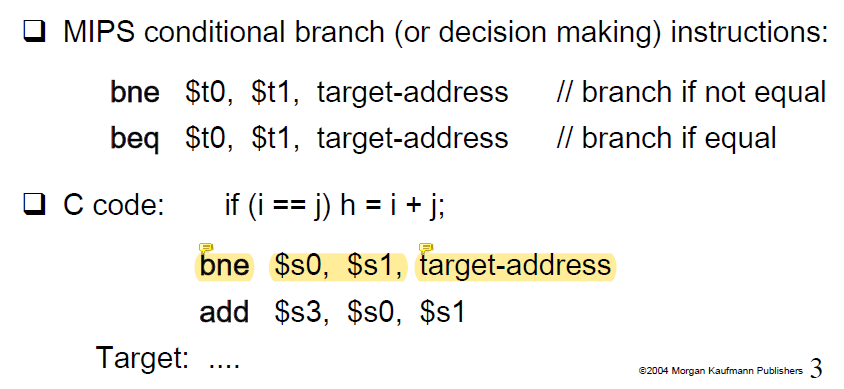
HW8 2017069598 박상지

control instructions

-conditional branch



MIPS에서 조건부 branch instruction은 bne, beq 두 가지의 instruction이 존재한다. bne, beq 둘 다 target-address를 사용하는데 주소는 항상 32-bit으로 사용하기 때문에 적은 bit으로 target-address를 표현을 해야 한다. 두 가지의 branch instruction은 현재 위치에서 얼만큼 jump를 할 것인지 표현하는 mode로 이것을 PC-relative addressing mode라고 한다. 두 가지 instruction은 상수 offset이 필요하기 때문에 I-format을 사용한다. offset에서는 address를 증가, 감소하는 방향을 jump를 할 경우가 있기 때문에 2의 보수를 사용한다.

\*PC(destination) <- current PC + offset \* 4

질문) I-format 에서는 offset 부분에서 16bit를 사용하는데 conditional branch에서 주소 값을 branch에 사용하는데 충분한가?

if문 컴파일할 때 jump bit은 얼마 들지 않는다. 벤치 마킹에서 jump distance bit을 이미 계산을 많이 했기 때문에 16-bit으로 충분하다.

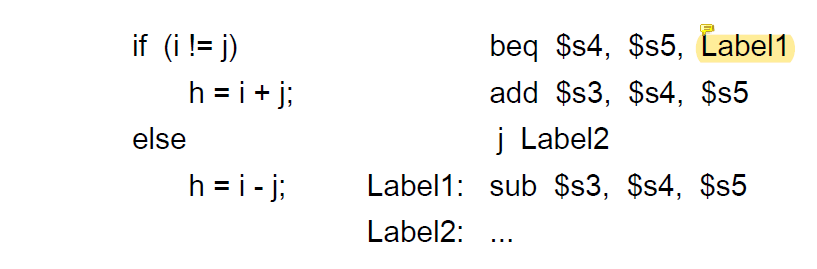
질문) branch에서 사용하는 16-bit offset은 byte 단위인가?

보통 branch offset에서는 byte단위로 설정하지 않는다. word offset을 사용하는데 byte offset을 사용할 경우 하나의 instruction을 사용하는데 1word 즉 4byte로 사용하는데 그러면 bit에서 사용할 때 0과 2byte는 쓰지 않는 bit가 된다. 프로그래머들은 이런 낭비하는 bit를 선호하지 않기 때문에 word offset을 사용한다. 그러면 낭비했던 bit까지 사용하는 것이기 때문에 18-bit를 쓰는 효과가 일어난다.

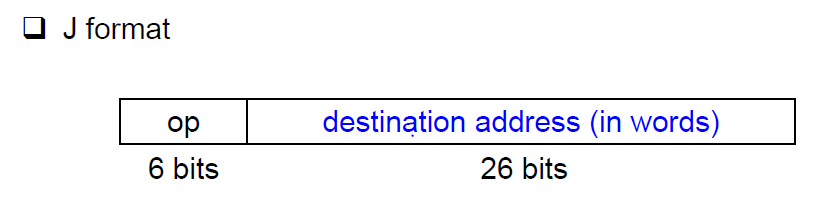
질문) 그러면 offset을 사용하지 않고 R-format처럼 레지스터안에 offset을 저장하여 사용하지 않는가? (ex. beq r1, r2,r3)

RISC instruction에서는 branch에서 R-format을 사용하지 않는다. 만약 offset 값을 레지스터에 저장을 한다면 레지스터에 값을 넣는 instruction을 사용하기 때문에 결과적으로 2개 이상의 instruction을 사용하는 꼴이다. 그렇다면 성능에 대한 치명적인 단점이 생긴다.

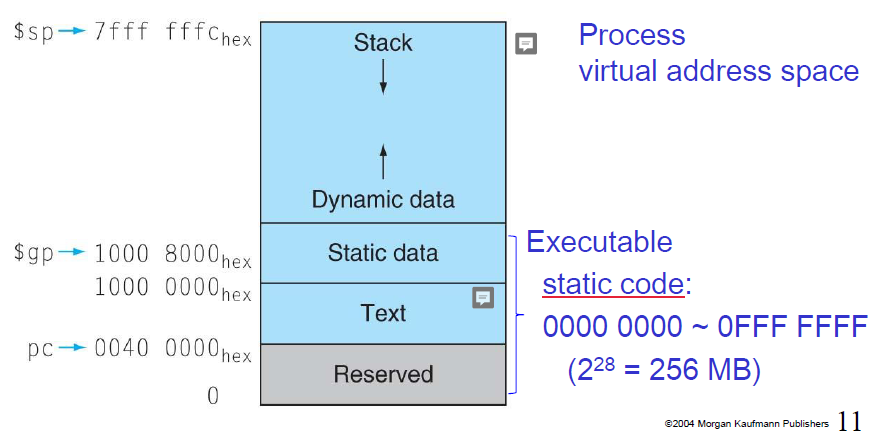
-unconditional branch



unconditional branch에서는 따로 offset값이 존재하지 않기 때문에 새로운 j target-address를 사용한다. 위에 C코드처럼 i와 j가 다를 경우와 같을 경우를 나누어지는 연산을 할 때에는 unconditional하다. (점프 target이 여러 개이기 때문에)



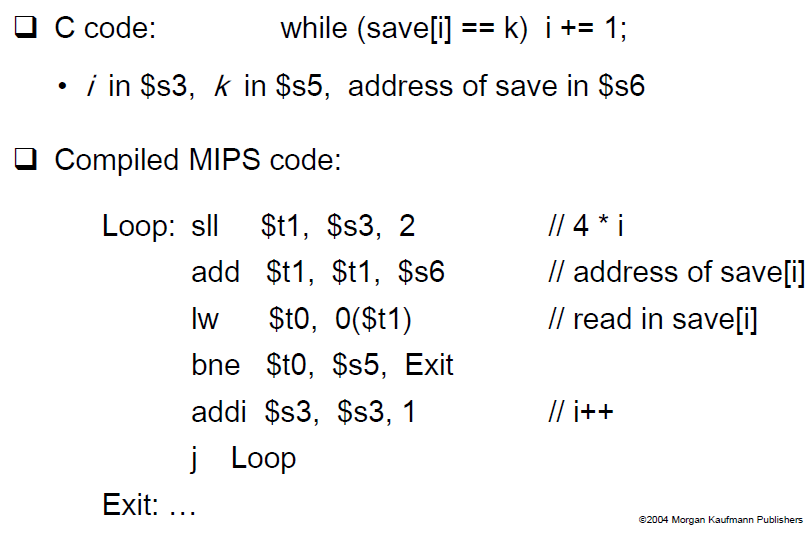
질문) 주소에서는 32-bit가 필요하다. 만약 J-format을 사용하면 op에서 6bits와 나머지 destination address에서 26bits를 사용한다. 26bit를 word로 사용하기 때문에 28bit를 확보한 상태이다. 그러면 나머지 4bit는 어떻게 처리하는가?

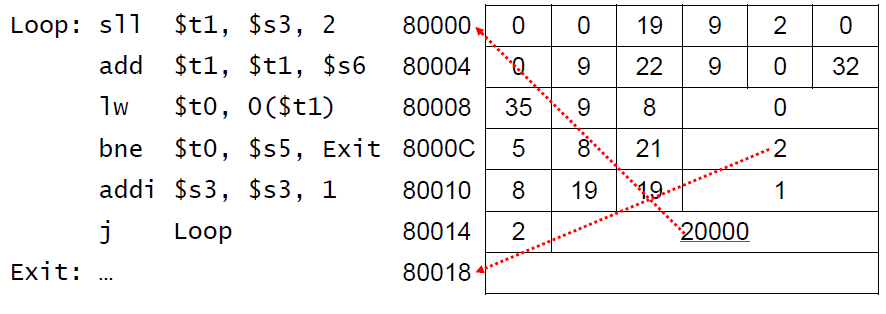


그림을 보면 process virtual address space를 나타내는 부분에서 아랫부분에 executable에서 클릭해서 실행(run)을 시키면 OS용어로 process가 만들어 진다. OS는 추가적인 공간을 만들어서 process virtual address space를 만들어 주고 이것을 이용해서 프로그램을 실행시킨다.

reserved 공간에서 4bit을 사용하는데 이 부분에서 사용하지 않는 4bit이기 때문에 나머지 text 부분에서 28-bit만 텍스트에 있는 어디든 function을 표현할 수 있다.

more on beq, bne, j

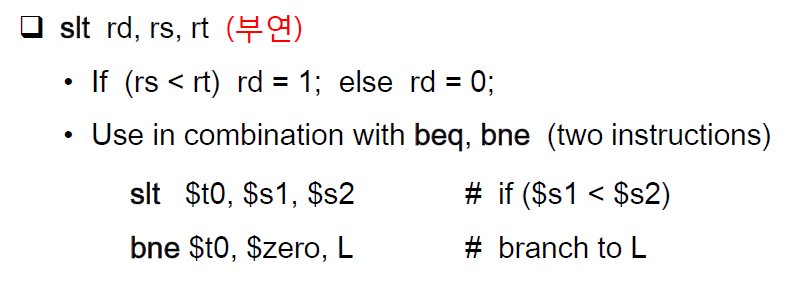




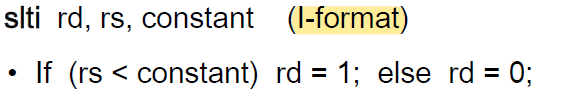
처음에 i는 save 변수의 주소를 가리키는 것이기 때문에 i를 나타내는 레지스터 s3을 shift left logic 연산을 사용하여 2bit를 shift하면 t1에는 4\*I 값이 들어가게 된다. t1과 s6 레지스터를 더하면 save[i] 배열 값의 생기게 되고 bne의 연산으로 t0과 s5가 다르면 loop에 나가게 되고, 같을 경우 다음 연산을 실행시킨다(loop). 위에 그림을 보면 80014에서 j-target에서 offset 값을 20000이라 주어져 있는데 해당 offset은 word단위 이기 때문에 80000byte꼴이다. 즉 다시 loop연산으로 jump하게 된다.

그림을 보면 1번째 instruction이 실행되면 4개의 instruction이 연속적으로 이루어진다. 이것을 basic block이라 부르며, 이 block에는 다른 instruction이 중간 개입을 할 수 없고 도중에 instruction이 빠져나가지도 못한다.

-comparison for branch



set less than: rs가 rt보다 작을 경우 rd에는 1의 true값이 저장되어 지고, 큰 경우에는 rd에는 0이라는 false 값이 저장된다. Slt에서는 대개 beq, bne와 많이 사용되어진다.

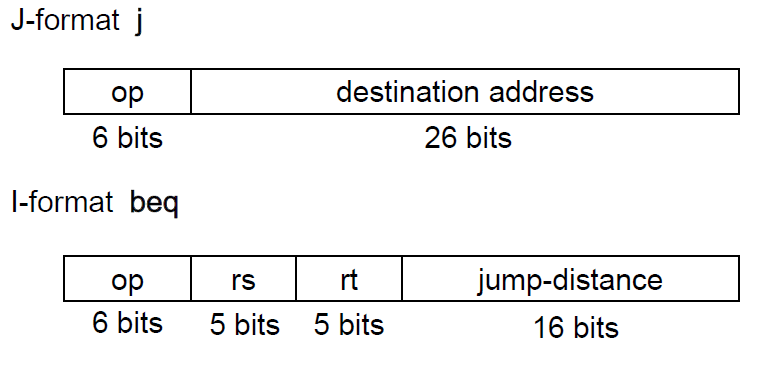


그렇다면 비교 연산으로 상수 값이 쓰이면 I-format으로 immediate 값을 사용하여 비교를 한다.

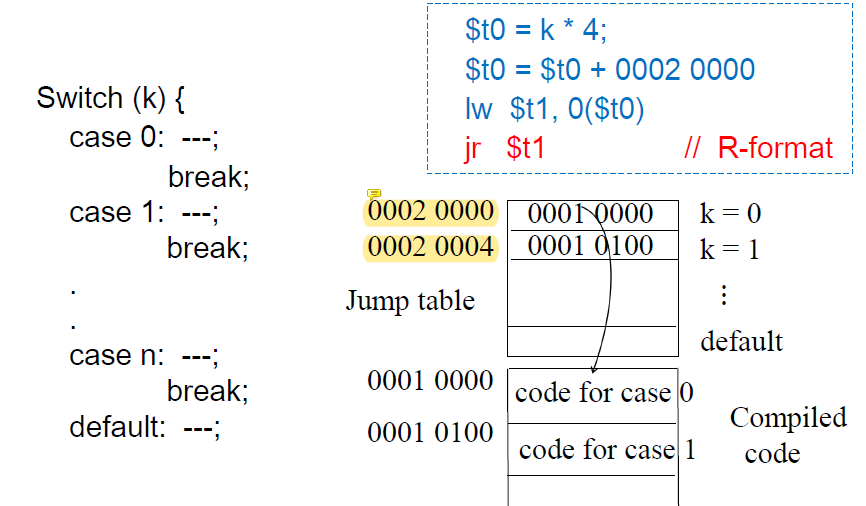
질문) 비교연산을 따로 만들지 않고 branch 자체에서 비교연산을 만들면 되지 않는가? ( ex. blt s1, s2, label)

equal, not equal 연산을 비교하는 것보다 less, more 비교 연산이 시간이 더 걸린다. 같다는 것을 비교하는 것은 bit by bit으로 병렬로 비교할 수가 있다. less than에서는 두 수를 빼고 더하기 연산을 sequence 하게 이루어진다. branch less than을 instruction을 하나로 만들어서 하나로 돌리게 되면 clock cycle time이 늘어나게 된다.

unconditional branch : jr(jump register) as well as j (jump)



컴파일러가 어디로 jump를 하는지 모를 때에 정확한 address를 모르기 때문에 jump register을 사용한다.



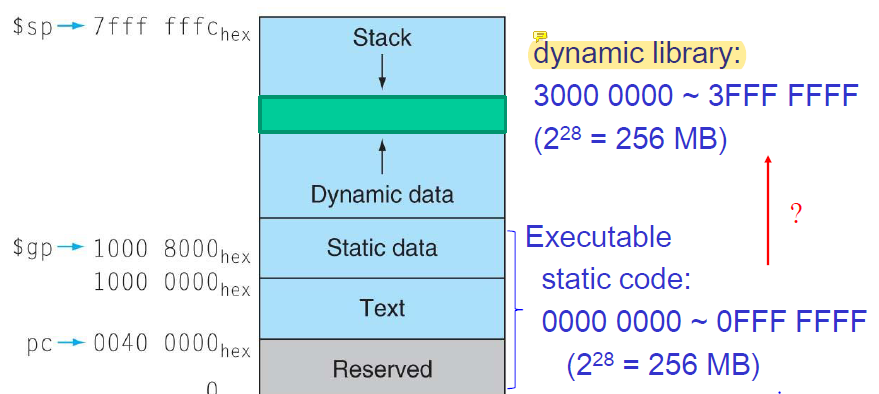
switch문을 예를 들면 각각의 case마다 실행시키는 코드가 다르다. 이럴 경우 해당 작업마다 다른 코드로 jump를 해야 한다. 이럴 경우 작업이 이루어지는 코드의 주소를 jump table로 따로 모아 놓아서 해당 table에 주소 값을 더해서 jump를 시켜준다.

질문) switch문을 사용하지 않고 if else문을 사용하지 않는가?

case가 많다면 코드가 엄청 느려진다. 하지만 switch문을 사용한다면 case가 여러 개가 있을 경우 속도가 향상된다.

\*indirection: 코드를 직접 컴파일하기 힘들 경우 jump table에 거쳐서 해당 코드로 넘어가는 것을 indirection이라 한다(약간 성능이 하향된다.)

\*OBJECT.print\_type(): function call이 일어나는데 run time마다 어떤 function call이 일어나는지 결정된다. 그래서 run time binding이 필요할 때는 jump register instruction이 일어난다.

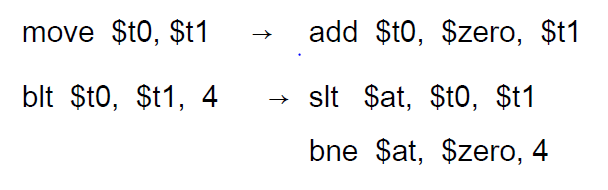


\*dynamic library: 프로그램이 실행하는 도중에 dynamic library가 들어오게 되는데 따라서 다이나믹 라이브러리에 들어있는 function은 그 해당 주소를 직접 줄 수는 없고 run time으로 어디다 binding되는지 위치를 확인해서 register에 저장해서 jump를 시켜준다. ->jump register

-procedure call and return

\*jr r32: caller는 return address를 r31에 저장한다.

\*link: return address register



\*pseudoinstructions(가상적인 머신 명령어): 가짜 instruction (어셈블러에서 사용함) move라는 instruction을 자주 사용하기 때문에 어셈블러 개발자의 편리함을 위해서 move연산을 따로 만들었지만 실제로는 zero 레지스터를 사용하여 add를 해준다.