2017069598 박상지

1.

1)해당 명령어는 먼저 r1 레지스터를 읽어 와야 하고 r1이 static variable이면 데이터를 global point에서 얼만큼 떨어져 있는지 load instruction으로 읽어서 레지스터에 저장하고 해당 데이터에서 8만큼 떨어져 있는 것을 offset을 이용해서 읽어와야 하기 때문에 해당 instruction은 단일 명령어로 처리할 수 없다

2) local variable은 레지스터에 저장된 값을 상수 값을 주어서 가져오면 되기 때문에 단일 명령어로 처리할 수 있다.

2.

1) c프로그램에서 if문 같은 경우 비교를 하여 참이면 해당 if문을 작동하고 참이 아니면 if문 밖에 있는 연산으로 넘어간다. 그러므로 beq는 두 레지스터를 비교해야 하기 때문에 branch instruction이 필요하다.

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

3) jr은 c프로그램에서 다른 함수나 switch 문 등에서 사용할 때 다음 연산이 어디인 줄 모르니 branch instruction을 이용하여야 한다.

4) jal은 c프로그램에서 똑같이 return을 연결시켜줄 때 branch instruction을 이용해야한다.

5) jalr도 점프에 쓰이기 때문에 branch instruction을 이용해야 한다.

3.

int f (int a, int b, int c, int d) { return func(func(a, b), c+d); }

f :

addi sp, sp, -20

sw ra , 16(sp)

sw a0, 12(sp)

sw a1, 8(sp)

sw a2, 4(sp)

sw a3, 0(sp)

jal func

move a0 v0

add a1 c d

lw a1 4(sp)

jal func

lw ra 0(sp)

addi sp sp 20

jr ra

4.

1) 2의보수에서overflow가 나는 것은 전반적으로 프로그래머 잘못이라 여기면 충분한 크기의 데이터타입을 사용해야 한다. 'addu', 'addiu', 'subu', 와 같은 instruction은 overflow를 무시하기 위해서 만들었다. 여기서u는 unsigned를 가리키는 것이 아니라 2의보수에대한 오버프로우를 무시하라는 의미이다.

2) 1980년도에 제작이 된 표준 표현법은, 컴퓨터가 처음으로 네트워킹이 되던 시절에 floating point를 표현 방식을 컴퓨터 회사간에 독자적으로 표현을 하였기 때문에 호환성이 떨어졌다. 그래서 floating point standard가 생겼다. 그리고 754 standard는 1985년에 두개의 표현방식을 정의하였는데 single precision과 double precision입니다. C언어에서의 float가 single precision, double이 double precision을 의미합니다.

single-precision과 double-precision 두 가지를 만든 이유는 single-precision으로 표현이 안된 수가 당연히 종종 나왔다. 하드웨어가 미약하였기 때문에 double을 사용하지 않은 이유도 있다. 하지만 하드웨어가 허용되면 일반적으로 double-precision을 사용하는 게 더 좋다. 오늘 날에 하드웨어는 범용 컴퓨터에서 사용할 때에는 double을 사용한다. float를 쓸 때에는 이유가 있어야 한다. 예를 들면 작고 쉬운 임베디드 연산을 사용할 때 사용하지만 그 외에는 double을 사용해야 한다.

Subword Parallelism은 정밀도가 낮은 데이터를 처리할 때 단어 중심의 데이터패스를 충분히 사용할 수 있는 기법이다.

5.

6.

7. 1) single cycle implementation은 lw와 같은 명령어는 같은 instruction을 수행하는데 있어서 MIPS에 구현된 대부분의 모듈에 access한다. 어려 모듈에 동시에 읽기,쓰기를 하다 보면 전체적인 처리시간이 길어지게 되므로 한 cycle내에서도 단계를 나눠서 instruction이 수행될 수 있도록 처리하자는 개념으로 multi cycle이 나왔다.

2) 파이프라인은 병렬로 하나의 하드웨어로 서로 다른 instruction을 돌리기 때문에 성능이 높아진다. 모든 instruction이 이상적으로 똑 같은 시간이 걸린다면 이상적인 속도는 각 stage의 수와 같게 된다. 하지만 각 stage가 균일한 속도를 갖고 있지 않기 때문에 가장 느린 stage를 speedup 기준을 잡는다. pipeline은 매 cycle 마다 instruction을 fetch를 하는 것이 이상적이지만 hazard 때문에 매 cycle마다 instruction을 fetch할 수 없다.

3)dynamic multiple issue는 프로그램이 실제로 실행 중에 지금 cycle의 instruction이 몇개를 돌리 수 있는지 run time으로 결정 한다.(컴파일러가 아니라 CPU가 결정한다.) static multiple issue는 주어진 사이클에 몇개의 instruction의 실행으로 넘길 수 있는지 compiler가 중심되는 역할을 한다. 한꺼번에 실행할 수 있는 instruction을 issue packets이라 불린다.

loop unrolling: 동시에 돌아가는 instruction을 늘려서 IPC를 2에 근사하게 만들어 준다. hazards를 피하기 위해서 사용한다.

8.

CPU가 입력하는 주소를 tag(캐시 블록이 어떤 위치에서 온 캐시 블록인지 지정), index(캐시의 몇 번째 블럭인지 지정), byte offset(블럭에서 찾고자 하는 item이 몇 번째 byte인지 지정)으로 나눠 생각할 수 있다. identification에서 byte offset은 내가 원하는 블록을 찾고 그 블록에서 몇 번째 바이트를 CPU에게 줄 것인지를 결정하는 것이기 때문에 사용하지 않는다. (placement와도 무관하다.) index가 찾아봐야 하는 캐시 메모리의 위치를 결정해준다. 캐시 메모리에 블록을 저장할 때 블록을 가지고 온 주소의 앞부분인 tag를 같이 저장하여 identification에서 지금 찾는 item이 캐시 메모리에 저장되어 있는지의 여부를 주소의 tag부분과 캐시 블록의 tag를 비교하여 확인한다. 즉 CPU가 입력하는 주소의 index 부분을 보고 캐시 메모리의 원하는 블록의 위치로 가서 캐시블록의 tag와 주소의 tag를 비교하여 같으면 캐시 히트라고 하고 다르면 캐시 미스라고 판단한다. 만약 캐시 히트라면 offset을 이용해서 캐시 블록 안에서 원하는 item을 꺼내는 것이다

9.

10

1) 각각 n-way conflict를 해소하여 miss rate를 감소시킬 수 있다. 각각 하드웨어가 조금씩 복잡해지기 때문에 hit time 조금씩 늘어난다. 만약 최대한의 자유도를 준다면 Fully-Associative Mapping이라고 한다. 몇 way인지 정해져 있는 것이 아니고 캐시 블럭의 크기에 따라서 다르다. index로 찾아 가는 것이 아니라 CPU가 보내준 Content를 패턴 매칭을 통해서 찾아낸다. Content addressable memory라고도 한다.

2) direct map 방식에서 캐시 메모리의 배열을 2열 종대로 바꾼 것이고 행을 set, 열을 way을 바꾼게 2-way associative cache이다. direct map 방식에서 발생하는 address conflict(2-way)를 해결하여 miss rate를 감소시킬 수 있기 때문에 2-way set associative cache가 miss rate면에서 더욱 우세하며, 하드웨어가 조금 늘어나 hit time이 2%정도 증가한다.