Foundations of Programming Languages

2OPM Assembly (1/3): Introduction

Prof. Dr. Christoph Reichenbach

Fachbereich 12 / Institut für Informatik

17. Oktober 2014

2OPM: Two-Operand Pseudo-MIPS

- Synthetic Assembly Language
- Based on MIPS64, but most instructions use only two operands:

```
add $t0, $t1 #$t0 := $t0 + $t1
```

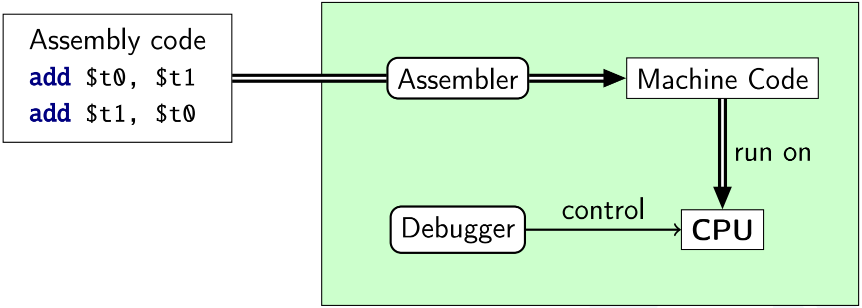
2OPM: Two-Operand Pseudo-MIPS

- Synthetic Assembly Language
- Based on MIPS64, but most instructions use only two operands:

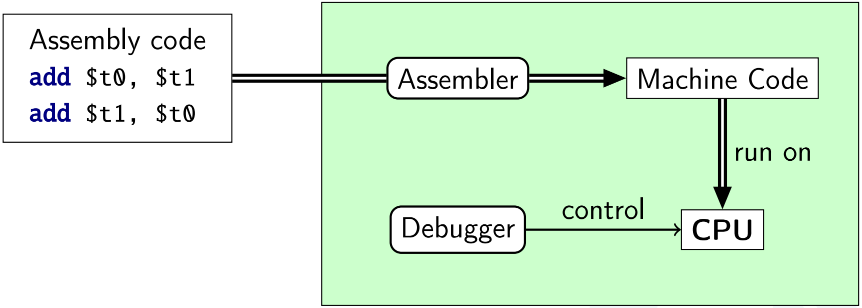
  ```
  add $t0, $t1 #$t0 := $t0 + $t1
  ```
- 16 Registers
- Translates (mostly) directly to Intel 64-bit machine code
  $\Rightarrow$ Suitable CPU can execute instructions directly

Easier than Intel assembly, almost as efficient

Using 2OPM


Assembly code
add $t0, $t1
add $t1, $t0
Assembler
Machine Code
run on
Debugger
control
CPU

Using 2OPM



Assembly code

```
add $t0, $t1
add $t1, $t0
```

We will also use 2OPM in a language implementation

2OPM Registers

2OPM features 16 general-purpose registers:

Name	**Purpose**
`$v0`	Return **V**alue
`$a0`, `$a1`, `$a2`, `$a3`, `$a4`, `$a5`	**A**rguments
`$s0`, `$s1`, `$s2`, `$s3`	**S**aved registers
`$t0`, `$t1`	**T**emporary registers
`$sp`	**S**tack **P**ointer
`$fp`	**F**rame **P**ointer
`$gp`	**G**lobal **P**ointer

2OPM Registers

2OPM features 16 general-purpose registers:

Name	**Purpose**
`$v0`	Return **V**alue
`$a0`, `$a1`, `$a2`, `$a3`, `$a4`, `$a5`	**A**rguments
`$s0`, `$s1`, `$s2`, `$s3`	**S**aved registers
`$t0`, `$t1`	**T**emporary registers
`$sp`	**S**tack **P**ointer
`$fp`	**F**rame **P**ointer
`$gp`	**G**lobal **P**ointer

Plus $pc:

- *Program Counter*, address of next instruction
- can't be referenced directly

Purpose defined by *convention*

Intel 64-bit Registers

64-bit x86 features 16 general-purpose registers:

Name	Purpose
`%rax`	Return Value
`%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`	Arguments
`%rbx`, `%r12`, `%r13`, `%r14`	Saved registers
`%r10`, `%r11`	Temporary registers
`%rsp`	Stack Pointer
`%rbp`	Frame Pointer
`%r15`	Global Pointer

Plus `%rip`:

- *instruction pointer*, address of next instruction
- can't be referenced directly

Historical naming; we will use the 2OPM names

2OPM Registers

2OPM features 16 general-purpose registers:

Name	**Purpose**
`$v0`	Return **V**alue
`$a0`, `$a1`, `$a2`, `$a3`, `$a4`, `$a5`	**A**rguments
`$s0`, `$s1`, `$s2`, `$s3`	**S**aved registers
`$t0`, `$t1`	**T**emporary registers
`$sp`	**S**tack **P**ointer
`$fp`	**F**rame **P**ointer
`$gp`	**G**lobal **P**ointer

Plus $pc

Basic Operations

```
li    $t0, 2      # $t0 := 2
```

- `li $r, v`
  Load 64 bit value `v` into `$r`

Basic Operations

```
li    $t0, 2      # $t0 := 2
move  $t1, $t0    # $t1 := $t0
```

- `li $r, v`
  Load 64 bit value `v` into `$r`
- `move $r0, $r1`
  Copy contents of `$r1` into `$r0`

Basic Operations

```
li    $t0, 2     # $t0 := 2
move  $t1, $t0   # $t1 := $t0
add   $t0, $t1   # $t0 := $t0 + $t1
```

- `li $r, v`
  Load 64 bit value `v` into `$r`
- `move $r0, $r1`
  Copy contents of `$r1` into `$r0`
- `add $r0, $r1`
  Add contents of `$r1` into `$r0`

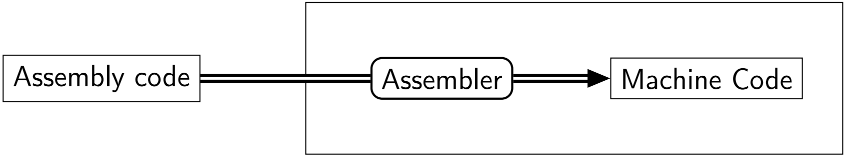
Basic Operations

```
li    $t0, 2      # $t0 := 2
move  $t1, $t0    # $t1 := $t0
add   $t0, $t1    # $t0 := $t0 + $t1
```

- `li $r, v`
  Load 64 bit value `v` into `$r`
- `move $r0, $r1`
  Copy contents of `$r1` into `$r0`
- `add $r0, $r1`
  Add contents of `$r1` into `$r0`

Operate on registers only

Machine Code



```
li      $t0, 2              49 ba 02 00 00 00 00 00 00 00
move    $t1, $t0            4d 89 d3
add     $t0, $t1            4d 01 da
```

CPU directly executes machine code

Operational Semantics

$$\text{Register file } \rho = \left\{ \begin{array}{ccc} \texttt{\$v0} & \mapsto & \dots \\ \texttt{\$t0} & \mapsto & \dots \\ & \dots & \\ \texttt{\$gp} & \mapsto & \dots \end{array} \right\}$$

Read: $\rho(\texttt{\$r})$
Update: $[\texttt{\$r := x}]\rho$

Operational Semantics

$$\text{Register file } \rho = \left\{ \begin{array}{ccc} \texttt{\$v0} & \mapsto & \ldots \\ \texttt{\$t0} & \mapsto & \ldots \\ & \ldots & \\ \texttt{\$gp} & \mapsto & \ldots \end{array} \right\}$$

Read: $\rho(\texttt{\$r})$
Update: $[\texttt{\$r := x}]\rho$

li $\$r, v$

Operational Semantics

$$\text{Register file } \rho = \left\{ \begin{array}{ccc} \texttt{\$v0} & \mapsto & \dots \\ \texttt{\$t0} & \mapsto & \dots \\ & \dots & \\ \texttt{\$gp} & \mapsto & \dots \end{array} \right\}$$

Read: $\rho(\texttt{\$r})$
Update: $[\texttt{\$r := x}]\rho$

$\texttt{li } \$r, v | \rho$

Operational Semantics

$$\text{Register file } \rho = \left\{ \begin{array}{ccc} \texttt{\$v0} & \mapsto & \ldots \\ \texttt{\$t0} & \mapsto & \ldots \\ & \ldots & \\ \texttt{\$gp} & \mapsto & \ldots \end{array} \right\}$$

Read: $\rho(\texttt{\$r})$
Update: $[\texttt{\$r := x}]\rho$

$$\texttt{li}\ \$r, v|\rho \longrightarrow$$

Operational Semantics

$$\text{Register file } \rho = \left\{ \begin{array}{ccc} \texttt{\$v0} & \mapsto & \ldots \\ \texttt{\$t0} & \mapsto & \ldots \\ & \ldots & \\ \texttt{\$gp} & \mapsto & \ldots \end{array} \right\}$$

Read: $\rho(\texttt{\$r})$
Update: $[\texttt{\$r := x}]\rho$

$$\texttt{li}\ \$r, v|\rho \longrightarrow \varepsilon|[\$r := v]\rho$$

Operational Semantics

$$\text{Register file } \rho = \left\{ \begin{array}{lcl} \texttt{\$v0} & \mapsto & \ldots \\ \texttt{\$t0} & \mapsto & \ldots \\ & \ldots & \\ \texttt{\$gp} & \mapsto & \ldots \end{array} \right\}$$

Read: $\rho(\texttt{\$r})$
Update: $[\texttt{\$r := x}]\rho$

$$\texttt{li}\ \$r, v|\rho \longrightarrow \varepsilon|[\$r := v]\rho$$

$$\texttt{move}\ \$r_0, \$r_1|\rho$$

Operational Semantics

Register file $\rho = \left\{ \begin{array}{ccc} \texttt{\$v0} & \mapsto & \ldots \\ \texttt{\$t0} & \mapsto & \ldots \\ & \ldots & \\ \texttt{\$gp} & \mapsto & \ldots \end{array} \right\}$

Read: $\rho(\texttt{\$r})$
Update: $[\texttt{\$r := x}]\rho$

$$\texttt{li}\ \$r, v|\rho \longrightarrow \varepsilon|[\$r := v]\rho$$

$$\texttt{move}\ \$r_0, \$r_1|\rho \longrightarrow \varepsilon|[\$r_0 := \rho(\$r_1)]\rho$$

Operational Semantics

Register file $\rho = \left\{ \begin{array}{ccc} \texttt{\$v0} & \mapsto & \ldots \\ \texttt{\$t0} & \mapsto & \ldots \\ & \ldots & \\ \texttt{\$gp} & \mapsto & \ldots \end{array} \right\}$

Read: $\rho(\texttt{\$r})$
Update: $[\texttt{\$r := x}]\rho$

$$\texttt{li}\ \$r, v|\rho \longrightarrow \varepsilon|[\$r := v]\rho$$

$$\texttt{move}\ \$r_0, \$r_1|\rho \longrightarrow \varepsilon|[\$r_0 := \rho(\$r_1)]\rho$$

$$\texttt{add}\ \$r_0, \$r_1|\rho \longrightarrow \varepsilon|[\$r_0 := \rho(\$r_0) + \rho(\$r_1)]\rho$$

Operational Semantics

$$\text{Register file } \rho = \left\{ \begin{array}{ccc} \texttt{\$v0} & \mapsto & \ldots \\ \texttt{\$t0} & \mapsto & \ldots \\ & \ldots & \\ \texttt{\$gp} & \mapsto & \ldots \end{array} \right\}$$

Read: $\rho(\texttt{\$r})$
Update: $[\texttt{\$r := x}]\rho$

$$\textbf{li } \$r, v|\rho \longrightarrow \varepsilon|[\$r := v]\rho$$

$$\textbf{move } \$r_0, \$r_1|\rho \longrightarrow \varepsilon|[\$r_0 := \rho(\$r_1)]\rho$$

$$\textbf{add } \$r_0, \$r_1|\rho \longrightarrow \varepsilon|[\$r_0 := \rho(\$r_0) + \rho(\$r_1)]\rho$$

$$\frac{i_0|\rho \longrightarrow \varepsilon|\rho'}{i_0; i_1|\rho \longrightarrow i_1|\rho'}$$

Operational Semantics

$$\text{Register file } \rho = \left\{ \begin{array}{ccc} \texttt{\$v0} & \mapsto & \ldots \\ \texttt{\$t0} & \mapsto & \ldots \\ & \ldots & \\ \texttt{\$gp} & \mapsto & \ldots \end{array} \right\}$$

Read: $\rho(\texttt{\$r})$
Update: $[\texttt{\$r := x}]\rho$

$$\textbf{li}\ \$r, v|\rho \longrightarrow \varepsilon|[\$r := v]\rho$$

$$\textbf{move}\ \$r_0, \$r_1|\rho \longrightarrow \varepsilon|[\$r_0 := \rho(\$r_1)]\rho$$

$$\textbf{add}\ \$r_0, \$r_1|\rho \longrightarrow \varepsilon|[\$r_0 := \rho(\$r_0) + \rho(\$r_1)]\rho$$

$$\frac{i_0|\rho \longrightarrow \varepsilon|\rho'}{i_0; i_1|\rho \longrightarrow i_1|\rho'}$$

Rule for add not quite accurate!

Overflow

- Real CPUs operate only on 64 bits:
  `0x8000000000000000` ‘+’ `0x8000000000000001`

Overflow

- Real CPUs operate only on 64 bits:
  `0x8000000000000000` ‘+’ `0x8000000000000001` = 1
  Overflow

Overflow

- Real CPUs operate only on 64 bits:
  `0x8000000000000000` '$+$' `0x8000000000000001` $=$ 1
  Overflow
- $a + b$ knows no bit bound
- We define:
  $a \boxplus^k b \equiv \mathrm{repr}^k(\mathrm{num}_s^k(a) + \mathrm{num}_s^k(b))$
- Analogous for $\boxminus^k$, $\boxtimes^k$, $\ldots$

Overflow

- Real CPUs operate only on 64 bits:
  `0x8000000000000000` '$+$' `0x8000000000000001` $= 1$
  Overflow
- $a + b$ knows no bit bound
- We define:
  $a \boxplus^k b \equiv \mathrm{repr}^k(\mathrm{num}_s^k(a) + \mathrm{num}_s^k(b))$
- Analogous for $\boxminus^k$, $\boxtimes^k$, ...

$$\textbf{add}\ \$r_0, \$r_1 | \rho \longrightarrow \varepsilon | [\$r_0 := \rho(\$r_0) \boxplus^{64} \rho(\$r_1)]\rho$$

Arithmetic

Additional operations for:

- Arithmetic: **`sub`**, **`mul`**, **`div_a2v0`**

Arithmetic

Additional operations for:

- Arithmetic: **sub**, **mul**, **div_a2v0**

$$\frac{\rho(\$r) \neq 0 \quad n = \rho(\$a2):\rho(\$v0) \quad \begin{array}{c} q = n \boxed{/}^{64} \rho(\$r) \\ r = n \boxed{\text{mod}}^{64} \rho(\$r) \end{array}}{\texttt{div\_a2v0}\ \$r|\rho \longrightarrow \varepsilon|[\$v0 := q][\$a2 := r]\rho}$$

Arithmetic

Additional operations for:

- Arithmetic: **sub**, **mul**, **div_a2v0**

$$\frac{\rho(\$r) \neq 0 \quad n = \rho(\$a2) : \rho(\$v0) \quad \begin{array}{c} q = n \boxed{/}^{64} \rho(\$r) \\ r = n \boxed{\text{mod}}^{64} \rho(\$r) \end{array}}{\texttt{div\_a2v0}\ \$r|\rho \longrightarrow \varepsilon|[\$v0 := q][\$a2 := r]\rho}$$

Semantics of realistic CPUs can be complicated!

Bit-Logic

- Negation: **`not`**
- Bit-Shift: **`sll`**, **`srl`**, **`sra`**
- Bit combinations: **`and`**, **`or`**, **`xor`**

Bit-Logic

- Negation: **`not`**
- Bit-Shift: **`sll`**, **`srl`**, **`sra`**
- Bit combinations: **`and`**, **`or`**, **`xor`**

$$\frac{n = \mathrm{num}_u^8(\rho(\$r_1)[7:0]) \qquad n < 64}{\texttt{srl}\ \ \$r_0, \$r_1 | \rho \longrightarrow \varepsilon | [\$r_0 := [0]^n : \rho(\$r_0)[63:n]]\rho}$$

Bit-Logic

- Negation: **`not`**
- Bit-Shift: **`sll`**, **`srl`**, **`sra`**
- Bit combinations: **`and`**, **`or`**, **`xor`**

$$\frac{n = \mathrm{num}_u^8(\rho(\$r_1)[7:0]) \quad n < 64}{\mathtt{srl}\ \ \$r_0, \$r_1 | \rho \longrightarrow \varepsilon | [\$r_0 := [0]^n : \rho(\$r_0)[63:n]]\rho}$$

$$\frac{n = \mathrm{num}_u^8(\rho(\$r_1)[7:0]) \quad n \geq 64}{\mathtt{srl}\ \ \$r_0, \$r_1 | \rho \longrightarrow \varepsilon | [\$r_0 := 0]\rho}$$

Bit-Logic

- Negation: **not**
- Bit-Shift: **sll**, **srl**, **sra**
- Bit combinations: **and**, **or**, **xor**

$$\frac{n = \mathrm{num}_u^8(\rho(\$r_1)[7:0]) \quad n < 64}{\texttt{srl}\ \ \$r_0, \$r_1 | \rho \longrightarrow \varepsilon | [\$r_0 := [0]^n : \rho(\$r_0)[63:n]]\rho}$$

$$\frac{n = \mathrm{num}_u^8(\rho(\$r_1)[7:0]) \quad n \geq 64}{\texttt{srl}\ \ \$r_0, \$r_1 | \rho \longrightarrow \varepsilon | [\$r_0 := 0]\rho}$$

$$\frac{n = \mathrm{num}_u^8(\rho(\$r_1)[7:0]) \quad n < 64}{\texttt{sra}\ \ \$r_0, \$r_1 | \rho \longrightarrow \varepsilon | [\$r_0 := (\rho(\$r_0)[63:63])^n : \rho(\$r_0)[63:n]]\rho}$$

Bit-Logic

- Negation: **not**
- Bit-Shift: **sll**, **srl**, **sra**
- Bit combinations: **and**, **or**, **xor**

$$\frac{n = \mathrm{num}_u^8(\rho(\$r_1)[7:0]) \quad n < 64}{\texttt{srl}\ \ \$r_0, \$r_1 | \rho \longrightarrow \varepsilon | [\$r_0 := [0]^n : \rho(\$r_0)[63:n]]\rho}$$

$$\frac{n = \mathrm{num}_u^8(\rho(\$r_1)[7:0]) \quad n \geq 64}{\texttt{srl}\ \ \$r_0, \$r_1 | \rho \longrightarrow \varepsilon | [\$r_0 := 0]\rho}$$

$$\frac{n = \mathrm{num}_u^8(\rho(\$r_1)[7:0]) \quad n < 64}{\texttt{sra}\ \ \$r_0, \$r_1 | \rho \longrightarrow \varepsilon | [\$r_0 := (\rho(\$r_0)[63:63])^n : \rho(\$r_0)[63:n]]\rho}$$

Immediate Operations

```
li  $t0, 8
srl $t1, $t0
```

Immediate Operations

Equivalent

```
li  $t0, 8
srl $t1, $t0
```

```
srli  $t1, 8
```

(but doesn't modify `$t0`)

Immediate Operations

Equivalent

```
li  $t0, 8
srl $t1, $t0
```

```
srli  $t1, 8
```
(but doesn't modify `$t0`)

Immediate operations:

- **addi**, **subi**
- **slli**, **srli**, **srai**
- **andi**, **ori**, **xori**

Summary

- 2OPM: synthetic assembly language
- Uses 16 general-purpose registers plus program counter
- Most operations use two register operands
- Operations for loading (**`li`**), copying (**`move`**)
- Basic arithmetic with combined division/modulo
- Bit-wise operations
- *Immediate* operations with register plus immediate operand available in many cases
- Describing semantics requires *register file*

Instructions are more fully described in documentation