Instructions and compilation of basic programming language constructs

Lubomír Bulej
KDSS MFF UK
Recall: computer is a machine

Executes a program
- Sequence of instructions stored in memory.
- Executes an instruction and moves to “next” one.
  - Does not “know” what it is doing, nor “understands” the big picture.

Instructions are very simple
- Mostly operations on numbers.

Everything is encoded into numbers
- Not only the input and output data...
  - Text, images, music, 3D scene, ...
- ... but also the program being executed!
It is easy to see by formal-logical methods that there exist certain [instruction sets] that are in abstract adequate to control and cause the execution of any sequence of operations...

... The really decisive considerations from the present point of view, in electing an [instruction set], are more of a practical nature: simplicity of the equipment demanded by the [instruction set], and the clarity of its application to the actually important problems together with the speed of its handling of those problems.

– Burks, Goldstine, and von Neumann, 1947
What is programming, really?

Semantic gap

Program

0101001010010
0110101001101
0111010110101

....

Shrink image
Delete paragraph
Play song

.....
Bridging the semantic gap

Compiler translates high-level programming language into machine-specific assembly language.

Assembler translates assembly language into binary code chunks. Linker merges binary code chunks into executable image.

011011011010101 0101001010010 0110101001101 0111010110101 ....

LW $16, 0($2) 
MULI $2, $5, 4 
ADD $2, $4, $2 
SW $16, 0($2) 
...
There must certainly be instructions for performing the fundamental arithmetic operations.

– Burks, Goldstine, and von Neumann, 1947
Adding (two variables)

- The most basic of basic operations.

```
add a, b, c  # a = b + c
```

- Add variables b and c and store result in a.

- One operation, always three variables.
  - Regularity helps make the hardware simple.

Adding three (four) variables

- Two (three) instructions needed

```
add a, b, c  # a = b + c
add a, a, d  # a = b + c + d
add a, a, e  # a = b + c + d + e
```
Compiling assignments (1)

Simple expression

\[ a = b + c; \]
\[ d = a - e; \]

Corresponding MIPS assembly

\[ add \ a, \ b, \ c \quad \# a = b + c \]
\[ sub \ d, \ a, \ e \quad \# d = a - e \]
Compiling assignments (2)

Complex expression
\[ f = (g + h) - (i + j); \]
- Compiler must break down the statement into multiple assembly instructions.

Corresponding MIPS assembly
```
add t0, g, h       # t0 = g + h
add t1, i, j       # t1 = i + k
sub f, t0, t1      # f = t0 - t1
```
- Programmer only deals with the 5 variables.
- Compiler determines where to store the (temporary) intermediate results.
Operands

**Instruction operands restricted to registers**
- Limited number of special locations in the hardware visible to programmer.
  - 32 on the MIPS architecture.
  - More than 16-32 not necessarily better. Why?
- The size of a register is limited as well.
  - 32 bits (word) on the 32-bit MIPS architecture.

**Effective use of registers critical to performance**
- Compiler allocates registers as necessary to hold different values at different stages of program execution.
Register number in the instruction code

- 5 bits required to express registers 0 – 31.

Symbolic name in the assembly language

- Reflects agreed-upon usage of a register.
- $r0 ($zero) and $r31 ($ra) are special.

<table>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
<th>Usage</th>
<th>Name</th>
<th>Number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>The constant value 0.</td>
<td>$t8 – $t9</td>
<td>24 – 25</td>
<td>More temporaries.</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>Reserved for assembler.</td>
<td>$k0 – $k1</td>
<td>26 – 27</td>
<td>Reserved for OS kernel.</td>
</tr>
<tr>
<td>$v0 – $v1</td>
<td>2 – 3</td>
<td>Values of results and expressions.</td>
<td>$gp</td>
<td>28</td>
<td>Global pointer.</td>
</tr>
<tr>
<td>$a0 – $a3</td>
<td>4 – 7</td>
<td>Function arguments.</td>
<td>$sp</td>
<td>29</td>
<td>Stack pointer.</td>
</tr>
<tr>
<td>$t0 – $t7</td>
<td>8 – 15</td>
<td>Temporaries.</td>
<td>$fp / $s8</td>
<td>30</td>
<td>Frame pointer (if used).</td>
</tr>
<tr>
<td>$s0 – $s7</td>
<td>16 – 23</td>
<td>Saved registers.</td>
<td>$ra</td>
<td>31</td>
<td>Return address.</td>
</tr>
</tbody>
</table>
Compiling assignments using registers

Complex expression
\[
f = (g + h) - (i + j);
\]

Corresponding MIPS assembly
- The compiler assigned variables \( f, g, h, i, \) and \( j \) to registers \( $s0, $s1, $s2, $s3, \) and \( $s4 \).

\[
\begin{align*}
\text{add} & \quad $t0, \quad $s1, \quad $s2 & \quad \# \quad $t0 = g + h \\
\text{add} & \quad $t1, \quad $s3, \quad $s4 & \quad \# \quad $t1 = i + k \\
\text{sub} & \quad $s0, \quad $t0, \quad $t1 & \quad \# \quad f = $t0 - $t1
\end{align*}
\]
Memory operands

Everything is primarily kept in memory
- Variables and data structures contain more data elements than there are registers in a computer.
  - Only small amount of data can be kept in registers.

Arithmetic operations only work with registers
- *Data transfer instructions* needed to transfer data between memory and registers.
- Instructions must supply the memory *address*.
- Memory is a 1-dimensional array of bytes.
  - The address serves as a zero-based index.
  - 32-bit word addresses must be aligned to 4 bytes.
Data transfer instructions

Load/store word
- \( \text{lw} \) \( $rd, \text{imm16} \) \( ($rs) \)
  \( R[rd] = M[R[rs] + \text{signext32} \ (\text{imm16})] \)
- \( \text{sw} \) \( $rt, \text{imm16} \) \( ($rs) \)
  \( M[R[rs] + \text{signext32} \ (\text{imm16})] = R[rt] \)

Load/store byte
- \( \text{lb} \) \( $rd, \text{imm16} \) \( ($rs) \)
  \( R[rd] = \text{signext32} \ (M[R[rs] + \text{signext32} \ (\text{imm16})][7:0]) \)
- \( \text{lbu} \) \( $rd, \text{imm16} \) \( ($rs) \)
  \( R[rd] = \text{zeroext32} \ (M[R[rs] + \text{signext32} \ (\text{imm16})][7:0]) \)
- \( \text{sb} \) \( $rt, \text{imm16} \) \( ($rs) \)
  \( M[R[rs] + \text{signext32} \ (\text{imm16})][7:0] = R[rt][7:0] \)

1 addressing mode:
Base address in register, immediate offset in instruction.
Program fragment

```c
int a[100];
g = h + a[8];
```

Corresponding MIPS assembly

- Variables `g` and `h` assigned to `$s1` and `$s2`.
- The base (starting) address of array `a` is in `$s3`.
- The offset of `a[8]` is `8*sizeof(int)`

```assembly
lw $t0, 32 ($s3)     # $t0 = a[8]
add $s1, $s2, $t0     # g = h + a[8]
```
Program fragment
- Single assignment, two memory operands.

```c
text a[100];
a[12] = h + a[8];
```

Corresponding MIPS assembly
- Variable `h` assigned to `$s2`.
- The base address of array `a` is in `$s3`.

```
lw $t0, 32 ($s3)       # $t0 = a[8]
add $t0, $s2, $t0      # $t0 = h + a[8]
sw $t0, 48 ($s3)       # a[12] = h + a[8]
```
Avoid extra memory reads for (common) constants
  ● Incrementing/decrementing a loop control variable or an index, initializing sums and products, ...
    ○ Common values: 0, 1, -1, 2, ... (constant structure sizes)

Immediate operands
  ● addi $rd, $rs, imm16
    add immediate, $RD[R[rs]] = R[rs] + signext32 (imm16)
  ● li $rd, imm32
    load immediate, R[rd] = imm32

Zero is special (hardwired in $r0)
  ● move $rd, $rs = add $rd, $rs, $r0
    R[rd] = R[rs]
Logical operations

Operations on bits and bit fields within words

- Isolating, setting, and clearing bits.

Bitwise operations

- **and/or/xor/nor** $rd$, $rs$, $rt$
  - not $rd$, $rs = nor$ $rd$, $rs$, $rs/$r0

- **andi/ori/xori** $rd$, $rs$, imm16
  $R[rd] = R[rs]$ and/or/xor **zeroext32** (imm16)

Shift operations

- **sll/slr** $rd$, $rs$, shamt
  shift logical left/right, $R[rd] = R[rs] << / >>$ shamt

- **sra** $rd$, $rs$, shamt
  shift arithmetic right, $R[rd] = R[rs] >>>$ shamt
Compiling logical operations

Program fragment

```
shamt = (insn & 0x000007C0) >> 6;
```

Corresponding MIPS assembly

- Variables `shamt`, `insn` assigned to `$s1`, `$s2`.

```
and $t0, $s2, 0x7C0  # $t0 = insn & 0x7C0
srl $s1, $t0, 6      # shamt = $t0 >> 6
```
Distinguishes computer from calculator

- Choose which instructions to execute based on inputs and values created during computation.
  - Control statements in programming languages.

Conditional branches / jumps

- `beq $rd, $rs, addr`
  - branch if eq, if \( R[rs] == R[rt] \) then \( PC = addr \) else \( PC = PC + 4 \)

- `bne $rd, $rs, addr`
  - branch not eq, if \( R[rs] <> R[rt] \) then \( PC = addr \) else \( PC = PC + 4 \)

Unconditional jumps

- `j addr`
  - jump, \( PC = addr \)
Compiling if-then-else statement

Program fragment

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

Corresponding MIPS assembly

```
  bne $s3, $s4, Else # (i != j) ⇒ PC = Else
  add $s0, $s1, $s2   # f = g + h
  j End               # PC = End
Else:
  sub $s0, $s1, $s2   # f = g - h
End:
  ...
```

- Variables \( f, g, h, i, \) and \( j \) assigned to registers \( $s0, $s1, $s2, $s3, \) and \( $s4 \).
Compiling while loop

Program fragment

while (save[i] == k) do
    i = i + 1;

Corresponding MIPS assembly

- Variables i, k assigned to $s3, $s5, and the base address of array save is in $s6.

Loop:

sll $t1, $s3, 2  # $t1 = i × 4
add $t1, $t1, $s6  # $t1 = &save[i]
lw $t0, 0 ($t1)  # $t0 = save[i]
bne $t0, $s5, End  # (save[i] != k) ⇒ PC = End
addi $s3, $s3, 1  # i = i + 1
j Loop  # PC = Loop

End:
Instructions for making decisions (2)

Set on less than

- Check all relations (together with beq/bne)

Signed variant

- \texttt{slt} $\text{rd}$, $\text{rs}$, $\text{rt}$
  \[
  \text{if } R[rs] <_s R[rt] \text{ then } R[r] = 1 \text{ else } R[r] = 0
  \]

- \texttt{slti} $\text{rd}$, $\text{rs}$, \text{imm16}
  \[
  \text{if } R[rs] <_s \text{ signext32}(\text{imm16}) \text{ then } R[r] = 1 \text{ else } R[r] = 0
  \]

Unsigned variant

- \texttt{sltu} $\text{rd}$, $\text{rs}$, $\text{rt}$
  \[
  \text{if } R[rs] <_U R[rt] \text{ then } R[r] = 1 \text{ else } R[r] = 0
  \]

- \texttt{sltiu} $\text{rd}$, $\text{rs}$, \text{imm16}
  \[
  \text{if } R[rs] <_U \text{ zeroext32}(\text{imm16}) \text{ then } R[r] = 1 \text{ else } R[r] = 0
  \]
Compiling repeat-until loop

Program fragment

```
int i = 0;
do {
    i = i + 1;
} while (i < k);
```

Corresponding MIPS assembly

- Variables $i$, and $k$ assigned to registers $s3$, and $s5$.

```
move $s3, $zero  # i = 0
Loop:
    addi $s3, $s3, 1  # i = i + 1
    slt $t0, $s3, $s5  # $t0 = (i < k)
    bne $t0, $zero, Loop  # ($t0 != 0) ⇒ PC = Loop
End:
```
Program fragment

```c
int a[5] = { 1, 2, 3, 4, 5 };  
...  
int s = 0;  
for (int i = 0; i < 5; i++) {  
    s = s + a[i];  
}  
```
Compiling for loop (2)

Corresponding MIPS assembly

```assembly
move $s2, $zero          # s = 0
move $s1, $zero          # i = 0
j Condition             # PC = Condition

Body:
sll $t0, $s1, 2          # $t0 = i × 4
add $t0, $t0, $s0        # $t0 = &a[i]
lw $t1, 0 ($t0)          # $t1 = a[i]
add $s2, $s2, $t1        # s = s + a[i]
addi $s1, $s1, 1         # i = i + 1

Condition:
slti $t2, $s1, 5         # $t2 = (i < 5)
bne $t2, $zero, Body     # ($t2 != 0) ⇒ PC = Body

End:
```
Supporting procedures/functions (1)

Fundamental tool for structuring programs
- Call from anywhere, with input parameters.
- Return to point of origin, with return value.
- One of the ways to abstraction and code reuse.

Basic steps to execute a routine
- Put parameters in a place accessible to routine.
- Transfer control to the routine code.
- Acquire storage needed for the routine.
- Perform the desired task.
- Put result in a place accessible to caller.
- Return control to point of origin.
Jump and link (call)
- **jal addr**
  \[ \text{$ra} = R[31] = \text{PC} + 4; \text{PC} = \text{addr} \]
- **jalr $rs**
  jump and link register, \[ \text{$ra} = R[31] = \text{PC} + 4; \text{PC} = R[rs] \]

Indirect jump / return
- **jr $rs**
  jump register, \[ \text{PC} = R[rs] \]

Registers used for calling routines
- First four arguments passed in \[ $a0 - a3 \]
- Return value passed back in \[ $v0 - v1 \]
- Address where to return passed in \[ $ra ($r31) \]
Compiling simple function call

Program fragment

    print (add_four (a, b, c, d));

Corresponding MIPS assembly

- Variables \texttt{a}, \texttt{b}, \texttt{c}, and \texttt{d} assigned to $s0$, $s1$, $s2$, and $s3$.

```
move $a0, $s0
move $a1, $s1
move $a2, $s2
move $a3, $s3
jal add_four
    add $v0, $a0, $a1
    add $v1, $a2, $a3
    add $v0, $v0, $v1
jr $ra
jal print
    move $a0, $v0
    jal print
jr $ra
...  ...
```
Mechanism to store register contents in memory
- Caller expects to find its own values in registers after a routine returns.
- Routine works with more values than there are registers available.

Mechanism to pass parameters through memory
- There may be more than 4 parameters.

Mechanism to return values through memory
- The returned value may be a structure.

Mechanism to acquire storage for local variables
- Loop control variables, temporaries, ...
Allocating local storage

In memory, but where?
- The location cannot be fixed, because any routine can be called multiple times.
  - A routine can call itself, either directly, or transitively.
  - A routine can be called from multiple threads.

Stack data structure (Last In First Out)
- Stack pointer to the top of the stack
  - Address of last used memory location
- Push and pop operations
  - Decrement/increment stack pointer, store/retrieve value
- Access local data relative to stack pointer
- Fits the need to make nested function calls
Stack space allocation

Stack and register contents
- Before, during, and after routine call

High address

Low address

Saved return address
Saved registers (if any)
Local variables (if any)
Function arguments (calling)

Function arguments (called)
Saved return address (if any)
Saved registers (if any)
Local variables (if any)
Function arguments (calling)

Stack frame size

$sp$
Compiling a function call using stack

Program fragment

\[ s = \text{add\_two}(1, 2); \]

Corresponding MIPS assembly for the call

- Note: arguments would normally go only through registers.

```
addi $sp, $sp, -40       # Allocate stack frame (including space
...                     # for locals and all possible call arguments)
li  $a1, 2
sw  $a1, 4 ($sp)        # Put 2nd parameter on stack
li  $a0, 1
sw  $a0, 0 ($sp)        # Put 1st parameter on stack
jal  \text{add\_two}    # Call (jump and link) the routine
...                     # Call (jump and link) the routine
addi $sp, $sp, 40       # Deallocate stack frame
```


Compiling a function using stack (1)

MIPS assembly for `add_two()`

- Note: saving $ra ($s0, $s1) is not strictly necessary.
- Note: arguments loaded from the caller’s stack frame.

`add_two`:

```mips
addi $sp, $sp, -12           # Allocate stack frame
sw $ra, 8 ($sp)              # Store return address
sw $s1, 4 ($sp)              # Save register $s1
sw $s0, 0 ($sp)              # Save register $s0
lw $s0, 12 ($sp)             # Load 1st argument from stack
lw $s1, 16 ($sp)             # Load 2nd argument from stack
add $v0, $s0, $s1            # Calculate return value
```

... to be continued
MIPS assembly for add_two()

... continued

lw $s0, 0 ($sp)       # Restore register $s0
lw $s1, 4 ($sp)       # Restore register $s1
lw $ra, 8 ($sp)       # Restore return address
addi $sp, $sp, 12     # Deallocate stack frame
jr  $ra               # Return to caller

Compared to machines with HW stack support

- Stack frame (activation record) for each function is allocated as a whole, $sp remains fixed after allocation.
  - Not incrementally using push instructions.
- Space for all possible arguments is part of the activation record → not need to change $sp during execution.
Stack allocation with frame pointer

Stack and register contents

- **Before, during, and after** routine call

  - **High address**
  - **Old $sp** → Saved registers (if any)
    - Local variables (if any)
  - **New $sp** → Function arguments (calling)
  - **$fp** → Saved return address
    - Saved frame pointer
  - **Low address**
  - **Caller pushes arguments on the stack** (often using `push` instructions) just before the call
  - **Caller “removes” arguments from stack** by adjusting the stack pointer. Can be done by the callee before returning if the number of arguments is known.

- **Function arguments (calling)**
  - **$fp** → Saved return address
    - Saved frame pointer
  - **$sp** → Saved registers (if any)
    - Local variables (if any)
  - **Function arguments (calling)**
MIPS assembly for `add_two()`

`add_two:`

```
addi $sp, $sp, -4
sw $ra, 0 ($sp)
addi $sp, $sp, -4
sw $fp, 0 ($sp)
move $fp, $sp

addi $sp, $sp, -4
sw $s0, -4 ($fp)

lw $s0, 8 ($fp)
lw $s1, 12 ($fp)
add $v0, $s0, $s1
```

# “Push” return address on stack
# “Push” old frame pointer on stack
# Establish new frame pointer
# Allocate the rest of the stack frame
# Save $s0 ($fp-based addressing)
# Load 1st argument ($fp-based addressing)
# Load 2nd argument ($fp-based addressing)
# Calculate return value

... to be continued
Compiling with frame pointer (2)

MIPS assembly for add_two()

- Note: explicit stack adjustments intended to mimic function prologue (push ebp; mov esp, ebp) and epilogue (mov ebp, esp; pop ebp) typical for Intel.

... continued

lw $s0, -4 ($fp)     # Restore $s0 ($fp-based addressing)
move $sp, $fp        # Deallocate stack frame

lw $fp, 0 ($sp)      # “Pop” frame pointer
addi $sp, $sp, 4     # “Pop” return address
lw $ra, 0 ($sp)      # Return to caller
addi $sp, $sp, 4
jr $ra