Politechnika Wrocławska

Computer Architecture and Organization

Lecture 5

Dr. Radosław Michalski
Department of Computational Intelligence, Faculty of Computer Science and Management, Wrocław University of Science and Technology

Version 1.0, spring 2017

Source and licensing

The most current version of this lecture is here:
https://github.com/rmhere/lecture-comp-arch-org

Overview of this lecture

Memory organization

Bytes and words - MIPS

In MIPS:

- byte - 8 bits
- word - 4 bytes - 32 bits

Yet, depending on the ISA, the word length may be different.
See this Wikipedia article.

Memory organization

Addressing - introduction

- each memory cell stores 8 bits (1 byte)
- each register stores 32 bits (4 bytes, 1 word)
- then how do we match both?

Memory organization

Addressing

- memory is indexed (0 ... X)
- in 32-bit architecture we have 2^{32} indexes
- memory upper bound for 32-bit architecture is 2^{32} bytes (4 GB)
- memory upper bound for 64-bit architecture is 2^{64} bytes (16 EB)

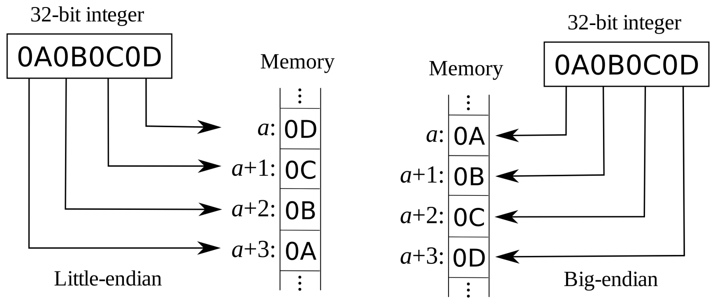
Memory organization

Endianness - introduction

- each register stores four 8-bit memory cells (1 word)
- we address the memory in the following way: 0, 4, ... X
- yet, how do we order the 8-bit chunks in the word?

Memory organization

Big-endian vs. little-endian



R. S. Shaw, public domain

Memory organization

nUxi problem

- assume 16-bit word (e.g., Intel 8086)
- each character is 8-bit encoded
- if we want to store the string "Unix", we'll use two words
- yet, the endianness heavily determines how it will be stored
- problems only when transferring to different machines/systems

Memory organization

Endianness - ctnd.

- **Little-endian** - Intel x86 and x86-64
- **Big-endian** (network byte order) - IBM System/360, z/Architecture, IPv4, IPv6, TCP, UDP
- **Bi-endian** - ARM (v 3+), PowerPC, Alpha, SPARC V9, MIPS, PA-RISC, SuperH SH-4 and IA-64

Memory organization

MIPS addressing modes

- register addressing
- immediate addressing
- PC-relative addressing
- base addressing
- pseudo-direct addressing

Memory organization

Addressing modes: register, immediate

Register (direct):

- operands in registers
- add $rd, $rs, $rt

Immediate:

- operand provided directly
- addi $rd, $rs, 5

Memory organization

Addressing modes: base

Base (displacement):

- address of operand is a sum of immediate and value in register
- the register is called base that may point to a structure or some other collection of data and immediate value is loaded at a constant offset from the beginning of the structure. The offset specifies how far the location of the operand data from the memory location pointed by the base.
- lw R4, 100(R1)

Data types

Data types (memory)

.ascii str

- string without a null terminator

.asciiz str

- string with a null terminator ("z" - zero), like in C

.byte b_1**, ...,** b_n

- n bytes contiguously

.halfword h_1**, ...,** h_n

- n halfwords contiguously

.word w_1**, ...,** w_n

- n words contiguously

.space numBytes

- numBytes of space in memory

Data types

MegaProcessor

Video

Computerphile - MegaProcessor

Data types

Sources & additional materials

- P.J. Jalics, T.S. Heines Transporting a portable operating system: UNIX to an IBM minicomputer, Communications of the ACM 26.12 (1983): 1066-1072 (scientific article)
- Summary of Addressing Modes in MIPS, University of Maryland, MD, United States (article)

Program execution

How the code is being stored and executed

General outlook:

- Princeton architecture: data and instructions share memory
- unless told otherwise, the CPU iterates through memory sequentially
- each instruction has its own address
- the CPU loads the word and tries to execute it
- question: is this word an instruction or data?
- knowing the address of the first one, you can determine the addresses of others

Program execution

Labels

- labels `label:` point to a section of code
- for your, not processor convenience
- we'll use them for controlling the flow of application

Program execution

Addressing - example in MARS

```
.data
    a:  .word 5
    b:  .word 6
    c:  .word 4
    d:  .word 3

.text
main:
    lw $t0, a
    lw $t1, b
    lw $t2, c
    lw $t3, d

    add $t4, $t0, $t1
    sub $t5, $t2, $t3
    sub $t6, $t4, $t5

    li $v0, 1
    add $a0, $zero, $t6
    syscall
```

Program execution

Labels - again

- what a variable declaration really is?
- `a:   .word 5`
- we point to the address in memory
- do we actually need `a:`?

Program execution

Branching

How to control the flow of application?

- until now - linear
- controlling the flow by branching
- `beq $r1,$r2,Label` - branch to label if equal
- `bne $r1,$r2,Label` - branch to label not equal
- otherwise - go to next instruction

Program execution

Jumping

- instruction `j` jumps to a given label
- unconditional branch

Program execution

Using beq/bne for conditions, jumping

How can we implement an IF instruction?

Pseudo code:

```
if t1 == t2 then t3=0
```

Assembly:

```
bne $t1, $t2, next
add $t3, $zero, $zero
next:  (...)
```

Program execution

Using beq/bne for conditions

How can we implement an IF ELSE instruction?

Pseudo code:

```
if t1 == t2 then t3=0 else t3=2
```

Assembly:

```
beq $t1, $t2, nullify
addi $t3, $zero, 2
j skip
nullify:  add $t3, $zero, $zero
skip:  (...)
```

Program execution

Loop implementation

How can we implement a FOR instruction?

Pseudo code:

```
for i = 1 ... 3 {exec}
```

Assembly:

```
add $t0, $zero, $zero
addi $t1, 3
loop:  beq $t0, $t1, exit
       addi $t0, $t0, 1
       exec:  ...
       j loop
exit:  (...)
```

Program execution

Less than (instructions vs. pseudoinstructions)

- so far we compare equality
- what about less/greater than?
- we have some pseudoinstructions: blt, bgt
- instruction: **SLT – set on less than**
- if `$s` is less than `$t`, `$d` is set to one. It gets zero otherwise.
- how to implement greater than using SLT?
- keep in mind that pseudoinstructions and instructions can even share the same name (operand types vary)

Program execution

Program counter - PC register

Program counter - register PC

- special register holding the address of the next instruction
- as the program execution is linear, it advances by word offset
- it can be modified indirectly (control flow)

Program execution

Jump vs. jump and link

- `j` jumps
- `jal` jumps and links
- `jal` copies the address of the next instruction into the register `$ra` (register 31) and then jumps to the address
- `jr $reg` jumps to register (sets PC to the value stored in $reg)

Program execution

Branch delay slot

- pipelining allows to execute many instructions in the same time
- jumping or branching instructions are not liked by pipelining
- the reason is that we have to optimize everything again
- branch delay slot simultaneously executes the next instruction with the branch
- how to avoid confusion: `nop` instruction, reordering
- is `JAL` actually storing PC + 4 or PC + 8 in $ra?

Program execution

Branch delay slot - example

How would this code behave?

```
j test
test:  addi $t3, $t3, 2
```

What happens if we have one jump after another?

```
j test
j test1
test:  addi $t3, $t3, 5
add $t3, $zero, $t3
test1:  addi $t3, $t3, 2
```

Program execution

Sources & additional materials

- MIPS32 Instruction Set Quick Reference, MIPS Technologies, Inc. (reference sheet)
- J.F. Frenzel, T.S. Heines, *MIPS Instruction Reference*, University of Idaho, ID, USA (course materials)
- M. Abrash, *“Michael Abrash’s Graphics Programming Black Book“*, Redline GmbH, 1997 (book)
- J. Pearson, *“Computer architecture“*, Uppsala University, Sweden (course materials)

Stack

Functions

How does a function work?

- usually, a function has some input and provides an output
- arguments are evaluated to values
- control flow jumps to function and executes it
- after the return clause, we come back

Stack

Functions - considerations

- functions also can declare variables (need memory)
- recursive functions require non-overlapping memory area
- how it is implemented in MIPS architecture?

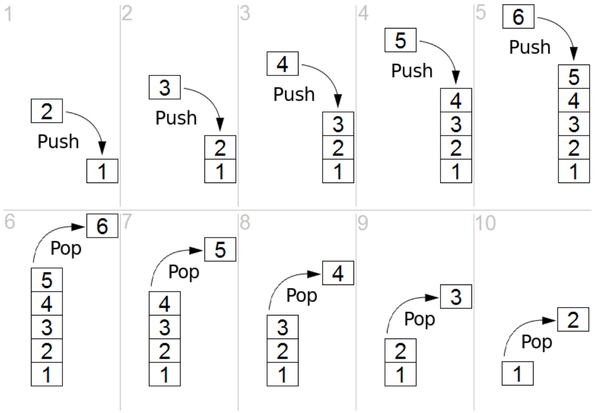
Stack

Stack - introduction

- stack - contiguous section of **memory**
- contains:
  - stack limit/origin (lowest valid address of the stack)
  - stack pointer (address of the stack)
  - stack bottom (highest valid address of the stack)
- **stack overflow** means that *stack pointer* < *stack limit*

Stack

Stack - operations



Maxtremus, public domain

Stack

Stack in MIPS - details

- stack pointer occupies register \$29 (`$sp`)
- it is not obligatory to use register \$29 as SP, just a convention
- have `$sp` set to the beginning of valid data in the stack

Stack

Stack - example

```
addi $t3, $zero, 9

push:  addi $sp, $sp, -4 # Decrement stack pointer by a word
       sw $t3, 0($sp) # Save $t3 to stack (indicated by $sp)
pop:   lw $t4, 0($sp) # Load the value at $sp to $t4
       addi $sp, $sp, 4 # Increment stack pointer by a word
```

Stack

Stack - example - multiple data

Push:

- decrement `$sp` once
- save multiple values (base addressing from `$sp`)

Pop:

- read multiple values (base addressing from `$sp`)
- increment `$sp` once

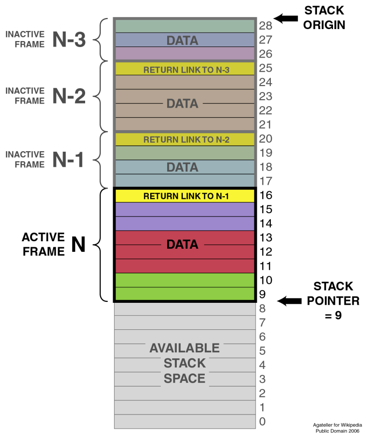
Stack

Stack and functions

- start with the main part of the code
- we need to know how much space we need for function calls
    - arguments
    - return value
- the above areas are called **stack frames**
- there is another pointer - **frame pointer** (`$fp`)
- frame pointer holds the last value of $sp before it moved to another stack frame

Stack

Stack frames



Agateller, public domain

Stack

Functions and registers

Caller and callee:

- *caller* calls *callee*
- callee does not know who called him

Architecture consideration:

- limited set of registers
- callee uses *saved registers* by convention (8)
- caller uses *argument registers* by convention (4)
- return values go to $v0 and $v1
- callee also can be a caller - what happens here?

Stack

The transistor

Video

AT&T Tech Channel - The Transistor: a 1953 documentary, anticipating its coming impact on technology

Stack

Sources & additional materials

- M. Hill, The MIPS Register Usage Conventions, University of Wisconsin-Madison, WI, United States (supplementary course materials)
- C. Lin, Computer Organization, University of Maryland, MD, United States (course materials)