ECE437: Introduction to Digital Computer Design

Chapter 2 – Instructions

Fall 2016

Announcements

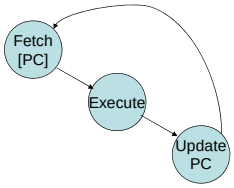
- Lab1 is on basic 270/337 material and is not covered in 437 lectures
- Homeworks are due in class, at the beginning of the lecture
- Slides and homeworks posted on the course website.
  - Slides will be posted every couple of lectures

Instructions

- Instructions are the "words" of a computer
- Instruction set architecture (ISA) is its "vocabulary"
- With a few other things, this defines the interface of computers
  - But implementations vary
- We will study MIPS™ ISA
  - but the next ISA is not too hard after the first
- We won't write programs - EE362

Computer

- What does it do?
  - Fetch instruction from address in Program Counter (PC)
  - Execute instruction
  - Update PC to point to next instruction



Outline

- Instructions
  - Basics
  - Registers and ALU ops
  - Memory and load/store
  - Branches and jumps
  - And more . . .
- Read Chapter 2 and skim through Appendix E (on CD) for MIPS ISA

Basics

- C statement
- f = (g + h) - (i + j)
- MIPS instructions*

```
add t0, g, h   // t0 = g + h
add t1, i, j   // t1 = i + j
sub f, t0, t1  // f  = t0 – t1
```

- Opcodes/mnemonic, operands, source/destination

Basics

- Opcode: Specifies the kind of operation (mnemonic)
- Operands: input and output data (source/destination)
- Operands t0, t1 are temps
- One operation, two inputs, one output
- Multiple instructions for one C statement

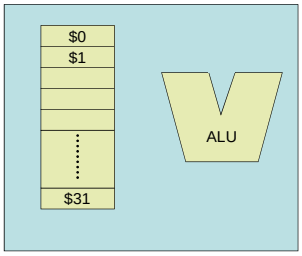
Why not have bigger instructions?

- Why not have "f = (g + h) - (i + j)" as one instruction?
- Church-Turing thesis: A very primitive computer can compute anything that a fancy computer can compute
  - need only logical functions, read and write memory and data-dependent decisions
- Therefore, ISA selected for practical reasons: performance and cost, not computability
- Simplicity/regularity tend to improve both
  - E.g., H/W to handle arbitrary number of operands
  - complex, slow and rarely usable and NOT NECESSARY

Registers and ALU ops

- Ok, I lied!
- Operands must be registers, not variables
  - add $8, $17, $18 // $8 <- $17 + $18 – destination LEFT MOST – if you confuse this, you won't get anything right in 437
  - add $9, $19, $20
  - sub $16, $8, $9
- MIPS has 32 registers $0-$31 (figure next slide)
- Registers $8, $9 are temps, $16 - f, $17 - g, $18 - h, $19 - i, $20 – j
- MIPS also allows one constant called "immediate"
  - later we will see immediate is 16 bits [-32768, 32767]

Registers and ALU



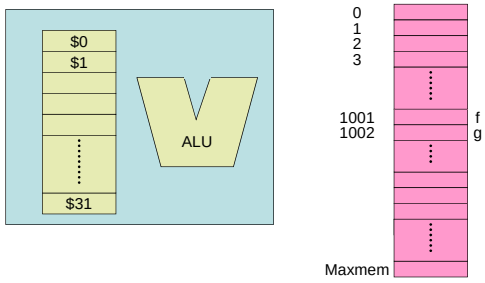
ALU ops

- Some ALU ops:
  - add, addi, addu, addiu (immediate, unsigned)
    - Caution: casting signed to unsigned in System Verilog may cause weirdness
  - sub . . .
  - mul, div - weird!
  - and, andi
  - or, ori
  - sll, srl, . . .
- Why registers? fit in instructions, smaller is faster
- Are registers enough?

Memory and load/store

- But need more than 32 words of storage
- Memory - an array of locations M[addr], indexed by addr (figure next slide)
- Data movement (on words or integers)
  - load word for reg <-- memory
  - lw $17, 1002 // get input g
  - store word for reg --> memory
  - sw $16, 1001 // save output f
    - Note: destination LAST for stores!

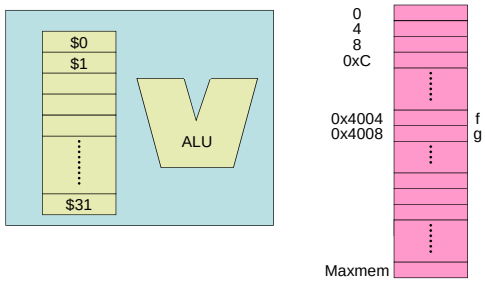
Memory and load/store



Byte vs. Word addresses

- I lied again!
  - we need to address bytes for characters
  - words take 4 bytes
  - therefore, word addresses must be multiples of 4 (i.e., end in 00)
  - Addresses are ALWAYS byte addresses (next slide)
    - NOT word addresses (previous slide)
- figure next slide

Memory and load/store



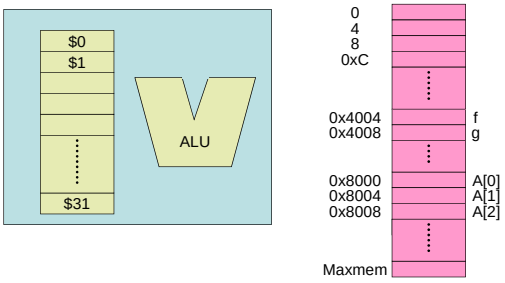
Memory and load/store

- Important for arrays
  - A[i] = A[i] + h (figure next slide)
  - # $8 - temp, $18 - h, $21 - (i x 4)
  - Astart is 0x8000, A is an array of words NOT bytes

```
lw $8, Astart($21)   // == 0x8000($21)
                     // Astart or 0x8000 is offset
                     // $8 <- M[Astart+$21]
                     // or $8 <- M[0x8000+$21]
add $8, $18, $8
sw $8, Astart($21)  //offset is signed, can be negative
```

- MIPS has other load/store for byte/halfword

Memory and load/store



Endian

- Word storage
- Word = bytes (0x400,0x401,0x402,0x403)?
- Word = bytes (0x403,0x402,0x401,0x400)?
  - It depends…
- Big endian: MSB at address xxxxxx00
  - e.g., IBM, SPARC
- Little endian: MSB at address xxxxxx11
  - e.g., Intel x86
- Mode selectable
  - e.g., PowerPC, MIPS

Branches and Jumps

```
While ( i != j) {
   j= j + i;
   i= i + 1;
}
```

- HLL-> Assembly, $8 is i, $9 is j

```
   Loop: beq $8, $9, Exit
         // note that the condition is reversed
         add $9, $9, $8
         addi $8, $8 , 1
         j    Loop
   Exit:
```

Branches and Jumps

- Better yet:

```
        beq $8, $9, Exit        // not !=
  Loop: add $9, $9, $8
        addi $8, $8 , 1
        bne $8, $9, Loop
  Exit:
```

- Let compilers worry about such optimizations

Branches and Jumps

- What does bne do really?
  - read $8; read $9, compare
  - set PC = PC + 4 or PC = Target
- To do compares other than "equal" or "not equal"
  - e.g., blt $8, $9, Target // assembler PSEUDO-instruction
- expands to
  - slt $1, $8, $9 // set less than $1 = ($8 < $9)?1:0
  - bne $1, $0, Target // $0 is always 0

Branches and Jumps

- Other MIPS branches/jumps
  - beq $8, $9, imm
  - // if ($8 == $9) PC = PC + imm<<2 else PC += 4
- bne ...
- slt, sle, sgt, sge
  - with immediate, unsigned
- j addr // PC = addr
- jr $12 // PC = $12
- jal addr // $31 = PC+4; PC = addr; used for procedure calls

Assembly Exercise

```
for(i=0;i<n;i++) {
  blah
  blah
}
```

```
Init:   mov $8,0
Test:   slt $5,$8,$9
        beq $5,$0,Exit
Loop:   blah
        blah
Ind:    addi $8,$8,1
        j Test
Exit:
```


Assembly Exercise

- HLL-> assembly

```
if (a=b)
   a = a+b
else
   b = a+b
```

Assembly Exercise

- HLL-> assembly

```
if (a=b)
    a = a+b
 else
    b = a+b
```

```
    lw $8, a  // a -> $8
    lw $9, b  // b -> $9
    beq $8,$9,L_IF
    j L_ELSE
L_IF:
    add $8,$8,$9  // a = a+b
    sw $8,a  // write a to mem
    j EXIT
L_ELSE:
    add $9,$8,$9 // b= a+b
    sw $9, b // write b to mem
EXIT:
```


Procedure Calls

- See section 2.7 for more details
  - save registers
  - set up parameters
  - call procedure
  - get results
  - restore registers

Proc: save more registers
do function
set up results
restore more registers
return

- jal is the ONLY operation in hardware: the rest is in software
- jal addr // $31 = PC+4 (return addr); PC = addr (UNLIKE 362 where return addr is saved on stack – too complicated/slow to do in fast clock)

Procedure Calls

- An important data structure is the stack
- Stack grows from larger to smaller addresses
- $29 is the stack pointer, it points just beyond valid data
- push $2:
  - addi $29, $29, -4
  - sw $2, 4($29)

pop $2:
    lw $2, 4($29)
    addi $29, $29, 4

- push, pop NOT real instructions
- Addi-sw/lw-addi order cannot change. Why?

Procedure Example

- HLL code for swap

```
{ temp = v[k];          /* code for swap */
  v[k] = v[k+1];
  v[k+1] = temp;
  return (temp); }
```

- Corresponding assembly code

```
swap:   addi  $29, $29, -8   // save registers
        sw    $15, 4($29)
        sw    $16, 8($29)
```

Procedure Example

```
muli*  $2, $5, 4      // procedure body; $2 <- k*4
add    $2, $4, $2     // $2 <- v + k*4 == address of v[k]
lw     $15, 0($2)       // $15 <- v[k]
lw     $16, 4($2)       // $16 <- v[k+1]
sw     $16, 0($2)       // v[k] <- $16
sw     $15, 4($2)       // v[k+1] <-  $15
mov*   $2, $15          // return value
lw     $15, 4($29)      // restore registers
lw     $16, 8($29)
addi   $29, $29, 8
jr     $31               // return
```

- All push/pop in s/w

Procedure Call Example

- Now, calling the procedure
  - HLL code

```
return_value = swap(arr_A, i);  /* call swap */

    mov   $4, $18      // first parameter
    mov   $5, $17      // second parameter
    jal    swap
    mov   $20, $2      // copy return value
```


Layers of Software

- Notation - program: input data -> output data
  - executable: input data -> output data
  - loader: executable file -> executable in memory
  - linker: object files -> executable file
  - assembler: assembly file -> object file
  - compiler: HLL file -> assembly file
  - editor: editor commands -> HLL file
- Only possible because programs can be manipulated as data
  - "stored program computer", "Von Neumann architecture".

362 vs. 437

- Shifts in 362 vs. 437
  - Freescale HCS (362) shifts include the Carry
  - MIPS (437) shifts do not
  - In MIPS, the Carry is not part of the ISA
    - Carry is an internal signal to detect overflows
    - Carry is not visible to the ISA
    - branches use general-purpose registers and not condition codes

MIPS Machine Language

- All 32-bit instructions
- Assembly: add $1, $2, $3
  - 14 ASCII (or Unicode) codes

61 64 64 20 24 31 2C 20 24 32 2C 20 24 33 (hex)

- Machine code: 1 word

000000	00010	00011	00001	00000	010000
alu-rr	2	3	1	zero	add/signed

Instruction Format

- R-format:

opcode	rs	rt	rd	shamt	funct
6	5	5	5	5	6

- Digression:
  - How do you store the number 4,392,976?
  - Same as add $1, $2, $3
- Stored program: instructions are represented as numbers
  - programs can be read/written in memory like numbers
- Other R-format: addu, sub*, and, or ….. etc

Instruction Format

- Assembly: lw $1, 100($2)
- Machine:

100011 00010 00001 0000000001100100
lw 2 1 100 (in signed binary)

- I-format:

opcode	rs	rt	address/immediate
6	5	5	16 (signed)

  - Addr/immediate is 16 bits signed 2's complement

Instruction Format

- I-format also used for ALU ops with immediates
  - addi $1, $2, -4
  - 001000 00010 00001 1111111111111100
- What about constants larger than 16 bits = [-32768, 32767]?
  - e.g., 0000 0000 0000 1100 0000 0000 0000 1111 or 0x000C000F?

```
lui $4, 12        // $4 <- 0x000C0000
ori $4,$4,15      // $4 <- 0x000C000F
```

- All loads and stores use I-format

Instruction Format

- I-format for branches: beq $1, $2, 7
- 000100 00001 00010 0000 0000 0000 0111
- PC = PC + (0000 0111 << 2) // word offset

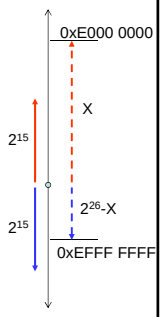
Finally, J-format: opcode addr

6 26

- addr is weird in J-format:
  - Target address = 4 MSB of PC:addr:00 (4+26+2 = 32)
  - Why MSB of PC? We have 26, we need 30
  - Can we Pad with 0s? Restricts jumps to top 16th of address space

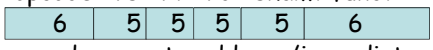
MIPS: branch vs jump

- Branch: 16 bit displacement
  - Current word +/- 2^{15}
  - Range is symmetric
    - $Range_{(forward)} = Range_{(backward)} = 2^{15}$
- Jump: 26 lower-bit concatenation in word address
  - Jump anywhere within a 2^{26} segment
  - Range may be asymmetric
    - $Range_{(forward)} + Range_{(backward)} = 2^{26}$



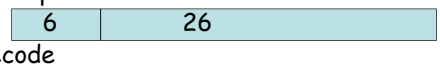
Summary: Instruction Formats

- R-format: opcode rs rt rd shamt funct



- I-format: opcode rs rt address/immediate

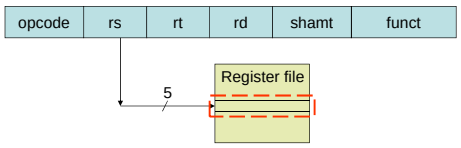
- J-format: opcode addr



- Instr decode
  - Theory: Inst bits -> identify instrs -> control signals
  - Practice: Instruction bits -> control signals

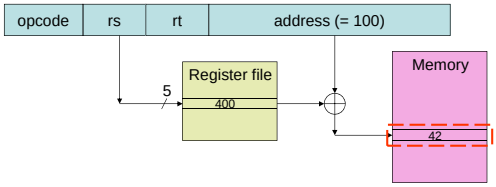
Addressing modes

- There are many ways of accessing data
- 1. Register addressing
  - add $1, $2, $3



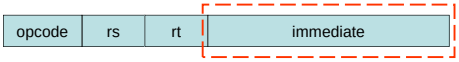
Addressing Modes

- 2. Base addressing (aka displacement)
  - lw $1, 100($2) // $2 == 400, M[500] == 42



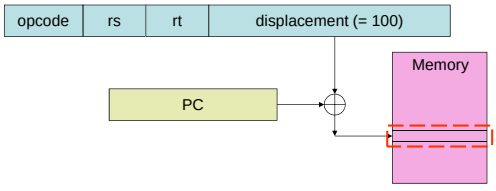
Addressing Modes

- 3. Immediate addressing
  - addi $1, $2, 100



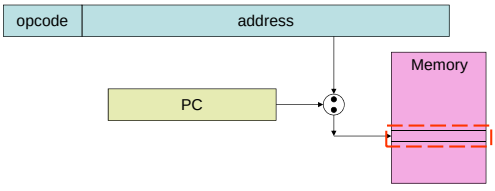
Addressing Modes

- 4. PC relative addressing
  - beq $1, $2, 100 // if ($1 == $2) PC = PC + 100



Addressing Modes

- Pseudodirect addressing
  - j Loop // instruction contains address*



Addressing Modes

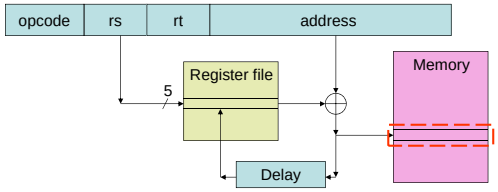
NOT found in MIPS

- 5. Indexed: add two registers - base + index
- 6. Indirect: M[M[addr]] - two memory references
- 7. Autoincrement/decrement: add data size
- 8. Autoupdate - found in IBM PowerPC, HP PA-RISC
  - like displacement but update register

Addressing Modes

- Autoupdate

lwupdate $1, 4[$2] // $1 == M[4+$2]; $2 == $2 + 4



Addressing Modes

```
for (i = 0, I < N, i +=1)
  sum += A[i];
```

- $7 - sum, $8 - address of a[i], $2 – tmp

```
inner:                  new inner:
  lw $2, 0($8)          lwupdate $2, 4($8)
  addi $8, $8, 4
  add $7, $7, $2        add $7, $7, $2
```

- any problems with new inner ?

How to Choose ISA

- Minimize what?
  - Two way constraint satisfaction
  - What can be implemented in hardware? From below
  - What ISA is better for optimized compilation? From above
- In 1985-1990 technology, simple modes like MIPS good
  - Easy to implement, not a real reason in the business world
  - Very compelling reason in the classroom
- As technology changes, computer design options change
- For small memory, dense instructions important and dense usually complex/irregular
- For high speed, pipelining important and for pipelining simple/regular important

Intel 80386 ISA

- 8 32-bit registers
- Two register machine: src1/dst src2
  - reg - reg, reg - immed, reg - mem, mem - reg, mem - imm
- seven addressing modes
- 16-bit and 32-bit ops on memory and registers
  - 8-bit prefix to override default data size
  - condition codes
    - part of normal operation, extends operation time, often not looked at

Intel 80386 ISA

- Decoding nightmare
  - instructions 1 to 17 bytes
  - prefixes, postfixes
  - crazy "formats" - e.g., register specifiers move around
  - but key instructions not terrible
  - yet have to make ALL work correctly

Current Approach

- Current Intel chips
  - Instruction decode logic translates into "RISCy Ops"
  - RISC – Reduced Instruction Set Computer
  - Execution unit runs RISCy ops
  - Backward compatibility
  - Complex decoding
  - Execution unit as fast as RISC
- We work with MIPS to keep it simple
- Learn x86 on the job!

Complex Instructions

- More powerful instructions not necessarily faster execution
- E.g. - string copy
  - Option 1: move with repeat prefix for memory to memory move
  - special-purpose
  - Option 2: use loads into register and then stores
  - generic instructions
  - Option 2 faster on the same machine!

Outline

- So far, too MIPS-centric
  - Rich design space:
  - Other possibilities: Stack and Accumulator architectures :Ch 2 (on CD)

General Purpose Architecture

- General Purpose Register architecture
  - 3 operands: 2 source + 1 dest
  - Depends on addressing modes
    - Register-register only
      - MIPS : all register operands except load/store
    - Memory-memory (all memory operands)
- Eg
  - f = (g+h) *(i-j)

```
Add t1, g, h
Sub t2, i, j
Mul f, t1, t2
```


Stack Machine

- Push, Pop with one address
  - Last in, first out
- Binary/Unary Op operates on
  - Implicit operands
  - Top one/two element(s) of stack
- Eg
  - f = (g+h) *(i-j)

```
Push g
Push h
Add
Push i
Push j
Subtract
Mul
```

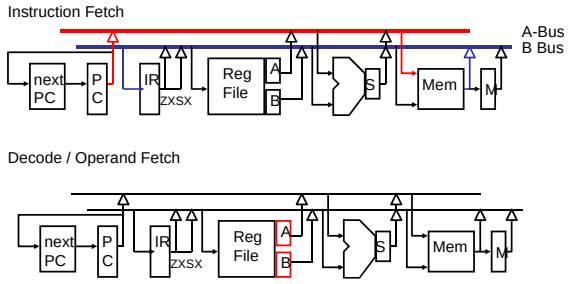

Accumulator Machine

- One address
  - 2nd implicit operand: accumulator
  - accumulator is 2nd source operand as well as destination
- Eg
  - f = (g+h) *(i-j)

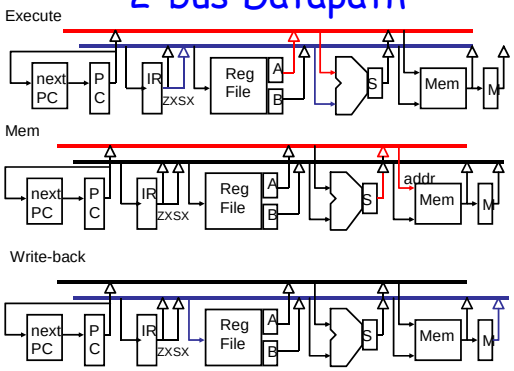
```
Load g   // Acc = g
Add h    // Acc = Acc+h
Store t  // t = g+h
Load i   // Acc = i
Subt j   // Acc = Acc-j
Mul t1   // Acc = Acc*t
Store f
```

2-Bus Datapath

- Alternate datapath: Shared bus
  - E.g. Load instruction



2-bus Datapath



Concluding Remarks

- Simple and regular
  - same length instructions, fields in same place
- Small and fast
  - Small number of registers
- Compromises inevitable
  - There is NO PERFECT ISA!
  - Pipelining (more later) should not be hindered
- Common case fast
- Read Chapter 2
- Read Chapter 4a 4.1-4.4 (coming up next)
  - Ch 4a+b is 1/3rds of this course, lab