CSE305 Project 3: Simulating Pipelined Execution

201811011 곽지혁

- Ubuntu 20.04.5 LTS 환경에서 gcc 9.4.0 버전을 이용하여 다음 명령어로 컴파일하였다.

  gcc -o <runfile> project3.c

- 실행 명령어는 다음과 같다.

  $ ./runfile <-atp 또는 -antp> [-m addr1:addr2] [-d] [-p] [-n num_instr] <binary file>

- 명령어 Option에 관한 설명은 다음과 같다.

  -atp: Always Taken 분기 예측기를 사용한다.

  -antp: Always Not Taken 분기 예측기를 사용한다.

  -m: 프로그램이 종료될 때 메모리 주소 범위(addr1 ~ addr2)에 있는 내용들을 출력한다. 해당 주소가 data section의 주소(0x10000000에서 시작) 혹은 text section의 주소(0x400000에서 시작)를 벗어난 범위를 가리키고 있을 경우, 해당 주소에는 값이 할당되지 않았으므로 0x0을 출력한다. 디폴트 값은 없기 때문에 본 플래그가 있을 때는 항상 주소 값을 입력해주어야 한다.

  -d: 매 사이클 마다 모든 레지스터의 내용을 출력한다. –m 옵션으로 출력할 메 모리 주소 범위가 지정되어 있다면 지정된 메모리 범위의 내용도 출력한다.

  -p: 매 사이클 마다 각 파이프라인 단계에서 실행되고 있는 인스트럭션의 PC를 아래와 같은 형태로 출력한다. e.g.) {0x400010|0x40000c|0x400008|0x400004|0x400000}

  -n: 수행될 명령어의 개수를 지정한다. 디폴트 값은 없기 때문에 본 플래그가 있 을 때는 항상 개수를 지정해주어야 한다.

  (입력 값 중 대괄호('[]')로 나타낸 부분은 옵션을 포함하지 않을 경우 기본 값으로 실행되는 옵션들이다. '<>'로 나타낸 옵션은 프로그램을 실행할 때 반드시 필요한 옵션이므로, 옵션이 들어가 있지 않을 경우 오류로 처리한다.)

- 전체적인 Flow는 다음과 같다.

1. 기본적인 세팅을 한다. (register array 생성 및 초기화, pc 생성 및 0x400000로 초기화)
2. 인자로 받은 옵션들을 저장한다.
3. 인자로 받은 파일을 열어 buf에 저장해준다.

   2-1. 만일 이때 파일의 사이즈가 0이라면, option d, m에 맞추어 레지스터와 메모리를 출력하고 종료한다.

4. buf에서 Text Segment와 Data Segment의 size를 읽어와 각각의 array를 생성하고, 해당하는 Text값과 Data값들을 저장해준다.
5. While 문을 돌면서 IF, ID, EX, MEM, WB각 파트 별로 상태 레지스터의 값을 이용해 작동한다.

   5-1. 작동한 이후에는 각 파트별로 next_(상태 레지스터)에 결과를 저장한다.

   5-2. 형식에 맞추어 출력한다.

   5-3. next_(상태 레지스터)에 있는 값을 상태 레지스터로 옮긴 후에 반복문을 수행한다.

6. 정해진 n만큼 instruction을 완료했거나 모든 instruction을 수행했다면 반복문을 종료한다.
7. 반복문이 종료되었다면 형식에 맞추어 출력해준다.
8. 종료.

- 상태 레지스터
  - IFID_StateRegister

int val : IFID 상태 레지스터에 값이 존재하는 지 여부 판단을 위해 0: noop

int pc : IFID 상태 레지스터에 저장되어 있는 instruction의 주소

int NPC : 상태 레지스터에 저장되어 있는 instruction의 다음주소

int instr : 32bit instruction

  - IDEX_StateRegister

int val : IDEX 상태 레지스터에 값이 존재하는 지 여부 판단을 위해 0: noop

int pc : IDEX 상태 레지스터에 저장되어 있는 instruction의 주소

int NPC : IDEX 레지스터에 저장되어 있는 instruction의 다음주소

int op : 상위 6bit, EX에서 ALU계산을 위해 어떤 OP인지에 대한 정보가 필요함.

int rs : ALU계산을 위해 rs에 해당하는 레지스터의 번호

int rt : ALU계산을 위해 rt에 해당하는 레지스터의 번호

int rd : ALU계산을 위해 rd에 해당하는 레지스터의 번호

int simm : ALU계산을 위해 하위 16bit을 sign extended한 값

int uimm : ALU계산을 위해 하위 16bit을 unsign extended한 값

int control : control bit들을 담고 있으며 각 비트별로 해당 내용을 담고 있음.

(MemtoReg ,RegWrite, eqBrch, neBrch, OneByte, MemRead, MemWrite)

예를 들어, lw는 MemtoReg, RegWrite, MemRead가 필요하다. 즉 64+32+2=98 값을 가짐

MemtoReg : 메모리에서 가져와 레지스터에 저장해야 함을 나타내는 control bit

RegWrite: 레지스터에 저장해야 함을 나타내는 control bit

eqBrch: 현재 instruction이 beq인지를 나타내는 control bit

neBrch: 현재 instruction이 bne인지를 나타내는 control bit

OneByte: sw, sb임을 나타내는 control bit

MemRead: 메모리에서 값을 읽어와야 함을 나타내는 control bit

MemWrite: 메모리에 값을 저장해야 함을 나타내는 control bit

- EXMEM_StateRegister

int val : EXMEM 상태 레지스터에 값이 존재하는 지 여부 판단을 위해 0: noop

int pc : EXMEM 상태 레지스터에 저장되어 있는 instruction의 주소

int ALUOUT : EX에서 계산된 ALU의 결과, MEM에서 메모리 주소 혹은 Branch여부를 위해 또는 WB에서 저장할 값을 위해 가져옴.

int rt : 메모리에 저장할 값 혹은 가져온 값 저장을 위한 rt에 해당하는 레지스터의 번호

int BRTarget : 해당 상태 레지스터의 pc + offset*4의 값

int REGDst : 레지스터에 값을 저장을 위한 레지스터의 번호

int control : control bit들을 담고 있다.

- MEMWB_StateRegister {

int val : MEMWB 상태 레지스터에 값이 존재하는 지 여부 판단을 위해 0: noop

int pc : MEMWB 상태 레지스터에 저장되어 있는 instruction의 주소

int ALUOUT : EX에서 계산된 ALU의 결과

int MEMOUT : MEM에서 읽어온 값

int REGDst : 레지스터에 값을 저장을 위한 레지스터의 번호

int control : control bit들을 담고 있으며 각 비트별로 해당 내용을 담고 있음.

(MemtoReg ,RegWrite)