**C language VAX Assembly language**

a = b MOV b, a

b = c + d \* e MPY d, e, b

ADD c, b, b

goto LBL BR LBL

트랜지스터 => Vin이 0이면 끊어져서 Vout에 1이 출력됨

**The Majority Sum-of-product (SOP) Form**

A B C F

0 0 0 0

0 0 1 0

0 1 0 0

0 1 1 1

1 0 0 0

1 0 1 1

1 1 0 1

1 1 1 1

=> 1과 0 중에서 더 많은 것이 출력으로 나간다 (Majority function 특징)

=> 1인 것만 긁어 모은다 (A`BC + AB`C + ABC` + ABC )

=> AND 후 OR => AND OR 2 level circuit

**POS Form**

=>OR후 AND => OR AND 2 level circuit

=> 0인 것만 긁어 모은다

Multiplexer : 다수의 입력중에 하나를 골라서 출력시키는 장치이다

<Combinational Logic> - 입력이 들어오면 그거에만 의존해서 출력이 바로 나온다

**A K-Map of the Majority Function**

-작은 짝수 단위로 묶어야 한다 (2개, 4개, ....)

-코너에 있는 것은 붙어 있는 것으로 생각하면 된다

16 -1 MUX 를 3 \* 4 – 1 MUX로 만드는 법 = > 4주차 15분

16 – 1 MUX는 Depth가 2이고, 3 개의 4 -1 MUX는 Depth가 4이다

=>따라서 , 3개의 4 – 1 mux가 propagation delay가 더 크지만 회로가 더 간단한 장점이 있다

<Sequential Logic> - 입력이 들어오면 이전 상태에 기반해서

-Flip flop 이라는 메모리 성분이 존재한다

-대표적인 예가 Vending machine 이 있다

Finite state machine Design : The modulo - 4 Counter

-modulo 4 Counter : 4가지만 쓸 수 있다 (ex. 0 1 2 3 0 1 2 3.... => 2bit면 된다)

AND , OR => Depth 2

Sequential Logic :

Flip flop => master slave(2개)구조, polling edge sensitive

latch => 1개의 구조, clock level sensitive

D, SR, JK, T Flip flop

-JK Flip flop은 SR Flip flop과 똑같은데 동시에 1이 되는것을 허용하고, 값이 반전된다

Tri-state Buffers : 공집합 같이 생긴거

-버스형태에서 하나의 회로가 출력을 하고 있을 때, 버스에 대한 공유가 문제없이 되기 위하여 회로적으로 분리시길 때 사용 (high impedence 상태)

Modulo 8 (3-bit) Ripple counter

-0 ~ 7 까지 세고 다시 0 으로 되돌아 간다

-T Flip flop : 게이트에 1이라는 값을 고정적으로 준다

=>Flip Flop은 CLK 신호가 들어올 때(Falling edge) 마다 Q 상태가 반전된다

br endloop : "endloop 이라는 곳으로 분기하시오 " 라는 의미

Accessing Memory – Reading from Memory

1) CPU가 읽고 싶은 메모리의 주소, R(Control Signal) 명령어를 Address bus 로 Main memory에 전달한다

2) Main memory는 해당 주소에 있는 값을 Data bus로 CPU에게 전달한다

3) Main memory는 CPU에게 COMPLETE Signal (Control Signal) 을 보낸다

=> 유일하게 양방향 인것은 Data bus 이다

Writing to memory

1) CPU가 쓸 주소 , 쓸 내용, W 명령어를 전달한다

2) 메모리는 해당 주소지로 가서 값을 쓴다

3) 완료하면 메모리는 COMPLETE Signal을 쓴다

Instruction Can be deviced into 3 Classes

1) Data movement instruction

-Load : Read 명령어 (메모리->레지스터)

-Store : Write 명령어 (레지스터->메모리)

2) ALU instruction

-Add, Sub, Shift, ....

3) Branch instructions ( Control flow instructions )

-Condition codes 가 branch 여부를 결정한다

-C : carry/borrow

-N : 연산의 결과가 음수일 때

-V : 연산의 결과가 오버플로우 시

-Z : 연산의 결과가 0일 때

=> 해당 조건을 만족하면 1로 flag 된다

Condition codes => C N V Z

-Br Loc, Brz Loc2

ex) C language => if NUM = 5 then SET = 7

Assembly => CMP.W #5, NUM

BNE L1

MOV.W #7, SET

3 & 2 & 1, 0 Address instructions

-명령어에서 메모리 주소를 몇개 쓰느냐에 따라 분류 (3개 ? 2개 ? 1개? 0개 ?)

-두개의 피연산자, 1개의 결과를 가지는 상황을 가정한다

**1) 3 Address instruction**

- R <- Op1 op Op2 (op1, op2, R이 전부 메모리 주소일 때)

**2) 2 Address instruction**

- Op2 <- Op1 op Op2 (피연산자 Op1, Op2 계선결과를 oP2에 덮어쓴다)

**3) 1 Address instruction**

- Acc <- Acc op Op1

-Special register인 accumulator register을 사용한다. (accumulator register은 CPU 안에 있다)

- 결과를 다시 accumulator에 집어넣는다

**4) 0 Address instruction**

-메모리 주소를 안쓰니까 CPU Register만 사용하는 형태이다

-TOS <- TOS op SOS (Top of stack, Second of Stack 을 빼내서 연산해서 결과를 다시 맨 위에다가 집어넣는다)

* 사용하는 operation

push (메모리 주소에 있는 값을 스택으로 가져오는 역할)

pop( top에 있는 것을 메모리로 보내는 역할)

**5) 4 Address instruction** => 피연산자 2개, 결과저장 메모리주소 1개, 다음 명령어 주소 1개 구조

-add Res, Op1, Op2, Nexti (Res <- Op1 + Op2)

-Op1, Op2를 더해서 Res에 집어넣고 Nexti 가 가리키는 다음 명령어 주소로 가라

- 2, 3 address instruction과 다른 것은 4 address instruction은 instruction format에 NextiAddr이 있지만, 2, 3 address instruction은 다음 명령어 주소(Nexti)가 Program counter에 존재한다

-1 Address instructions은 Accumulator을 가져오고 계산된 결과를 저장하는 별도의 2개의 명령어가 필요하다

=> LDA opAddr , STA opAddr

=> 메모리(opAddr)에 어떤 값을 Accumulator에서 LDA(읽기) 해서 연산을 하고 다시 메모리로 STA(쓰기) 한다

**a = (b + c ) \* d – e 라면**

**1) 3 -Address (메모리주소 3개를 쓴다)**

add a, b, c

mpy a, a, d

sub a, a, e

**2) 2 -Address**

**load** a, b => b를 a에다가 집어넣어라

add a, c

mpy a, d

sub a, e

**3) Accumulator (메모리주소 1개 쓴다)**

**lda** b => Accumulator에 b를 집어넣어라

add c

mpy d

sub e

**sta** a => Accumulator에 있는 것을 빼다가 메모리 a에 집어넣어라

**4) stack (메모리주소 0개 쓴다)**

push b

push c

add

push d

mpy

push e

sub

pop a

* load R8, Op1 => 메모리에서 레지스터로 값을 가져온다
* sto R1, Op2Addr => 레지스터에서 메모리로 값을 보낸다 (store 명령어)

-오직 load, store 명령어만이 메모리를 참조할 수가 있다

-컴파일 타임에서는 전역 변수의 주소가 결정되고, 실행시간(런타임) 에서는 포인터와 지역 변수 주소가 결정된다

**1 ) struct a** { : 구조체

int age;

char name[10];

int count

} \*rec;

**rec : full address variable**

count : filed name ; fixed byte offset

**2) v[i]** : 배열

**v** : fixed base address of array; **full address constant**

i : name of variable index; no larger than array size

**Common Addressing Modes**

**1) immediate addressing : 명령어가 상수를 포함함**

2) Direct addressing : 명령어가 피연산자의 주소를 가지고 있음

3) inDirect addressing : 명령어가 어떤 주소를 가지고 있는데 글로 가면 피연산자의 주소가 있다

4) Register direct addressing : 명령어가 레지스터를 가지고 있는데 거기에 피연산자가 있다

5) Register indirect addressing : 명령어가 레지스터가 있는데 3번과 같은 경우

6) Displacement addressing : address of operand = register + constant (배열, 구조체 같은 경우)

**7) Relative addressing : 명령어가 PC를 포함함**; address of operand = PC + constant

**SRC( Simple RISC Computer )**

-32비트 범용 레지스터(R0 ~ R31) 32개를 가지고 있다

-32비트 PC, IR

-2^32개의 memory address space

R[7] : contents of register 7

**SRC Basic Instruction Format ( r : 레지스터, c : 상수 )**

1) op **ra** c1 // op : 5비트(2^5 =32개 명령어 가능), ra : 5비트, c1 : 22비트

2) op **ra rb** c2 // op : 5비트, ra : 5비트, rb : 5비트, c2 : 17 비트

3) op **ra rb rc** c3 // op : 5비트, ra : 5비트, rb : 5비트, rc : 5비트, c3 : 12비트

**<시험에 나옴>**

**SRC 명령어 의미 Addressing Mode**

ld r1, 32 => R[1] <- M[32] Direct

ld r22, 24(r4) => R[22] <- M[24+R[4]] Displacement

st r4, 0(r9) => M[R[9]] <- R[4] Register indirect

la r7, 32 => R[7] <- 32 Immediate

ldr r12, -48 => R[12] <- M[PC-48] Relative

lar r3, 0 => R[3] <- PC Register

neg r1, r2 =>;Negate(r1 = -r2)

not r2, r3 =>;Not (r2 = r3`)

addi r1, r3, 1 => addi 는 맨 마지막이 상수다 (immediate)

add r2, r3, r4 => add는 맨 마지막에 상수가 오면 안된다

andi ra, rb, c2 => ra <- rb & c2

ori ra, rb, c2 => ra <- rb & c2

**Branch Instruction Format**

1) br rb, rc, c3<2..0> // R[ rc ]가 c3<2..0>의 조건을 만족하면 R[ rb ] 로 branch 한다

2) brl ra, rb, rc, c3<2..0> // R[ra] <- PC 저장 후 위에꺼랑 동일

It is c3<2..0>, the 3 lsbs of c3, that governs what the branch condition is:

lsbs condition Assy language Example

000 never brlnv brlnv r6

001 always br, brl br r5, brl r5, r4

010 if rc = 0 brzr, brlzr

011 if rc != 0 brnz, brlnz

100 if rc >= 0 brpl, brlpl

101 if rc < 0 brmi, brlmi

br r4 PC <- R[4]

brl r6, r4 R[6] <- PC

PC <- R[4]

brzr r5, r1 if(R[1] == 0)

PC <- R[5]

brlzr r7, r5, r1 R[7] <- PC;

brnz r1, r0 if(R[0] != 0) PC <- R[1]

brpl r3, r2 if(R[2] > 0) PC <- R[3]

brlpl r4, r3, r2 R[4] <- PC

if( R[2] > 0) PC <- R[3]

brmi r0, r1 if( R[1] <0 ) PC <- R[0]

brlmi r3, r0, r1 R[3] <- PC

if(R[1] < 0) PC <- R[0]

**<예제 1>**

C : goto Label3

SRC:

lar r0, Label3 (R[0] <- PC + offset ) // **lar : br 하기전에 분기주소를 넣는 명령어**

br r0 // offset : lar 주소부터 Label 3 까지의 offset

...

Label 3

**<예제2>**

C: #define Cost 125

if( X < 0) then X = - X

SRC:

Cost .equ 125 // Cost = 125

.org 1000 // next word will be loaded at address 1000 (10)

X: .dw 1 // reserve 1 word for variable X ( X변수 선언)

.org 5000 // next word will be loaded at location 5000 (10)

lar r31, Over // R[31] <- Over 주소

ld r1, X // R[1] <- X 주소에 있는 값 (아직은 값이 없음)

brpl r31, r1 // R[1] >= 0 이면 분기

neg r1, r1 // negative value

Over: ...