1DT157 Digitalteknik och datorarkitekt

Digital technology and computer architecture, 5p
Announcements

• You can give Homeworks to the TA’s in hardcopy, drop them in their mailboxes
• Second Homework Tonight On the WEBPAGE
  Chapter 5, problems 7 and 10, Optimize the loop example of these slides
  – Second HW due by the next lab: 23rd May.
• ORAL EXAMS are arranged – see announcement in the WEBPAGE
• 2 classes on architecture
  – Pipelining, dynamic-execution, caches, etc
• 1 class on OS – virtual memory, segments
• 1 class odds+ends
• May 28: Review class (TAs)
• May 29: Computer History Guest Lecture
The Instruction Set Architecture Level

Chapter 5
The ISA level is the interface between the compilers and the hardware.
Instruction Formats (1)

Four common instruction formats:
(a) Zero-address instruction. (b) One-address instruction
(c) Two-address instruction. (d) Three-address instruction.
Some possible relationships between instruction and word length.
Expanding Opcodes (1)

An instruction with a 4-bit opcode and three 4-bit address fields.
An expanding opcode allowing 15 three-address instructions, 14 two-address instructions, 31 one-address instructions, and 16 zero-address instructions. The fields marked $xxxx$, $yyyy$, and $zzzz$ are 4-bit address fields.
## Addressing MODES

- **Immediate:**
  - MOV R1, 4

- **Direct:**
  - MOV R1, [0x1000]

- **Accumulator:**
  - ADD operand

- **Register:**
  - ADD R1,R2,R3

- **Register Indirect:**
  - MOV R1,(R2)

- **Indexed:**
  - MOV R1,8(R2)

- **Base + Index:**
  - MOV R1,8(R2+R3)

- **Stack:**
  - ADD
Addressing

<table>
<thead>
<tr>
<th>MOV</th>
<th>R1</th>
<th>4</th>
</tr>
</thead>
</table>

MOV R1,#0 ; accumulate the sum in R1, initially 0
MOV R2,#A ; R2 = address of the array A
MOV R3,#A+4096 ; R3 = address of the first word beyond A
LOOP: ADD R1,(R2) ; register indirect through R2 to get operand
      ADD R2,#4 ; increment R2 by one word (4 bytes)
      CMP R2,R3 ; are we done yet?
      BLT LOOP ; if R2 < R3, we are not done, so continue

An immediate instruction for loading 4 into register 1.
Register Indirect Addressing: a generic assembly program for computing the sum of the elements of an array.
Indexed Addressing (1)

A generic assembly program for computing the OR of \( A_i \) AND \( B_i \) for two 1024-element arrays.

```assembly
MOV R1,#0 ; accumulate the OR in R1, initially 0
MOV R2,#0 ; R2 = index, i, of current product: A[i] AND B[i]
MOV R3,#4096 ; R3 = first index value not to use
LOOP:    MOV R4,A(R2) ; R4 = A[i]
        AND R4,B(R2) ; R4 = A[i] AND B[i]
        OR R1,R4 ; OR all the Boolean products into R1
        ADD R2,#4 ; i = i + 4 (step in units of 1 word = 4 bytes)
        CMP R2,R3 ; are we done yet?
        BLT LOOP ; if R2 < R3, we are not done, so continue
```
Indexed Addressing (2)

| MOV | R4 | R2 | 124300 |

A possible representation of MOV R4, A(R2).
Reverse Polish Notation (1)

Each railroad car represents one symbol in the formula to be converted from infix to reverse Polish notation.
Reverse Polish Notation (2)

Decision table used by the infix-to-reverse Polish notation algorithm

<table>
<thead>
<tr>
<th>Car at the switch</th>
<th>↓</th>
<th>+</th>
<th>−</th>
<th>x</th>
<th>/</th>
<th>(</th>
<th>)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Most recently arrived car on the Texas line</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>↓</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>+</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>−</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>x</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>/</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>(</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Reverse Polish Notation (3)

<table>
<thead>
<tr>
<th>Infix</th>
<th>Reverse Polish notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A + B \times C$</td>
<td>$A B C \times +$</td>
</tr>
<tr>
<td>$A \times B + C$</td>
<td>$A B \times C +$</td>
</tr>
<tr>
<td>$A \times B + C \times D$</td>
<td>$A B \times C D \times +$</td>
</tr>
<tr>
<td>$(A + B) / (C - D)$</td>
<td>$A B + C D - /$</td>
</tr>
<tr>
<td>$A \times B / C$</td>
<td>$A B \times C /$</td>
</tr>
<tr>
<td>$((A + B) \times C + D)/(E + F + G)$</td>
<td>$A B + C \times D + E F + G + /$</td>
</tr>
</tbody>
</table>

Some examples of infix expressions and their reverse Polish notation equivalents.
### Evaluation of Reverse Polish notation Formulas

<table>
<thead>
<tr>
<th>Step</th>
<th>Remaining string</th>
<th>Instruction</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8 2 5× + 1 3 2× + 4 ÷ /</td>
<td>BIPUSH 8</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>2 5× + 1 3 2× + 4 ÷ /</td>
<td>BIPUSH 2</td>
<td>8, 2</td>
</tr>
<tr>
<td>3</td>
<td>5× + 1 3 2× + 4 ÷ /</td>
<td>BIPUSH 5</td>
<td>8, 2, 5</td>
</tr>
<tr>
<td>4</td>
<td>× + 1 3 2× + 4 ÷ /</td>
<td>IMUL</td>
<td>8, 10</td>
</tr>
<tr>
<td>5</td>
<td>+ 1 3 2× + 4 ÷ /</td>
<td>IADD</td>
<td>18</td>
</tr>
<tr>
<td>6</td>
<td>1 3 2× + 4 ÷ /</td>
<td>BIPUSH 1</td>
<td>18, 1</td>
</tr>
<tr>
<td>7</td>
<td>3 2× + 4 ÷ /</td>
<td>BIPUSH 3</td>
<td>18, 1, 3</td>
</tr>
<tr>
<td>8</td>
<td>2× + 4 ÷ /</td>
<td>BIPUSH 2</td>
<td>18, 1, 3, 2</td>
</tr>
<tr>
<td>9</td>
<td>× + 4 ÷ /</td>
<td>IMUL</td>
<td>18, 1, 6</td>
</tr>
<tr>
<td>10</td>
<td>+ 4 ÷ /</td>
<td>IADD</td>
<td>18, 7</td>
</tr>
<tr>
<td>11</td>
<td>4 ÷ /</td>
<td>BIPUSH 4</td>
<td>18, 7, 4</td>
</tr>
<tr>
<td>12</td>
<td>÷ /</td>
<td>ISUB</td>
<td>18, 3</td>
</tr>
<tr>
<td>13</td>
<td>/</td>
<td>IDIV</td>
<td>6</td>
</tr>
</tbody>
</table>

Use of a stack to evaluate a reverse Polish notation formula.
Orthogonality of Opcodes and Addressing Modes (1)

A simple design for the instruction formats of a three-address machine.
Orthogonality of Opcodes and Addressing Modes (2)

A simple design for the instruction formats of a two-address machine.
Overview of the Pentium 4 ISA Level

The Pentium 4’s primary registers.
## Data Types on the Pentium 4

The Pentium 4 numeric data types. Supported types are marked with ×.

<table>
<thead>
<tr>
<th>Type</th>
<th>1 Bit</th>
<th>8 Bits</th>
<th>16 Bits</th>
<th>32 Bits</th>
<th>64 Bits</th>
<th>128 Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signed integer</td>
<td>×</td>
<td>×</td>
<td></td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unsigned integer</td>
<td>×</td>
<td>×</td>
<td></td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary coded decimal integer</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating point</td>
<td></td>
<td></td>
<td></td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
</tbody>
</table>
The Pentium 4 Instruction Formats

The Pentium 4 instruction formats.
The Pentium 4 Addressing Modes

(1)

<table>
<thead>
<tr>
<th>R/M</th>
<th>MOD</th>
<th>MOD</th>
<th>MOD</th>
<th>MOD</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>M[EAX]</td>
<td>M[EAX + OFFSET8]</td>
<td>M[EAX + OFFSET32]</td>
<td>EAX or AL</td>
</tr>
<tr>
<td>100</td>
<td>SIB</td>
<td>SIB with OFFSET8</td>
<td>SIB with OFFSET32</td>
<td>ESP or AH</td>
</tr>
<tr>
<td>101</td>
<td>Direct</td>
<td>M[EBP + OFFSET8]</td>
<td>M[EBP + OFFSET32]</td>
<td>EBP or CH</td>
</tr>
</tbody>
</table>
The Pentium 4 Addressing Modes

(2)

Stack frame

Other local variables

a [0] EBP + 8

a [1] EBP + 12

a [2] EBP + 16

EBP

i in EAX

SIB Mode references
M[4 * EAX + EBP + 8]
# MIPS ISA Level

- MIPS Registers

<table>
<thead>
<tr>
<th>Register name</th>
<th>Number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>constant 0</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>reserved for assembler</td>
</tr>
<tr>
<td>$v0</td>
<td>2</td>
<td>expression evaluation and results of a function</td>
</tr>
<tr>
<td>$v1</td>
<td>3</td>
<td>expression evaluation and results of a function</td>
</tr>
<tr>
<td>$a0</td>
<td>4</td>
<td>argument 1</td>
</tr>
<tr>
<td>$a1</td>
<td>5</td>
<td>argument 2</td>
</tr>
<tr>
<td>$a2</td>
<td>6</td>
<td>argument 3</td>
</tr>
<tr>
<td>$a3</td>
<td>7</td>
<td>argument 4</td>
</tr>
<tr>
<td>$t0</td>
<td>8</td>
<td>temporary (not preserved across call)</td>
</tr>
<tr>
<td>$t1</td>
<td>9</td>
<td>temporary (not preserved across call)</td>
</tr>
<tr>
<td>$t2</td>
<td>10</td>
<td>temporary (not preserved across call)</td>
</tr>
<tr>
<td>$t3</td>
<td>11</td>
<td>temporary (not preserved across call)</td>
</tr>
<tr>
<td>$t4</td>
<td>12</td>
<td>temporary (not preserved across call)</td>
</tr>
<tr>
<td>$t5</td>
<td>13</td>
<td>temporary (not preserved across call)</td>
</tr>
<tr>
<td>$t6</td>
<td>14</td>
<td>temporary (not preserved across call)</td>
</tr>
<tr>
<td>$t7</td>
<td>15</td>
<td>temporary (not preserved across call)</td>
</tr>
</tbody>
</table>
MIPS ISA Level

- MIPS registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s0</td>
<td>16</td>
<td>saved temporary (preserved across call)</td>
</tr>
<tr>
<td>$s1</td>
<td>17</td>
<td>saved temporary (preserved across call)</td>
</tr>
<tr>
<td>$s2</td>
<td>18</td>
<td>saved temporary (preserved across call)</td>
</tr>
<tr>
<td>$s3</td>
<td>19</td>
<td>saved temporary (preserved across call)</td>
</tr>
<tr>
<td>$s4</td>
<td>20</td>
<td>saved temporary (preserved across call)</td>
</tr>
<tr>
<td>$s5</td>
<td>21</td>
<td>saved temporary (preserved across call)</td>
</tr>
<tr>
<td>$s6</td>
<td>22</td>
<td>saved temporary (preserved across call)</td>
</tr>
<tr>
<td>$s7</td>
<td>23</td>
<td>saved temporary (preserved across call)</td>
</tr>
<tr>
<td>$t8</td>
<td>24</td>
<td>temporary (not preserved across call)</td>
</tr>
<tr>
<td>$t9</td>
<td>25</td>
<td>temporary (not preserved across call)</td>
</tr>
<tr>
<td>$k0</td>
<td>26</td>
<td>reserved for OS kernel</td>
</tr>
<tr>
<td>$k1</td>
<td>27</td>
<td>reserved for OS kernel</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>pointer to global area</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address (used by function call)</td>
</tr>
</tbody>
</table>
### MIPS Registers

#### Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>constant 0</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>reserved for assembler</td>
</tr>
<tr>
<td>$v0 - $v1</td>
<td>2-3</td>
<td>expression evaluation and function results</td>
</tr>
<tr>
<td>$a0 - $a3</td>
<td>4-7</td>
<td>arguments</td>
</tr>
<tr>
<td>$t0 - $t7</td>
<td>8-15</td>
<td>temporary, saved by caller</td>
</tr>
<tr>
<td>$s0 - $s7</td>
<td>16-23</td>
<td>temporary, saved by called function</td>
</tr>
<tr>
<td>$t8 - $t9</td>
<td>24-25</td>
<td>temporary, saved by caller</td>
</tr>
<tr>
<td>$k0 - $k1</td>
<td>26-27</td>
<td>reserved for kernel (OS)</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>points to middle of a 64K block in the data segment</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer (top of stack)</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer (beginning of current frame)</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address</td>
</tr>
<tr>
<td>HL, LO</td>
<td>-</td>
<td>store partial result of mult and div operations</td>
</tr>
<tr>
<td>PC</td>
<td>-</td>
<td>contains the address of the next instruction to be fetched (this is not a real MIPS register, and is only used to define instructions)</td>
</tr>
<tr>
<td>status</td>
<td>-</td>
<td>register 12 in coprocessor 0, stores interrupt mask and enable bits</td>
</tr>
<tr>
<td>cause</td>
<td>-</td>
<td>register 13 in coprocessor 0, stores exception type and pending interrupt bits</td>
</tr>
<tr>
<td>epc</td>
<td>-</td>
<td>register 14 in coprocessor 0, stores address of instruction causing exception</td>
</tr>
</tbody>
</table>
MIPS Addressing Modes

- Simple, Indexed is a pseudo instruction

<table>
<thead>
<tr>
<th>Format</th>
<th>Address computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(register)</td>
<td>contents of register</td>
</tr>
<tr>
<td>imm</td>
<td>Immediate</td>
</tr>
<tr>
<td>imm (register)</td>
<td>immediate + contents of register</td>
</tr>
<tr>
<td>label</td>
<td>address of label</td>
</tr>
<tr>
<td>label ± imm</td>
<td>address of label + or − immediate</td>
</tr>
<tr>
<td>label ± imm (register)</td>
<td>address of label + or − (immediate + contents of register)</td>
</tr>
</tbody>
</table>
MIPS Instruction formats

- 3 basic types: R (3 reg), I (immediate), J (jump …)

<table>
<thead>
<tr>
<th>Format</th>
<th>Bits 31-26</th>
<th>Bits 25-21</th>
<th>Bits 20-16</th>
<th>Bits 15-11</th>
<th>Bits 10-6</th>
<th>Bits 5-0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
</tr>
<tr>
<td>I</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td>imm</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>address</td>
</tr>
</tbody>
</table>
MIPS Tutorial: General

- Think of assembly language as a very low level programming language.
- There are very few instructions.
- Many things that you can do in one step in high level languages you have to do in many steps

Why learn Assembly Language?
- It is the language of the machine. Computers don’t understand C or Java directly.
- We’ll see how we can implement assembly language.
- It helps you to understand how compilers work
MIPS Tutorial: Registers

- The MIPS processor has 32 special variables called registers.
- These registers can hold 32 bits (4 Bytes).
- Some of the registers have special uses. The registers have the names $0$-$31$, they also have other names.
- In the next few lectures we will be concerned with the following registers:

<table>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t0$-$t7$</td>
<td>8-15</td>
<td>Temporaries</td>
</tr>
<tr>
<td>$s0$-$s7$</td>
<td>16-23</td>
<td>Saved</td>
</tr>
<tr>
<td>$t8$-$t9$</td>
<td>24-25</td>
<td>more temporaries</td>
</tr>
</tbody>
</table>
MIPS Tutorial: Register usage

• When you are writing your assembly language programs, a $ means that there is a register.

• While the assembler can often guess what you mean it is better to write what you mean.

• The above instruction could be rewritten as
  addi $8,$8,1

• If you wrote
  add 8,8,1

• The assembler would have a hard job of guessing what you mean.
MIPS Tutorial: 3-operand inst.

Pseudo C code
$s0 = $s5 + $t0

Assembly language:
add $s0,$s5,$t0

Arithmetic instructions have three arguments the first two must be registers and the last is a register or a small constant (more later).

Arithmetic instructions, first argument is the destination.

*Important*: Arithmetic instructions can only have 3 arguments.
MIPS Tutorial: Temp. registers

Pseudo C code
$s0 = $s1 + $s2 + $s4 + 2*$s5

Assembly language:
add $t0,$s1,$s2
add $t0,$s4,$t0
add $t1,$s5,$s5
add $s0,$t0,$t1

The add instruction does not get confused if the destination register is the same as a source register.
MIPS Tutorial: Immediates

- What about constants?
- How do we do things like $t0 = t0 + 1$?
- We can’t just magic the values into registers we have to load values in there.

- The last operand of an add instruction can be a small constant (a 16-bit number). The new form of the instruction is called addi, the i stands for immediate.
- addi $t0,$t0,1
MIPS Tutorial: Immediates

- How do we put a value in a register?
- The MIPS processor has a special register, number 0, which is hardwired to be the value 0. No matter what you do to that register it stays at that value.

- This register is called $0 or $zero.

- How do I set $s0 to be 34?
  
  add $s0,$zero,34
MIPS Tutorial: Immediates

- There is no direct way of loading large constants into a register. It must be done in two steps.
- For example to load the value 0x0fff0123 into the register $s0 we have to do the following:

  \[ \text{lui } \text{ } \text{lui } s0,0xffff0 \]

- This places the value 0xffff00000 into the register $s0.
- lui stands for load upper immediate.

  \[ \text{add } s0,s0,0x0123 \]
MIPS Tutorial: Immediates

• The assembler provides a number of pseudo instructions, that is instructions that look like atomic instructions that get turned into a sequence of instructions.

• One of them is li which allows you to load large constants into registers.

• The instructions on the previous slide can be abbreviated to:
  
  li $s0,0xffff00123
MIPS Tutorial: Recap Arithmetic

- 32 registers, each can hold 32 bit integers.
- register $0$ is fixed at the value 0.
- Arithmetic instructions, very limited format, all ways three arguments.
- Destination is the first register.
MIPS Tutorial: Load

• To read information from memory you use the `lw`, load word instruction.
• Assume $s0 holds the address 0x8000000 then
  \[ \text{\texttt{lw $t0,0($s0)}} \]
  \[ \text{\texttt{lw $t0,0($s0)}} \]
  \[ \text{\texttt{lw $t0,4($s0)}} \]
  \[ \text{\texttt{lw $t0,4($s0)}} \]
  \[ \text{\texttt{lw $t0,4($s0)}} \]
  \[ \text{\texttt{lw $t0,4($s0)}} \]

• Will load the contents of memory location 0x8000000 into $t0 and
  \[ \text{\texttt{lw $t0,0($s0)}} \]
  \[ \text{\texttt{lw $t0,4($s0)}} \]
  \[ \text{\texttt{lw $t0,4($s0)}} \]
  \[ \text{\texttt{lw $t0,4($s0)}} \]
  \[ \text{\texttt{lw $t0,4($s0)}} \]
  \[ \text{\texttt{lw $t0,4($s0)}} \]

• Will load the contents of memory location 0x8000004 into $t0.
• Format of `lw`: \texttt{lw register,constant(register)}
• The constant cannot be a register
MIPS Tutorial: Load

- How do we load an address into a register? We can use the same trick as in the previous lecture, but there is a pseudo instruction:
  ```
  la $t0,address
  ```

- There is a reason that you use `la` rather than `li`, but I can’t tell you what it is yet.

- When you start doing your labs you’ll start to learn how to use labels.
MIPS Tutorial: Store

• To store a value from a register into a memory location you use, \texttt{sw}, store word. This instruction has the same format as \texttt{lw}.

  \begin{verbatim}
  la $s0,0x8000000
  li $t0,10
  sw $t0,0($s0)
  sw $t0,4($s0)
  \end{verbatim}

• This puts the value 10 into locations 0x8000000 and 0x8000004.
MIPS Tutorial: Addressing Modes

• To access the ith element of an integer array you need to access the memory address
  – Base Address + 4 * i

• Sometimes, especially when you are dealing strings you want to read and write bytes.

• The MIPS processor has two instructions lb and sb which read and write bytes, these have the same format as lw and sw.
Recap Load/Store

• `la` to load an address into a register.
• Remember there is a distinction between the address and the value stored at that and `swaddress`. (Pointers and values).

• `lw` load a value into a register from memory, `sw` store a value and `sw` from a register to memory.
• Integer arrays, multiply by 4.
• Strings and byte arrays, use `lb` and `sb`. 
MIPS Tutorial: Control Flow

- Labels
- Making Decisions beq and bne
- Jumps.
- Some example programs

- Programs and Data live in Memory.
- The processor does not understand strings of characters such as add for instructions.
- Each instruction is a unique number. For example the instruction
  \[ \text{lw } $4, \ 0($29) \]
  has the code 0x8fa40000.
- Each instruction has an address.
MIPS Tutorial

- Programs and data are in the same memory. The processor just fetches numbers from memory executes instructions. Almost every combination of bits is an instruction. Thus you could ask the machine to execute a sound file with unpredictable consequences.

- Also, the machine has no way of knowing if a value in memory is data or an instruction. So if you do a lw or a sw you better know what your reading or writing.

- All this has a positive side, you can write programs that write other programs (Can you think of any?).

- Your assembly language programs will fail in very strange ways, you better get used to understanding the code you write.
MIPS Tutorial: PC addresses

• Every instruction has an address.
• Sometimes you need to know the address of an instruction.
• Every instruction takes up exactly 4 bytes of memory (on the MIPS), it is possible in theory to work out the address of any instruction if you know the address of the first instruction.

• Working out the address of an instruction can be tedious, luckily
• there is a program called an assembler which works this out.
MIPS Tutorial

```
.data
n1: .word 10
.text
.globl main
main: la $t0,n1
lw $s0,0($t0)
addi $s0,$s0,1
sw $s0,0($t0)
jr $31
```

- main is the address of the first instruction. n1 is the address of a piece of data.
MIPS Tutorial: BEQ, BNE

- All the programs we have looked at so far have been linear. We need a way of doing different things depending on the values of registers.
- The MIPS processor provides two instructions for making decisions:
  
  \[
  \text{beq} \quad \text{Branch if Equal}
  \]
  
  \[
  \text{bne} \quad \text{Branch if not equal}
  \]

- General format:
  
  \[
  \text{beq} \; \$\text{register1},\$\text{register2},\text{Label}
  \]
beq $register1,$register2,Label

If $register1$ is equal to $register2$ then goto Label otherwise execute the next instruction.

bne $register1,$register2,Label

If $register1$ is not equal to $register2$ then goto Label otherwise execute the next instruction.
MIPS Tutorial: If then

Pseudo C code:
if $s1 == $s2 then $s3 = 0 ;

Assembly language:

```
bne $s1,$s2,skip
add $s3,$0,$0
```

skip:    Next Instruction
MIPS Tutorial: if then else

Pseudo C code:
if $s1 = $s2 then $s3 = 0 else $s3 = $s3 + 1 ;

Assembly code:

```
beq $s1,$s2,set_zero
addi $s3,$s3,1
j skip2

set_zero:     add $s3,$0,$0
skip2:        Next Instruction
```

- Jump instruction. This is like an unconditional branch: make the next instruction the label.
MIPS Tutorial

There is often more than one way of writing the same piece of code.

For example:

```
bne $s1,$s2,skip
add $s3,$0,$0
```

skip: Next Instruction

Can also be written as:

```
beq $s1,$s2,settozero
j skip
```

settozero: add $s3,$0,$0

skip: Next Instruction
MIPS Tutorial: Loops

Assume that the base of the integer array A is stored in $s0.

```
addi $t0, $0, 0
loop:      addi $t1, $0, 10
           beq $t0, $t1, exit
           add $t2, $t0, $t0  # $t2 = $t0*4
           add $t2, $t2, $t2  # $t2 = address of A[$t0]
           sw $0,0($t2)
           addi $t0, $t0, 1
           j loop

exit:      Pseudo C code
           for($t0 = 0; $t0 != 10 ; $t0 = $t0 +1 ) {
              A[$t0] = 0
           }
```
Look at the loop on the previous slide. How can we make it more efficient?

- We don’t have to load 10 in $t1$ each time around the loop because $t1$ does not change (this is called a loop invariant).
- We can do the test to exit the loop at the end of the loop, because we know the loop executes at least once.
- Instead of multiplying by 4 each time around the loop, we can add 4 to $t0$ each time and exit when $t0$ is equal to 40.

Exercise: rewrite the above code.
MIPS Tutorial: SLT

• So far we have only compare if registers are equal or different.
• The MIPS provides no branch on less than.

• Instead there is the slt instruction.

• General format
  slt $Rdest,$Rsrs1,$RSrc2

• Set Register Rdest to 1 if register Rsrs is less than Rsrs2, set to zero otherwise.
MIPS Tutorial

• Pseudo C code:
  if $s1 < $s2 then $s3 = 0

• Assembly:

  slt $t0,$s1,$s2
  beq $t0,$zero,skip
  add $s3,$zero,$zero

skip:
MIPS Tutorial: functions

• As in high level languages, when programming in assembly language you should split up your program into smaller functions, that you can reuse.

• One of the key ideas with functions is that you can call them from any where and return back to where you called the function from.

• The MIPS processor has two instructions that enable you to call functions, jr and jal.
MIPS Tutorial: functions

• Jump and link.
  
  jal label

• Copies the address of the next instruction into the register $ra (register 31) and then jumps to the address label.

  jr $register jumps to the address in $register
  most common use

  jr $ra
MIPS Tutorial: functions

.data
str: .asciiz "Hello World!.

.text
.globl main #necessary for the assembler
main: jal message
ejal message
li $v0,10
syscall #exit the program gracefully
message: la $a0,str
li $v0,4
syscall #Magic to printhings on the screen.
jr $ra
MIPS Tutorial: functions

- There are many ways of passing values to functions, but there is a convention that most programs on the MIPS follow.
- `$a0-$a3` (registers 4 to 7) arguments 1-4 of a function.
- `$v0-$v1` (registers 2 and 3) results of a function.

```
li $a0,10
li $a1,21
li $a3,31
jal silly  #result is is $v0.
li $v0,10
syscall

silly:
    add $t0,$a0,$a1
    sub $v0,$a3,$t0
    jr $ra
```
MIPS Tutorial: functions

On the previous slide in our function we needed an extra register to do part of a calculation. How do we know what registers to use?

• As with function calls there is a convention.
• $s0$-$s7$ the saved registers, these registers should be unchanged after a function call.
• $t0$-$t9$ these are temporaries, are are not necessarily preserved across function calls.
• So in the previous example it would of been a bad thing to use $s0$ in the function silly.

• What happens if we run out of registers? What happens if we have to use $s0$?
MIPS Tutorial: functions

jal silly

after1:  ...
...
silly:  jal silly2
after2:  ...
...
jr $ra

silly2:  ...
jr $ra

• So we have to save $ra as well!
MIPS Tutorial: STACK

• A stack is a data structure, at least two operations:
  – push put a value on the top of the stack
  – pop remove an item from the top of the stack.

• The important thing about a stack is that it is a LIFO (Last in First Out) data structure. This is useful for nested functions.

• You store your temporary data by pushing it onto the stack and restore things by popping things from it
MIPS Tutorial: STACK

- The MIPS has no specialized push and pop instructions (Other processors do).
- Instead the stack is implemented using the register $sp (number 29), lw and sw.
- Unless you are writing an operating system the register $sp points to the top of the stack.
- On the MIPS stacks grow downwards.
- You have to manipulate the value of the register $sp and then use store and load.
MIPS Tutorial: STACK

• To push the contents of register $s0 onto the stack. Do the following:
  
  addi $sp,$sp,−4
  sw $s0,0($sp)

• To pop the top of the stack into register $s0 do the following:
  
  lw $s0,0($sp)
  add $sp,$sp,4

• Basic rules:
  – Everything you push onto the stack, you must pop from the stack.
  – Never touch anything on the stack that does not belong to you.
MIPS Tutorial: STACK

silly:

```
addi $sp,$sp,-4     # PUSH
sw $ra,0($sp)       # ra
jal silly2
lw $ra,0($sp)       # POP
add $sp,$sp,4       # ra
jr $ra
```
MIPS Tutorial: STACK

silly:

```
addi $sp,$sp,-4  # PUSH
sw $s0,0($sp)    # s0
addi $sp,$sp,-4  # PUSH
sw $ra,0($sp)    # ra
...
jal silly2
...
lw $ra,0($sp)    # POP
add $sp,$sp,4    # ra
lw $s0,0($sp)    # POP
add $sp,$sp,4    # s0
jr $ra
```
MIPS Tutorial: STACK

silly:
  addi $sp,$sp,-8
  sw $s0,4($sp)
  sw $ra,0($sp)
...
  jal silly2
...
  lw $ra,0($sp)
  lw $s0,4($sp)
  addi $sp,$sp,8
  jr $ra

- General rule (applies to all programs you’ll ever write):
  - Write the inefficient version once that is correct; then optimize.
MIPS Tutorial: functions

- All arguments and all results via the stack

- Optimization:
  - $a0$-$a3$ (registers 4 to 7) arguments 1-4 of a function.
  - $v0$-$v1$ (registers 2 and 3) results of a function
Example

- Factorial
  - \( N! = N \times (N-1) \times (N-2) \times \ldots \times 1 \)

- Recursive function: a function that calls itself
  - Can only be implemented on the stack
Example: factorial

result: .space 4 # the place for the result
    .text
    .globl main
main:  addi $sp,$sp,−4 # save the return address.
    sw $ra,0($sp)

    lw $a0, 5
    jal fact
    la $t0,result
    sw $v0, 0($t0)

    lw $ra,0($sp)
    addi $sp,$sp,4
    jr $ra
Example: factorial

```assembly
fact:
addi $sp,$sp,-4
sw $ra,0($sp) #push $ra on the stack

# fact of 0 is 1
bne $a0,$zero,not_zero

# Set $v0 to be 1
addi $v0,$zero,1

# Restore $ra from the stack
lw $ra,0($sp) # Read $ra from the stack
addi $sp,$sp,4 # restore stack pointer.
jr $ra
```
Example: factorial

not_zero:

    addi $sp,$sp,-4
    sw $a0,0($sp) #push n on the stack ($a0=n)
    #So call fact with our new parameter
    addi $a0,$a0,-1
    jal fact # $v0=fact(n-1)

    lw $t0,0($sp) #restore n from the stack.
    addi $sp,$sp,4

    mul $v0,$v0,$t0 #$v0 = fact(n-1)($v0)*n($t0)

    #Restore the stack for $ra
    lw $ra,0($sp)
    addi $sp,$sp,4
    jr $ra