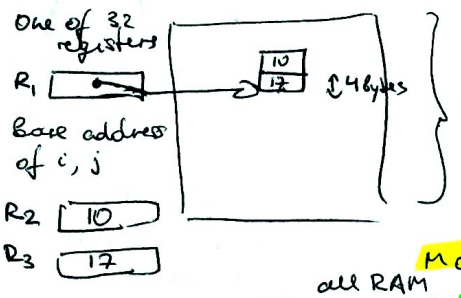
CS 107 Lecture 9 Assembly




```
int i;
int j;
i = 10;
j = i + 7;
j++;
```

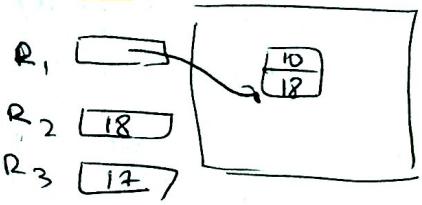
Mock assembly for the above code

all RAM as array

```
M[R1 + 4] = 10;   // store operation
R2 = M[R1 + 4];   // load operation
R3 = R2 + 7;      // ALU operation
M[R1] = R3;       // store operation
```

common load-ALU-store sequence

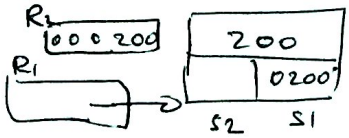
designed to deal with 4 byte figures (int, char*)



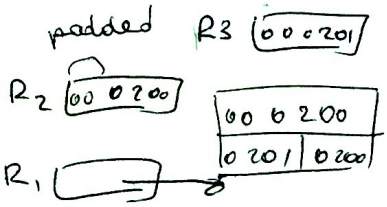

```
R2 = M[R1]
R2 = R2 + 1;
M[R1] = R2;
```

---

big-endian



padded

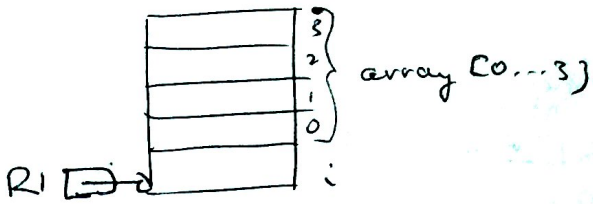



```
int i;
short s1;
short s2;

i = 200;   =>  M[R1 + 4] = 200;
s1 = i;    =>  R2 = M[R1 + 4];
              M[R1 + 2] = .2 R2;
s2 = s1 + 1;  R2 = M[R1 + 2];
              R3 = R2 + 1;
              M[R1] = .2 R3;
```

each instruction is 32-bit

each op - 1 clock cycle

```
int array[4];
int i;

for (i=0; i<4; i++){
   array[i]=0;
}
i--;
```



The hardware makes sure that base address of an relevant activation record is stored in a register (e.g. R1)

```
M[R1] = 0;
R2 = M[R1];
BGE R2, 4, PC+40
R3 = M[R1];
R4 = R3 * 4;
R5 = R1 + 4;
R6 = R4 + R5;
M[R6] = 0;
R2 = M[R1];
R2 = R2 + 1;
M[R1] = R2;
JMP PC - 40;
```

reload after JMP

program counter (address of currently executed instr.)

greater or equal Branching

reload to be context insensitive

offset in Bytes

base address

array[i]=0;

jump when test passes

reload to be context insensitive

i++

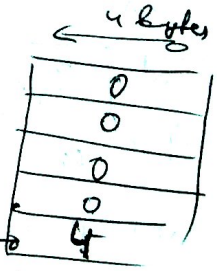
i--;

(10 instructions, each 4 Bytes)

each i is used, reload it to keep context insensitivity

Branch instructions

BEQ	(equal)
BNE	(not equal to)
BLE	(less equal to)
BGT	(greater than)
BGE	(greater equal to)



Encoding of instructions: instruction structure indicator looked at during first x% of clock cycle

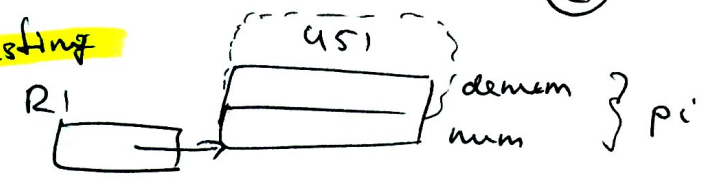

```
R1 = M[R2 + 4];     000000
R1 = 100;           000001
R3 = R6 * R10;      010011
M[R1 - 20] = R19;   111111
```

fill out the instruction structure (example numbers)

e.g. Huffman encoding

CS 107 Lecture 9

Struct and pointers, casting




```
struct fraction {
    int num;
    int denum;
};
```

```
struct fraction pi;
pi.num = 22;    => M[R1] = 22;
pi.denum = 7;   => M[R1+4] = 7;
```

(struct fraction *) &pi.denum → denum = 451;

=> M[R1+8] = 451;

A cast does not add code;
it allows compiler to generate code that it would otherwise not generate.