CSED311: Lab 5 (due May. 14)

김태연(20220140), 손량(20220323)

Last compiled on: Thursday 30th May, 2024, 16:44

1 Introduction

RISC-V architecture를 바탕으로 RV32I instruction을 수행하는 CPU를 설계하되, 지금까지의 Lab에서 사용한 "Magic memory" 대신 실제와 유사한 memory hierarchy를 가지도록 memory 접근에 penalty를 부여하고, 높은 hierarchy를 가지는 256 byte의 data cache를 구현한다.

2 Design

이번에 구현한 RISC-V Multicycle CPU는 다음과 같은 주요 기능을 가진 Submodule로 나누었으며, 생성된 값들을 선택하기 위해 추가적으로 Multiplexer를 구현하여 사용했다. 기존 Single-cycle CPU에 추가적으로 Forwarding unit과 Forwarding path를 설계하여 Pipelining을 지원하여 Data hazard를 피했으며, GShare branch predictor를 디자인하여 Control hazard를 피했다.

2.1 Program Counter

Program counter는 CPU clock에 synchronous하게 현재 program counter를 출력하고, 들어오는 32-bit 입력으로 현재 program counter를 갱신한다. 해당 모듈 내부에는 branch prediction 기능을 가지고 있지 않다.

2.2 Instruction Memory and Data Memory

구현한 CPU는 instruction memory 영역과 data memory 영역을 분리하여 가지고 있으며, 각 memory의 특징은 다음과 같다.

2.3 Register File

2.4 ALU

기존의 Single cycle CPU와 동일한 모듈 배치를 이루고 있으며, 각 모듈의 Asynchronous 혹은 Synchronous한 Clock signal 또한 Single-cycle CPU와 동일한 구조를 가지고 있다. `opcodes.v`에서 자세히 확인할 수 있다.

2.5 Inter-Stage Registers

Lab 4에서 구현한 Pipelined CPU와 동일한 구조를 가지고 있다. 해당 Pipeline을 통해 서로 다른 instruction을 CPU 내에서 동시에 처리할 수 있다. 각각 Instruction fetch – Instruction decode – Execute – Memory access – Write back stage의 서로 다른 단계 사이에 신호를 전달한다.

2.6 Control Unit

Instruction Fetch 과정을 거친 이후 Control Unit으로 `IF_ID_reg_inst_out` Interstage register의 값이 입력되며, 해당 instruction의 type으로부터 다음과 같은 종류의 Control signal을 만들어낼 수 있다. Signal이 생성되어 필요한 Pipelining stage까지 Interstage register를 통해 Clock Synchronous하게 전달된다.

2.7 Hazard Detection Unit

2.8 Hazard Detection

Hazard Detection을 구현하기 위해 `HazardDetectionUnit` 모듈을 구현하였다. Hazard Detection Unit은 EX stage에 Load instruction이 배치됨으로써 발생하는 불가피한 "Stall" 을 처리하도록 설계하였다. Hazard Detection Unit의 출력인 `is_hazardous`는 각 `rs1`, `rs2`와 `ecall` 시 `x17` register를 관리하기 위해 총 3개의 `HazardDetectionUnit`을 두었다. 이 중 하나만이라도 Hazardous하게 판단이 된 경우 `pc_write_enable` 및 `IF_ID_reg_write_enable` 신호를 Disable시켜 하나의 Clock cycle을 강제로 소비하여 Hazard를 해결한다 (Bubbling).

2.9 Data Forwarding

Data Forwarding을 구현하기 위해 `ForwardingUnit` 모듈을 두었다. Forwarding Unit은 각 단계 사이의 Inter-stage register로부터 ALU에 Forward된 값을 전달하며, multiplexer로 선택하기 위해 Forwarded 여부를 같이 출력한다. Forwarding에는 다음과 같은 가능한 조합이 있으며, 두 조합 모두 만족하는 경우 Pipeline에서 가장 최근에 갱신되는 값인 EX/MEM stage의 값을 전달해야 한다. 만약 Hazard distance가 3보다 커지는 경우, 이미 Data memory로 Write back이 진행된 상태이기 때문에 이 부분에 대해선 고려하지 않는다. 최종적으로 ALU에게 ALU가 진행중인 instruction의 operand를 전달할 수 있도록 구현하였다.

2.10 Branch Prediction

32개의 Branch target buffer entry를 가지는 GShare Branch Predictor를 구현하여 사용하였다. 해당 구현에서 사용한 Pattern History Table은 2-bit saturation counter를 사용하였다. Branch predictor은 다음 Program counter를 예측하는 부분과, 실제 EX stage의 결과로부터 Branch predictor를 갱신하는 부분으로 나뉘어져 있다. 다음 Program counter의 예측은 Combinational하게 생성되며, Branch predictor 내부 상태의 갱신은 Clock에 Synchronous하게 동작한다. 실제로 수행된 branch prediction 여부에 따라 branch history shift register 및 pattern history table이 업데이트된다.

3 Cache

3.1 Cache Structure

구현한 Cache의 총 Data block은 256byte이며, 4-way Associative Cache로 구현하였다. 다음과 같은 Cache specification을 가지고 있다.

32bit WORD address를 다음과 같이 나누어서 256 byte cache에 접근하기 위해 사용한다. 256 byte cache는 4개의 64Byte cache set으로 이루어지며, 각 Set은 4개의 index를 가진 16Byte cache line으로 이루어진다. 마지막으로 16byte cache line은 4byte WORD를 4개로 묶어서 하나의 Tag를 공유하도록 설계했다.

- **Tag** 26 bit – 아래에 사용할 Index, Block offset, Granularity를 제외한 상위 MSB를 모두 Tag로 관리한다.

- **Index** 2 bit – 각 Set마다 $4 = 2^2$ 개의 index를 가지고 있으며 접근하기 위해 2bit의 index를 사용한다.
- **Block Offset** 2 bit – Cache Tag를 저장하기 위한 Overhead를 줄이기 위해, 여러 Word를 하나로 묶어서 관리하며, 본 Cache의 구현에서는 $4 = 2^2$ 개의 WORD를 하나의 Tag로 묶어서 관리했다.
- **Granularity** 2 bit – WORD size 32bit로 접근하기 위해, Address의 아래 2bit는 사용하지 않는다.

3.2 Cache Replacement

이번 구현에서는 Bit-PLRU (Pseudo-LRU) 를 사용하여 Cache replacement를 구현하였다. 4개의 Set을 사용하기 때문에 다음과 같이 Pseudo LRU를 구현할 수 있다.

- **Initialize** – 4개의 Set이 존재하기 때문에, 각 Index별로 4bit의 bit array를 `4'b0000`으로 초기화한다.
- **Access on Hit**
  - Hit이 일어나는 경우 hit된 set에 해당하는 bit를 `1`으로 설정한다.
  - 만약, 해당 bit가 마지막으로 켜지게 되는 경우 (해당 bit가 켜지면 `4'b1111`이 되는 경우), bit array를 해당 set에 해당하는 bit만 `1`, 이외는 `0`인 상태로 설정한다.
- **Eviction** – Bit array를 MSB부터 스캔하며, 처음에 등장하는 `0`에 해당하는 set의 값을 evict한다.

4 Implementation

각 베릴로그 파일에 대한 세부 설명은 다음과 같다.

4.1 `top.v` – Top module

`cpu.v`에서 구현한 `cpu` 모듈을 사용하여 Pipelined CPU를 구현하였다.

4.2 `cpu.v` – 내부 Module을 연결하여 Pipelined CPU 구성

`cpu.v`는 다음과 같은 submodule을 연결하여 구성하였다.

- `PC pc` – Program Counter이다.
- `InstMemory imem` – Instruction Memory이다.
- `IFIDRegister if_id_reg` – Instruction Fetch stage에서 Instruction Decode stage로 넘어가는 Interstage register 전달 모듈이다.
- `ControlUnit ctrl_unit` – Control unit으로, ALU 및 각 Memory module, Hazard Detection, Forwarding 등에 사용되는 전반적인 Control signal을 instruction으로부터 생성한다.
- `MUX2X1 rs1_mux` – `ecall`의 경우 고정적으로 `x17`을 `rs1`으로 넘겨주기 위한 multiplexer이다.
- `RegisterFile reg_file` – Register file이다.

- `EcallUnit ecall_unit` – `ecall`을 처리하기 위한 Unit으로, Pipeline을 모두 수행한 이후에 해당 모듈에 Signal이 전달되도록 설계되었다.
- `ImmediateGenerator imm_gen` – Immediate generator이다.
- `HazardDetectionUnit rs1_hdu` – Execution stage에 `rs1`으로 인한 Hazard를 감지, Forwarding이 필요한 경우 Forwarding module인 `rs1_fwd` 에 신호를 제공한다.
- `HazardDetectionUnit rs2_hdu` – Execution stage에 `rs2`으로 인한 Hazard를 감지, Forwarding이 필요한 경우 Forwarding module인 `rs2_fwd` 에 신호를 제공한다.
- `EcallHazardDetectionUnit ecall_hdu` – Execution stage에 `x17`으로 인한 Hazard가 발생할 경우, `ecall`의 동작을 보장하기 위한 전용 Hazard Detection Unit이다.
- `IDEXRegister id_ex_reg` – Instruction Decode stage에서 Execution stage로 넘어가는 Interstage register 전달 모듈이다.
- `ALUControlUnit alu_ctrl_unit` – ALU Control Unit으로, Single cycle CPU에서 Control flow instruction에 필요한 operation을 제외하고 구성되었다.
- `ForwardingUnit rs1_fwd` – EX/MEM stage와 MEM/WB stage의 `rs1` ID와 그 값, 현재 ALU에 들어간 `rs1` ID로부터 ALU에 Forwarding할 값과 Forward 여부를 결정한다.
- `MUX2X1 rs1_fwd_mux` – ALU의 `rs1` 입력으로 Register file의 값과 `rs1_fwd` Forwarding unit의 Forwarded value 중 필요한 값을 선택한다.
- `MUX2X1 op1_mux` – ALU의 `in1` 입력으로 현재 Program counter 및 Register file의 값 중 선택하는 multiplexer이다.
- `ForwardingUnit rs2_fwd` – EX/MEM stage와 MEM/WB stage의 `rs2` ID와 그 값, 현재 ALU에 들어간 `rs2` ID로부터 ALU에 Forwarding할 값과 Forward 여부를 결정한다.
- `MUX2X1 rs2_fwd_mux` – ALU의 `rs2` 입력으로 Register file의 값과 `rs2_fwd` Forwarding unit의 Forwarded value 중 필요한 값을 선택한다.
- `MUX2X1 mux_alu_in_2_select` – ALU의 `in2` 입력으로 `rs2_fwd_mux`를 통과한 최종적인 `rs2` 의 값과 Immediate generator에서 생성된 값, +4 중 필요한 값을 선택한다.
- `ALU alu` – ALU를 구현하였다.
- `PCGenerator pc_gen` – Execution stage를 거친 실제 연산 결과로부터 다음 instruction을 fetch해야하는 Program counter를 계산한다.
- `ControlHazardDetectionUnit ctrl_hdu` – 해당 Instruction이 Control Hazard를 가지고 있는지 판단한다.
- `MUX2X1 mux_next_pc` – 다음 Program counter를 Branch predictor으로부터 갱신할 것인지, `pc_gen`을 통해 계산된 Program counter으로 갱신할 것인지 결정하는 multiplexer이다.
- `BranchPredictor branch_predictor` – GShare Branch Predictor를 통해 다음 Instruction fetch가 일어날 Instruction address를 예측한다.

- `EXMEMRegister ex_mem_reg` – Execution stage에서 Memory stage로 넘어가는 Interstage register 전달 모듈이다.
- `Cache cache` – **Cache가 구현되었으며, 해당 모듈 내에 기존의 `DataMemory`를 두어 CPU에서 Data memory에 접근하기 위해서 오직 Cache만 접근할 수 있도록 구현했다.**
- `CacheCounter cache_counter` – **Cache Hit, Cache Miss 및 Memory access 횟수를 세며, 이를 통해 Cache Hit Ratio를 계산할 수 있다.**
- `MUX2X1 rd_mux` – Write back 될 value를 Register file의 출력 `rs2`와 Data Memory의 `dmem_dout`으로부터 선택한다.
- `MEMWBregister mem_wb_reg` – Memory stage에서 Write back stage로 넘어가는 Interstage register 전달 모듈이다.

4.3 `PC.v` – Program Counter

Clock에 Synchronous하게 현재 PC를 출력하고, 들어오는 32-bit 입력으로 현재 PC를 갱신하는 모듈이다.

4.4 `PCGenerator.v` – 다음 PC 계산

Unconditional Jump (`JAL`, `JALR`)의 경우 해당 Offset을 더하여 다음 PC를 출력한다. Branch의 경우 입력된 `imm`만큼을 더해 다음 PC를 출력하게 된다. 그 이외의 경우는 다음 Instruction (PC + 4)를 출력한다.

4.5 `RegisterFile.v` – Register File

`RegisterFile` 모듈의 입력 신호 중 `rd`와 `rd_din`은 Write Back stage까지 Pipeline을 진행한 이후 입력되며, 이외의 입력은 Instruction Decode Stage에서 입력된다.

4.6 `ControlUnit.v` – Control Unit

Single Cycle CPU의 기능에 더해, 다양한 Data hazard / Control Hazard를 Resolve하기 위한 Control signal output이 존재한다.

4.7 `InstMemory.v` – Instruction Memory

4.8 `DataMemory.v` – Data Memory

Pipelined CPU의 구현에서는 Instruction Memory와 Data Memory가 분리되어 있는 구조이다.

4.9 `ALU.v` – ALU

4.10 `ALUControlUnit.v` – ALU Control Unit

4.11 `ImmediateGenerator.v` – Immediate Generator

Single Cycle CPU의 구성과 동일한 ALU 설계와 ALU Opcode, Immediate generator의 형태를 가지고 있다.

4.12 `IFIDRegister.v` – Intsruction Fetch stage / Instruction Decode stage 간 Interstage Register 전달

4.13 `IDEXRegister.v` – Instruction Decode stage / Execution stage 간 Interstage Register 전달

4.14 `EXMEMRegister.v` – Execution stage / Memory stage 간 Interstage Register 전달

4.15 `MEMWBregister.v` – Memory Stage / Write back 간 Interstage Register 전달

각 Stage 간 전달되는 Interstage Register를 CPU clock에 Synchronous하게 다음 Stage로 전달해주는 Unit들이다.

4.16 `BranchPredictor.v` – GShare Branch Predictor

GShare branch predictor를 구현하였으며, 해당 모듈 내에서 Prediction 성공 여부로부터 Branch predictor 내부 구성 요소(BHSR, PHT, BTB 등)의 갱신이 Clock synchronous하게 이루어지며, 다음 Program counter의 예측은 해당 모듈의 Combinational block에 구현하였다.

4.17 `HazardDetectionUnit.v` – Data Hazard Detection Unit

Data Hazard Detection Unit을 구현했다.

4.18 `ControlHazardDetectionUnit.v` – Control Hazard Detection Unit

Control Hazard Detection Unit을 구현했다.

4.19 `ForwardingUnit.v` – Forwarding Unit

EX, MEM stage의 register ID, register value로부터 Forwarding을 감지하고, Forwarding을 진행한다.

4.20 `ECallUnit.v` – `ecall` Unit

4.21 `EcallHazardDetectionUnit.v` – `ecall` Hazard Detection Unit

`ecall`을 구현하기 위한 모듈이며, 해당 모듈으로 들어오는 Signal은 모든 Stage를 지나서 전달된다. 이를 구현하기 위해 `cpu.v`의 Interstage register로 `is_halted`를 두어 구현했다. 또한, `halt`는 `x17` register에 값을 작성해서 이룰 수 있으며, 마찬가지로 Register의 Hazard가 발생할 수 있다. 특별히 Ecall만을 관리하기 위해 `EcallHazardDetectionUnit` 모듈을 만들어서 관리했다.

4.22 `Cache.v` – Cache

4.23 `cache_def.v` – Cache FSM & Structure Definition

4.24 `CacheCounter.v` – Cache Hit/Miss Counter

256 byte의 4-way associative cache를 구현하였으며, Bit-PLRU cache algorithm을 사용해 cache eviction될 대상을 결정하였다. 해당 Cache에서 발생하는 memory access 및 cache hit / miss를 `CacheCounter.v`에서 생성한 `CacheCounter` 모듈에서 세고, 이를 수정된 `tb_top.cpp`에서 값을 불러와 출력하게 된다.

4.25 `Adder.v` – Adder

4.26 `MUX2X1.v` – 2-to-1 Multiplexer

4.27 `MUX4X1.v` – 4-to-1 Multiplexer

Pipelined CPU의 구현에 필요한 Multiplexer 및 Adder를 개별의 모듈로 분리하였다.

5 Discussion

5.1 Replacement Policy

이번에 구현한 Cache는 Direct-mapped cache가 아니기 때문에 여러 개의 set에 들어있는 값들 중 Cache replacement를 진행하는 알고리즘이 필요하다. 그러나 상호 Associative한 Cache line이 증가할수록 Least-Recently Used를 추적하기 위한 bit 수가 증가하여, Cache 구현에 overhead가 생길 수 있다. 따라서 본 구현에서는 해당 LRU를 근사하는 Pseudo-LRU 알고리즘 중 하나인 Bit-PLRU를 사용하여 구현하였다.

5.2 Associativity

이번 구현에서 4-way set을 사용한 이유는 다음과 같이 실험을 수행하였을 때의 결과로부터 최적의 Hit ratio를 4-way set에서 얻었기 때문이다. 각각 Naive 및 Optimized matrix multiplication benchmark에서의 실험 결과는 다음과 같으며, 모든 Testbench에서 Memory 접근은 공통적으로 2499번 발생하였다. Direct-Mapped Cache 이외의 Set-associative한 cache의 Cache Algorithm으로 동일하게 Bit-PLRU를 사용하였다.

Testbench `naive_matmul_unroll`

Set associativity `a`	# of Index `b`	Cache Hit	Hit %	Total Cyc.
1	16	1605	0.642	76801
2	8	1140	0.456	103480
4	4	1995	0.798	36529
8	2	1140	0.456	103480

Testbench `opt_matmul_unroll`

Set associativity `a`	# of Index `b`	Cache Hit	Hit %	Total Cyc.
1	16	1687	0.675	71568
2	8	1203	0.481	98980
4	4	2248	0.900	25617
2	8	1203	0.481	98980

다른 (Set, Index)의 조합보다 이번에 구현한 4-way set associative cache가 Matrix multiplication에 대해 상당히 높은 Hit rate를 보여주고 있다.

5.3 Naïve vs. Optimized Matrix Multiplication

아래의 분석은 우리가 구현한 4-way associative 16-byte block Cache를 기준으로 분석하였다.

5.3.1 Naïve Matrix Multiplication

Naïve한 행렬 곱셈 알고리즘은 다음과 같은 구성을 가지고 있다.

```
#pragma unroll (M)
for (int m = 0; m < M; ++m) {
  #pragma unroll (N)
  for (int n = 0; n < N; ++n) {
    #pragma unroll (K)
    for (int k = 0; k < K; ++k) {
      c[m][n] += a[m][k] + b[k][n];
    }
  }
}
```

Source array a Row와 b의 Column을 각각 Cache하여 연산 결과를 저장하게 된다. Destination array c의 인접한 16byte (4개의 WORD) 에 대해 아래와 같은 작업이 반복적으로 수행된다.

먼저, 각 Array의 element가 어떤 index를 가지는 지 나타내면 다음과 같다. 실제로 캐싱되어 사용하는 값은 **굵게** 나타내었으며, 캐싱되는 값들에 대해서만 해당하는 Index를 나타내었다.

Source a							
0	**0**	**0**	**0**	**1**	**1**	**1**	**1**

Source b							
0	0	0	0				
2	2	2	2				
0	0	0	0				
2	2	2	2				
0	0	0	0				
2	2	2	2				
0	0	0	0				
2	2	2	2				

Destination c							
0	**0**	**0**	**0**				

상기한 구성에서, 인접한 4개의 WORD를 계산하기 위해 Cache되는 Index 0의 블럭은 6개이다. 이 값은 현재 구현에서 사용하고 있는 Cache set의 개수인 4보다 크기 때문에, 해당 과정에서 필연적으로 Cache Miss가 발생하게 되며 이에 따라 Memory에 접근하는 Overhead가 발생하게 된다. 또한, Matrix b를 Cache하는데 있어서 실제로 계산에 사용하지 않으나 함께 Caching되는 block으로 인해 Cache 사용 측면에서 Utilization이 떨어지게 된다.

5.3.2 Optimized Matrix Multiplication

Optimize된 행렬 곱셈 알고리즘은 다음과 같은 구성을 가지고 있다.

```
#pragma unroll (M)
for (int tile_m = 0; tile_m < M; tile_m += TILE) {
  #pragma unroll (N)
  for (int tile_n = 0; tile_n < N; tile_n += TILE) {
    #pragma unroll (K)
    for (int tile_k = 0; tile_k < K; tile_k += TILE) {
      #pragma unroll (TILE)
      for (int m = tile_m; m < tile_m + TILE; ++m) {
        #pragma unroll (TILE)
        for (int n = tile_n; n < tile_n + TILE; ++n) {
          #pragma unroll (TILE)
          for (int k = tile_k; k < tile_k + TILE; ++k) {
```

```
                c[m][n] += a[m][k] + b[k][n];
              }
            }
          }
        }
      }
    }
  }
```

8x8 행렬을 4개의 Sub 4x4 `TILE`으로 나누어 접근하는 형태를 가지게 되며, 메모리 접근 패턴 또한 같은 `TILE`에 접근하여 Submatrix의 행렬 곱셈 패턴이 발생하게 되며, 2가지의 패턴이 각각 2번 발생한다. 먼저, 각 Array의 element가 어떤 index를 가지는 지 나타내면 다음과 같다. 실제로 캐싱되어 사용하는 값은 **굵게** 나타내었으며, 캐싱되는 값들에 대해서만 해당하는 Index를 나타내었다.

첫 번째 패턴은 다음과 같다.

Source a								**Source b**								**Destination c**							
0	**0**	**0**	**0**					**0**	**0**	**0**	**0**					**0**	**0**	**0**	**0**				
2	**2**	**2**	**2**					**2**	**2**	**2**	**2**					**2**	**2**	**2**	**2**				
0	**0**	**0**	**0**					**0**	**0**	**0**	**0**					**0**	**0**	**0**	**0**				
2	**2**	**2**	**2**					**2**	**2**	**2**	**2**					**2**	**2**	**2**	**2**				

첫 번째 패턴은 다음과 같다.

Source a								**Source b**								**Destination c**							
0	**0**	**0**	**0**									**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**				
2	**2**	**2**	**2**									**3**	**3**	**3**	**3**	**2**	**2**	**2**	**2**				
0	**0**	**0**	**0**									**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**				
2	**2**	**2**	**2**									**3**	**3**	**3**	**3**	**2**	**2**	**2**	**2**				

Optimized approach에서는 두 가지의 패턴이 나타나는데, 각기 분석한 결과는 다음과 같다.

- 첫 번째 패턴의 경우, (`Index 0` ×6), (`Index 2` ×6) 으로 캐시에 접근하고 있다. ((0, 2)와 (1, 3)으로, 동일 Index의 set을 6개 사용하는 Cache access pattern이다) 이 때, 특정 index만 사용하는 점에서 Cache 사용에 비효율적으로 나타나지만, Naïve approach와 다르게 Cache block을 최대한으로 Utilize하고 있다.
- 두 번째 패턴의 경우, (`Index 0` ×4), (`Index 2` ×4), (`Index 1` ×2), (`Index 3` ×2) 으로 캐시에 접근하고 있다. 이 경우에는 모든 Cache index에 대해서 구현된 Cache set보다 사용하는 Set이 작거나 같게 되어, 새로운 `TILE`으로 접근해 Working set을 바꾸며 발생하는 Miss 이외에 동일 `TILE` 내에서 Cache miss가 이론상으로 발생하지 않는다.

Matrix multiplication을 Cache에 Utilize할 수 있는 size로 divide-and-conquer하여 Cache hit ratio를 증가시켰고, 이에 따라 유의미한 성능 개선을 이룰 수 있었다.

6 Conclusion

아래의 benchmark들은 모두 4-way set associative한 Cache를 구현한 결과이다.

`naive_matmul_unroll.mem`를 사용하여 Benchmark를 실행했을 때, 다음과 같은 결과를 얻었으며 모든 Testcase 및 Register가 맞는 것을 확인했다. 이 때 나타나는 Cache hit ratio는 $1995/2499 = 0.798$임을 확인할 수 있었다.

```
### SIMULATING ###
TEST END
SIM TIME : 73060
TOTAL CYCLE : 36529 (Answer : 71567)
FINAL REGISTER OUTPUT
MEMORY ACCESSES : 2499
CACHE HITS : 1995
 0 00000000 (Answer : 00000000)
 1 00000000 (Answer : 00000000)
 2 00002ffc (Answer : 00002ffc)
 3 00000000 (Answer : 00000000)
 4 00000000 (Answer : 00000000)
 5 00000000 (Answer : 00000000)
 6 00000000 (Answer : 00000000)
 7 00000000 (Answer : 00000000)
 8 00000000 (Answer : 00000000)
 9 00000000 (Answer : 00000000)
10 00000000 (Answer : 00000000)
11 00000000 (Answer : 00000000)
12 00000000 (Answer : 00000000)
13 0000007e (Answer : 0000007e)
14 000004f3 (Answer : 000004f3)
15 000005f0 (Answer : 000005f0)
16 00000000 (Answer : 00000000)
17 0000000a (Answer : 0000000a)
18 00000000 (Answer : 00000000)
19 00000000 (Answer : 00000000)
20 00000000 (Answer : 00000000)
21 00000000 (Answer : 00000000)
22 00000000 (Answer : 00000000)
23 00000000 (Answer : 00000000)
24 00000000 (Answer : 00000000)
25 00000000 (Answer : 00000000)
26 00000000 (Answer : 00000000)
27 00000000 (Answer : 00000000)
28 00000000 (Answer : 00000000)
29 00000000 (Answer : 00000000)
30 00000000 (Answer : 00000000)
31 00000000 (Answer : 00000000)
Correct output : 32/32
```

`opt_matmul_unroll.mem`를 사용하여 Benchmark를 실행했을 때, 다음과 같은 결과를 얻었으며 모든 Testcase 및 Register가 맞는 것을 확인했다. 이 때 나타나는 Cache hit ratio는 $2248/2499 = 0.900$임을 확인할 수 있었으며, `naive_matmul_unroll.mem`에 비교했을 때

$36529/25617 = 142.6\%$ 빨라진 것을 확인할 수 있었다.

```
### SIMULATING ###
TEST END
SIM TIME : 51236
TOTAL CYCLE : 25617 (Answer : 76800)
FINAL REGISTER OUTPUT
MEMORY ACCESSES : 2499
CACHE HITS : 2248
 0 00000000 (Answer : 00000000)
 1 00000000 (Answer : 00000000)
 2 00002ffc (Answer : 00002ffc)
 3 00000000 (Answer : 00000000)
 4 00000000 (Answer : 00000000)
 5 00000000 (Answer : 00000000)
 6 00000000 (Answer : 00000000)
 7 00000000 (Answer : 00000000)
 8 00000000 (Answer : 00000000)
 9 00000000 (Answer : 00000000)
10 00000000 (Answer : 00000000)
11 00000000 (Answer : 00000000)
12 00000000 (Answer : 00000000)
13 0000007e (Answer : 0000007e)
14 000004f3 (Answer : 000004f3)
15 000005f0 (Answer : 000005f0)
16 00000000 (Answer : 00000000)
17 0000000a (Answer : 0000000a)
18 00000000 (Answer : 00000000)
19 00000000 (Answer : 00000000)
20 00000000 (Answer : 00000000)
21 00000000 (Answer : 00000000)
22 00000000 (Answer : 00000000)
23 00000000 (Answer : 00000000)
24 00000000 (Answer : 00000000)
25 00000000 (Answer : 00000000)
26 00000000 (Answer : 00000000)
27 00000000 (Answer : 00000000)
28 00000000 (Answer : 00000000)
29 00000000 (Answer : 00000000)
30 00000000 (Answer : 00000000)
31 00000000 (Answer : 00000000)
Correct output : 32/32
```