Computer Architecture

Lab-01

담당 교수	[redacted]
제 출 일	[redacted]
학 번	[redacted]
이 름	[redacted]

1. 과제의 목표

(1) Binary Code로 주어진 RISC-V 명령어를 Decode하고 Execution 할 수 있는 프로그램을 작성하여 주어진 instruction을 실행하고 동작을 분석한다.

2. Part 1 (Program A), Part 2 (Program B)

주어진 Program A, B는 명령어의 종류만 1~2개 차이나는 근본적으로 같은 동작을 하는 코드이다. 따라서, **Part 1과 Part 2**를 분리하지 않았으며, Instruction을 수행하기 위해 다음과 같이 주어진 뼈대코드를 수정하였다. 주석이 포함된 코드를 첨부하고 자세한 설명은 **Part 3**에서 하도록 하겠다.

코드

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
// registers
unsigned int registers[32];
// program counter
unsigned int pc = 0;
// instruction/data memory
#define INST_MEM_SIZE 32*1024
#define DATA_MEM_SIZE 32*1024
unsigned int inst_mem[INST_MEM_SIZE]; //instruction memory
unsigned int data_mem[DATA_MEM_SIZE]; //data memory
// example instruction set
enum OPCODE {      // Instruction을 구분하기 위한 OPCODE로서 7bit의 값을 16진수로 작성하였다.
    ADD = 0x33,    // R-type
    ADDI = 0x13,   // addi I-type
    LW = 0x03,     // load I-type
    SW= 0x23,      // S-type
    BEQ = 0x63,    // SB-type
    LUI = 0x37,    // U-type
    HALT = 0x7F    // END
};

void my_print_register() {  // Register에 저장된 값을 출력하는 함수.
    printf("Current Register Values \n");   // Register에 저장되어 있는 값을 출력
    int i = 0;
    while (i < 32) {
        if (registers[i] != 0)
            printf("| \033[31mx%02d : %02d \033[0m", i, registers[i]);  // 0이 아닌 값이 있으면 빨간색
 출력
        else
            printf("| x%02d : %02d ", i, registers[i]);
        i++;
        if (i % 8 == 0 && i != 0)
            printf("\n");
    }
    printf("Current Data_Memory Values \n");    // Data Memory에 저장되어 있는 값을 출력
    i = 0;
    while (i < 32) {
        if (data_mem[i] != 0)
            printf("| \033[31mMEM%02d : %02d \033[0m", i, data_mem[i]); // 0이 아닌 값이 있으면 빨간색
 출력
        else
```

```
            printf("| MEM%02d : %02d ", i, data_mem[i]);
        i++;
        if (i % 8 == 0 && i != 0)
            printf("\n");
    }
}

void execute_instructions(void) { // 주어진 Instruction을 Execution 하는 함수.
    // flag for end-of-program
    int running = 1;
    int step = 0;
    printf("\033[31mInitial values\n\033[0m"); // Instruction을 수행하기 전의 Register의 초기값 출력
.
    my_print_register();
    printf("==============================\n");
    while (running) {
        //Fetch
        int instruction = inst_mem[pc]; // 현재 PC에 해당하는 Instruction을 저장.
        int opcode = (instruction & 0x7F); // instruction[6:0] 의 값을 Masking
        int funct3 = ((instruction >> 12) & 0x7); //instruction[14:12]의 값을 추출하기 위해 12bit만큼
 bit-shift 후 Masking
        printf("\033[35mSTEP : %d\033[0m\n",step++); // 현재 Step을 출력
        printf("\033[36mPC : %d\033[0m\n",pc); // 현재 PC값을 출력
        int rs1,    // source reg.1
            rs2,    // source reg.2
            rd,     // destination reg.
            imm,    // immediate
            addr;   // base address

        switch (opcode) {
            case ADD:
                rd = (instruction >> 7) & 0x1F;     // bit shift이후 Masking하여 rd 추출
                rs1 = (instruction >> 15) & 0x1F;   // bit shift이후 Masking하여 rs1 추출
                rs2 = (instruction >> 20) & 0x1F;   // bit shift이후 Masking하여 rs2 추출
                printf("\033[32madd x%02d, x%02d, x%02d\n\033[0m",rd, rs1,rs2); // Binary Code를 D
eocde한 결과를 Assembly로 출력
                registers[rd] = registers[rs1] + registers[rs2]; // add 명령어 수행
                pc++; // 다음 Instruction 수행을 위해 PC증가.
                break;
            case ADDI:
                rd = (instruction >> 7) & 0x1F;     // bit shift이후 Masking하여 rd 추출
                rs1 = (instruction >> 15) & 0x1F;   // bit shift이후 Masking하여 rs1 추출
                imm = (instruction >> 20) & 0xFFF;  // bit shift이후 Masking하여 imm값추출
                if (((instruction >> 31) & 0x1) == 0x1) { imm = (imm | 0xFFFFF000); }   // imm의 MSB
가 1이라면 Sign Extension
                printf("\033[32maddi x%02d, x%02d, %d\n\033[0m",rd, rs1,imm); // Binary Code를 Deo
cde한 결과를 Assembly로 출력
                registers[rd] = registers[rs1] + imm; // addi 명령어 수행
                pc++;
                break;
            case LW:
                rd = (instruction >> 7) & 0x1F;     // bit shift이후 Masking하여 rd 추출
                rs1 = (instruction >> 15) & 0x1F;   // bit shift이후 Masking하여 rs1 추출
```

```
            imm = (instruction >> 20) & 0xFFF;  // bit shift이후 Masking하여 imm값 추출
            if (((instruction >> 31) & 0x1) == 0x1) { imm = (imm | 0xFFFFF000); }    // imm의 MSB가 1이라면 Sign Extension
            printf("\033[32mlw x%02d ,%d(x%02d)\n\033[0m",rd,imm,rs1);
            registers[rd] = data_mem[registers[rs1] + imm/4]; // LW 명령어 수행. data mem 1개가 4바이트 덩어리기 때문에 imm 값을 4로 나눠줌.
            pc++;
            break;
        case LUI :
            rd = (instruction >> 7) & 0x1F;
            imm = (((instruction >> 21) & 0x3FF) << 1) | (((instruction >> 20) & 0x1) << 11) | \
               (((instruction >> 12) & 0xFF) << 12) | (((instruction >> 31) & 0x1) << 20); // imm[31:12] 값을 추출.
            imm = imm << 11;       // 11-bit만큼 bit-shift.
            printf("\033[32mlui x%02d, %d\n\033[0m",rd,imm);
            registers[rd] = imm;    //LUI 명령어 수행. rd에 imm값을 load
            pc++;
            break;
        case SW:
            rs1 = (instruction >> 15) & 0x1F;
            rs2 = (instruction >> 20) & 0x1F;
            imm = ((instruction >> 7) & 0x1F) | (((instruction >> 25) & 0x7F) << 5); // imm[11:0]값을 추출
            if (((instruction >> 31) & 0x1) == 0x1) { imm = (imm | 0xFFFFF000); }    // imm의 MSB가 1이라면 Sign Extension
            printf("\033[32msw x%02d, %d(x%02d)\n\033[0m",rs2,imm,rs1);
            data_mem[registers[rs1] + imm/4] = registers[rs2]; // SW 명령어 수행. data mem 1-unit이 4바이트 덩어리기 때문에 imm 값을 4로 나눠줌.
            pc++;
            break;
        case BEQ: // BEQ와 BLT의 OPCODE가 같기 때문에 아래에서 func3를 활용하여 구분.
            imm = (((instruction >> 8) & 0xF) << 1) | (((instruction >> 25) & 0x3F) << 5) |\
             (((instruction >> 7) & 0x1) << 11) | (((instruction >> 31) & 0x1) << 12);  // imm[12:1]값을 추출
            if (((instruction >> 31) & 0x1) == 0x1) { imm = (imm | 0xFFFFF000); }      // imm의 MSB가 1이라면 Sign Extension
            rs1 = ((instruction >> 15) & 0x1F);
            rs2 = ((instruction >> 20) & 0x1F);
            printf("\033[32mbeq x%02d, x%02d, %d\n\033[0m",rs1,rs2,imm);
            if (funct3 == 0) { // BEQ (func3 == 0)
               if(registers[rs1] == registers[rs2]) { pc += (imm)/4; } // PC값이 4byte단위기 때문에 imm<<1 값을 4로 나눠서 PC에 더해줌
               else pc++;
            }
            else if (funct3 == 4) { // BLT (func3 == 4)
               if(registers[rs1] < registers[rs2]) { pc += (imm)/4; } // PC값이 4byte단위기 때문에 imm<<1 값을 4로 나눠서 PC에 더해줌
               else pc++;
            }
            break;
        case HALT:
            running = 0; // while문을 동작시키는 flag를 0으로 수정하여 종료.
            printf("\033[32mEND\n\033[0m");
```

```
            printf(" ** END OF THE PROGRAM ** \n");
            break;
        default:
            printf("\033[32mUnknown instruction\033[0m\n");
            running = 0;
            break;
      }
      printf("==============================\n");
   }
   my_print_register(); // 최종 register값을 출력.
}

void step_execution(void) { // Program을 Step별로 실행하여 Debugging하는 Mode의 함수.
   int running = 1;
   int step = 0;
   printf("\033[31mInitial values\n\033[0m");
   my_print_register();
   printf("================================================================================
====================\n");
   while (running) {
      //Fetch
      int instruction = inst_mem[pc];
      int opcode = (instruction & 0x7F);
      int funct3 = ((instruction >> 12) & 0x7);
      printf("\n\033[35mSTEP : %d\033[0m\n",step++); // 현재 Step을 출력
      printf("\033[36mPC : %d\033[0m\n",pc); // 현재 PC값을 출력
      int rs1,    // source reg.1
          rs2,    // source reg.2
          rd,     // destination reg.
          imm,    // immediate
          addr;   // base address

      switch (opcode) {
          case ADD:
              rd = (instruction >> 7) & 0x1F; // bit shift이후 Masking하여 원하는 값 추출
              rs1 = (instruction >> 15) & 0x1F;
              rs2 = (instruction >> 20) & 0x1F;
              printf("\033[32madd x%02d, x%02d, x%02d\n\033[0m",rd, rs1,rs2); // Binarycode를 As
sembly로 변환하여 출력
              printf("\033[0;33mx%02d = %d, x%02d = %d, x%02d = %d\n\033[0m",rd,registers[rd],rs
1,registers[rs1],rs2,registers[rs2]); // 명령어 실행 전 Target Register의 값 출력
              registers[rd] = registers[rs1] + registers[rs2]; // add 명령어 수행
              pc++;
              printf("\033[31mResult : x%02d = %d\033[0m\n",rd,registers[rd]); // 명령어 수행 후 T
arget Register의 값
              break;
          case ADDI:
              rd = (instruction >> 7) & 0x1F;
              rs1 = (instruction >> 15) & 0x1F;
              imm = (instruction >> 20) & 0xFFF;
              if (((instruction >> 31) & 0x1) == 0x1) { imm = (imm | 0xFFFFF000); }
              printf("\033[32maddi x%02d, x%02d, %d\n\033[0m",rd, rs1,imm);
              printf("\033[0;33mx%02d = %d , x%02d = %d\n\033[0m",rd,registers[rd],rs1,registers
[rs1]);
              registers[rd] = registers[rs1] + imm; // addi 명령어 수행
```

```
            pc++;
            printf("\033[31mResult : x%02d = %d\033[0m\n",rd,registers[rd]);
            break;
        case LW:
            rd = (instruction >> 7) & 0x1F;
            rs1 = (instruction >> 15) & 0x1F;
            imm = (instruction >> 20) & 0xFFF;
            if (((instruction >> 31) & 0x1) == 0x1) { imm = (imm | 0xFFFFF000); }
            printf("\033[32mlw x%02d ,%d(x%02d)\n\033[0m",rd,imm,rs1);
            printf("\033[0;33mx%02d = %d, data_mem[%d] = %d\033[0m\n",rd, registers[rd], regis
ters[rs1] + imm/4, data_mem[registers[rs1] + imm/4]);
            registers[rd] = data_mem[registers[rs1] + imm/4]; // LW 명령어 수행. data mem 1개가 4
바이트 덩어리기 때문에 imm 값을 4로 나눠줌.
            printf("\033[31mResult : x%02d = %d\033[0m\n",rd, registers[rd]);
            pc++;
            break;
        case LUI :
            rd = (instruction >> 7) & 0x1F;
            imm = (((instruction >> 21) & 0x3FF) << 1) | (((instruction >> 20) & 0x1) << 11) | \
                (((instruction >> 12) & 0xFF) << 12) | (((instruction >> 31) & 0x1) << 20);
            imm = imm << 11;
            printf("\033[32mlui x%02d, %d\n\033[0m",rd,imm);
            printf("\033[0;33mx%02d = %d\n",rd, registers[rd]);
            registers[rd] = imm;
            printf("\033[31mResult : x%02d = %d\033[0m\n",rd, registers[rd]);
            pc++;
            break;
        case SW:
            rs1 = (instruction >> 15) & 0x1F;
            rs2 = (instruction >> 20) & 0x1F;
            imm = ((instruction >> 7) & 0x1F) | (((instruction >> 25) & 0x7F) << 5);
            if (((instruction >> 31) & 0x1) == 0x1) { imm = (imm | 0xFFFFF000); }
            printf("\033[32msw x%02d, %d(x%02d)\n\033[0m",rs2,imm,rs1);
            printf("\033[0;33mx%02d = %d, data_mem[%d] = %d\n\033[0m",rs2, registers[rs2], reg
isters[rs1] + imm/4, data_mem[registers[rs1] + imm/4]);
            data_mem[registers[rs1] + imm/4] = registers[rs2]; // SW 명령어 수행. data mem 1-uni
t이 4바이트 덩어리기 때문에 imm 값을 4로 나눠줌.
            printf("\033[31mResult : data_mem[%d] = %d\033[0m\n",registers[rs1] + imm/4, data_
mem[registers[rs1] + imm/4]);
            pc++;
            break;
        case BEQ:
            imm = (((instruction >> 8) & 0xF) << 1) | (((instruction >> 25) & 0x3F) << 5) |\
             (((instruction >> 7) & 0x1) << 11) | (((instruction >> 31) & 0x1) << 12);
            if (((instruction >> 31) & 0x1) == 0x1) { imm = (imm | 0xFFFFF000); }
            rs1 = ((instruction >> 15) & 0x1F);
            rs2 = ((instruction >> 20) & 0x1F);

            if (funct3 == 0) {
                printf("\033[32mbeq x%02d, x%02d, %d\n\033[0m",rs1,rs2,imm);
                printf("\033[0;33mx%02d = %d, x%02d = %d\033[0m\n",rs1, registers[rs1], rs2,reg
isters[rs2]);
                if(registers[rs1] == registers[rs2]) { pc += (imm)/4; } // PC값이 4byte단위기 때문
에 imm<<1 값을 4로 나눠서 PC에 더해줌
                else pc++;
            }
```

```
            else if (funct3 == 4) {
                printf("\033[32mblt x%02d, x%02d, %d\n\033[0m",rs1,rs2,imm);
                printf("\033[0;33mx%02d = %d, x%02d = %d\033[0m\n", rs1,registers[rs1], rs2, re
gisters[rs2]);
                if(registers[rs1] < registers[rs2]) { pc += (imm)/4; } // PC값이 4byte단위기 때문
에 imm<<1 값을 4로 나눠서 PC에 더해줌
                else pc++;
            }
            break;
        case HALT:
            running = 0;
            printf("\033[32mEND\n\033[0m");
            printf(" ** END OF THE PROGRAM ** \n");
            break;
        default:
            printf("\033[32mUnknown instruction\033[0m\n");
            running = 0;
            break;
    }
    my_print_register();
    printf("Press Enter to continue....");
    getchar(); // Step별로 실행하기 위해 getchar()함수를 사용하여 개행을 입력받아 진행.
    printf("\n==========================================================================
==========================\n");
  }
}

void Decode_instructions(void) { // execute_instructions 함수와 유사하지만, 연산을 수행하지 않고 모든
명령어를 순회하며 Assembly를 출력하는 함수.
  int temp = 0;
  int running = 1;
  while (running) {
    int instruction = inst_mem[temp];
    int opcode = (instruction & 0x7F); //TODO
    int funct3 = ((instruction >> 12) & 0x7); //TODO
    int rs1,   // source reg.1
        rs2,   // source reg.2
        rd,    // destination reg.
        imm,   // immediate
        addr;  // base address
    switch (opcode) {
        case ADD:
            rd = (instruction >> 7) & 0x1F;
            rs1 = (instruction >> 15) & 0x1F;
            rs2 = (instruction >> 20) & 0x1F;
            printf("add  x%02d, x%02d, x%02d\n",rd, rs1,rs2); // Assembly 출력.
            temp++;
            break;
        case ADDI:
            rd = (instruction >> 7) & 0x1F;
            rs1 = (instruction >> 15) & 0x1F;
            imm = (instruction >> 20) & 0xFFF;
            if (((instruction >> 31) & 0x1) == 0x1) { imm = (imm | 0xFFFFF000); }
            printf("addi x%02d, x%02d, %d\n",rd, rs1,imm);
            temp++;
            break;
```

```
            case LW:
                rd = (instruction >> 7) & 0x1F;
                rs1 = (instruction >> 15) & 0x1F;
                imm = (instruction >> 20) & 0xFFF;
                if (((instruction >> 31) & 0x1) == 0x1) { imm = (imm | 0xFFFFF000); }
                printf("lw   x%02d, %d(x%02d)\n",rd,imm,rs1);
                temp++;
                break;
            case LUI :
                rd = (instruction >> 7) & 0x1F;
                imm = (((instruction >> 21) & 0x3FF) << 1) | (((instruction >> 20) & 0x1) << 11) | \
                   (((instruction >> 12) & 0xFF) << 12) | (((instruction >> 31) & 0x1) << 20);
                imm = imm << 11;
                printf("lui  x%02d, %d\n",rd,imm);
                temp++;
                break;
            case SW:
                rs1 = (instruction >> 15) & 0x1F;
                rs2 = (instruction >> 20) & 0x1F;
                imm = ((instruction >> 7) & 0x1F) | (((instruction >> 25) & 0x7F) << 5);
                if (((instruction >> 31) & 0x1) == 0x1) { imm = (imm | 0xFFFFF000); }
                printf("sw   x%02d, %d(x%02d)\n",rs2,imm,rs1);
                temp++;
                break;
            case BEQ:
                imm = (((instruction >> 8) & 0xF) << 1) | (((instruction >> 25) & 0x3F) << 5) |\
                 (((instruction >> 7) & 0x1) << 11) | (((instruction >> 31) & 0x1) << 12);
                rs1 = (instruction >> 15) & 0x1F;
                rs2 = (instruction >> 20) & 0x1F;
                if (((instruction >> 31) & 0x1) == 0x1) { imm = (imm | 0xFFFFF000); }
                if (funct3 == 0) { printf("beq  x%02d, x%02d, %d\n",rs1,rs2,imm);}
                else if (funct3 == 4) { printf("blt  x%02d, x%02d, %d\n",rs1,rs2,imm);}
                temp++; // 조건문과 무관하게 PC를 1 증가 시킴
                break;
            case HALT:
                running = 0;
                printf("END\n");
                break;
            default:
                printf("Unknown instruction\n");
                running = 0;
                break;
        }
    }
}

int main(int ac, char **av) {
    ////////     수정금지 영역 시작 (프로그램 B의 주석을 해제하는 수정만 가능)    ///////////
    // Instructions for Program A
    // ISA: ADD, ADDI, LW, BEQ, HALT
    // inst_mem[0] = 0b00000000011000101000001110110011;
    // inst_mem[1] = 0b00000000010100101000001010010011;
    // inst_mem[2] = 0b00000000010100101000001100110011;
    // inst_mem[3] = 0b00000000011000101000001110110011;
    // inst_mem[4] = 0b00010000010100110000000001100011;
    // inst_mem[5] = 0b00000000000001001001000110000011;
```

```
    // inst_mem[6] = 0b11111111111111111111111111111111; // END

    // // Instructions for Program B
    // ISA: ADD, ADDI, LW, BEQ, HALT + (SW, BLT)
    inst_mem[0] = 0b00000000000001000010110000100011; //sw zero, 24(x8)
    inst_mem[1] = 0b00000000000001000010101000100011; //sw zero, 20(x8)
    inst_mem[2] = 0b00000001010001000010011100000011; //lw x14, 20(x8) : curr i
    inst_mem[3] = 0b00000000000000000000011110110111; //lui x15, 0
    inst_mem[4] = 0b00000000010001111100011110010011;  //addi x15, x15, 4
    inst_mem[5] = 0b00000010111001111100000001100011; //blt x15, x14, 32 : 4 < i
    inst_mem[6] = 0b00000001100001000010011110000011; //lw x15, 24(x8)
    inst_mem[7] = 0b00000000010001111100011110010011; //addi x15, x15, 4
    inst_mem[8] = 0b00000000111101000010110000100011; //sw x15, 24(x8)
    inst_mem[9] = 0b00000001010001000010011110000011; //lw x15, 20(x8)
    inst_mem[10] = 0b00000000000101111000011110010011; //addi x15, x15, 1
    inst_mem[11] = 0b00000000111101000010101000100011; //sw x15, 20(x8)
    inst_mem[12] = 0b11111110000000000000110011100011; //beq zero, zero, -40
    inst_mem[13] = 0b00000001100001000010011110000011; //lw x15, 24(x8)
    inst_mem[14] = 0b00000000101001111000011110010011; //addi x15, x15, 10
    inst_mem[15] = 0b00000000111101000010111000100011; //sw x15, 28(x8)
    inst_mem[16] = 0b11111111111111111111111111111111; //END

    // Given Data for Program A and B
    data_mem[0] = 0b00000000000000000000000000000000; // 0
    data_mem[1] = 0b00000000000000000000000000000001; // 1
    data_mem[2] = 0b00000000000000000000000000000010; // 2
    data_mem[3] = 0b00000000000000000000000000000011; // 3
    data_mem[4] = 0b00000000000000000000000000000100; // 4
    data_mem[5] = 0b00000000000000000000000000000101; // 5
    data_mem[6] = 0b00000000000000000000000000000110; // 6
    data_mem[7] = 0b00000000000000000000000000000111; // 7

    // Given Register values for Program A and B
    registers[5] = 10;
    registers[6] = 20;
    registers[7] = 30;

    ////////            수정금지 영역 끝          ///////////

    if (ac != 2) {  // parameter 개수 예외 처리.
        printf("\033[31mplease input the parameter! ex)./test \033[32m1\n\033[0m");
        printf("\033[32m1st parameter: Mode Selection (1: Decode, 2: Execute, 3: Debugging)\n\033
[0m");
        return -1;
    }
    /* String으로 입력받은 argument를 atoi함수를 활용해 integer로 변환하여 Mode 선택 */
    if (atoi(av[1]) == 1) // mode 1 : Decode
        Decode_instructions();
    else if(atoi(av[1]) == 2) // mode 2 : Execute
        execute_instructions();
    else if (atoi(av[1]) == 3) // mode 3 : Debugging
        step_execution();
    else  // parameter의 범위 예외 처리.
        printf("\033[31mError : The first parameter must \033[32mrange from 1 to 3.\n\033[0m");
    return 0;
}
```

3. Part 3 (분석 및 이해)

3.1 코드 설명

이번 과제는 이진코드로 주어진 RISC-V Instruction을 Decode하고 Execution하는 과제이다. 따라서 명령어의 분석을 좀더 편하게 하기 위해 argument를 추가적으로 받아 3가지 모드(Decode, Execute, Debugging)로 작동하도록 코드를 작성하였다. 각 Line에 대한 설명은 코드에 달린 주석으로 대체하고, 이번 Chapter에서는 각 함수의 전체적인 동작방식에 대해 설명하도록 하겠다.

`void my_print_register()  // Register, Data_Memory 에 저장된 값을 출력하는 함수.`

이 함수는 현재 register와 Data Memory에 담긴 값을 출력하는 함수이다. 주어진 초기조건에 의해 register와 Data_Memory의 값들이 대부분 0이기 때문에 알아보기 쉽도록 Escape code, Color Code를 사용하여 0이 아닌 값을 빨간색으로 출력하도록 하였다.

`void execute_instructions(void)  // 주어진 Instruction 을 Execution 하는 함수.`

이 함수는 각 Step에서 수행한 Instruction을 assembly로 출력하고 최종적으로 Register, Memory에 담긴 정보를 출력하는 함수이며 다음과 같은 순서로 동작한다.

1. **`void my_print_register()`** 함수를 호출하여 Register 및 Memory의 초기값을 출력.
2. 현재 PC값에 해당하는 Instruction을 inst_mem에서 가져와서 instruction 변수에 저장.
3. **아래의 그림을 참고**하여 instruction을 bit-shifting과 Masking을 통해 opcode 및 funct3 추출.

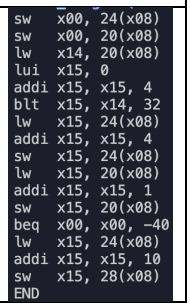
32-bit RISC-V instruction formats

Format	Bit 31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/ register	funct7							rs2					rs1					funct3			rd					opcode						
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						
Upper immediate	imm[31:12]																				rd					opcode						
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						

4. 현재 STEP 및 PC값을 출력.
5. 앞서 구한 opcode를 통해 Instruction의 Type을 결정(R-type, S-type, SB-type, U-type, I-type, HALT)
6. switch문을 통해 case를 지정하고, 필요한 Register, imm를 위의 그림을 참고하여 bit-shifting, Masking, Sign-Extension하여 결정함.
7. 연산을 수행함.
8. 연산된 결과를 출력하고 PC값을 증가.
9. HALT 명령어, Unknown 명령어가 발생하면 flag를 0으로 Loop를 탈출.
10. 최종 Register 및 Memory 값을 출력하고 프로그램 종료.

`void Decode_instructions(void)`

이 함수는 근본적으로 위에 작성한 **`void execute_instructions( )`**함수와 동일한 매커니즘으로 작동하지만, 연산을 수행하지는 않으며 단순히 Inst_mem의 모든 명령어 set을 순회하며 주어진 이진코드를 assembly code로 변환하여 출력하는 함수이다. 실행 결과를 보기 전에 assembly code를 보고 결과를 예측해보기 위해서 작성하였다.



```
void step_execution(void)
```

이 함수 또한 근본적으로 위에 작성한 **void execute_instructions()** 함수와 동일한 매커니즘으로 작동하지만, 코드 작성 과정에서 Debugging을 위해 while문을 반복할 때 마다 **getchar()함수**를 활용하여 개행을 입력 받아 단계적으로 현재 Step, PC, assembly code, 수행 전 Target Register 및 Memory 값, 실행 결과, 수행 후 Register 및 Memory의 값을 출력하도록 작성한 함수이다.

```
STEP : 10
PC : 10
addi x15, x15, 1
x15 = 0 , x15 = 0
Result : x15 = 1
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 00 | x15 : 01
| x16 : 00 | x17 : 00 | x18 : 00 | x19 : 00 | x20 : 00 | x21 : 00 | x22 : 00 | x23 : 00
| x24 : 00 | x25 : 00 | x26 : 00 | x27 : 00 | x28 : 00 | x29 : 00 | x30 : 00 | x31 : 00
Current Data_Memory Values
| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 00 | MEM06 : 04 | MEM07 : 07
| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
| MEM16 : 00 | MEM17 : 00 | MEM18 : 00 | MEM19 : 00 | MEM20 : 00 | MEM21 : 00 | MEM22 : 00 | MEM23 : 00
| MEM24 : 00 | MEM25 : 00 | MEM26 : 00 | MEM27 : 00 | MEM28 : 00 | MEM29 : 00 | MEM30 : 00 | MEM31 : 00
| MEM32 : 00 | MEM33 : 00 | MEM34 : 00 | MEM35 : 00 | MEM36 : 00 | MEM37 : 00 | MEM38 : 00 | MEM39 : 00
```

```
int main(int ac, char **av)
```

main함수는 다음과 같은 순서로 동작한다.

1. Program A, B에 해당하는 instruction memory 할당.
2. Program A, B에 사용되는 Data_mem, Register값 할당.
3. mode 선택에 따른 argument를 검사하여 예외처리.
4. 선택한 mode에 맞는 함수를 실행.

bit-shifting 및 masking을 통해 원하는 값을 추출하는 과정을 추가적으로 설명하면 다음과 같다.

Format	Bit																													
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						

가장 복잡했던 Branch 명령어의 imm 값을 얻는 과정은 다음과 같다.

```
imm = (((instruction >> 8) & 0xF) << 1) | (((instruction >> 25) & 0x3F) << 5) |\
        (((instruction >> 7) & 0x1) << 11) | (((instruction >> 31) & 0x1) << 12);
```

- imm[4:1]의 값을 얻기 위해 위의 표를 참고하여 8만큼 right shift 후 0xF와 AND연산을 하여 Masking하고 1만큼 left-shift하여 위치를 맞춰준다.
- imm[10:5]의 값을 얻기 위해 위의 표를 참고하여 25만큼 right shift 후 0x3F와 AND연산을 하여 Masking하고 5만큼 left-shift하여 위치를 맞춰준다.
- imm[11]의 값을 얻기 위해 위의 표를 참고하여 7만큼 right shift 후 0x1와 AND연산을 하여 Masking하고 11만큼 left-shift하여 위치를 맞춰준다.
- imm[12]의 값을 얻기 위해 위의 표를 참고하여 31만큼 right shift 후 0x1와 AND연산을 하여 Masking하고 12만큼 left-shift하여 위치를 맞춰준다.

```
if (((instruction >> 31) & 0x1) == 0x1) { imm = (imm | 0xFFFFF000); }
```

- Sign-extension이 필요한 명령어의 경우 MSB가 1일 경우 Masking하여 앞부분을 1로 채워준다.

3.2 Program A 동작 예측 및 결과 분석.

(1) 동작 예측

Assembly code	Initial Values

add x07, x05, x06
addi x05, x05, 5
add x06, x05, x05
add x07, x05, x06
beq x06, x05, 256
lw x06, 0(x18)
END
	Initial values
	Current Register Values
	| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
	| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 00 | x15 : 00
	| x16 : 00 | x17 : 00 | x18 : 00 | x19 : 00 | x20 : 00 | x21 : 00 | x22 : 00 | x23 : 00
	| x24 : 00 | x25 : 00 | x26 : 00 | x27 : 00 | x28 : 00 | x29 : 00 | x30 : 00 | x31 : 00
	Current Data_Memory Values
	| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 05 | MEM06 : 06 | MEM07 : 07
	| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
	| MEM16 : 00 | MEM17 : 00 | MEM18 : 00 | MEM19 : 00 | MEM20 : 00 | MEM21 : 00 | MEM22 : 00 | MEM23 : 00
	| MEM24 : 00 | MEM25 : 00 | MEM26 : 00 | MEM27 : 00 | MEM28 : 00 | MEM29 : 00 | MEM30 : 00 | MEM31 : 00
	| MEM32 : 00 | MEM33 : 00 | MEM34 : 00 | MEM35 : 00 | MEM36 : 00 | MEM37 : 00 | MEM38 : 00 | MEM39 : 00

Assembly code와 초기 Register 및 Memory값을 통해 실행 결과를 예측해보면 다음과 같다.

1. Memory에 Write하는 instruction이 없기 때문에 Memory의 값은 변화하지 않을 것이다.
2. x07 = 10 + 20 = 30
3. x06 = 10 + 5 = 15
4. x06 = 10 + 10 = 20
5. x07 = x05 + x06 = 45
6. if (x06 == x05) {pc += 256 / 4} else {pc++;} -> x06과 x05값이 다르기 때문에 단순히 pc++으로 동작.
7. x06에 x18(base address)에서 0(offset)만큼 떨어진 값을 load -> x06 = data_mem[0] = 0.

따라서, x05 = 10, x06 = 0, x07 = 45 값을 갖을 것으로 예상된다.

(2) 결과 분석

```
Initial values
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 00 | x15 : 00
| x16 : 00 | x17 : 00 | x18 : 00 | x19 : 00 | x20 : 00 | x21 : 00 | x22 : 00 | x23 : 00
| x24 : 00 | x25 : 00 | x26 : 00 | x27 : 00 | x28 : 00 | x29 : 00 | x30 : 00 | x31 : 00
Current Data_Memory Values
| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 05 | MEM06 : 06 | MEM07 : 07
| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
| MEM16 : 00 | MEM17 : 00 | MEM18 : 00 | MEM19 : 00 | MEM20 : 00 | MEM21 : 00 | MEM22 : 00 | MEM23 : 00
| MEM24 : 00 | MEM25 : 00 | MEM26 : 00 | MEM27 : 00 | MEM28 : 00 | MEM29 : 00 | MEM30 : 00 | MEM31 : 00
| MEM32 : 00 | MEM33 : 00 | MEM34 : 00 | MEM35 : 00 | MEM36 : 00 | MEM37 : 00 | MEM38 : 00 | MEM39 : 00
```

주어진 초기 Register 및 Memory값은 위와 같다.

```
STEP : 0
PC : 0
add x07, x05, x06
x07 = 30, x05 = 10, x06 = 20
Result : x07 = 30
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 00 | x15 : 00
| x16 : 00 | x17 : 00 | x18 : 00 | x19 : 00 | x20 : 00 | x21 : 00 | x22 : 00 | x23 : 00
| x24 : 00 | x25 : 00 | x26 : 00 | x27 : 00 | x28 : 00 | x29 : 00 | x30 : 00 | x31 : 00
```

Add 명령어를 수행하여 x07값이 30으로 바뀐 것을 확인할 수 있다.

```
STEP : 1
PC : 1
addi x05, x05, 5
x05 = 10 , x05 = 10
Result : x05 = 15
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 15 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 00 | x15 : 00
| x16 : 00 | x17 : 00 | x18 : 00 | x19 : 00 | x20 : 00 | x21 : 00 | x22 : 00 | x23 : 00
| x24 : 00 | x25 : 00 | x26 : 00 | x27 : 00 | x28 : 00 | x29 : 00 | x30 : 00 | x31 : 00
```

Addi 명령어를 수행하여 x05값이 15로 바뀐 것을 확인할 수 있다.

```
STEP : 2
PC : 2
add x06, x05, x05
x06 = 20, x05 = 15, x05 = 15
Result : x06 = 30
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 15 | x06 : 30 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 00 | x15 : 00
| x16 : 00 | x17 : 00 | x18 : 00 | x19 : 00 | x20 : 00 | x21 : 00 | x22 : 00 | x23 : 00
| x24 : 00 | x25 : 00 | x26 : 00 | x27 : 00 | x28 : 00 | x29 : 00 | x30 : 00 | x31 : 00
```

Add 명령어를 수행하여 x06값이 30으로 바뀐 것을 확인할 수 있다.

```
STEP : 3
PC : 3
add x07, x05, x06
x07 = 30, x05 = 15, x06 = 30
Result : x07 = 45
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 15 | x06 : 30 | x07 : 45
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 00 | x15 : 00
| x16 : 00 | x17 : 00 | x18 : 00 | x19 : 00 | x20 : 00 | x21 : 00 | x22 : 00 | x23 : 00
| x24 : 00 | x25 : 00 | x26 : 00 | x27 : 00 | x28 : 00 | x29 : 00 | x30 : 00 | x31 : 00
```

Add 명령어를 수행하여 x07값이 45로 바뀐 것을 확인할 수 있다.

```
STEP : 4
PC : 4
beq x06, x05, 256
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 15 | x06 : 30 | x07 : 45
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 00 | x15 : 00
| x16 : 00 | x17 : 00 | x18 : 00 | x19 : 00 | x20 : 00 | x21 : 00 | x22 : 00 | x23 : 00
| x24 : 00 | x25 : 00 | x26 : 00 | x27 : 00 | x28 : 00 | x29 : 00 | x30 : 00 | x31 : 00
```

X06과 x05값이 다르기 때문에 branch가 일어나지 않고 단순히 pc가 증가할 것이다.

```
STEP : 5
PC : 5
lw x06 ,0(x18)
x06 = 30, data_mem[0] = 0
Result : x06 = 0
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 15 | x06 : 00 | x07 : 45
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 00 | x15 : 00
| x16 : 00 | x17 : 00 | x18 : 00 | x19 : 00 | x20 : 00 | x21 : 00 | x22 : 00 | x23 : 00
| x24 : 00 | x25 : 00 | x26 : 00 | x27 : 00 | x28 : 00 | x29 : 00 | x30 : 00 | x31 : 00
```

Pc값이 1증가 했음을 확인할 수 있고, x06에 0이 load되어서 x06값이 0으로 바뀐 것을 확인할 수 있다.

```
STEP : 6
PC : 6
END
 ** END OF THE PROGRAM **
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 15 | x06 : 00 | x07 : 45
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 00 | x15 : 00
| x16 : 00 | x17 : 00 | x18 : 00 | x19 : 00 | x20 : 00 | x21 : 00 | x22 : 00 | x23 : 00
| x24 : 00 | x25 : 00 | x26 : 00 | x27 : 00 | x28 : 00 | x29 : 00 | x30 : 00 | x31 : 00
Current Data_Memory Values
| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 05 | MEM06 : 06 | MEM07 : 07
| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
| MEM16 : 00 | MEM17 : 00 | MEM18 : 00 | MEM19 : 00 | MEM20 : 00 | MEM21 : 00 | MEM22 : 00 | MEM23 : 00
| MEM24 : 00 | MEM25 : 00 | MEM26 : 00 | MEM27 : 00 | MEM28 : 00 | MEM29 : 00 | MEM30 : 00 | MEM31 : 00
| MEM32 : 00 | MEM33 : 00 | MEM34 : 00 | MEM35 : 00 | MEM36 : 00 | MEM37 : 00 | MEM38 : 00 | MEM39 : 00
```

최종적인 Register, Memory값을 확인해보면 **3.2.(1)의 결과 예측**과 같이 Memory의 값들은 변화하지 않았고, **x05 = 10, x06 = 0, x07 = 45** 값을 갖을 것을 확인할 수 있다.

3.3 Program B 동작 예측 및 결과 분석.

(1) 동작 예측

Assembly code	Initial Values

sw x00, 24(x08)
sw x00, 20(x08)
lw x14, 20(x08)
lui x15, 0
addi x15, x15, 4
blt x15, x14, 32
lw x15, 24(x08)
addi x15, x15, 4
sw x15, 24(x08)
lw x15, 20(x08)
addi x15, x15, 1
sw x15, 20(x08)
beq x00, x00, -40
lw x15, 24(x08)
addi x15, x15, 10
sw x15, 28(x08)
END
	Initial values
	Current Register Values
	| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
	| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 00 | x15 : 00
	| x16 : 00 | x17 : 00 | x18 : 00 | x19 : 00 | x20 : 00 | x21 : 00 | x22 : 00 | x23 : 00
	| x24 : 00 | x25 : 00 | x26 : 00 | x27 : 00 | x28 : 00 | x29 : 00 | x30 : 00 | x31 : 00
	Current Data_Memory Values
	| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 05 | MEM06 : 06 | MEM07 : 07
	| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
	| MEM16 : 00 | MEM17 : 00 | MEM18 : 00 | MEM19 : 00 | MEM20 : 00 | MEM21 : 00 | MEM22 : 00 | MEM23 : 00
	| MEM24 : 00 | MEM25 : 00 | MEM26 : 00 | MEM27 : 00 | MEM28 : 00 | MEM29 : 00 | MEM30 : 00 | MEM31 : 00
	| MEM32 : 00 | MEM33 : 00 | MEM34 : 00 | MEM35 : 00 | MEM36 : 00 | MEM37 : 00 | MEM38 : 00 | MEM39 : 00

Assembly code와 초기 Register 및 Memory값을 통해 실행 결과를 예측해보면 다음과 같다.

가. 초기화 단계

1. sw x00, 24(x08) : x08 레지스터에 있는 주소의 24바이트 오프셋 위치에 있는 값을 **0으로 초기화** 시킨다.
2. sw x00, 20(x08) : x08 레지스터에 있는 주소의 20바이트 오프셋 위치에 있는 값을 **0으로 초기화** 시킨다.

나. LOOP 단계

3. lw x14, 20(x08) : x08 레지스터에 있는 주소의 20바이트 오프셋 위치에 있는 값을 x14 레지스터에 저장. **(조건문 검사를 위해 12단계에서 i값을 저장해둔 data_mem[5]의 값을 load하는 동작)**
4. lui x15, 0 : x15 레지스터의 상위 20비트 값을 0으로 설정.
5. addi x15, x15, 4 : x15값을 4로 설정. **(조건문 검사를 위해 x15를 4로 설정하는 동작)**
6. blt x15, x14, 32 : x14가 4보다 크면 현재 PC로부터 32바이트 오프셋 만큼 분기. **(LOOP 검사)**
7. lw x15, 24(x08) : x08 레지스터에 있는 주소의 24바이트 오프셋 위치에 있는 값을 x15 레지스터에 저장.
8. addi x15, x15, 4 : x15에 4를 더함.
9. sw x15, 24(x08) : x08 레지스터에 있는 주소의 24바이트 오프셋 위치에 x15 레지스터의 값을 저장. **(data_mem[6]의 값을 4만큼 증가시키는 동작)**
10. lw x15, 20(x08) : x08 레지스터에 있는 주소의 20바이트 오프셋 위치에 있는 값을 x15 레지스터에 저장.
11. addi x15, x15, 1 : x15 레지스터의 값을 1만큼 증가.
12. sw x15, 20(x08) : x08 레지스터에 있는 주소의 20바이트 오프셋 위치에 x15 레지스터의 값을 저장. **(i값을 1만큼 증가시키고 data_mem[5]에 저장하는 동작)**
13. beq x00, x00, -40 : 무조건 3번 단계로 돌아가서 LOOP문 다시 수행. **(무한루프)**

다. LOOP 탈출

14. lw x15, 24(x08) : x08 레지스터에 있는 주소의 24바이트 오프셋 위치에 있는 값을 x15 레지스터에 저장.
15. addi x15, x15, 10 : x15 레지스터의 값을 10만큼 증가.
16. sw x15, 28(x08) : x08 레지스터에 있는 주소의 28바이트 오프셋 위치에 x15 레지스터의 값을 저장.
17. HALT : 프로그램 종료.

Program B의 동작을 분석하자면 위와 같다. **반복문을 총 5번 수행** 하기 때문에 **x14 = 5가 될 것으로 예상**되며, x15의 값은 반복문을 수행할 때 마다 data_mem[6]가 0으로 시작하여 4만큼 증가 하여 20, 루프를 탈출 후 10 증가하기 때문에 **x15 = 30으로 예상**된다.

Memory의 경우 **data_mem[5]에는 i값이 저장되어 5**, **data_mem[6]에는 20**, **data_mem[7]에는 20에 10을더한 30**이 저장될 것으로 예상된다.

(2) 결과 분석

```
Initial values
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 00 | x15 : 00
| x16 : 00 | x17 : 00 | x18 : 00 | x19 : 00 | x20 : 00 | x21 : 00 | x22 : 00 | x23 : 00
| x24 : 00 | x25 : 00 | x26 : 00 | x27 : 00 | x28 : 00 | x29 : 00 | x30 : 00 | x31 : 00
Current Data_Memory Values
| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 05 | MEM06 : 06 | MEM07 : 07
| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
| MEM16 : 00 | MEM17 : 00 | MEM18 : 00 | MEM19 : 00 | MEM20 : 00 | MEM21 : 00 | MEM22 : 00 | MEM23 : 00
| MEM24 : 00 | MEM25 : 00 | MEM26 : 00 | MEM27 : 00 | MEM28 : 00 | MEM29 : 00 | MEM30 : 00 | MEM31 : 00
| MEM32 : 00 | MEM33 : 00 | MEM34 : 00 | MEM35 : 00 | MEM36 : 00 | MEM37 : 00 | MEM38 : 00 | MEM39 : 00
```

주어진 초기 Register 및 Memory값은 위와 같다.

```
STEP : 0
PC : 0
sw x00, 24(x08)
x00 = 0, data_mem[6] = 6
Result : data_mem[6] = 0
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 00 | x15 : 00
Current Data_Memory Values
| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 05 | MEM06 : 00 | MEM07 : 07
| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
STEP : 1
PC : 1
sw x00, 20(x08)
x00 = 0, data_mem[5] = 5
Result : data_mem[5] = 0
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 00 | x15 : 00
Current Data_Memory Values
| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 00 | MEM06 : 00 | MEM07 : 07
| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
```

data_mem[5], data_mem[6]가 0으로 초기화 된것을 확인할 수 있다.

```
STEP : 2
PC : 2
lw x14 ,20(x08)
x14 = 0, data_mem[5] = 0
Result : x14 = 0
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 00 | x15 : 00
Current Data_Memory Values
| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 00 | MEM06 : 00 | MEM07 : 07
| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
```

x14에 data_mem[5]의 값이 load된것을 확인할 수 있다.

```
STEP : 3
PC : 3
lui x15, 0
x15 = 0
Result : x15 = 0
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 00 | x15 : 00
Current Data_Memory Values
| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 00 | MEM06 : 00 | MEM07 : 07
| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
```

```
STEP : 4
PC : 4
addi x15, x15, 4
x15 = 0 , x15 = 0
Result : x15 = 4
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 00 | x15 : 04
Current Data_Memory Values
| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 00 | MEM06 : 00 | MEM07 : 07
| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
```

STEP3, 4에 의해 x15의 값이 4가 되는것을 확인할 수 있다.

```
STEP : 5
PC : 5
blt x15, x14, 32
x15 = 4, x14 = 0
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 00 | x15 : 04
Current Data_Memory Values
| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 00 | MEM06 : 00 | MEM07 : 07
| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
```

Loop 조건문 검사를 수행하는 단계로 x15 = 4 > 0이기 때문에 반복문을 수행할 것으로 예상된다.

```
STEP : 6
PC : 6
lw x15 ,24(x08)
x15 = 4, data_mem[6] = 0
Result : x15 = 0
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 00 | x15 : 00
Current Data_Memory Values
| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 00 | MEM06 : 00 | MEM07 : 07
| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
```

예상과 같이 분기가 일어나지 않았고, x15에 data_mem[6]의 값을 load하는 것을 확인할 수 있다.

```
STEP : 7
PC : 7
addi x15, x15, 4
x15 = 0 , x15 = 0
Result : x15 = 4
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 00 | x15 : 04
Current Data_Memory Values
| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 00 | MEM06 : 00 | MEM07 : 07
| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
STEP : 8
PC : 8
sw x15, 24(x08)
x15 = 4, data_mem[6] = 0
Result : data_mem[6] = 4
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 00 | x15 : 04
Current Data_Memory Values
| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 00 | MEM06 : 04 | MEM07 : 07
| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
```

STEP 7,8에 의해 4*1의 값이 data_mem[6]에 저장되는것을 확인할 수 있다.

```
STEP : 9
PC : 9
lw x15 ,20(x08)
x15 = 4, data_mem[5] = 0
Result : x15 = 0
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 00 | x15 : 00
Current Data_Memory Values
| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 00 | MEM06 : 04 | MEM07 : 07
| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
STEP : 10
PC : 10
addi x15, x15, 1
x15 = 0 , x15 = 0
Result : x15 = 1
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 00 | x15 : 01
Current Data_Memory Values
| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 00 | MEM06 : 04 | MEM07 : 07
| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
STEP : 11
PC : 11
sw x15, 20(x08)
x15 = 1, data_mem[5] = 0
Result : data_mem[5] = 1
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 00 | x15 : 01
Current Data_Memory Values
| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 01 | MEM06 : 04 | MEM07 : 07
| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
```

Data_mem[5]에 i값이 저장되는 모습을 확인할 수 있다.

```
STEP : 12
PC : 12
beq x00, x00, -40
x00 = 0, x00 = 0
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 00 | x15 : 01
Current Data_Memory Values
| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 01 | MEM06 : 04 | MEM07 : 07
| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
STEP : 13
PC : 2
lw x14 ,20(x08)
x14 = 0, data_mem[5] = 1
Result : x14 = 1
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 01 | x15 : 01
Current Data_Memory Values
| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 01 | MEM06 : 04 | MEM07 : 07
| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
```

STEP 12에 의해 PC가 10만큼 감소하여 PC=2로 변화하였음을 확인할 수 있고 다시 반복문 검사를 위해 data_mem[5]에 저장된 i값을 x14에 load하는것을 확인할 수 있다.

```
STEP : 14
PC : 3
lui x15, 0
x15 = 1
Result : x15 = 0
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 01 | x15 : 00
Current Data_Memory Values
| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 01 | MEM06 : 04 | MEM07 : 07
| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
STEP : 15
PC : 4
addi x15, x15, 4
x15 = 0 , x15 = 0
Result : x15 = 4
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 01 | x15 : 04
Current Data_Memory Values
| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 01 | MEM06 : 04 | MEM07 : 07
| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
STEP : 16
PC : 5
blt x15, x14, 32
x15 = 4, x14 = 1
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 01 | x15 : 04
Current Data_Memory Values
| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 01 | MEM06 : 04 | MEM07 : 07
| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
```

앞선 과정과 동일하게 반복문 수행을 위한 조건문을 검사하고 있음을 확인할 수 있다.

```
STEP : 17
PC : 6
lw x15 ,24(x08)
x15 = 4, data_mem[6] = 4
Result : x15 = 4
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 01 | x15 : 04
Current Data_Memory Values
| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 01 | MEM06 : 04 | MEM07 : 07
| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
STEP : 18
PC : 7
addi x15, x15, 4
x15 = 4 , x15 = 4
Result : x15 = 8
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 01 | x15 : 08
Current Data_Memory Values
| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 01 | MEM06 : 04 | MEM07 : 07
| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
STEP : 19
PC : 8
sw x15, 24(x08)
x15 = 8, data_mem[6] = 4
Result : data_mem[6] = 8
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 01 | x15 : 08
Current Data_Memory Values
| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 01 | MEM06 : 08 | MEM07 : 07
| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
```

마찬가지로 앞선 과정과 동일하게 data_mem[6]에 저장된 값을 4만큼 증가시켜 다시 저장하는 모습을 확인할 수 있다.

```
STEP : 20
PC : 9
lw x15 ,20(x08)
x15 = 8, data_mem[5] = 1
Result : x15 = 1
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 01 | x15 : 01
Current Data_Memory Values
| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 01 | MEM06 : 08 | MEM07 : 07
| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
STEP : 21
PC : 10
addi x15, x15, 1
x15 = 1 , x15 = 1
Result : x15 = 2
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 01 | x15 : 02
Current Data_Memory Values
| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 01 | MEM06 : 08 | MEM07 : 07
| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
STEP : 22
PC : 11
sw x15, 20(x08)
x15 = 2, data_mem[5] = 1
Result : data_mem[5] = 2
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 01 | x15 : 02
Current Data_Memory Values
| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 02 | MEM06 : 08 | MEM07 : 07
| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
```

I값을 1만큼 증가시켜 data_mem[5]에 저장하는 것을 확인할 수 있다.

```
STEP : 23
PC : 12
beq x00, x00, -40
x00 = 0, x00 = 0
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 01 | x15 : 02
Current Data_Memory Values
| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 02 | MEM06 : 08 | MEM07 : 07
| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
STEP : 24
PC : 2
lw x14 ,20(x08)
x14 = 1, data_mem[5] = 2
Result : x14 = 2
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 02 | x15 : 02
Current Data_Memory Values
| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 02 | MEM06 : 08 | MEM07 : 07
| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
```

마찬가지로 다시 반복문 검사를 위해 PC를 10만큼 감소시키고 반복문 검사를 수행하는것을 확인할 수 있다. 이와 같은 과정을 총 **5번 반복한다.** 파일 용량상의 문제로 중간 과정은 생략하고 마지막 반복문을 살펴보도록 하겠다.

```
STEP : 60
PC : 5
blt x15, x14, 32
x15 = 4, x14 = 5
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 05 | x15 : 04
Current Data_Memory Values
| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 05 | MEM06 : 20 | MEM07 : 07
| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
```

드디어 x14의 값이 5가 되어 더이상 반복문을 반복하지 않고 분기가 발생할 것으로 예상된다.

```
STEP : 61
PC : 13
lw x15 ,24(x08)
x15 = 4, data_mem[6] = 20
Result : x15 = 20
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 05 | x15 : 20
Current Data_Memory Values
| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 05 | MEM06 : 20 | MEM07 : 07
| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
```

조건을 충족하여 PC값이 8만큼 증가하였음을 확인할 수 있고 반복문이 총 5회 수행되어 data_mem[6]의 값이 20이 되었고 이를 x15에 load하였음을 확인할 수 있다.

```
STEP : 62
PC : 14
addi x15, x15, 10
x15 = 20 , x15 = 20
Result : x15 = 30
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 05 | x15 : 30
Current Data_Memory Values
| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 05 | MEM06 : 20 | MEM07 : 07
| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
STEP : 63
PC : 15
sw x15, 28(x08)
x15 = 30, data_mem[7] = 7
Result : data_mem[7] = 30
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 05 | x15 : 30
Current Data_Memory Values
| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 05 | MEM06 : 20 | MEM07 : 30
| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
```

X15에 10을 더해 30이 x15에 저장된것을 확인할 수 있고, 해당 값이 data_mem[7]에 저장된 것을 확인할 수 있다.

```
STEP : 64
PC : 16
END
 ** END OF THE PROGRAM **
Current Register Values
| x00 : 00 | x01 : 00 | x02 : 00 | x03 : 00 | x04 : 00 | x05 : 10 | x06 : 20 | x07 : 30
| x08 : 00 | x09 : 00 | x10 : 00 | x11 : 00 | x12 : 00 | x13 : 00 | x14 : 05 | x15 : 30
Current Data_Memory Values
| MEM00 : 00 | MEM01 : 01 | MEM02 : 02 | MEM03 : 03 | MEM04 : 04 | MEM05 : 05 | MEM06 : 20 | MEM07 : 30
| MEM08 : 00 | MEM09 : 00 | MEM10 : 00 | MEM11 : 00 | MEM12 : 00 | MEM13 : 00 | MEM14 : 00 | MEM15 : 00
```

마지막 HALT명령어를 만나 프로그램이 종료되었음을 확인할 수 있으며, **3.3.(1)의 동작 예측**과 동일한 결과값을 가지는것을 확인할 수 있다.