Computer Systems Architecture

http://cs.nott.ac.uk/~txa/g51csa/

Thorsten Altenkirch and Liyang Hu

School of Computer Science and IT
University of Nottingham

Lecture 04: Machine Code, Data Transfer and Control Flow

The University of Nottingham

Assembly to Machine Code

- So far we've been using assembly language
- Assembler turns *symbolic instructions* to *machine code*
- Each instruction is 32-bits (or 4 bytes) long
- The 32 bit machine code consists of several fields
  - Field format differ depending on instruction type
  - MIPS uses three basic formats: R, I and J

Assembly	`add $s0, $s1, $s2`
Hexadecimal	$0232\,8020_{16}$
Binary	$0000\,0010\,0011\,0010\,1000\,0000\,0010\,0000_2$

The University of Nottingham

Instruction Encoding (Register Operands)

R-Format

Field	*op*	src_0	src_1	*dst*	*shamt*	*func*
Size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
		←	32 bits total		→	

op Basic operation code

src Source operand register

dst Destination operand register

shamt Shift amount (not relevant now; more later)

func Function / operation variant

The University of Nottingham

Instruction Encoding (Immediate Operand)

I-Format

Field	*op*	*src*	*dst*	*imm*
Size	6 bits	5 bits	5 bits	16 bits
	←	32 bits total		→

op Basic operation code

src Source operand register

dst Destination operand register

imm Immediate constant

- Also a J format: 6 bit opcode, 26-bit immediate

Encoding Addition

`add $s0, $s1, $s2` – 02328020_{16}

op	src_0	src_1	*dst*	*shamt*	*func*
000000	10001	10010	10000	00000	100000
`add`	`$s1`	`$s2`	`$s0`		

`addi $t0, $t1, 42` – $2128002A_{16}$

op	*src*	*dst*	*imm*
001000	01001	01000	0000 0000 0010 1010
`addi`	`$t1`	`$t0`	42

The University of Nottingham

Registers vs Memory

- Processor can access registers directly
- Limit of 32 registers
  - Most programs require much more data
- What about larger data structures?
  - Document in a text editor
  - Program code
  - Graphics image displayed on screen
- Extra data must be kept in memory

load transfer data from memory to register

store transfer data from register to memory

The University of Nottingham

Load Word

`lw` *dst*, *n*(*src*)

- Load word at address *src* offset *n* into *dst*
- *dst* := `M[`*src* + *n*`]`
- `M[`*addr*`]` – word/half/byte at address `addr`
- Offset *n* is 16 bits (`lw` is an I-format instruction)
  - Can write `lw` *dst*, (*src*) if *n* is zero
- The address `src` + *n* must be *word-aligned*
  - i.e. divisible by 4

Store Word

`sw` *dst*, *n*(*src*)

- Stores word in *dst* into address *src* offset *n*
- `M[`*src* `+` *n*`]` `:=` *dst*
- Note *dst* is not the 'destination' this time
- Offset *n* is 16 bits (`sw` is an I-format instruction)
- Again, the address `src` `+` *n* must be *word-aligned*

Example: Loads and Stores

Before

Address	Data
10010000	$7C0802A6_{16}$
10010004	$BE81FFD0_{16}$

After

Address	Data
10010000	$BE81FFD0_{16}$
10010004	$7C0802A6_{16}$

Assembly Code

- Initially `$s0` $= 10010000_{16}$

```
lw $t0, ($s0)
lw $t1, 4($s0)
sw $t0, 4($s0)
sw $t1, ($s0)
```

- Afterwards,
  `$t0` $= 7C0802A6_{16}$
  `$t1` $= BE81FFD0_{16}$

Bytes

- Not all data is word-sized, e.g. ASCII characters
- No alignment requirements
  - Addresses are always byte-aligned

Byte Transfer Instructions

- `lbu` *dst*, *n*(*src*) – load byte unsigned
  - Loads `M[`*src* + *n*`]` into the lower 8 bits of *dst*
  - Sets remaining 24 bits of *dst* to zero
  - Ignore '*unsigned*' for now; more later
  - There's also `lb` *dst*, *n*(*src*) – load byte
- `sb` *dst*, *n*(*src*) – store byte
  - Stores bits 0 to 7 of *dst* into `M[`*src* + *n*`]`
  - Ignores remaining 24 bits of *dst*

Half-Words

- Half-words are 16 bits long (i.e. 2 bytes)
- Address *src* + *n* must be half-word aligned
  - Sum of *src* + *n* must be even

Half-Word Transfer Instructions

- `lhu` *dst*, *n*(*src*) – load half-word unsigned
  - Loads `M[`*src* + *n*`]` into the lower 16 bits of *dst*
  - Sets remaining 16 bits of *dst* to zero
  - Ignore '*unsigned*' for now; more later
  - There's also `lh` *dst*, *n*(*src*) – load half-word
- `sh` *dst*, *n*(*src*) – store half-word
  - Stores bits 0 to 15 of *dst* into `M[`*src* + *n*`]`
  - Ignores remaining 16 bits of *dst*

What does this program do?

```
        .data
num:    .word 0
        .text
        .globl main
main:   la $s0, num
        li $v0, 5 # read_int
        syscall

        sw $v0, ($s0)
        lbu $a0, 1($s0)

        li $v0, 1 # print_int
        syscall
        j $ra
```

The University of Nottingham

High-Level Control Structures

- Unstructured programming leads to 'spaghetti code'
- The `goto` keyword is avoided in high-level programming
  - Reserved keyword in Java, but does nothing
  - Can be used for jumps in C, C++ and C#.
- Instead we use control structures like
  - `if-then-else`
  - `while` and `for`, `break` and `continue`
- Edsgar Dijkstra's "*Goto Statement Consdered Harmful*" Communications of the ACM, March 1968 http://www.acm.org/classics/oct95/
  - See also http://en.wikipedia.org/wiki/GOTO or http://reference.com/browse/wiki/GOTO

The University of Nottingham

Control Flow in Assembly

- No high-level control structures in assembly
- *Branch* (or jump) instructions change the PC
  - Hence affects the next instruction to be executed
  - The sequence of branches in a program is its *control flow*
- *Conditional* branches depend on a previous comparison
  - Otherwise we carry on with the following instruction
- *Unconditional* branches always jumps to the given location

Basic Branching

`j` *label* – jump (unconditional branch)

- Continues execution at *label* — `PC := [`*label*`]`

`beq` *dst*`,` *src*`,` *label* – branch on equal

- Continues execution at *label* if $dst = src$
- `if(`*dst* `==` *src*`) PC := [`*label*`]`

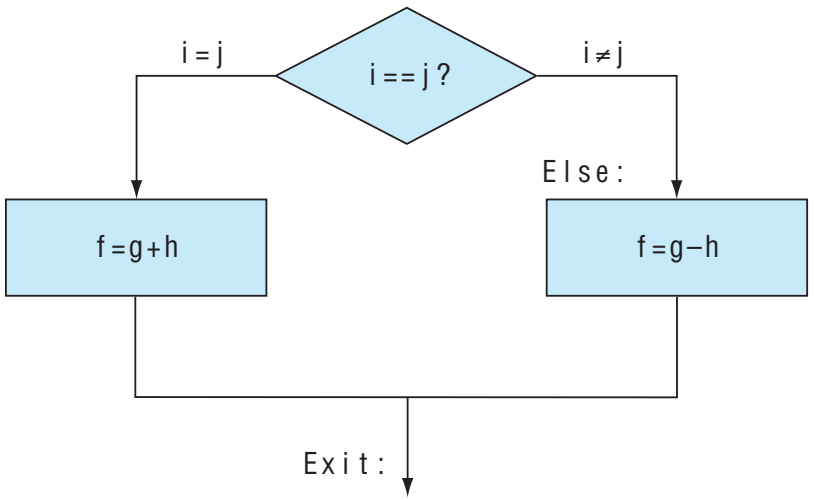
`bne` *dst*`,` *src*`,` *label* – branch on not-equal

- Continues execution at *label* if $dst \neq src$
- `if(`*dst* `!=` *src*`) PC := [`*label*`]`

- Labels identify addresses of program code as well as data

The University of Nottingham

If-Then-Else



- How do we implement the following Java fragment?

```
if(i == j)
        f = g + h;
else
        f = g - h;
```

- Assume

i	j	f	g	h
$s0	$s1	$s2	$s3	$s4

The University of Nottingham

Implementing Decisions

```
        bne $s3, $s4, else_i_ne_j
        add $s0, $s1, $s2    # then-block
        j if_i_eq_j_end
else_i_ne_j:
        sub $s0, $s1, $s2    # else-block
if_i_eq_j_end:
```

- Branch on opposite condition (`i` $\neq$ `j`) to the `else`-block
- So if `i` $=$ `j`, fall through to the `add` instruction
- Last instruction of `if`-block jumps over the `else`-block
- What if there is no `else`-block?

The University of Nottingham

Example: Checking for a Passcode

```
        # ...
main:   li $v0, 5 # read_int
        syscall

        li $t0, 42    # secret reply
        beq $v0, $t0, correct
        la $a0, go_away
        j check_end
correct:
        la $a0, hello
check_end:

        li $v0, 4 # print_string
        syscall
        j $ra
```

Thursday quiz

Most of the following questions are multiple choice. There is at least one correct choice but there may be several. For each of the questions list all the roman numerals corresponding to correct answers but none of the incorrect ones.
Questions are marked as follows:

no errors	5 points
1 error	3 points
2 errors	1 point
$\geq$**3 errors**	0 points

Thursday quiz

1. What are important aspects of the von Neumann architecture?
   a. Instructions are data.
   b. Separate memory for programs and data.
   c. Invented in Germany.
   d. Uses a Program Counter (PC).
   e. Superseded by the Harvard Architecture.

Thursday quiz

2. What is MIPS?
   a. Multiple Instructions per step
   b. Microprocessor without Interlocked Processor States
   c. Microprocessor with Interlocked Processor States
   d. A typical RISC processor
   e. A typical CISC processor

Thursday quiz

3. What are typical features of assembly language?
   a. Universal language for different processors
   b. Processor specific
   c. Uses labels for memory addresses
   d. Uses hexadecimal notation for instructions
   e. Is a compromise between C and Java.

Thursday quiz

4. What can we say about MIPS' registers?
   a. Designed for different purposes
   b. 32 registers because the word size is 32.
   c. Saved registers are not changed by system calls
   d. Temporary registers are not changed by system calls
   e. Register 0 always contains 0.

Thursday quiz

5. What is the effect of `add $s1, $s1, $s0`?
   a. Stores the sum of $s1 and $s0 in $s0.
   b. Stores the sum of $s1 and $s1 in $s0.
   c. Stores the sum of $s1 and $s0 in $s1.
   d. Causes an interrupt because $s1 is used twice.
   e. $s1 remains unchanged because $s0 always contains 0.