ECE243

NIOS ISA Advanced Topics

Prof. Enright Jerger

Extending and Uncommon Insts

Signed data types

- interpret value as a 2's complement number
- the most-significant bit is a "sign bit"
  - if 0 means positive, if 1 means negative
  - bit7 of a byte
  - bit15 of a halfword
  - bit31 of a word
- Example:
  - signed byte for -25: 0b1110 0111

- Note: unsigned data types
  - not 2's complement, no sign bit

Max & Min Values

	signed byte	unsigned byte	signed halfword	unsigned halfword
min	$-2^7 = -128$	0	$-2^{15} = -32768$	0
max	$2^7 - 1 = 127$	$2^8 - 1 = 255$	$2^{15} - 1 = 32767$	$2^{16} - 1 = 65535$

Conditional branches (recall)

- only branch if a certain condition is true

bCC rA, rB, LABEL

- if rA CC rB goto LABEL

- signed comparisons

bCCu rA, rB, LABEL

- if rA CC rB goto LABEL
- unsigned comparisons

- CC:
  - eq (=), gt (>), ge (>=), lt (<), le (<=), ne (!=)

COMPARE INSTRUCTIONS

- used to compare two registers
- useful to compare with a constant
  - can't do that with bCC's

cmpCC rA,rB,rC

- if rB CC rC then rA=1 else rA=0
- signed comparison
- CC = same list as for bCC

CmpCCu rA,rB,rC

- unsigned comparison

CmpCCi rA,rB,Imm16

- signed comparison

CmpCCui rA,rB,Imm16

- unsigned comparison

Zero-Extending

- for unsigned data types when converting to a larger size
- Converting an unsigned byte to unsigned halfword:
  - new bits are zeros
  - Called “zero-extend to 16bits”
  - my notation: ZE16()
- Converting an unsigned byte or halfword to word:
  - new bits are zeros
  - Called “zero-extend to 32bits”
  - my notation: ZE32()
- Example: compute ZE16(byte 5)

  = ZE16(0b0005 0101)
  = 0b0000 0000 0000 0101

- Example: compute ZE16(byte 255)

  ZE16(0b1111 1111)
  = 0b0000 0000 1111 1111

Sign-Extending

- for signed data types when converting to a larger size
- Converting a signed byte to a signed halfword:
  - replicate the sign bit
  - called “sign-extend to 16bits”
  - my notation: SE16()
- Converting a signed byte or halfword to a signed word
  - replicate the sign bit
  - called “sign-extend to 32bits”
  - my notation: SE32()
- Example: compute SE16(byte –5)

SE16(0b1111 1011)
= 0b1111 1111 1111 1011

- Example: compute SE16(byte 5)

SE16(0b0000 0101)
= 0b0000 0000 0000 0101

Instrs that use Extending

- addi r8, r9, IMM16
  - Imm16 is a 16-bit signed value

r8 = r9 + SE32(Imm16)

- ldw r8,Imm16(r9)
  - Imm16 is 16-bit signed value

    r8 = mem[SE32(Imm16) + r9]

  - same for all loads and stores
- ldb r8, Imm16(r9)
  - Imm16 is 16-bit signed value

    r8 = SE32(mem[SE32(Imm16) + r9])

  - ldb, ldh are signed by default
- ldbu r8, Imm16(r9)
  - 'u' means 'unsigned'

    r8 = ZE32(mem[SE32(Imm16) + r9])

- ldh and ldhu behave similarly

Examples

- addi r8, r0, 0x1

  # r8 = r0 + SE32(0x1)
  = 0x0 + 0x0000 0001
  = 0x0000 0001

- addi r8, r0, 0xF001

  # r8 = r0 + SE32(0xF001)
  = 0x0 + 0xFFFF F001
  = 0xFFFF F001

- addi r8, r0, -1

  # r8 = r0 + SE32(0xFFFF)
  = 0x0 + 0xFFFF FFFF
  = 0xFFFF FFFF

- movia r8, 0x1000
- ldw r9, 0(r8) # r9 = mem[SE32(0x0) + 0x1000]
  = mem[0x00001000]

- ldw r9, 4(r8)

  # r9 = mem[SE32(0x4) + 0x1000]
  = mem[0x00001004]

- ldw r9, -4(r8)

  # r9 = mem[SE32(0xFFFC) + 0x1000]
  = mem[0xFFFF FFFC + 0x00001000]
  = mem[0x0FFC]

- movia r9, 0xFF
- stb r9, 0(r8)

  # mem[SE32(0x0) + 0x1000] = r9
  mem[0x00001000] = 0xFF

- ldb r9, 0(r8)

  r9 = SE32(mem[SE32(0x0) + 0x1000])
  = SE32(mem[0x00001000])

= SE32(0xFF)

= 0xFFFF FFFF

- ldbu r9, 0(r8)

r9 = ZE32(mem[SE32(0x0) + 0x1000])
= ZE32(mem[0x00001000])
= ZE32(0xFF)
= 0x0000 00FF

LOGICAL SHIFTS

- SLL/SLLI: shift left logical
  - like '<<' in C
- SRL/SRLI: shift right logical
  - like '>>' in C
- NOTE: new bits shifted in are zeros
- Ex: SLLI r8, r9, 0x4

r9 = 1101 1011 result(r8) = 1011 0000

- Ex: SRLI r8, r9, 0x3

r9 = 1101 1011 result(r8) = 0001 1011

Math with Shifts

- Can use shifts for cheap multiplication/division:
  - SLLI r8, r9, N => r8 = r9 * 2^N
  - SRLI r8, r9, N => r8 = r9 / 2^N

- Ex: compute 8/4: $= 8/2^2$

0b0000 1000 >> 2 = 0b0000 0010 = 2

- Ex: compute 5*2 $= 5 \times 2^1$

0b0000 0101 << 1 = 0b0000 1010 = 10

ARITHMETIC SHIFT RIGHT

- used if signed integer
- fill new bits with the value of the MSbit
- Why?
  - What if number to be shifted is –2?
  - Ex: want –8 >> 2 = -2

- Ex: SRAI r8,r9,0x3
  - r9: 1001 0110 Result (r8): 0b1111 0010

- r9: 0001 0110 Result (r8): 0b0000 0010

- Ex: SRAI r8, r9, 0x2
  - r9: 1111 1000 (-8) Result: (r8): 0b1111 1110
    = -2

ROTATES

- ROR r8, r9, r10

- Example: r9: 1011 0001
  - r10=1: r8 = 0b1101 1000
  - r10=2: r8 = 0b0110 1100
  - r10=4: r8 = 0b0001 1011

- ROL/I:
  - same thing but: left r9: 1011 0001

Office Hours this week:

Wed 12-1pm
Thurs 12-1pm

Discuss midterm in class on Wed

Data Structures

Arrays

char myarray[5] = { 1, 2, 3, 4, 5 }; // byte size locations

index:	0	1	2	3	4
Value:	1	2	3	4	5

value of myarray[1]: 2

value of myarray[4]: 5

Layout in memory assuming myarray starts at 0x1000 (i.e., base_addr == 0x1000)

Addr	Value	
0x1000	1	myarray[0]
0x1001	2	←
0x1002	3	
0x1003	4	
0x1004	5	myarray[4]
0x1005		

Address of an element: base_addr + index * sizeof(element)

address of myarray[1]: 0x1000 + 1 x 1 = 0x1001

address of myarray[4]: 0x1000 + 4 x 1 = 0x1004

Arrays Example

Note: addr = base_addr + index * sizeof(element)

```
short myarray[5]      // 2B
    = { 1, 2, 3, 4, 5 };
short sum = 0;
sum += myarray[0];
sum += myarray[4];
```

```
        .section .data
        .align 1        # align on 2' byte boundary
myarray:  .hword 1,2,3,4,5
sum:      .hword  0
```

Addr	Value
0x1000	1
0x1002	2
0x1004	3
0x1006	4
0x1008	5
0x100a	0
0x100c	
0x100e	

sum:

```
movia r8, sum
ldh r9, 0(r8) # r9 has sum
movia r10, myarray # base addr of myar
ldh r11, 0(r10) # r11 = 1
add r9, r9, r11 # r9 = 1
ldh r11, 8(r10) # element 4 offset by 8B
                # r11 = 5
add r9, r9, r11 # r9 = 1+5 = 6
sth r9, 0(r8)
```

ARRAYS AND FOR LOOPS

```
short myarray[5]
    = { 1, 2, 3, 4, 5 };
short n = 5;
short sum = 0;

for (int r8=0; r8<n; r8++){
   sum = sum + myarray[r8];
}
```

```
.section .data
.align 1
```

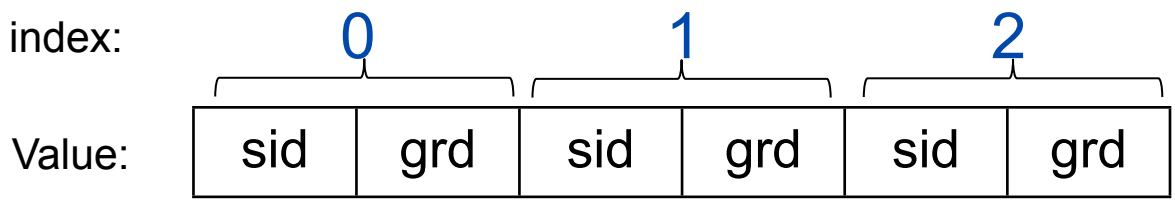
```
myarray:    .hword 1,2,3,4,5
n:          .hword 5
sum:        .hword 0
```

```
        mov r8, r0
        movia r9, n
        ldh r9, 0(r9)  #r9 holds n
LOOP:   bge r8, r9, DONE
        movia r10, sum
        ldh r11, 0(r10)  # r11 has sum
        movia r12, myarray
        add r13, r12, r8  # r13 = myarray + r8
        add r13, r13, r8  # r13 = myarray + 2x r8
        ldh r13, 0(r13)
        add r11, r11, r13
        sth r11, 0(r10)
        addi r8, r8, 1
        br LOOP
DONE:   
```

ARRAY OF STRUCTS

```
typedef struct {
  unsigned student_id;   // 4 B
  unsigned grade;
} student_t;
```

student_t ece243[300];



ece243[2].grade :

ece243[1].student_id :

Addr	Value	
0x1000	student_id	← ece243[0]
0x1004	grade	
0x1008	student_id	← ece243[1]
0x100c	grade	
0x1010	student_id	
0x1014	grade	
0x1018	student_id	
0x101c	grade	

Layout in memory assuming:
ece243 starts at 0x1000
(i.e., base_addr == 0x1000)
Unsigned == 4B == word

Address of an element: base_addr + index * sizeof(struct) + offset_within_struct

address of ece243[5].grade:

0x1000 + 5x8 + 4 = 0x102C

address of ece243[4].student_id:

0x1000 + 4x8 + 0 = 0x1020

ARRAY OF STRUCTS

```
typedef struct {
  unsigned student_id;
  unsigned grade;
} student_t;

student_t ece243[300];

main(){
  ece243[5].student_id = 999999999;
  ece243[5].grade = 81;
}
```

assume unsigned = 4B = word

```
        .section .data
        .align 2
ece243: .skip 2400   # 300 x 2 x 4
        .section .text
        .global main
main:   movia r8, ece243
        movi r9, 5
        slli r9, r9, 3 # 5 x 2^3 = 5 x 8
```

```
add r9, r9, r8
movia r10, 999999999
stw r10, 0(r9)
movi r10, 81
stw r10, 4(r9) # grade @ offset 4
ret
```

Pseudo-Instructions

Pseudo-instruction:

- doesn't actually exist
- assembler replaces any pseudo-inst
  - with one or more 'real' insts
  - real inst or insts implement the pseudo-inst
- **Why pseudo-instructions?**
  - for convenience:
    - to make code more readable
    - to hide the use of r0
  - no real instructions support Imm32:
    - must break into two Imm16 instructions

But first: lo, hi, and hiadj

%lo (value)

- returns SE32(least-significant 16 bits of value)
- Ex: %lo(0x200000) = 0x0
- Ex: %lo(0x53) = 0x0000 0053

%hi(value)

- returns the most-significant 16 bits of value
- Ex: %hi (0x200000) = 0x0020 0000
- Ex: %hi (0x53) = 0x0000 0000

%hiadj(value)

- = %hi(value) + bit15(value)
- Ex: %hiadj(0x200000) = 0x20+0x0 = 0x20
- Ex: %hiadj(0x20f000) = 0x20 + 0x1 = 0x21

Pseudo: Mov and Movi

- mov r8,r9
  - becomes: add r8, r0, r9
  - check: r8 = r0+r9 = 0 + r9 = r9

Midterm Info

March 4 6-8pm

SF 3201: A-Gr
SF 3202: Gu-Lv
HA 403: M-Sw
HA 402: T-Wu
HA 316: X-Z

* no calculator
* open book/notes

Topics for midterm

- Number representation
  - 2s complement, floating point
- Basic Assembly
  - → Instruction operation
    - memory, endian (loads/stores)
    - → signed/unsigned/extending
    - C to assembly
      - loops, if/else, subroutines, arrays
      - structs

I/O

- memory mapped I/O
- polling
- Interrupts · configuring, handler

- movi r8, Imm16
  - becomes: addi r8, r0, Imm16
  - check: r8 = r0 + Imm16 = 0 + Imm16 = Imm16

Pseudo: Movhi

- movhi r8, Imm16
  - copies Imm16 into the MS 16bits of r8
    - r8 = Imm16 << 16
  - clears the LS 16 bits of r8
- Ex: movhi r8,0x25 # r8 = 0x00250000

- movhi r8, 0x25
  - becomes: orhi r8, r0, 0x25
  - check: r8 = r0 | (0x25 << 16)

    0x0 | 0x00250000 = 0x00250000

Pseudo: Movia

- movia r8, Imm32
  - becomes: orhi r8, r0, %hiadj(Imm32)
    addi r8, r8, %lo(Imm32)

- Ex1: movia r8, 0x200008

  orhi r8, r0, %hiadj(0x200008)

  addi r8, r8, %lo(0x200008)

① r8 = r0 | ((0x20 + 0) << 16) = 0x0 | 0x200000

= 0x200000

② r8 = 0x200000 + SE32(0x0008) =

0x200008

- Ex2: movia r8, 0x20f000

  orhi r8, r0, %hiadj(0x20f000)

  addi r8, r8, %lo(0x20f000)

① r8 = r0 | ((0x20 + 1) << 16 = 0x0 | 0x210000

= 0x210000

② r8 = 0x210000 + SE32(0xF000)

= 0x210000 + 0xFFFF000

= 0x20F000

Typical Loads (and Stores)

- A Typical load sequence:

  movia r8,0x20f000 # pseudo-inst

  ldw r9,0(r8)

- Becomes:

  orhi r8,r0,%hiadj(0x20f000)

  addi r8,r8,%lo(0x20f000)

ldw r9,0(r8)

r8 = 0 | ((0x20+1) << 16) = 0x210000

r8 = 0x210000 + SE32(0xF000) = 0x20F000

r9 = mem[0x0 + 0x20F000]

More Efficient load Sequence:

- load sequence with only 2 instructions
- no pseudo-insts, so becomes 2 real insts

```
movhi r8, %hiadj(0x20f000)
ldw r9, %lo(0x20f000)(r8)
```

r8 = 0x0 | ((0x20+1) << 16) = 0x210000

r9 = mem[SE32(0xF000) + 0x210000]
= mem[0xFFFFF000 + 0x210000]
= mem[0x20F000]

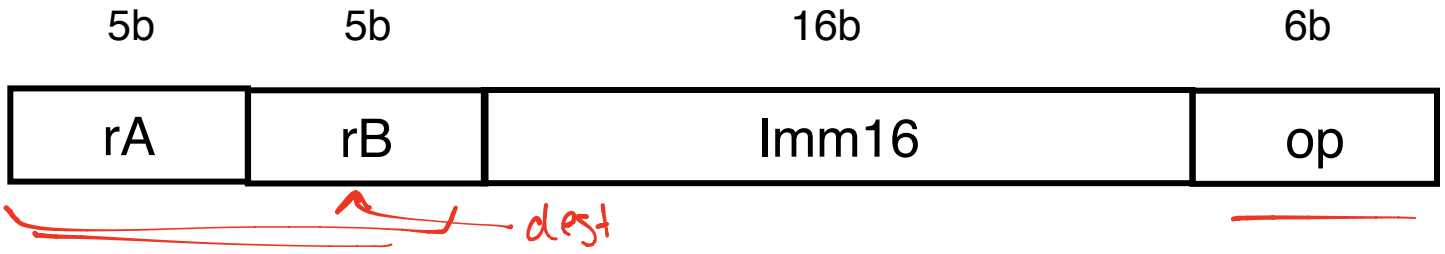
- Summary:
  - can use %hiadj and %lo to load from a constant address in 2 instructions
  - same thing works for stores
  - downsides: have to do it “by hand”, ugly code

Instruction Representation

Machine Instructions

- binary encoding of assembly instructions
- NIOS: all instructions are 32bits (4B)
  - must be aligned properly
  - i.e., with word alignment
- NIOS Instruction formats:
  - see NIOS II ISA reference (very beginning)

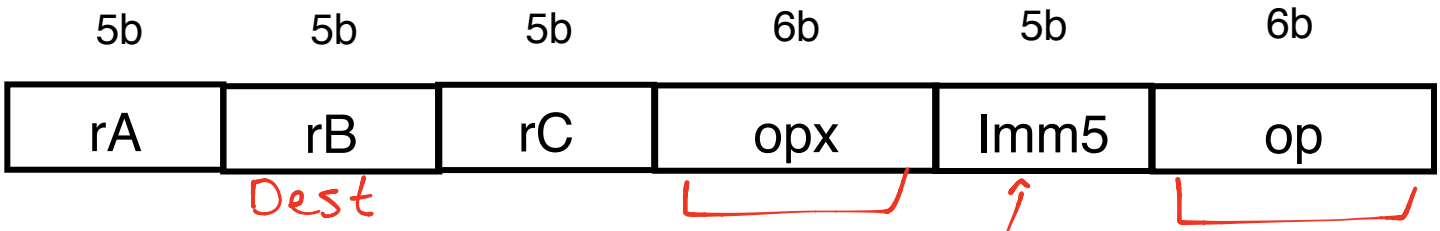
I-TYPE



- anything with an immediate value
  - addi, movi, ori, etc.
- branches, loads and stores

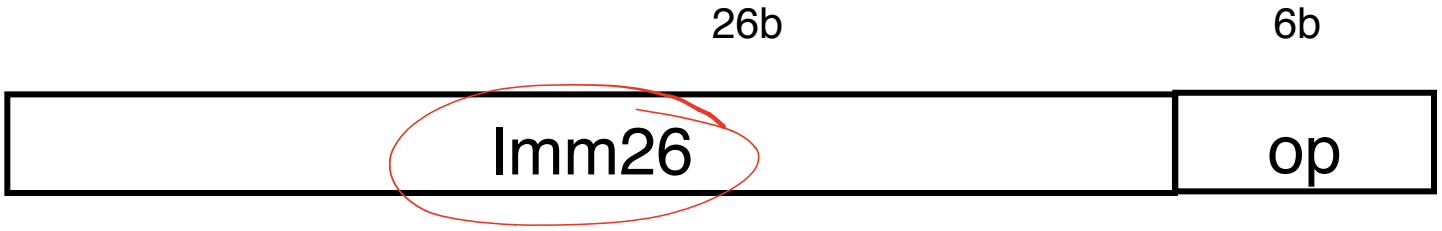
- op tells you what instruction
  - 0x4 means addi
  - 0x16 means blt
  - 0x17 means ldw

R-TYPE

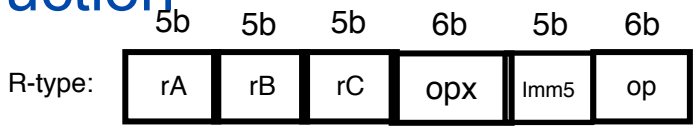


- arithmetic & logical operations
  - add, nor, etc.
- comparisons
  - cmpCC etc.
- when op is set to 0x3a
  - then you know it is an R-type instruction
- the opx field tells you what kind of R-type instruction
  - 0x31 means add
  - 0x39 means sub
- Imm5 is used for some advanced instructions

J-TYPE



- only used for the call instruction
- op is set to 0x00 - call
  0x01 - jmpi



Example

- give the encoding for "add r3,r5,r1"
  - add rC,rA,rB (from NIOS spec)

rA = 0b00101

rB = 0b00001

rC = 0b00011

opx = 0x31 = 0b110001

Imm5 = 0b00000

op = 0x3A = 0b111010

0b00101 10000 10000 01111 10001 10000 01110 10 = 0x28478838

3A

- give the encoding for "addi r7,r3,0x1234"
  - addi rB,rA,Imm16

rA	rB	Imm16	op

rA = 0b00011

rB = 0b00111

Announcements

Away Mon - Wed (3/3 - 3/5)

- No office hours Tues (3/4)
- Prof Anderson - exam review in lecture Monday
- Wed (3/5) lecture cancelled

Post midterm questions to piazza

Imm16 = 0b 0001 0010 0011 0100

op = 0x4 = 0b 000100

⇒ 0b 0001 1001 1100 0100 1000 1101 0000 0100
= 0x19C48D04

Branch Encoding

- In assembly:
  - bCC rA,rB,LABEL
- Actual encoding:
  - bCC rA,rB,Imm16
  - Imm16 is displacement
    - from current instr to target instr
- If rA CC rB then
  - PC = PC + Imm16 + 4
  - else PC = PC + 4
- Compute Imm16 by solving for it:
  - TARGET_PC = CURRENT_PC + Imm16 +4
  - Imm16 = TARGET_PC – CURRENT_PC - 4

(TPC) (CPC)

Branch Example

- give the encoding for "blt r8,r9,LOOP"

I - type

– blt rA,rB,LABEL

| rA | rB | Imm16 | op |

- Assume this code:

LOOP: add r8,r9,r9 ← TPC

blt r8,r9,LOOP ← CPC

Solve for Imm16

CPC = TPC + 4 (for this ex!)

Imm16 = TPC - CPC - 4

= TPC - (TPC + 4) - 4

= TPC - TPC - 4 - 4

= -8

binary for -8? Compute 2s comp

8 in 16-bit: 0000 0000 0000 1000

2scomp: 1111 1111 1111 0111

\+ 1

1111 1111 1111 1000

rA = 0b01000

rB = 0b01001

Imm16 = 0b1111 1111 1111 1000

op = 0x16 = 0b010110

⇒ 0b0100 0010 0111 1111 1111 1110 0001 0110

= 0x427FFE16