분의 branch는 local임



▼ Jump instructions(unconditional)

- *jump* instructions

```
j label #go to label
```



▼ aside : branching far away

16bit보다 더 넘는 branch destination이 있다면?(jump한 것이 더 클 경우)

→ assembler : comes to the rescue

- branch target으로 unconditional jump를 insert + inverts the condition

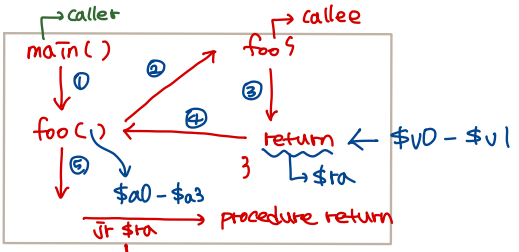
```
beq $s0, $s1, L1

//두 가지를 함께 씀 -> jump
```

```
bne $s0,$s1, L2
j L1
L2 :
```

▼ 2.8 hardware의 procedure 지원

▼ Six steps in Procedure Calling

1. Main routine(caller) : procedure(callee)가 접근할 수 있는 곳에 parameter 할당
   a. $a0 - $a3 : 4개의 argument registers
2. caller : callee에게 제어권을 줌 → PC를 넘겨 줌
3. callee : 필요로 하는 메모리 자원 획득
4. callee : 필요한 작업 수행
5. callee : caller가 접근할 수 있는 곳에 결과값을 넣음
   a. callee : 자기가 썼던 operation들을 거꾸로 하고 돌아감
   b. $v0 - $v1 : 결과 값에 대한 두 개의 return register
6. callee : caller가 제어권을 다시 돌려줌
   a. $ra : 호출한 곳으로 되돌아가기 위한 return address를 가지고 있는 register



▼ Register Usage

Name	Register NO	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

▼ Instructions for Procedure Call

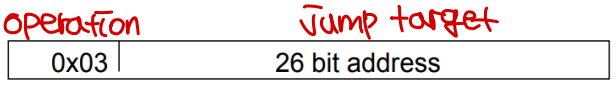
- **MIPS** procedure call instruction

```
jal ProcedureAddress #jump and link
```

  - ***jump*** : 지정된 주소로 jump
  - ***link*** : 다음 명령어의 주소를 $ra에 저장(PC+4) ⇒ return address
    - procedure 종료 후 올바른 주소로 되돌아 올 수 있도록 호출한 곳과 procedure 사이에 address or link 생성

⇒ ***jal*** (jump + link 동시에)

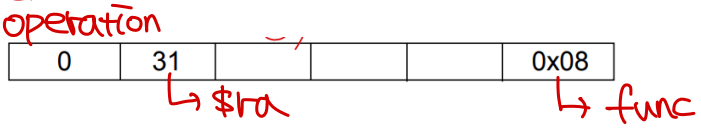
- machine format (J format)



- MIPS procedure return instruction

```
jr $ra #return
```

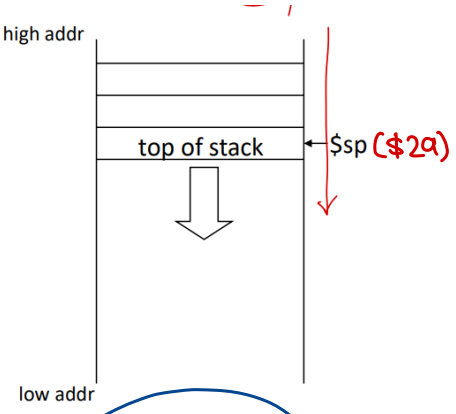
  - register($ra)에 저장된 주소로 무조건 점프 → case, switch 구문에서 많이 사용
  - instruction format(R format)



▼ Procedure Example

aside)

- Spilling Registers
  - 1) 컴퓨터가 갖고 있는 register보다 program에서 사용하는 더 많은 경우
    - 자주 사용되는 변수를 가능한 많이 넣고 사용하지 않는 건 메모리에 저장
  - 2) callee가 할당된 register보다 더 많은 것을 필요로 한다면?
    - stack 사용!
  - ex)

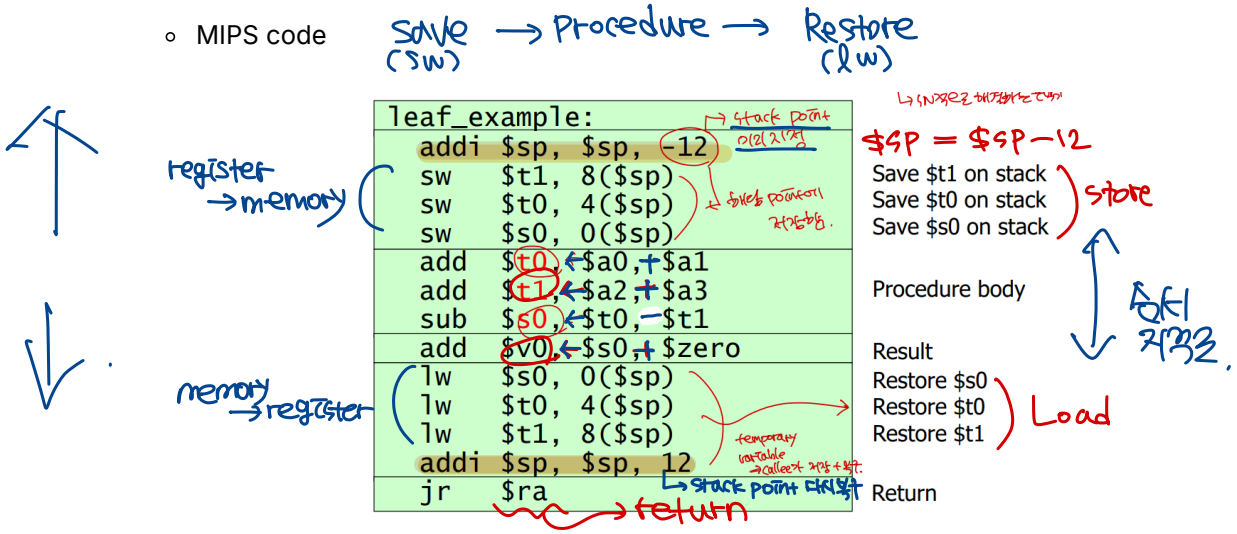


    - $29 → stack address에 사용됨(high에서 low로 'grow')
    - push : $sp = $sp -4 ⇒ address down
    - pop : $sp = $sp + 4 ⇒ address up

- Leaf Procedure Example → 다른 procedure 호출하지 않음.
  - c code

```
int leaf_example (int g, h, i, j){ //$a0, $a1, $a2, $a3
   int f;
   f = (g+h) - (i+j); //$s0, $t0, $t1
   return f; //result -> $v0
}
```

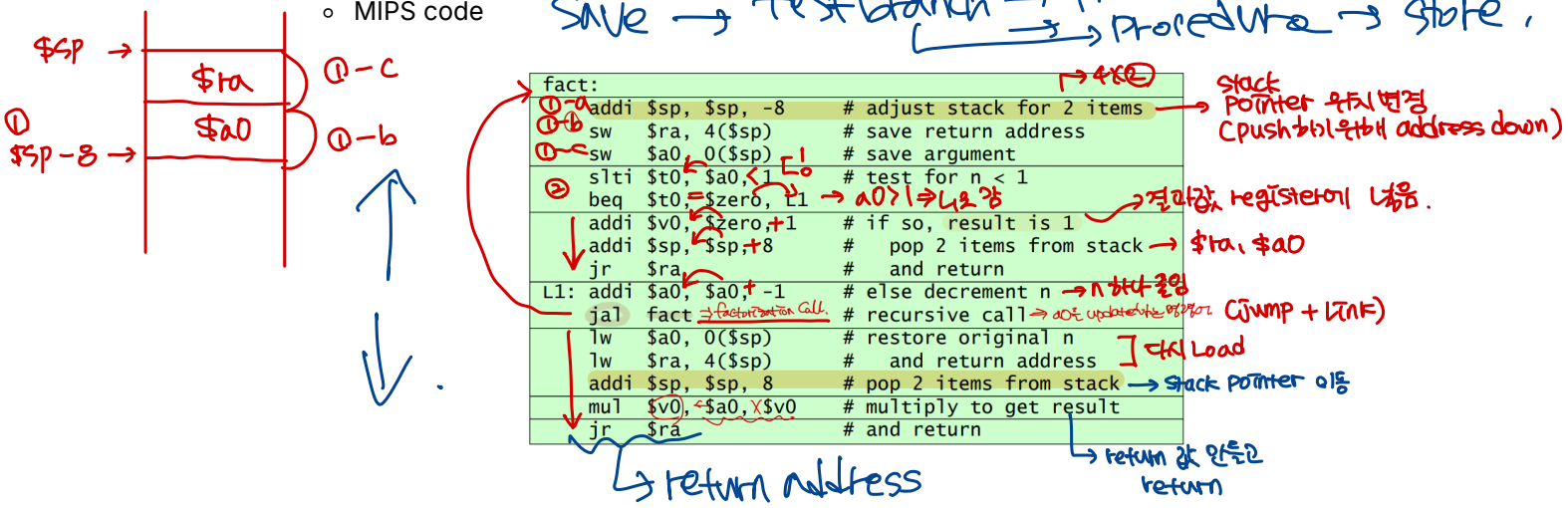
- MIPS code



- Non-leaf Procedure Example → 다른 procedure 호출
  - 중첩 함수 → caller는 stack에 return address, argument 등을 저장해야 함
  - restore from the stack after the call
  - c code → 자기자신함수

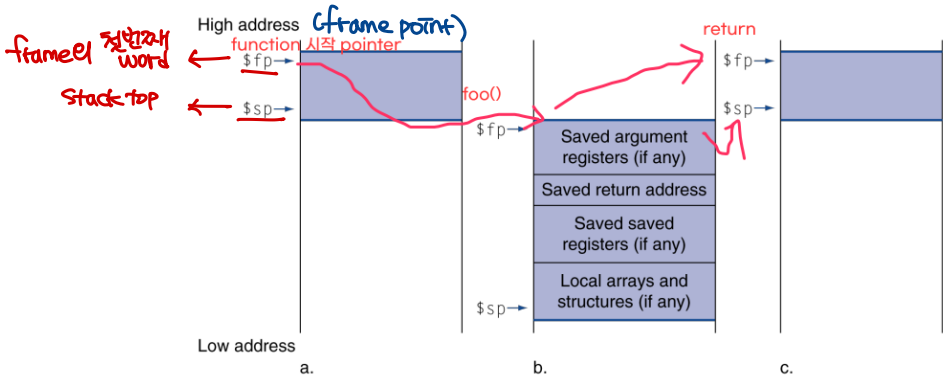
```
int fact (int n){ //n -> $a0
    if (n < 1) return(1);
    else return n * fact(n - 1); //result -> $v0
}
```

- MIPS code



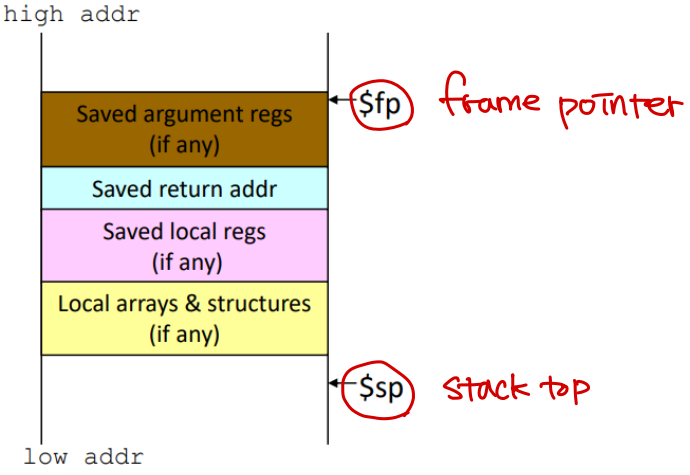
▼ 새 데이터를 위한 스택 공간의 할당

*레지스터에 들어가지 못할만큼 큰 배열 OR 구조체 같은 local variable 저장 → stack.
⇒ Procedure에 해당 영역 fp. (activation record)



return 직전에 sp, fp를 원위치로 돌려놓고 return.

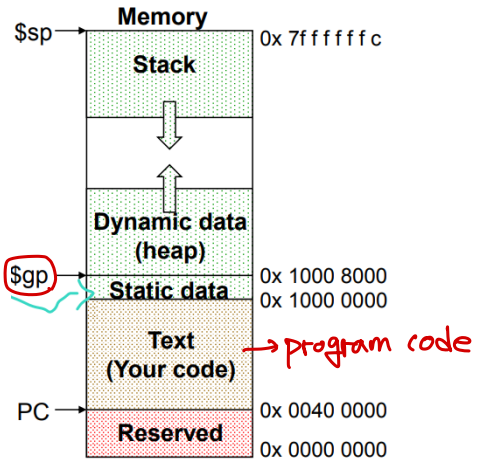
- local data allocated by callee
  - C 자동 변수
- procedure frame (activation record)



  - procedure에 저장된 레지스터와 지역 변수 가지고 있는 스택 영역
  - stack storage 관리하는 몇몇 compiler에 의해 사용됨 (frame 첫번째 word 표시)
  - $fp(frame pointer) : procedure의 저장된 register와 지역 변수 위치 표시
    - $sp에 의해 호출될 때 초기화 → $sp가 return될 $fp에 restore

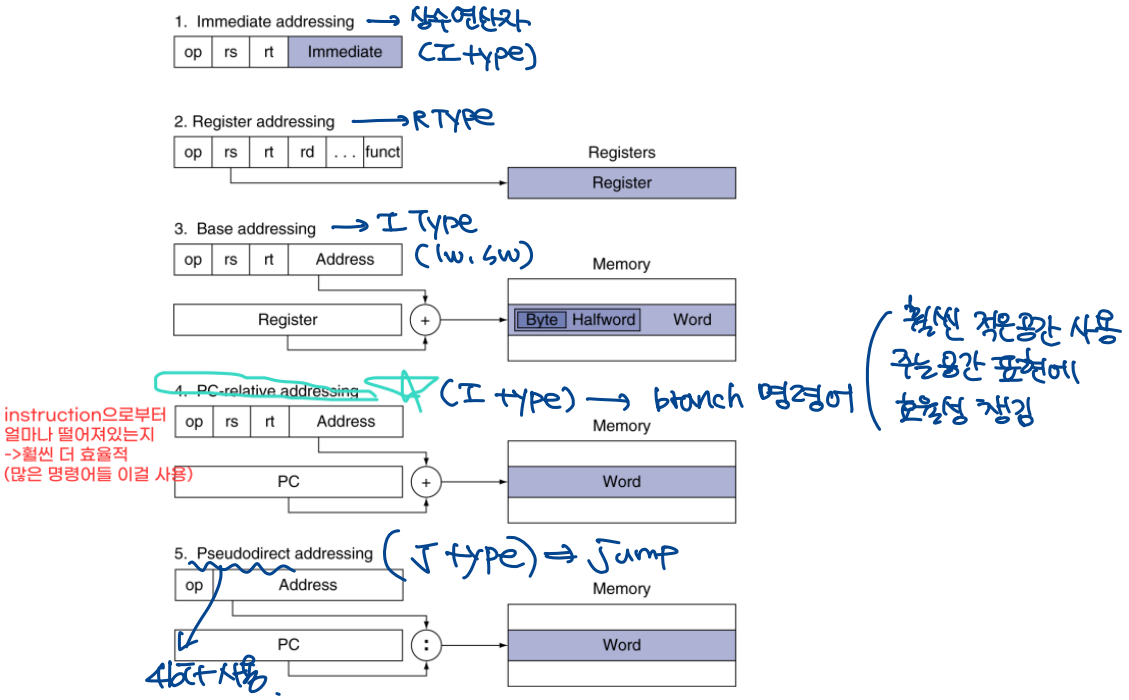
▼ 새 데이터를 위한 힙 공간의 할당

procedure 내에서만 사용되는 변수들 이외에도 메모리 공간이 필요



- text : program code
- static data segment : global variables
  - constants, other static variables (arrays) in C
  - $gp : 데이터에 쉽게 접근할 수 있도록 주소 초기화 해줌 (전역포인터)
- Dynamic data segment(aka heap)
  - 늘어났다 줄었다 하는 자료구조(linked list)
  - malloc(), free()로 공간 할당
- Stack

▼ Address mode summary



▼ 2.11 병렬성과 명령어 : 동기화

▼ Synchronization

- data race의 위험을 막기 위해 synchoronization 필요할 때가 있음
  - program의 결과가 events가 어떻게 일어났는지에 따라 바뀔 수 있을 때
- data race : 다른 thread가 같은 위치에 있는 메모리에 접근할 때
  - p1: write, p2: read ⇒ synchoronize 실패 → 순서에 따라 결과 달라짐

⇒ hw의 support가 필요함

1. atomic r/w memory operation
2. location에 대한 다른 접근이 r/w 사이에 허락되지 x

- Atomic Exchange (atomic swap)
  - synchronization 연산 구축을 위한 연산 방법
  - 레지스터의 값을 메모리 값과 서로 맞바꿈 ⇒ 교환 명령어 실행
    - 두 processor가 동시에 exchange 시도하더라도 누군가 먼저 수행한 processor가 교환 전의 값을 읽게 됨 ⇒ 경쟁 사라짐 ⇒ single operation
  - 교환 명령어 2개 실행 → 어느 processor에서도 이 명령어 쌍 사이에 수행 x
    - load linked : ll rt, offset(rs) ⇒ ll 이후에 해당 위치에 있는 값이 변하지 않은 경우 성공, 변한 경우 → 실패 (다시 0을 return)
    - store conditional : sc rt, offset(rs) (다시 0을 return)
      1. succeed → 위치가 11에서 바뀌지 x → return 1
      2. falls → 위치 바뀜 → return 0
    - ll 명령어에 의해 명시된 메모리 주소의 내용이 같은 주소에 대한 sc 명령어가 실행되기 전에 바뀐다면 sc는 실패

```
try: add $t0, $zero, $s4 #$t0=$s4 (exchange value)
     ll $t1, 0($s1) #load memory value to $t1
     sc $t0, 0($s1) #try to store exchange
     #value to memory, if fail
     #$t0 will be 0
     beq $t0, $zero, try #try again if store fails
     add $s4, $zero, $t1 #load value in $s4
```

    - ll 명령어와 sc 명령어 사이에 어떤 processor가 끼어들어서 메모리 값 수정
      - sc: failled → return 0 (to $t0)
      - 코드 시퀀스를 다시 실행

▼ 2.12 프로그램 번역과 실행

① assembler ▼ Assembler Pseudoinstructions

- 대부분의 assembler instructions : machine instructions one-to-one
- hw가 지원하지 않지만 assembler가 처리
  - 마치 실제 instruction처럼 사용되는 언어의 변형 ⇒ assembler의 imagination
- example 1

- MIPS hw : $zero → 항상 0으로 사용, 바꿀 수 없음
- MIPS hw : move 명령어 없음

```
move $t0, $t1    →  add $t0, $zero, $t1
```

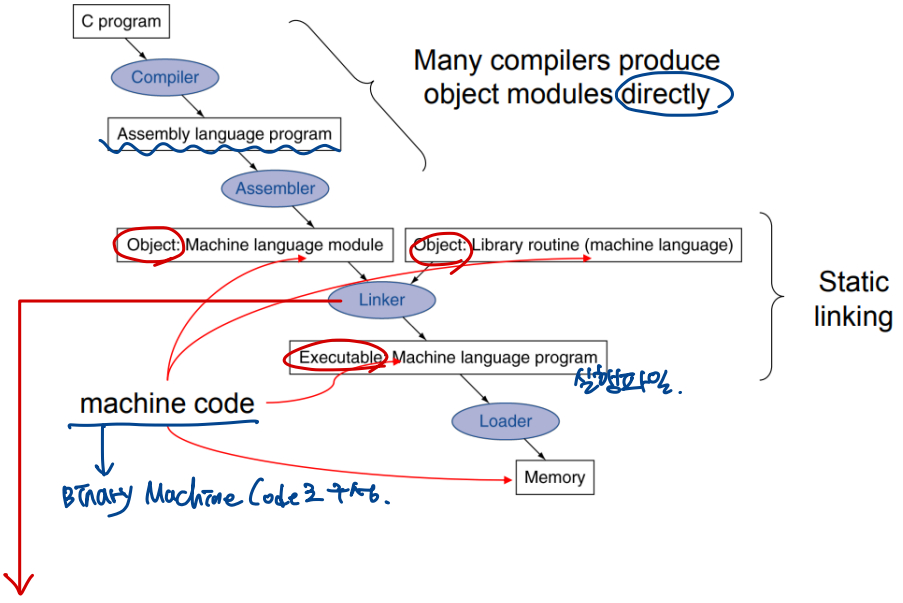
- MIPS assembler : move 명령어 받아들일 수 있음 → add로 사용해서
  - 해당 명령어를 기계 명령어로 바꾸는 작업 실행

- example 2

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

- $at (register 1): assembler temporary

▼ translation and startup



② Linker

▼ Linking object modules

- produces an executable image
  1. merges segments → code, data module을 symbol 형태로 저장
  2. resolve labels → data, code lable 주소 결정
  3. patch location-dependent and external refs → 외부 및 내부 참조 해결

  ⇒ linker : 프로그램 전체 컴파일, 어셈블 대신 번역된 모듈을 해결하면 편리함 (각 Procedure 따로 컴파일)

  ⇒ execute program : linker가 생성하는 파일(object file과 같은 형식)
- linker
  - relocating loader의 fixing을 위해 location dependencies 남겨 놓을 수 있음 (각 module의 relocated 정보, symbol table)
    - virtual memory : 굳이 남겨놓을 필요는없음

- program : virtual memory에 절대적인 위치를 located 가능

③ Loader

▼ Loading a program ⇒ 프로그램을 실행하는 과정.

- load from image file on disk into memory
  1. execute file read → text, data segment size 알아냄
  2. text, data 들어갈 만한 address space 확보
  3. copy text(instruction), and initialized data into memory
     a. or set page table entries
  4. stack에 전달해야 할 parameter 복사
  5. register 초기화, stack pointer → 사용 가능한 첫 주소 ($sp, $fp, $gp)
  6. start-up routine으로 점프
     a. argument → argument register에 넣음
     b. main이 return 될 때 exit system call 사용

▼ compiler benefits → 프로그램 성능

- 버블정렬과 compiler의 성능 비교
  - 10만개의 word를 다음과 같은 성능과 환경에서 랜덤하게 초기화

▶ To sort 100,000 words with the array initialized to random values on a Pentium 4 with a 3.06 clock rate, a 533 MHz system bus, with 2 GB of DDR SDRAM, using Linux version 2.4.20

gcc opt	Relative Performance	Clock cycles (M)	Instr count (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 integ)	2.41	65,747	44,993	1.46

optimization Level 설정가능.

→ none(unoptimized code) ⇒ best CPI

→ O1 version : 가장 낮은 instruction count

→ O3 version : 가장 빠른 version

⇒ 시간이 가장 direct한 성능 평가 지표.

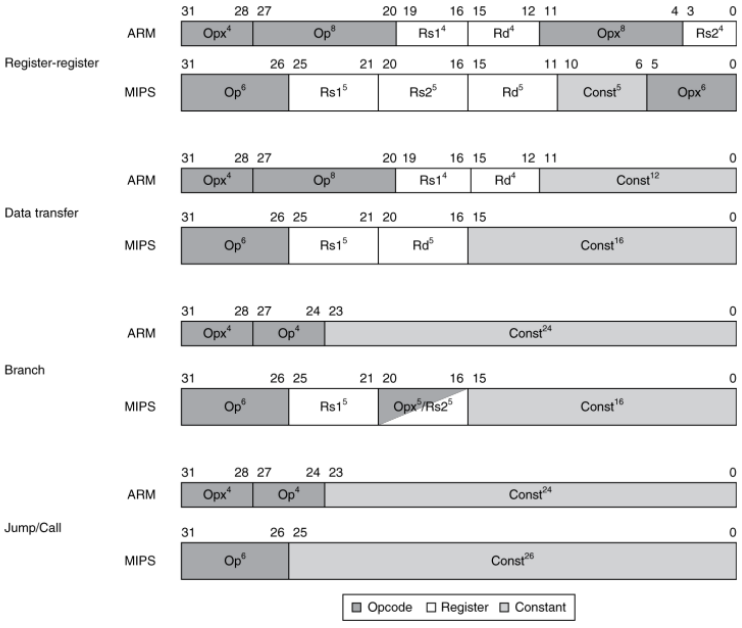
▼ 2.16 실례 : ARM

- ARM : 임베디드용으로는 가장 인기 있는 명령어 집합 구조
  - MIPS와 같은 설계 철학을 다르고 있음

	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

- 비교 및 conditional branch
  - MIPS : conditional branch를 판단하기 위해 register 값 사용
  - ARM : program state word에 저장되는 4개의 전통적인 조건 코드 사용
    - negative, zero, carry, overflow bit 사용
    - result를 저장하지 않고 instruction 비교하여 condition code 설정
    - 각 명령어는 조건부가 될 수 있다
      - 4비트 조건부 실행 필드를 가지고 있음
      - 명령어 하나를 뛰어넘기 위해 branch를 사용하는 일 방지

- ARM, MIPS, RISC-V instruction encoding



▼ 2.17 실례 : ARMv8(64bit 명령어)

In moving to 64-bit, ARM did a complete overhaul

ARM v8 resembles MIPS

- Changes from v7:
  - No conditional execution field
  - Immediate field is 12-bit constant
  - Dropped load/store multiple
  - PC is no longer a GPR
  - GPR set expanded to 32
  - Addressing modes work for all word sizes
  - Divide instruction
  - Branch if equal/branch if not equal instructions

▼ 2.19 실례 : x86 ISA

▼ intel x86 ISA

- 발전

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

- 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

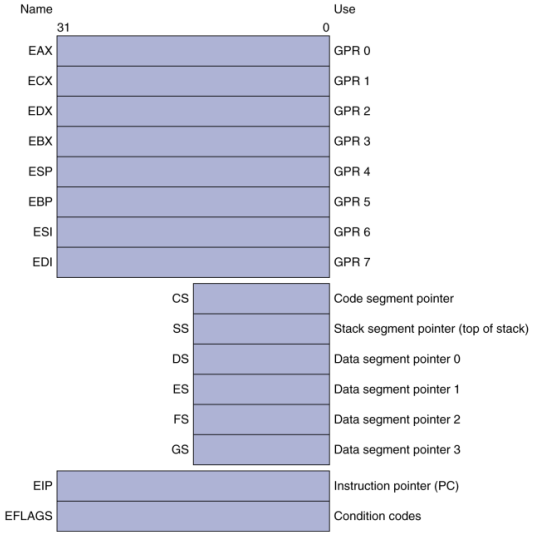
And further…

- 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

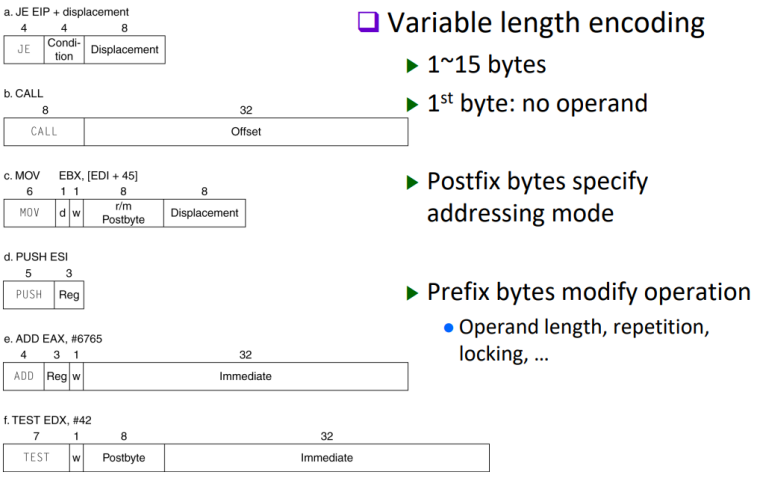
둘이 같음

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



▼ implementing IA-32

- CISC → implementation 어렵게 만들었음
- RISC와 비교하였을 때 compiler가 복잡한 instruction 너무 많이 수행해야 함

→ 스타일의 약점을 양으로 승부하고 있었던 것!

▼ 2.21 오류 및 함정

▼ Fallacies : 기능이 좋은 명령어 → 성능도 좋다?

- 기능이 좋다 → 명령어를 적게 사용해도 됨 ⇒ CISC : 한 번에 처리
  - but, 복잡한 명령어 → implement 어려움 (메모리 접근 불편함)
    - 단순한 명령어까지 모두 느려질 수 있다

  ⇒ complier : fast code 만들기 위해 단순한 명령어 만듦.
- 과거 : 높은 성능을 가지기 위해 assembly code를 쓰는게 좋았음.
  - 현재의 complier : modern processor 다루는게 훨씬 좋음
  - assembley code가 많으면 error, 생산성 낮음

▼ Fallacies : backward compatiablility(호환성) → 명령어 집합 변하지 x

- 과거의 sw에 쓰던 instruction 계속 지원 → 계속 증가, 명령어 집합 계속 변함
  - 더 많은 명령어를 만들 뿐임.

▼ Pitfalls : byte 주소 사용할 때 주소 차이?

- word 단위 일 때 → 4씩 증가, byte 단위일 때 → 1씩 증가

▼ Pitfalls : 자동 variable이 정의된 procedure 외부에서 해당 변수 포인터 사용

- pointer : stack이 pop될 때 invaild 됨.
- example) argument를 통해 pointer 돌려 받음