

Instruction Selection

Akim Demaille Étienne Renault Roland Levillain
first.last@lrde.epita.fr

EPITA — École Pour l'Informatique et les Techniques Avancées

April 24, 2017

Instruction Selection

- 1 Microprocessors
- 2 A Typical risc: mips
- 3 The EPITA Tiger Compiler
- 4 Instruction Selection

- 1 Microprocessors
- 2 A Typical risc: mips
- 3 The EPITA Tiger Compiler
- 4 Instruction Selection

*Instruction set architecture is the structure of a computer that a machine language programmer (or a compiler) must understand to write a correct (timing independent) program for that machine
IBM introducing 360 (1964)*

The Instruction Set Architecture (ISA) is the part of the processor that is visible to the programmer or compiler writer.

What is an instruction set?

An instruction set specifies a processor functionality:

- what operations are supported
- what storage mechanisms are used
- how to access storage
- how to communicate program to processor

Technical aspect of instruction set

- 1 format: length, encoding
- 2 operations: data type (floating or fixed point) , number and kind of operands
- 3 storage:
 - internal: accumulator, stack, register
 - memory: address size, addressing modes
- 4 control: branch condition, support for procedures, predication

What makes a good instruction set?

An instruction set specifies a processor functionality:

- **implementability**: support for a (high performances) range of implementation
- **programmability**: easy to express program (by Humans before 80's, mostly by compiler nowadays)
- **backward/forward compatibility**: implementability & programmability across generation

- large number of instructions (100-250)
- 6, 8, 16 registers, some for pointers, others for integer computation
- arithmetic in memory can be processed
- two address code
- many possible effects (e.g., self-incrementation)

Pros:

- Simplified compiler: translation from IR is straightforward
- Smaller assembly code than risc assembly code
- Fewer instructions will be fetched
- Special purpose register available: stack pointer, interrupt handling ...

Cons:

- Variable length instruction format
- Many instruction require many clock for execution
- Limiter number of general purpose register
- (often) new version of cisc include the subset of instructions of the previous version

Motivations for something else!

Though the CISC programs could be small in length, but number of bits of memory occupies may not be less

The complex instructions do not simplify the compilers: many clock cycles can be wasted to find the appropriate instruction.

risc architectures were designed with the goal of executing one instruction per clock cycle.

risc – Reduced Instruction Set Chip

- 32 generic purpose registers
- arithmetic only available on registers
- 3 address code
- load and store relative to a register
($M[r + \text{const}]$)
- only one effect or result per instruction

Pipelining is the overlapping the execution of several instructions in a pipeline fashion.

A pipeline is (typically) decomposed into five stages:

- 1 Instruction Fetch (IF)
- 2 Instruction Decode (ID)
- 3 Execute (EX)
- 4 Memory Access (MA)
- 5 Write Back (WB)

risc – Pipeline 2/3

inst1:	IF	ID	EX	MA	WB				
inst2:		IF	ID	EX	MA	WB			
inst3:			IF	ID	EX	MA	WB		
inst4:				IF	ID	EX	MA	WB	
inst5:					IF	ID	EX	MA	WB

The slowest stage determines the speed of the whole pipeline!

Ex introduces latency

- Register-Register Operation: 1 cycle
- Memory Reference: 2 cycles
- Multi-cycle Instructions (floating point): many cycles

Data hazard: When an instruction depends on the results of a previous instruction still in the pipeline.

- inst1 write in \$s1 during WB
- inst2 read in \$s1 during ID

inst1:	IF	ID	EX	MA	WB	
inst2:		IF	ID	EX	MA	WB

inst2 must be split, causing delays...

other dependencies can appears

Pros:

- Fixed length instructions: decoding is easier
- Simpler hardware: higher clock rate
- Efficient Instruction pipeline
- Large number of general purpose register
- Overlapped register windows to speed up procedure call and return
- One instruction per cycle

Cons:

- Minimal number of addressing modes: only Load and Store
- Relatively few instructions

- the classification pure-risc or pure-cisc is becoming more and more inappropriate and may be irrelevant
- modern processors use a calculated combination elements of both design styles
- decisive factor is based on a tradeoff between the required improvement in performance and the expected added cost
- Some processors that are classified as CISC while employing a number of RISC features, such as pipelining

ARM provides the advantage of using a CISC (in terms of functionality) and the advantage of an RISC (in terms of reduced code lengths).

Lessons to be learned

Implementability

Driven by technology: microcode, VLSI, FPGA, pipelining, superscalar, SIMD, SSE

Programmability

Driven by compiler technology

Sum-up

- Many non technical issues influence ISA's
- Best solutions don't always win (Intel X86)

Intel X86 (IA32)

- Introduced in 1978
- 8×32 bits "general" register
- variable length instructions (1–15 byte)
- long life to the king! 13 generations from Intel 8086 to Intel Skylake/Cannonlake (i3/i7)

Intel's trick?

Decoder translates cisc into risc micro-operations

A Typical risc: mips

1 Microprocessors

2 A Typical risc: mips

- Integer Arithmetics
- Logical Operations
- Control Flow
- Loads and Stores
- Floating Point Operations

3 The EPITA Tiger Compiler

4 Instruction Selection

mips Registers and Use Convention [Larus, 1990]

Name	Number	Usage
zero	0	Constant 0
at	1	Reserved for assembler
v0-v1	2-3	Expression evaluation and results of a function
a0-a3	4-7	Function argument 1-4
t0-t7	8-15	Temporary (not preserved across call)
s0-s7	16-23	Saved temporary (preserved across call)
t8-t9	24-25	Temporary (not preserved across call)
k0-k1	26-27	Reserved for OS kernel
gp	28	Pointer to global area
sp	29	Stack pointer
fp	30	Frame pointer
ra	31	Return address (used by function call)

Typical risc Instructions

The following slides are based on [Larus, 1990].

- The assembler translates pseudo-instructions (marked with † below).
- In all instructions below, Src2 can be
 - a register
 - an immediate value (a 16 bit integer).
- The immediate forms are included for reference.
- The assembler translates the general form (e.g., add) into the immediate form (e.g., addi) if the second argument is constant.

Integer Arithmetics

1 Microprocessors

2 A Typical risc: mips

- Integer Arithmetics
- Logical Operations
- Control Flow
- Loads and Stores
- Floating Point Operations

3 The EPITA Tiger Compiler

4 Instruction Selection

Arithmetic: Addition/Subtraction

`add` Rdest, Rsrc1, Src2 *Addition (with overflow)*

`addi` Rdest, Rsrc1, Imm *Addition Immediate (with overflow)*

`addu` Rdest, Rsrc1, Src2 *Addition (without overflow)*

`addiu` Rdest, Rsrc1, Imm *Addition Immediate (without overflow)*

Put the sum of the integers from Rsrc1 and Src2 (or Imm) into Rdest.

`sub` Rdest, Rsrc1, Src2 *Subtract (with overflow)*

`subu` Rdest, Rsrc1, Src2 *Subtract (without overflow)*

Put the difference of the integers from Rsrc1 and Src2 into Rdest.

Arithmetic: Division

If an operand is negative, the remainder is unspecified by the mips architecture and depends on the conventions of the machine on which spim is run.

`div Rsrc1, Rsrc2` *Divide (signed)*[†]

`divu Rsrc1, Rsrc2` *Divide (unsigned)*[†]

Divide the contents of the two registers. Leave the quotient in register `lo` and the remainder in register `hi`.

`div Rdest, Rsrc1, Src2` *Divide (signed, with overflow)*[†]

`divu Rdest, Rsrc1, Src2` *Divide (unsigned, without overflow)*[†]

Put the quotient of the integers from `Rsrc1` and `Src2` into `Rdest`.

`rem Rdest, Rsrc1, Src2` *Remainder*[†]

`remu Rdest, Rsrc1, Src2` *Unsigned Remainder*[†]

Likewise for the the remainder of the division.

Arithmetic: Multiplication

`mul` Rdest, Rsrc1, Src2

Multiply (without overflow) †

`mulo` Rdest, Rsrc1, Src2

Multiply (with overflow) †

`mulou` Rdest, Rsrc1, Src2

Unsigned Multiply (with overflow) †

Put the product of the integers from Rsrc1 and Src2 into Rdest.

`mult` Rsrc1, Rsrc2

Multiply

`multu` Rsrc1, Rsrc2

Unsigned Multiply

Multiply the contents of the two registers. Leave the low-order word of the product in register `lo` and the high-word in register `hi`.

Arithmetic Instructions

`abs` Rdest, Rsrc *Absolute Value* †
Put the absolute value of the integer from Rsrc in Rdest.

`neg` Rdest, Rsrc *Negate Value (with overflow)* †

`negu` Rdest, Rsrc *Negate Value (without overflow)* †
Put the negative of the integer from Rsrc into Rdest.

Logical Operations

1 Microprocessors

2 A Typical risc: mips

- Integer Arithmetics
- **Logical Operations**
- Control Flow
- Loads and Stores
- Floating Point Operations

3 The EPITA Tiger Compiler

4 Instruction Selection

Logical Instructions

`and` Rdest, Rsrc1, Src2

AND

`andi` Rdest, Rsrc1, Imm

AND Immediate

Put the logical AND of the integers from Rsrc1 and Src2 (or Imm) into Rdest.

`not` Rdest, Rsrc

NOT †

Put the bitwise logical negation of the integer from Rsrc into Rdest.

Logical Instructions

`nor` Rdest, Rsrc1, Src2 *NOR*

Put the logical NOR of the integers from Rsrc1 and Src2 into Rdest.

`or` Rdest, Rsrc1, Src2 *OR*

`ori` Rdest, Rsrc1, Imm *OR Immediate*

Put the logical OR of the integers from Rsrc1 and Src2 (or Imm) into Rdest.

`xor` Rdest, Rsrc1, Src2 *XOR*

`xori` Rdest, Rsrc1, Imm *XOR Immediate*

Put the logical XOR of the integers from Rsrc1 and Src2 (or Imm) into Rdest.

Logical Instructions

`rol` Rdest, Rsrc1, Src2

Rotate Left †

`ror` Rdest, Rsrc1, Src2

Rotate Right †

Rotate the contents of Rsrc1 left (right) by the distance indicated by Src2 and put the result in Rdest.

`sll` Rdest, Rsrc1, Src2

Shift Left Logical

`sllv` Rdest, Rsrc1, Rsrc2

Shift Left Logical Variable

`sra` Rdest, Rsrc1, Src2

Shift Right Arithmetic

`srav` Rdest, Rsrc1, Rsrc2

Shift Right Arithmetic Variable

`srl` Rdest, Rsrc1, Src2

Shift Right Logical

`srlv` Rdest, Rsrc1, Rsrc2

Shift Right Logical Variable

Shift the contents of Rsrc1 left (right) by the distance indicated by Src2 (Rsrc2) and put the result in Rdest.

1 Microprocessors

2 A Typical risc: mips

- Integer Arithmetics
- Logical Operations
- **Control Flow**
- Loads and Stores
- Floating Point Operations

3 The EPITA Tiger Compiler

4 Instruction Selection

Comparison Instructions

`seq Rdest, Rsrc1, Src2`

Set Rdest to 1 if Rsrc1 equals Src2, otherwise to 0.

Set Equal †

`sne Rdest, Rsrc1, Src2`

Set Rdest to 1 if Rsrc1 is not equal to Src2, otherwise to 0.

Set Not Equal †

Comparison Instructions

<code>sge</code> Rdest, Rsrc1, Src2	<i>Set Greater Than Equal</i> [†]
<code>sgeu</code> Rdest, Rsrc1, Src2	<i>Set Greater Than Equal Unsigned</i> [†]
Set Rdest to 1 if Rsrc1 \geq Src2, otherwise to 0.	
<code>sgt</code> Rdest, Rsrc1, Src2	<i>Set Greater Than</i> [†]
<code>sgtu</code> Rdest, Rsrc1, Src2	<i>Set Greater Than Unsigned</i> [†]
Set Rdest to 1 if Rsrc1 $>$ Src2, otherwise to 0.	
<code>sle</code> Rdest, Rsrc1, Src2	<i>Set Less Than Equal</i> [†]
<code>sleu</code> Rdest, Rsrc1, Src2	<i>Set Less Than Equal Unsigned</i> [†]
Set Rdest to 1 if Rsrc1 \leq Src2, otherwise to 0.	
<code>slt</code> Rdest, Rsrc1, Src2	<i>Set Less Than</i>
<code>slti</code> Rdest, Rsrc1, Imm	<i>Set Less Than Immediate</i>
<code>sltu</code> Rdest, Rsrc1, Src2	<i>Set Less Than Unsigned</i>
<code>sltiu</code> Rdest, Rsrc1, Imm	<i>Set Less Than Unsigned Immediate</i>
Set Rdest to 1 if Rsrc1 $<$ Src2 (or Imm), otherwise to 0.	

Branch and Jump Instructions

Branch instructions use a signed 16-bit offset field: jump from -2^{15} to $+2^{15} - 1$) *instructions* (not bytes). The *jump* instruction contains a 26 bit address field.

- b** label *Branch instruction* †
Unconditionally branch to *label*.
- j** label *Jump*
Unconditionally jump to *label*.
- jal** label *Jump and Link*
jalr Rsrc *Jump and Link Register*
Unconditionally jump to *label* or whose address is in Rsrc. Save the address of the next instruction in register 31.
- jr** Rsrc *Jump Register*
Unconditionally jump to the instruction whose address is in register Rsrc.

Branch and Jump Instructions

`bczt label`

Branch Coprocessor z True

`bczf label`

Branch Coprocessor z False

Conditionally branch to *label* if coprocessor *z*'s condition flag is true (false).

Branch and Jump Instructions

Conditionally branch to *label* if the contents of `Rsrc1 * Src2`.

`beq Rsrc1, Src2, label`

Branch on Equal

`bne Rsrc1, Src2, label`

Branch on Not Equal

`beqz Rsrc, label`

Branch on Equal Zero †

`bnez Rsrc, label`

Branch on Not Equal Zero †

Branch and Jump Instructions

Conditionally branch to *label* if the contents of $Rsrc1 * Src2$.

<code>bge Rsrc1, Src2, label</code>	<i>Branch on Greater Than Equal</i> †
<code>bgeu Rsrc1, Src2, label</code>	<i>Branch on GTE Unsigned</i> †
<code>bgez Rsrc, label</code>	<i>Branch on Greater Than Equal Zero</i>
<code>bgezal Rsrc, label</code>	<i>Branch on Greater Than Equal Zero And Link</i>

Conditionally branch to *label* if the contents of $Rsrc$ are greater than or equal to 0. Save the address of the next instruction in register 31.

<code>bgt Rsrc1, Src2, label</code>	<i>Branch on Greater Than</i> †
<code>bgtu Rsrc1, Src2, label</code>	<i>Branch on Greater Than Unsigned</i> †
<code>bgtz Rsrc, label</code>	<i>Branch on Greater Than Zero</i>

Branch and Jump Instructions

Conditionally branch to *label* if the contents of Rsrc1 are * to Src2.

<code>ble</code> Rsrc1, Src2, label	<i>Branch on Less Than Equal</i> †
<code>bleu</code> Rsrc1, Src2, label	<i>Branch on LTE Unsigned</i> †
<code>blez</code> Rsrc, label	<i>Branch on Less Than Equal Zero</i>
<code>bgezal</code> Rsrc, label	<i>Branch on Greater Than Equal Zero And Link</i>
<code>bltzal</code> Rsrc, label	<i>Branch on Less Than And Link</i>

Conditionally branch to *label* if the contents of Rsrc are greater or equal to 0 or less than 0, respectively. Save the address of the next instruction in register 31.

<code>blt</code> Rsrc1, Src2, label	<i>Branch on Less Than</i> †
<code>bltu</code> Rsrc1, Src2, label	<i>Branch on Less Than Unsigned</i> †
<code>bltz</code> Rsrc, label	<i>Branch on Less Than Zero</i>

Exception and Trap Instructions

<code>rfe</code>	<i>Return From Exception</i>
Restore the Status register.	
<code>syscall</code>	<i>System Call</i>
Register <code>\$v0</code> contains the number of the system call provided by <code>spim</code> .	
<code>break n</code>	<i>Break</i>
Cause exception n . Exception 1 is reserved for the debugger.	
<code>nop</code>	<i>No operation</i>
Do nothing.	

Loads and Stores

1 Microprocessors

2 A Typical risc: mips

- Integer Arithmetics
- Logical Operations
- Control Flow
- **Loads and Stores**
- Floating Point Operations

3 The EPITA Tiger Compiler

4 Instruction Selection

Constant-Manipulating Instructions

`li Rdest, imm`

Load Immediate †

Move the immediate `imm` into `Rdest`.

`lui Rdest, imm`

Load Upper Immediate

Load the lower halfword of the immediate `imm` into the upper halfword of `Rdest`.

The lower bits of the register are set to 0.

Load: Byte & Halfword

`lb Rdest, address`

Load Byte

`lbu Rdest, address`

Load Unsigned Byte

Load the byte at *address* into Rdest. The byte is sign-extended by the `lb`, but not the `lbu`, instruction.

`lh Rdest, address`

Load Halfword

`lhu Rdest, address`

Load Unsigned Halfword

Load the 16-bit quantity (halfword) at *address* into register Rdest. The halfword is sign-extended by the `lh`, but not the **`lhu`**, instruction

Load: Word

- `lw` Rdest, address *Load Word*
Load the 32-bit quantity (word) at *address* into Rdest.
- `lwcz` Rdest, address *Load Word Coprocessor*
Load the word at *address* into Rdest of coprocessor *z* (0–3).
- `lwl` Rdest, address *Load Word Left*
`lwr` Rdest, address *Load Word Right*
Load the left (right) bytes from the word at the possibly-unaligned *address* into Rdest.
- `ulh` Rdest, address *Unaligned Load Halfword* †
`ulhu` Rdest, address *Unaligned Load Halfword Unsigned* †
Load the 16-bit quantity (halfword) at the possibly-unaligned *address* into Rdest.
The halfword is sign-extended by the `ulh`, but not the `ulhu`, instruction
- `ulw` Rdest, address *Unaligned Load Word* †
Load the 32-bit quantity (word) at the possibly-unaligned *address* into Rdest.

Load Instructions

- la** Rdest, address *Load Address* †
Load computed *address*, not the contents of the location, into Rdest.
- ld** Rdest, address *Load Double-Word* †
Load the 64-bit quantity at *address* into Rdest and Rdest + 1.

Store: Byte & Halfword

sb Rsrc, address

Store the low byte from Rsrc at *address*.

Store Byte

sh Rsrc, address

Store the low halfword from Rsrc at *address*.

Store Halfword

Store: Word

- sw** Rsrc, address *Store Word*
Store the word from Rsrc at *address*.
- swcZ** Rsrc, address *Store Word Coprocessor*
Store the word from Rsrc of coprocessor *z* at *address*.
- swl** Rsrc, address *Store Word Left*
swr Rsrc, address *Store Word Right*
Store the left (right) bytes from Rsrc at the possibly-unaligned *address*.
- ush** Rsrc, address *Unaligned Store Halfword*[†]
Store the low halfword from Rsrc at the possibly-unaligned *address*.
- usw** Rsrc, address *Unaligned Store Word*[†]
Store the word from Rsrc at the possibly-unaligned *address*.

Store: Double Word

`sd Rsrc, address`

Store Double-Word †

Store the 64-bit quantity in Rsrc and Rsrc + 1 at *address*.

Data Movement Instructions

`move Rdest, Rsrc`

Move †

Move the contents of Rsrc to Rdest.

The multiply and divide unit produces its result in two additional registers, hi and lo (e.g., `mul Rdest, Rsrc1, Src2`).

`mfhi Rdest`

Move From hi

`mflo Rdest`

Move From lo

Move the contents of the hi (lo) register to Rdest.

`mthi Rdest`

Move To hi

`mtlo Rdest`

Move To lo

Move the contents Rdest to the hi (lo) register.

Data Movement Instructions

Coprocessors have their own register sets. These instructions move values between these registers and the CPU's registers.

mfcz Rdest, CPsrc *Move From Coprocessor z*
Move the contents of coprocessor z's register CPsrc to CPU Rdest.

mfc1.d Rdest, FRsrc1 *Move Double From Coprocessor 1 †*
Move the contents of floating point registers FRsrc1 and FRsrc1 + 1 to CPU registers Rdest and Rdest + 1.

mtcz Rsrc, CPdest *Move To Coprocessor z*
Move the contents of CPU Rsrc to coprocessor z's register CPdest.

Floating Point Operations

1 Microprocessors

2 A Typical risc: mips

- Integer Arithmetics
- Logical Operations
- Control Flow
- Loads and Stores
- **Floating Point Operations**

3 The EPITA Tiger Compiler

4 Instruction Selection

mips Floating Point Instructions

- Floating point coprocessor 1 operates on single (32-bit) and double precision (64-bit) FP numbers.
- 32 32-bit registers \$f0-\$f31.
- Two FP registers to hold doubles.
- FP operations only use even-numbered registers including instructions that operate on single floats.
- Values are moved one word (32-bits) at a time by `lwc1`, `swc1`, `mtc1`, and `mfc1` or by the `l.s`, `l.d`, `s.s`, and `s.d` pseudo-instructions.
- The flag set by FP comparison operations is read by the CPU with its `bc1t` and `bc1f` instructions.

Floating Point: Arithmetics

Compute the * of the floating float doubles (singles) in FRsrc1 and FRsrc2 and put it in FRdest.

`add.d` FRdest, FRsrc1, FRsrc2

`add.s` FRdest, FRsrc1, FRsrc2

`div.d` FRdest, FRsrc1, FRsrc2

`div.s` FRdest, FRsrc1, FRsrc2

`mul.d` FRdest, FRsrc1, FRsrc2

`mul.s` FRdest, FRsrc1, FRsrc2

`sub.d` FRdest, FRsrc1, FRsrc2

`sub.s` FRdest, FRsrc1, FRsrc2

`abs.d` FRdest, FRsrc

`abs.s` FRdest, FRsrc

`neg.d` FRdest, FRsrc

`neg.s` FRdest, FRsrc

Floating Point Addition Double

Floating Point Addition Single

Floating Point Divide Double

Floating Point Divide Single

Floating Point Multiply Double

Floating Point Multiply Single

Floating Point Subtract Double

Floating Point Subtract Single

Floating Point Absolute Value Double

Floating Point Absolute Value Single

Negate Double

Negate Single

Floating Point: Comparison

Compare the floating point double in FRsrc1 against the one in FRsrc2 and set the floating point condition flag true if they are *.

`c.eq.d` FRsrc1, FRsrc2

Compare Equal Double

`c.eq.s` FRsrc1, FRsrc2

Compare Equal Single

`c.le.d` FRsrc1, FRsrc2

Compare Less Than Equal Double

`c.le.s` FRsrc1, FRsrc2

Compare Less Than Equal Single

`c.lt.d` FRsrc1, FRsrc2

Compare Less Than Double

`c.lt.s` FRsrc1, FRsrc2

Compare Less Than Single

Floating Point: Conversions

Convert between (i) single, (ii) double precision floating point number or (iii) integer in FRsrc to FRdest.

`cvt.d.s` FRdest, FRsrc

Convert Single to Double

`cvt.d.w` FRdest, FRsrc

Convert Integer to Double

`cvt.s.d` FRdest, FRsrc

Convert Double to Single

`cvt.s.w` FRdest, FRsrc

Convert Integer to Single

`cvt.w.d` FRdest, FRsrc

Convert Double to Integer

`cvt.w.s` FRdest, FRsrc

Convert Single to Integer

Floating Point: Moves

<code>l.d</code> FRdest, address	<i>Load Floating Point Double</i> †
<code>l.s</code> FRdest, address	<i>Load Floating Point Single</i> †
Load the floating float double (single) at address into register FRdest.	
<code>mov.d</code> FRdest, FRsrc	<i>Move Floating Point Double</i>
<code>mov.s</code> FRdest, FRsrc	<i>Move Floating Point Single</i>
Move the floating float double (single) from FRsrc to FRdest.	
<code>s.d</code> FRdest, address	<i>Store Floating Point Double</i> †
<code>s.s</code> FRdest, address	<i>Store Floating Point Single</i> †
Store the floating float double (single) in FRdest at address.	

The EPITA Tiger Compiler

- 1 Microprocessors
- 2 A Typical risc: mips
- 3 The EPITA Tiger Compiler**
- 4 Instruction Selection

The EPITA Tiger Project

We aim at mips because:

- mips is a nice assembly language
- mips is more modern
- mips is meaningful:
 - Million Instructions Per Second (10 mips = 1 mil)
 - Meaningless Indication of Processor Speed
 - Meaningless Information Provided by Salesmen
 - Meaningless Information per Second
 - Microprocessor without Interlocked Piped Stages
- spim is a portable mips emulator
- spim has a cool modern gui, xspim!

The EPITA Tiger Project

We aim at mips because:

- mips is a nice assembly language
- mips is more modern
- mips is meaningful:
 - Million Instructions Per Second (10 mips, 1 mip)
 - Meaningless Indication of Processor Speed
 - Meaningless Information Provided by Salesmen
 - Meaningless Information per Second
 - Microprocessor without Interlocked Piped Stages
- spim is a portable mips emulator
- spim has a cool modern gui, xspim!

The EPITA Tiger Project

We aim at mips because:

- mips is a nice assembly language
- mips is more modern
- mips is meaningful:
 - Million Instructions Per Second (10 mips, 1 mip)
 - Meaningless Indication of Processor Speed
 - Meaningless Information Provided by Salesmen
 - Meaningless Information per Second
 - Microprocessor without Interlocked Piped Stages
- spim is a portable mips emulator
- spim has a cool modern gui, xspim!

The EPITA Tiger Project

We aim at mips because:

- mips is a nice assembly language
- mips is more modern
- mips is meaningful:
 - Million Instructions Per Second (10 mips, 1 mip)
 - Meaningless Indication of Processor Speed
 - Meaningless Information Provided by Salesmen
 - Meaningless Information per Second
 - Microprocessor without Interlocked Piped Stages
- spim is a portable mips emulator
- spim has a cool modern gui, xspim!

The EPITA Tiger Project

We aim at mips because:

- mips is a nice assembly language
- mips is more modern
- mips is meaningful:
 - Million Instructions Per Second (10 mips, 1 mip)
 - Meaningless Indication of Processor Speed
 - Meaningless Information Provided by Salesmen
 - Meaningless Information per Second
 - Microprocessor without Interlocked Piped Stages
- spim is a portable mips emulator
- spim has a cool modern gui, xspim!

The EPITA Tiger Project

We aim at mips because:

- mips is a nice assembly language
- mips is more modern
- mips is meaningful:
 - Million Instructions Per Second (10 mips, 1 mip)
 - Meaningless Indication of Processor Speed
 - Meaningless Information Provided by Salesmen
 - Meaningless Information per Second
 - Microprocessor without Interlocked Piped Stages
- spim is a portable mips emulator
- spim has a cool modern gui, xspim!

The EPITA Tiger Project

We aim at mips because:

- mips is a nice assembly language
- mips is more modern
- mips is meaningful:
 - Million Instructions Per Second (10 mips, 1 mip)
 - Meaningless Indication of Processor Speed
 - Meaningless Information Provided by Salesmen
 - Meaningless Information per Second
 - Microprocessor without Interlocked Piped Stages
- spim is a portable mips emulator
- spim has a cool modern gui, xspim!

The EPITA Tiger Project

We aim at mips because:

- mips is a nice assembly language
- mips is more modern
- mips is meaningful:
 - Million Instructions Per Second (10 mips, 1 mip)
 - Meaningless Indication of Processor Speed
 - Meaningless Information Provided by Salesmen
 - Meaningless Information per Second
 - Microprocessor without Interlocked Piped Stages
- spim is a portable mips emulator
- spim has a cool modern gui, xspim!

The EPITA Tiger Project

We aim at mips because:

- mips is a nice assembly language
- mips is more modern
- mips is meaningful:
 - Million Instructions Per Second (10 mips, 1 mip)
 - Meaningless Indication of Processor Speed
 - Meaningless Information Provided by Salesmen
 - Meaningless Information per Second
 - Microprocessor without Interlocked Piped Stages
- spim is a portable mips emulator
- spim has a cool modern gui, xspim!

The EPITA Tiger Project

We aim at mips because:

- mips is a nice assembly language
- mips is more modern
- mips is meaningful:
 - Million Instructions Per Second (10 mips, 1 mip)
 - Meaningless Indication of Processor Speed
 - Meaningless Information Provided by Salesmen
 - Meaningless Information per Second
 - Microprocessor without Interlocked Piped Stages
- spim is a portable mips emulator
- spim has a cool modern gui, xspim!

xspim

PC = 00000000 EPC = 00000000 Cause = 00000000 BadVaddr = 00000000
 Status = 00000000 HI = 00000000 LO = 00000000

General Registers

R0 (r0) = 00000000 R8 (t0) = 00000000 R16 (s0) = 00000000 R24 (t8) = 00000000
 R1 (at) = 00000000 R9 (t1) = 00000000 R17 (s1) = 00000000 R25 (s9) = 00000000
 R2 (v0) = 00000000 R10 (t2) = 00000000 R18 (s2) = 00000000 R26 (k0) = 00000000
 R3 (v1) = 00000000 R11 (t3) = 00000000 R19 (s3) = 00000000 R27 (k1) = 00000000
 R4 (a0) = 00000000 R12 (t4) = 00000000 R20 (s4) = 00000000 R28 (gp) = 00000000
 R5 (a1) = 00000000 R13 (t5) = 00000000 R21 (s5) = 00000000 R29 (gp) = 00000000
 R6 (a2) = 00000000 R14 (t6) = 00000000 R22 (s6) = 00000000 R30 (s8) = 00000000
 R7 (a3) = 00000000 R15 (t7) = 00000000 R23 (s7) = 00000000 R31 (ra) = 00000000

Double Floating Point Registers

FP0 = 0.000000 FP8 = 0.000000 FP16 = 0.000000 FP24 = 0.000000
 FP2 = 0.000000 FP10 = 0.000000 FP18 = 0.000000 FP26 = 0.000000
 FP4 = 0.000000 FP12 = 0.000000 FP20 = 0.000000 FP28 = 0.000000
 FP6 = 0.000000 FP14 = 0.000000 FP22 = 0.000000 FP30 = 0.000000

Single Floating Point Registers

Register Display

Control Buttons

User and Kernel Text Segments

Data and Stack Segments

SPIM Messages

quit load run step clear set value
 print breakpt help terminal mode

Text Segments

```
[0x00400000] 0x8fa40000 lw R4, 0(R29) []
[0x00400004] 0x27a50004 addiu R5, R29, 4 []
[0x00400008] 0x24a60004 addiu R6, R5, 4 []
[0x0040000c] 0x00041090 sll R2, R4, 2
[0x00400010] 0x00c23021 addu R6, R6, R2
[0x00400014] 0x0c000000 jal 0x00000000 []
[0x00400018] 0x3402000a ori R0, R0, 10 []
[0x0040001c] 0x0000000c syscall
```

Data Segments

```
[0x10000000] ... [0x10010000] 0x00000000
[0x10010004] 0x74706563 0x206e6f69 0x636f2000 0x726f6e67
[0x10010010] 0x72727563 0x61206465 0x6920646e 0x74707572
[0x10010020] 0x000a6465 0x495b2020 0x7265746e 0x6e67696c
[0x10010030] 0x0000205d 0x20200000 0x616e555b 0x6e67696c
[0x10010040] 0x61206465 0x65726464 0x69207373 0x6e69206e
[0x10010050] 0x642f7473 0x20617461 0x63746566 0x00205d68
[0x10010060] 0x555b2020 0x696c616e 0x64656e67 0x64646120
[0x10010070] 0x73736572 0x206e6920 0x726f7473 0x00205d65
```

SPIM Version 3.2 of January 14, 1990

A Sample: fact

```
/* Define a recursive function. */  
let  
  /* Calculate n! */  
  function fact (n : int) : int =  
    if n = 0  
    then 1  
    else n * fact (n - 1)  
in  
  print_int (fact (10));  
  print ("\n")  
end
```

```

# Routine: fact
10:   sw      $fp, -8 ($sp)
      move   $fp, $sp
      sub    $sp, $sp, 16
      sw     $ra, -12 ($fp)
      sw     $a0, ($fp)
      sw     $a1, -4 ($fp)
15:   lw      $t0, -4 ($fp)
      beq    $t0, 0, l1
12:   lw      $a0, ($fp)
      lw      $t0, -4 ($fp)
      sub    $a1, $t0, 1
      jal    l0
      lw      $t0, -4 ($fp)
      mul    $t0, $t0, $v0
13:   move    $v0, $t0
      j      l6
11:   li      $t0, 1
      j      l3
16:   lw      $ra, -12 ($fp)
      move   $sp, $fp
      lw     $fp, -8 ($fp)
      jr     $ra

```

```

.data
l4:
      .word 1
      .asciiz "\n"

.text
# Routine: Main
t_main: sw     $fp, ($sp)
        move   $fp, $sp
        sub    $sp, $sp, 8
        sw     $ra, -4 ($fp)
17:     move   $a0, $fp
        li     $a1, 10
        jal    l0
        move   $a0, $v0
        jal    print_int
        la     $a0, l4
        jal    print
18:     lw      $ra, -4 ($fp)
        move   $sp, $fp
        lw     $fp, ($fp)
        jr     $ra

```

Nolimips (formerly Mipsy)

- Another mips emulator
- Interactive loop
- Unlimited number of \$x42 registers!

Routine: fact

```
10:  sw      $a0, ($fp)
      sw      $a1, -4($fp)
      move   $x11, $s0
      move   $x12, $s1
      move   $x13, $s2
      move   $x14, $s3
      move   $x15, $s4
      move   $x16, $s5
      move   $x17, $s6
      move   $x18, $s7
15:  lw      $x5, -4($fp)
      beq    $x5, 0, 11
12:  lw      $x6, ($fp)
      move   $a0, $x6
      lw      $x8, -4($fp)
      sub    $x7, $x8, 1
      move   $a1, $x7
      jal   10
      move   $x3, $v0
      lw      $x10, -4($fp)
      mul    $x9, $x10, $x3
      move   $x0, $x9
13:  move   $v0, $x0
      j      16
11:  li      $x0, 1
      j      13
16:  move   $s0, $x11
      move   $s1, $x12
      move   $s2, $x13
      move   $s3, $x14
      move   $s4, $x15
      move   $s5, $x16
      move   $s6, $x17
      move   $s7, $x18
```

Routine: fact

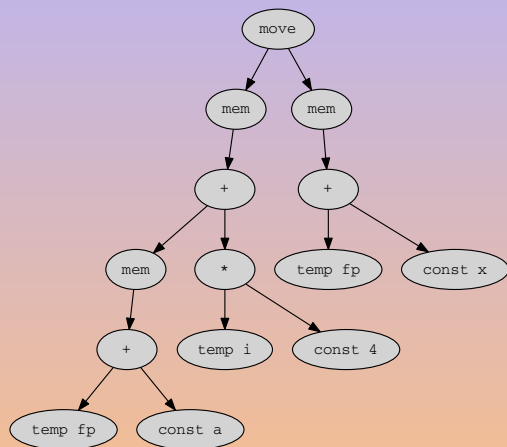
```
10:  sw      $fp, -8($sp)
      move   $fp, $sp
      sub    $sp, $sp, 16
      sw      $ra, -12($fp)
      sw      $a0, ($fp)
      sw      $a1, -4($fp)
15:  lw      $t0, -4($fp)
      beq    $t0, 0, 11
12:  lw      $a0, ($fp)
      lw      $t0, -4($fp)
      sub    $a1, $t0, 1
      jal   10
      lw      $t0, -4($fp)
      mul    $t0, $t0, $v0
13:  move   $v0, $t0
      j      16
11:  li      $t0, 1
      j      13
16:  lw      $ra, -12($fp)
      move   $sp, $fp
      lw      $fp, -8($fp)
      jr     $ra
```

Instruction Selection

- 1 Microprocessors
- 2 A Typical risc: mips
- 3 The EPITA Tiger Compiler
- 4 Instruction Selection**

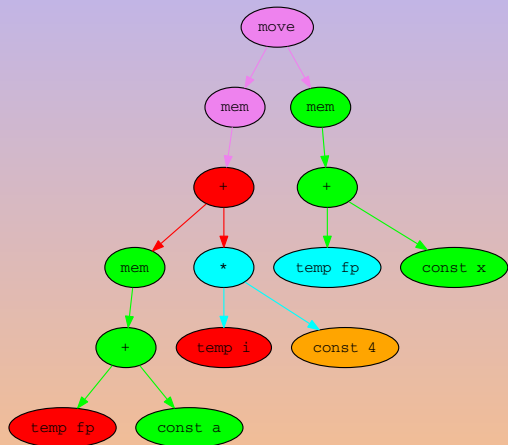
Translating a Simple Instruction

How would you translate
`a[i] := x`
where `x` is frame resident, and
`i` is not? [Appel, 1998]



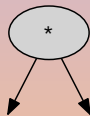
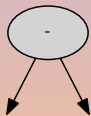
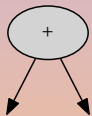
Simple Instruction: Translation 1

```
load  t17 <- M[fp + a]
addi  t18 <- r0 + 4
mul   t19 <- t1 * t18
add   t20 <- t17 + t19
load  t21 <- M[fp + x]
store M[t20 + 0] <- t21
```



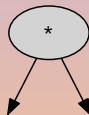
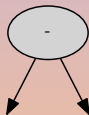
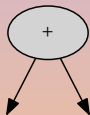
Tree Patterns

- Translation from Tree to Assembly corresponds to *parsing a tree*.
- Looking for a covering of the tree, using tiles.
- The set of tiles corresponds to the instruction set.



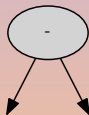
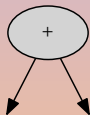
Tree Patterns

- Translation from Tree to Assembly corresponds to *parsing a tree*.
- Looking for a covering of the tree, using tiles.
- The set of tiles corresponds to the instruction set.

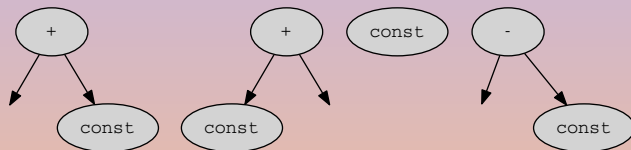


Tree Patterns

- Translation from Tree to Assembly corresponds to *parsing a tree*.
- Looking for a covering of the tree, using tiles.
- The set of tiles corresponds to the instruction set.

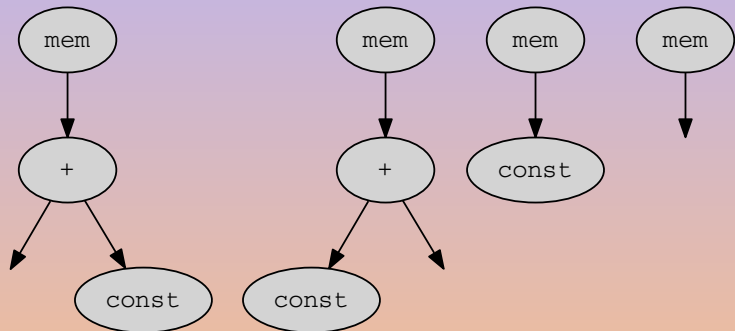


Missing nodes are plugs for *temporaries*: tiles read from temps, and create temps.

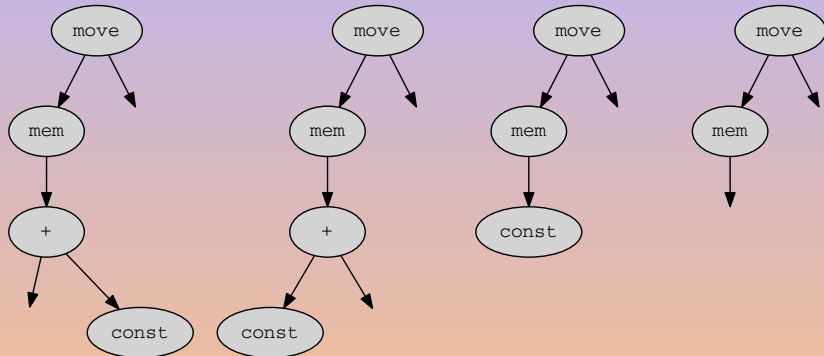


Some architectures rely on a special register to produce 0.

Tiles: Loading load $r_i \leftarrow M[r_j + c]$

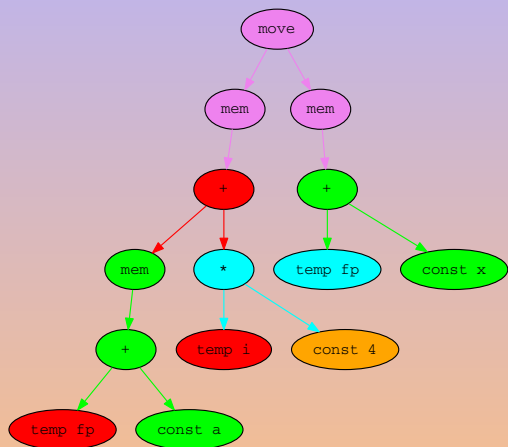


Tiles: Storing store $M[r_j + c] \leftarrow r_i$



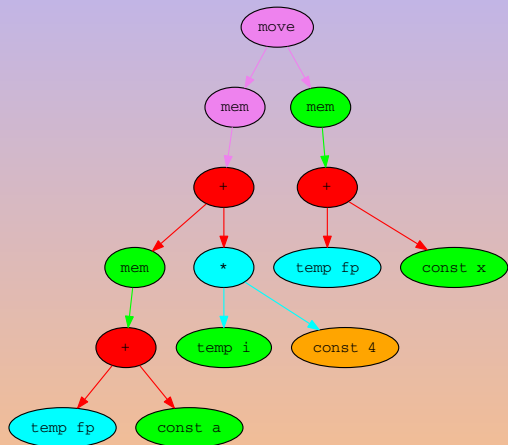
Simple Instruction: Translation 2

```
load  t17 <- M[fp + a]
addi  t18 <- r0 + 4
mul   t19 <- t1 * t18
add   t20 <- t17 + t19
addi  t21 <- fp + x
movem M[t20] <- M[t21]
```



Simple Instruction: Translation 3

```
addi t17 <- r0 + a
add t18 <- fp + t17
load t19 <- M[t18 + 0]
addi t20 <- r0 + 4
mul t21 <- ti * t20
add t22 <- t19 + t21
addi t23 <- r0 + x
add t24 <- fp + t23
load t25 <- M[t24 + 0]
store M[t22 + 0] <- t25
```



Translating a Simple Instruction

- There is always a solution
(provided the instruction set is reasonable)
- there can be several solutions
- given a cost function, some are better than others:
 - local optimum coverings (no fusion can reduce the cost)
 - some are globally better, *optimum coverings*.

Nowadays this approach is too naive:

- cpus are really layers of units that work in parallel.
- Costs are therefore interrelated.

Translating a Simple Instruction

- There is always a solution
(provided the instruction set is reasonable)
- there can be several solutions
- given a cost function, some are better than others:
 - some are locally better, *optimal coverings*
(no fusion can reduce the cost),
 - some are globally better, *optimum coverings*.

Nowadays this approach is too naive:

- cpus are really layers of units that work in parallel.
- Costs are therefore interrelated.

Translating a Simple Instruction

- There is always a solution
(provided the instruction set is reasonable)
- there can be several solutions
- given a cost function, some are better than others:
 - some are locally better, *optimal coverings*
(no fusion can reduce the cost),
 - some are globally better, *optimum coverings*.

Nowadays this approach is too naive:

- cpus are really layers of units that work in parallel.
- Costs are therefore interrelated.

Translating a Simple Instruction

- There is always a solution
(provided the instruction set is reasonable)
- there can be several solutions
- given a cost function, some are better than others:
 - some are locally better, *optimal coverings*
(no fusion can reduce the cost),
 - some are globally better, *optimum coverings*.

Nowadays this approach is too naive:

- cpus are really layers of units that work in parallel.
- Costs are therefore interrelated.

Translating a Simple Instruction

- There is always a solution (provided the instruction set is reasonable)
- there can be several solutions
- given a cost function, some are better than others:
 - some are locally better, *optimal coverings* (no fusion can reduce the cost),
 - some are globally better, *optimum coverings*.

Nowadays this approach is too naive:

- cpus are really layers of units that work in parallel.
- Costs are therefore interrelated.

Translating a Simple Instruction

- There is always a solution
(provided the instruction set is reasonable)
- there can be several solutions
- given a cost function, some are better than others:
 - some are locally better, *optimal coverings*
(no fusion can reduce the cost),
 - some are globally better, *optimum coverings*.

Nowadays this approach is too naive:

- cpus are really layers of units that work in parallel.
- Costs are therefore interrelated.

Algorithms for Instruction Selection

Maximal Munch Find an optimal tiling.

- Top-down strategy.
- Cover the current node with the largest tile.
- Repeat on subtrees.
- Generate instructions in reverse-order after tile placement.

Dynamic Programming Find an optimum tiling.

- Bottom-up strategy.
- Assign cost to each node.
- $\text{Cost} = \text{cost of selected tile} + \text{cost of subtrees}$.
- Select a tile with minimal cost and recurse upward.
- Implemented by code generator generators (Twig, Burg, iBurg, MonoBURG, ...).

Algorithms for Instruction Selection

Maximal Munch Find an optimal tiling.

- Top-down strategy.
- Cover the current node with the largest tile.
- Repeat on subtrees.
- Generate instructions in reverse-order after tile placement.

Dynamic Programming Find an optimum tiling.

- Bottom-up strategy.
- Assign cost to each node.
- Cost = cost of selected tile + cost of subtrees.
- Select a tile with minimal cost and recurse upward.
- Implemented by code generator generators (Twig, Burg, iBurg, MonoBURG, ...).

Tree Matching

- The basic operation is the *pattern matching*.
- Not all the languages stand equal before pattern matching. . .

Tree Matching

- The basic operation is the *pattern matching*.
- Not all the languages stand equal before pattern matching. . .

... in Stratego

Select-swri :

```
MOVE(MEM(BINOP(PLUS, e1, CONST(n))), e2) →  
SEQ(MOVE(r2, e2), SEQ(MOVE(r1, e1), sw-ri(r2, r1, n)))  
where <new-atemp> e1 ⇒ r1; <new-atemp> e2 ⇒ r2
```

Select-swr :

```
MOVE(MEM(e1), e2) → SEQ(MOVE(r2, e2), SEQ(MOVE(r1, e1), sw-r(r2, r1)))  
where <new-atemp> e1 ⇒ r1; <new-atemp> e2 ⇒ r2
```

Select-nop :

```
MOVE(TEMP(r), TEMP(r)) → NUL
```

Select-nop :

```
MOVE(REG(r), REG(r)) → NUL
```

Select-mover :

```
MOVE(TEMP(r), TEMP(t)) → move(TEMP(r), TEMP(t)) where <not(eq)> (r, t)
```

Select-mover :

```
MOVE(TEMP(r), REG(t)) → move(TEMP(r), REG(t)) where <not(eq)> (r, t)
```

Select-mover :

```
MOVE(REG(r), TEMP(t)) → move(REG(r), TEMP(t)) where <not(eq)> (r, t)
```

Select-mover :

```
MOVE(REG(r), REG(t)) → move(REG(r), REG(t)) where <not(eq)> (r, t)
```

... in Haskell: Ir.hs [Anisko, 2003]

```
module Ir (Stm (Move, Sxp, Jump, CJump, Seq, Label,
              LabelEnd, Literal),
          ...)

```

```
where

```

```
data Stm a =
  Move { ma :: a, lval :: Exp a, rval :: Exp a }
| Sxp a (Exp a)
| Jump a (Exp a)
| CJump { cja :: a,
         rop :: Relop, cleft :: Exp a, cright :: Exp a,
         iftrue :: Exp a, iffalse :: Exp a }
| Seq a [Stm a]
| Label { la :: a,
         name :: String, size :: Int }
| LabelEnd a
| Literal { lita :: a,
          litname :: String, litcontent :: [Int] }

```

... in Haskell Eval.hs [Anisko, 2003]

```
module Eval (evalStm, ...)
where
import Ir
import Monad (Mnd, rfetch, rstore, rpush, rpop, ...)
import Result (Res (IntRes, UnitRes))
import Profile (profileExp, profileStm)

evalStm :: Stm Loc -> Mnd ()
evalStm stm@(Move loc (Temp _ t) e) =
    do (IntRes r) <- evalExp e
       verbose loc ["move", "(", "temp", t, ")", show r]
       profileStm stm
       rstore t r

evalStm stm@(Move loc (Mem _ e) f) =
    do (IntRes r) <- evalExp e
       (IntRes s) <- evalExp f
       verbose loc ["move", "(", "mem", show r, ")", show s]
       profileStm stm
       mstore r s
```

... in Haskell Low.hs [Anisko, 2003]

```
module Low (lowExp, lowStms)
where import ...

lowStms :: Int -> [Stm Ann] -> Mnd Bool
lowStms _ [] = return True

lowStms level
  ((CJump _ _ e f _ (Name _ s)) : (Label _ s' _) : stms)
  | s == s' =
do a <- lowExp (level + 1) e
  b <- lowExp (level + 1) f
  c <- lowStms level stms
  return $ a && b && c

lowStms level (CJump l _ e f _ _ : stms) =
do awarn l ["invalid cjump"]
  lowExp (level + 1) e
  lowExp (level + 1) f
  lowStms level stms
  return False
```


... in Haskell High.hs [Anisko, 2003]

```
module High (highExp, highStms)
where import ...

highStms :: Int -> [Stm Ann] -> Mnd Bool
highStms level ss =
    do a <- sequence $ map (highStm level) ss
       return (and a)

highStm :: Int -> Stm Ann -> Mnd Bool
highStm level (Move l dest src) =
    do a <- highExp (level + 1) dest
       a' <- case dest of
            Temp _ _ -> return True
            Mem _ _ -> return True
            _ -> do awarn (annExp dest)
                  ["invalid move destination:",
                   show dest]
                  return False
       b <- highExp (level + 1) src
       return $ a && a' && b
```

... in C++

52 lines matching "switch\\|case\\|default\\|\\|/" in buffer codegen.cc.

```
28: switch (stm.kind_get ())
30:   case Tree::move_kind :
36:     switch (dst->kind_get ())
38:       case Tree::mem_kind : // dst
41:         // MOVE (MEM (...), ...)
42:         switch (src.kind_get ())
44:           // MOVE (MEM (...), MEM (...))
45:           case Tree::mem_kind : // src
55:           default : // src
57:             // MOVE (MEM (...), e1)
59:             switch (addr->kind_get ())
61:               case Tree::binop_kind : // addr
63:                 // MOVE (MEM (BINOP (...), ..., ...)) , e1)
69:                 switch (binop.oper_get ())
71:                   case Binop::minus:
73:                   case Binop::plus:
74:                     // MOVE (MEM (BINOP (+/-, e1, CONST (i))),
77:                     // MOVE (MEM (BINOP (+/-, CONST (i), e1)) ,
87:                   default:
88:                     // MOVE (MEM (BINOP (<?.., ?.., ?..)> , e1))
```

... in C++

```
case Node::move_kind :
{
  DOWN_CAST (Move, move, stm);
  const Exp* dst = move.dst_get (); const Exp* src = move.src_get ();
  switch (dst->kind_get ()) {
  case Node::mem_kind : { // dst
    DOWN_CAST (Mem, mem, *dst);
    // MOVE (MEM (...), ...)
    switch (src.kind_get ()) {
      // MOVE (MEM (...), MEM (...))
    case Node::mem_kind : // src
      ...
    default : { // src
      // MOVE (MEM (...), e1)
      const Exp* addr = dst.exp_get ();
      switch (addr->kind_get ()) {
      case Node::binop_kind : { // addr
        // MOVE (MEM (BINOP (...), ..., ...)) , e1)
        DOWN_CAST (Binop, binop, *addr);
        const Exp* binop_left = binop.left_get ();
        const Exp* binop_right = binop.right_get ();
        short sign = 1;
        switch (binop.oper_get ()) {
        case Binop::minus: sign = -1;
        case Binop::plus:
          // MOVE (MEM (BINOP (+/-, e1, CONST (i))), e2)
          if (binop_right->kind_get () == Node::const_kind)
            std::swap (binop_left, binop_right);
          // MOVE (MEM (BINOP (+/-, CONST (i), e1)) , e2)
          if (binop_left->kind_get () == Node::const_kind) {
            DOWN_CAST (Const, const_left, *binop_left);
            emit (_assembly->store_build (munchExp (src),
            munchExp (* binop_right),
```

Break down long switches into smaller functions.

Twig, Burg, iBurg [Fraser et al., 1992]

```
%{ /* ... */  
enum { ADDI=309, ADDRLP=295, ASGNI=53, CNSTI=21, CVCI=85,  
      IOI=661, INDIRC=67 };  
/* ... */  
%}  
%term ADDI=309 ADDRLP=295 ASGNI=53  
%term CNSTI=21 CVCI=85 IOI=661 INDIRC=67  
%%  
/* ... */
```





Twig, Burg, iBurg [Fraser et al., 1992]

```
/* ... */  
%%  
stmt:  ASGNI(dispatch,reg) = 4 (1);  
stmt:  reg = 5;  
reg:   ADDI(reg,rc) = 6 (1);  
reg:   CVCI(INDIRC(dispatch)) = 7 (1);  
reg:   IOI = 8;  
reg:   dispatch = 9 (1);  
dispatch: ADDI(reg,con) = 10;  
dispatch: ADDRPLP = 11;  
rc:    con = 12;  
rc:    reg = 13;  
con:   CNSTI = 14;  
con:   IOI = 15;  
%%  
/* ... */
```

```
binop: Binop(lhs : exp, rhs : Const)
{
    auto binop = tree.cast<Binop>();
    auto cst = rhs.cast<Const>();
    EMIT(IA32_ASSEMBLY
        .binop_build(binop->oper_get(), lhs->asm_get(),
                    cst->value_get(), tree->asm_get()));
}
```

```
binop: Binop(lhs : exp, rhs : exp)
{
    auto binop = tree.cast<Binop>();
    EMIT(IA32_ASSEMBLY
        .binop_build(binop->oper_get(), lhs->asm_get(),
                    rhs->asm_get(), tree->asm_get()));
}
```

Bibliography I

-  Anisko, R. (2003).
Havm.
<http://tiger.lrde.epita.fr/Havm>.
-  Appel, A. W. (1998).
Modern Compiler Implementation in C, Java, ML.
Cambridge University Press.
-  Fraser, C. W., Hanson, D. R., and Proebsting, T. A. (1992).
Engineering a simple, efficient code-generator generator.
ACM Letters on Programming Languages and Systems, 1(3):213–226.
-  Larus, J. R. (1990).
SPIM S20: A MIPS R2000 simulator.
Technical Report TR966, Computer Sciences Department, University of Wisconsin–Madison.