More Instructions & Instruction Encoding

In this lecture, you will learn:

1 - unsigned arithmetic: mulu, divu, movui, etc.

2 - unsigned branches: bltu, etc.

3 - comparison instructions: cmplt, cmpltu, cmpltui, etc.

4 - jump instructions

5 - program counter

6 - instruction encoding for R-type, I-type, J-type

Unsigned Arithmetic and Branches

add, sub use exact same { instruction for signed and unsigned ALU logic gates

but some operations need different logic gates

```
mul   rC, rA, rB   <-- rA, rB, rC all signed
mulu  rC, rA, rB   <-- rA, rB, rC all unsigned
divu  rC, rA, rB   <--/
```

Also:
movui
eg:
movi r3, 0xffff
movui r3, 0xffff
these are different!

```
blt   rA, rB, label  <-- rA, rB  signed
bltu  rA, rB, label  <-- rA, rB  unsigned
```

Also: bgtu
bgeu
bleu

example: suppose rA = -1 = 0xffff ffff
rB = +1 = 0x0000 0001

rA < rB signed

rA > rB unsigned

∴ need to carefully choose blt or bltu

WARNING: addresses are always unsigned

example:

```
movia r4, TABLE1
movia r5, TABLE2
bltu  r4, r5, table2 is AfterTable1
```

Comparisons

① [bltu rA, rB, label

② [cmpltu rC, rA, rB
  bne rC, r∅, label

← these do the same thing except ② also modifies rC

$$rC = \begin{cases} 1 & \text{if } rA < rB \\ 0 & \text{otherwise} \end{cases}$$

- when is this useful?

consider:

```
if ( r4 < r5 and r5 < r6) {
    r10 = r10 + 1
}
```

- this becomes:

```
        cmplt   r2, r4, r5   ← flags
        cmplt   r3, r5, r6   ←
        and     r2, r2, r3
        beq     r2, r∅, skip
        addi    r10, r10, 1
skip:   ...
```

- like branches, there are signed and unsigned versions

- unlike branches, you can compare a register to Imm16:

```
cmplt   rC, rA, rB
cmplti  rB, rA, Imm16S   ← signed
cmpltui rB, rA, Imm16U   ← unsigned
```

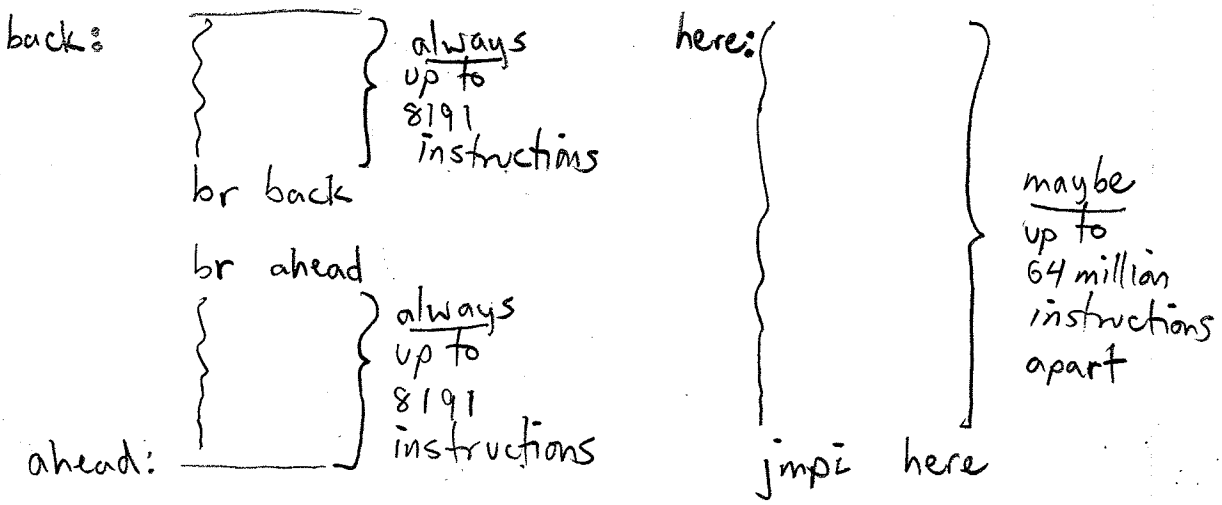
example

```
cmpeqi  r3, r4, 100
```

$$r3 = \begin{cases} 1 & \text{if } r4 == 100 \\ 0 & \text{otherwise} \end{cases}$$

		R-type	I-type
cmpeq	cmpeqi		
cmpne	cmpnei		
cmpgt	cmpgti	cmpgtu	cmpgtui
cmpge	cmpgei	cmpgeu	cmpgeui
cmplt	cmplti	cmpltu	cmpltui
cmple	cmplei	cmpleu	cmpleui
R-type	I-type	unsigned	

Jump Instructions



br and jmpi are almost identical

except how far they can go

and br can always go ± 8191 instructions

jmpi can go anywhere within 256MB window

→ windows are fixed, do not overlap
→ you cannot jmpi between windows

jmp has no restrictions, but uses a register

```
movia   r2, here
jmp     r2
```

Program Counter or PC

- special register in CPU
- holds address of the currently executing instruction
- automatically increases by 4 to execute next instruction
- to get the value of PC: "nextpc rC" puts PC+4 into rC (R-type)

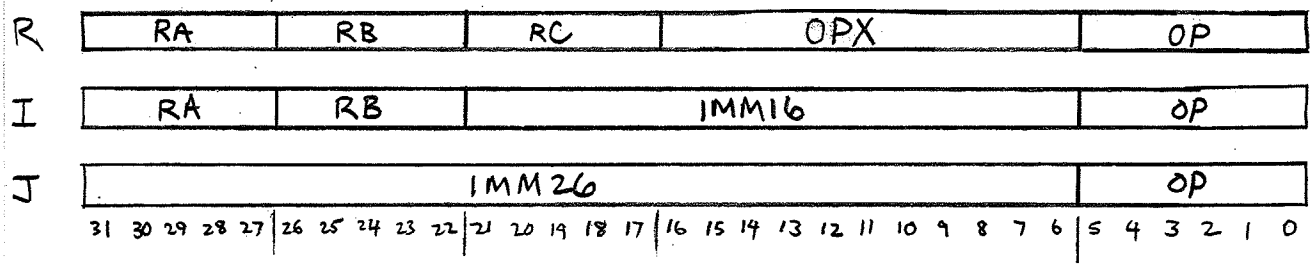
example:

```
        nextpc  r6
loop:   subi    r2, r2, 1
        beq     r2, r0, done
        jmp     r6
done:   ____
```

- to change PC, use: any branch, jmp, jmpi, call, or ret instruction

Instruction Encoding

Three types:



fields:
- RA, RB, RC — 5 bit register number 0 – 31
- IMM16 — 16 bit constant, usually signed
- IMM26 — 26 bit constant, unsigned

R-type:
- add RC, RA, RB ← jmp, nextpc, ret are all R-type

I-type:
- addi RB, RA, IMM16
- ldw RB, IMM16(RA)
- stw RB, IMM16(RA)
- beq RA, RB, label ⇝ IMM16 ⇝ ± offset

J-type:
- call label ⇝ IMM26
- jmpi label ⇝ IMM26

} changes only 26 middle bits of PC:
- lowest 2 bits always %00
- highest 4 bits are unchanged

Examples of OP, OPX (OP ≡ operation code)

Instruction	OP (6 bits)	OPX (11 bits)
roli	all R-type use 0x3A	0x002
ret		0x005
and		0x00e
mul		0x027
add		0x031
sub		0x039
call	0x00	—
jmpi	0x01	—
br	0x06	—
andi	0x0C	—
stw	0x15	—
ldw	0x17	—
bne	0x1E	—

- How is IMM16 computed for branch instructions (I-type)?

  IMM16 is an offset relative to the instruction immediately after the branch

example

```
here:   beq   r2,r3, there
        ⋮
there:
```

there = here + 4 + IMM16

IMM16 = there − (here + 4)

offset (difference) is signed!

if too far assembler reports error

- Assembler computes IMM16 from label automagically!

- How is IMM26 computed for J-type instructions?

```
here:   call  f
```

f = [4 bits | IMM26 (26 bits) | 00 (2 bits)]

always 0 – instruction addresses are always a multiple of 4

f must have same 4 MSB as here

if not, assembler reports error