The SPIM Target Machine

W. M. Waite

November 27, 2005

This document assumes familiarity with the structure of assembly languages and access to the Eli documentation. It illustrates the use of Eli to describe a target machine.

The appendix provides a specification for a component that makes intermediate data structures visible.

Contents

1 The Abstract SPIM Program Tree

The basis of an abstract target tree design is the desired separation of concerns between the translator and the code generator: Certain decisions will be made during translation and others during code generation. Because the abstract target tree is the interface between these two processes, in must reflect the decisions made by the former and provide information to support the decisions made by the latter. This specification defers two classes of decision, allocation of machine registers to intermediate results and determination of operand evaluation order, to the code generator.

Most MIPS instructions belong to groups, such that the register allocation and operand evaluation order decisions for all instructions within a group are made in the same way. Thus most of the rules describing abstract target constructs are associated with groups rather than individual instructions. One of the right-hand-side symbols of each such rule is a nonliteral terminal symbol whose value specifies the instruction within the group.

Eli generates a constructor function for each rule. In order to simplify the use of the generated constructor functions, this specification defines a set of "factory methods" as macros. Many of these macros correspond to single MIPS instructions, and because SPIM supports an extended instruction set a particular instruction might have more than one possible format. The names of the macros are therefore combinations of the SPIM operation code and the desired format. For example, `add_iRegrr` is the name of the macro that constructs an integer addition of two registers and `add_iRegri` is the name of the macro that constructs an integer addition of a register and an integer constant.

The first argument of a function provides the source text coordinates of the construct corresponding to the rule, for use in error reporting during tree computations. Errors uncovered during computations over the target program tree are inevitably compiler implementation errors, however, and are not related to anything appearing in the source program. Thus all factory method macros set the first argument of the node constructor function to `NoPosition` because the source text coordinates are usually irrelevant. If source coordinates are required in a specific situation, then either call the constructor function directly or provide a different factory method.

1.1 Terminal Symbols

Terminal symbols represent target machine operations, constants, labels, and registers. Each has a value that can be used in target tree computations.

Terminal Symbols[1]:

> *Target machine operations*[2]
> *Constants*[3]
> *Labels*[4]
> *Registers*[5]

This macro is invoked in definition 71.

1.1.1 Target machine operations

A `Mop` represents a target machine operation. The value of the terminal is a definition table key specifying the relevant properties of the operator.

Target machine operations[2]:

```
TERM Mop: DefTableKey;
```

This macro is invoked in definition 1.

Known keys for the relevant MIPS operations are defined in later sections of this document. The property `Pname` of each is initialized to a string giving the SPIM assembly language opcode for the operation. A translator can define additional keys for extended operations, but in that case an additional specification of how to handle those extensions must be provided.

1.1.2 Constants

A `Denoter` represents a literal constant. The value of the terminal is the string table index of the constant's value.

Constants[3]:

```
TERM Denoter: int;
```

This macro is invoked in definition 1.

1.1.3 Labels

A `Label` represents a location known to the translator. The value of the terminal is a definition table key.

Labels[4]:

```
TERM Label: DefTableKey;
```

This macro is invoked in definition 1.

1.1.4 Registers

Registers are normally assigned by a computation over the abstract target program tree, but in some cases the translator must specify fixed registers. For example, information stored in the activation record is addressed relative to the frame pointer – a register fixed by convention.

A `Register` represents a register assigned by the translator. The value of the terminal is the number of the register assigned.

Registers[5]:

```
TERM Register: int;
```

This macro is invoked in definition 1.

1.2 Machine Operations

Most of the constructs of the target program tree correspond directly to machine operations. This section describes those constructs. It is organized in parallel with the SPIM description.

Although the SPIM manual gives immediate forms for many instructions, they are only included for reference. The assembler will translate the more general form of an instruction into the immediate form if the relevant argument is a constant. Therefore a compiler need only produce the more general instruction.

Instructions that move values from one register to another are not part of this abstraction because they imply that register assignment has been done. Register assignment may be carried out by computations over the abstract target program tree defined by this abstraction, or may be deferred until after optimizing transformations have been carried out.

1.2.1 Addressing modes

MIPS has a load/store architecture, which means that only load and store instructions access memory. Computation instructions operate only on integer constants and values in registers. The bare machine provides only one memory addressing mode: `c(rx)`, which uses the sum of the integer `c` and content of register `rx` as the address. Register `rx` might be an arbitrary register containing the result of an array indexing or indirect address computation, or it might be a fixed base register (usually `$gp` or `$fp`). An indexing or indirect address computation would be represented by the translator as an abstract target program tree rooted in an `IntReg` node; a fixed base register would be a property of the symbol being addressed. Thus two abstractions are provided for memory references:

Addressing modes[6]:

```
RULE Indexed: Memory  ::= Address IntReg   END;
RULE Based:   Memory  ::= Address Register END;
```

This macro is defined in definitions 6 and 8.
This macro is invoked in definition 71.

Factory methods[7]:

```
#define indexed_memory(x,y)   MkIndexed(NoPosition,x,y)
#define based_memory(x,y)     MkBased(NoPosition,x,y)
```

This macro is defined in definitions 7, 9, 16, 19, 22, 25, 36, 41, 46, 49, 58, 60, 63, 65, 67, and 69.
This macro is invoked in definition 74.

`Address` represents the `c` component of the memory addressing mode `c(rx)` above. It may take any of three forms:

Addressing modes[8]:

```
RULE RelAddr: Address ::= Label Denoter    END;
RULE SymAddr: Address ::= Label            END;
RULE AbsAddr: Address ::=       Denoter    END;
```

This macro is defined in definitions 6 and 8.
This macro is invoked in definition 71.

Factory methods[9]:

```
#define indexed_memory(x,y)   MkIndexed(NoPosition,x,y)
#define based_memory(x,y)     MkBased(NoPosition,x,y)
#define relative_address(x,y) MkRelAddr(NoPosition,x,y)
#define symbolic_address(x)   MkSymAddr(NoPosition,x)
#define absolute_address(x)   MkAbsAddr(NoPosition,x)
```

This macro is defined in definitions 7, 9, 16, 19, 22, 25, 36, 41, 46, 49, 58, 60, 63, 65, 67, and 69.
This macro is invoked in definition 74.

1.2.2 Arithmetic and logical instructions

SPIM allows the right operand of every arithmetic and logical instruction to be either a register or an immediate value.

Arithmetic and logical instructions[10]:

```
RULE iRegrr: IntReg ::= Mop IntReg IntReg  END;
RULE iRegri: IntReg ::= Mop IntReg Denoter END;
RULE iRegr:  IntReg ::= Mop IntReg         END;
RULE iRegi:  IntReg ::= Mop        Denoter END;
```

This macro is invoked in definition 71.

`IntReg` represents an integer-valued intermediate result held in a register. That result is created by an operation corresponding to a rule with `IntReg` on the left-hand side, and is used by an operation corresponding to a rule with `IntReg` on the right-hand side. The tree structure guarantees that each intermediate result is used exactly once.

Machine operation codes[11]:

```
absOp   -> Pname={"abs"};
addOp   -> Pname={"add"};
adduOp  -> Pname={"addu"};
andOp   -> Pname={"and"};
divOp   -> Pname={"div"};
divuOp  -> Pname={"divu"};
mulOp   -> Pname={"mul"};
muloOp  -> Pname={"mulo"};
mulouOp -> Pname={"mulou"};
negOp   -> Pname={"neg"};
neguOp  -> Pname={"negu"};
norOp   -> Pname={"nor"};
notOp   -> Pname={"not"};
orOp    -> Pname={"or"};
remOp   -> Pname={"rem"};
remuOp  -> Pname={"remu"};
rolOp   -> Pname={"rol"};
rorOp   -> Pname={"ror"};
sllOp   -> Pname={"sll"};
sraOp   -> Pname={"sra"};
srlOp   -> Pname={"srl"};
subOp   -> Pname={"sub"};
subuOp  -> Pname={"subu"};
xorOp   -> Pname={"xor"};
```

This macro is defined in definitions 11, 18, 24, 28, 39, 44, and 52.
This macro is invoked in definition 72.

The names of the factory methods used to construct arithmetic and logical instructions are formed from the MIPS operation code for the desired instruction and the name of the desired rule. Each factory method takes only the operands as arguments, supplying `NoPosition` and the value of the `Mop` terminal literally:

Construct an iRegrr node for[12](◇1):

```
#define ◇1_iRegrr(x,y) MkiRegrr(NoPosition,◇1Op,x,y)
```

This macro is invoked in definitions 16 and 25.

Construct an iRegri node for[13](◇1):

```
#define ◇1_iRegri(x,y) MkiRegri(NoPosition,◇1Op,x,y)
```

This macro is invoked in definitions 16 and 25.

Construct an iRegr node for[14](◇1):

```
#define ◇1_iRegr(x) MkiRegr(NoPosition,◇1Op,x)
```

This macro is invoked in definition 16.

Construct an iRegi node for[15](◇1):

```
#define ◇1_iRegi(x) MkiRegi(NoPosition,◇1Op,x)
```

This macro is invoked in definitions 16 and 19.

Factory methods[16]:

Construct an iRegr node for[14](‘`abs`’)
Construct an iRegrr node for[12](‘`add`’)
Construct an iRegrr node for[12](‘`and`’)
Construct an iRegrr node for[12](‘`div`’)
Construct an iRegrr node for[12](‘`mul`’)
Construct an iRegr node for[14](‘`neg`’)
Construct an iRegrr node for[12](‘`nor`’)
Construct an iRegr node for[14](‘`not`’)
Construct an iRegrr node for[12](‘`or`’)
Construct an iRegrr node for[12](‘`rem`’)
Construct an iRegrr node for[12](‘`rol`’)
Construct an iRegrr node for[12](‘`ror`’)
Construct an iRegrr node for[12](‘`sll`’)
Construct an iRegrr node for[12](‘`sra`’)
Construct an iRegrr node for[12](‘`srl`’)
Construct an iRegrr node for[12](‘`sub`’)
Construct an iRegrr node for[12](‘`xor`’)

Construct an iRegi node for[15](‘`abs`’)
Construct an iRegri node for[13](‘`add`’)
Construct an iRegri node for[13](‘`and`’)
Construct an iRegri node for[13](‘`div`’)
Construct an iRegri node for[13](‘`mul`’)
Construct an iRegi node for[15](‘`neg`’)
Construct an iRegri node for[13](‘`nor`’)
Construct an iRegi node for[15](‘`not`’)
Construct an iRegri node for[13](‘`or`’)
Construct an iRegri node for[13](‘`rem`’)

Construct an iRegri node for[13](‘`rol`’)
Construct an iRegri node for[13](‘`ror`’)
Construct an iRegri node for[13](‘`sll`’)
Construct an iRegri node for[13](‘`sra`’)
Construct an iRegri node for[13](‘`srl`’)
Construct an iRegri node for[13](‘`sub`’)
Construct an iRegri node for[13](‘`xor`’)

This macro is defined in definitions 7, 9, 16, 19, 22, 25, 36, 41, 46, 49, 58, 60, 63, 65, 67, and 69.
This macro is invoked in definition 74.

Tree parser rules[17]:

```
IntReg: NODEPTR;

IntReg ::= absTp(IntReg)        :  abs_iRegr;
IntReg ::= addTp(IntReg,IntReg) :  add_iRegrr;
IntReg ::= andTp(IntReg,IntReg) :  and_iRegrr;
IntReg ::= divTp(IntReg,IntReg) :  div_iRegrr;
IntReg ::= mulTp(IntReg,IntReg) :  mul_iRegrr;
IntReg ::= negTp(IntReg)        :  neg_iRegr;
IntReg ::= norTp(IntReg,IntReg) :  nor_iRegrr;
IntReg ::= notTp(IntReg)        :  not_iRegr;
IntReg ::= orTp(IntReg,IntReg)  :  or_iRegrr;
IntReg ::= remTp(IntReg,IntReg) :  rem_iRegrr;
IntReg ::= rolTp(IntReg,IntReg) :  rol_iRegrr;
IntReg ::= rorTp(IntReg,IntReg) :  ror_iRegrr;
IntReg ::= sllTp(IntReg,IntReg) :  sll_iRegrr;
IntReg ::= sraTp(IntReg,IntReg) :  sra_iRegrr;
IntReg ::= srlTp(IntReg,IntReg) :  srl_iRegrr;
IntReg ::= subTp(IntReg,IntReg) :  sub_iRegrr;
IntReg ::= xorTp(IntReg,IntReg) :  xor_iRegrr;

IntReg ::= absTp(IntVal)        :  abs_iRegi;
IntReg ::= addTp(IntReg,IntVal) :: add_iRegri;
IntReg ::= andTp(IntReg,IntVal) :: and_iRegri;
IntReg ::= divTp(IntReg,IntVal) :  div_iRegri;
IntReg ::= mulTp(IntReg,IntVal) :: mul_iRegri;
IntReg ::= negTp(IntVal)        :  neg_iRegi;
IntReg ::= norTp(IntReg,IntVal) :: nor_iRegri;
IntReg ::= notTp(IntVal)        :  not_iRegi;
IntReg ::= orTp(IntReg,IntVal)  :: or_iRegri;
IntReg ::= remTp(IntReg,IntVal) :  rem_iRegri;
IntReg ::= rolTp(IntReg,IntVal) :  rol_iRegri;
IntReg ::= rorTp(IntReg,IntVal) :  ror_iRegri;
IntReg ::= sllTp(IntReg,IntVal) :  sll_iRegri;
IntReg ::= sraTp(IntReg,IntVal) :  sra_iRegri;
IntReg ::= srlTp(IntReg,IntVal) :  srl_iRegri;
IntReg ::= subTp(IntReg,IntVal) :  sub_iRegri;
IntReg ::= xorTp(IntReg,IntVal) :: xor_iRegri;
```

This macro is defined in definitions 17, 20, 23, 26, 37, 42, 47, 50, 59, 61, and 70.

This macro is invoked in definition 73.

1.2.3 Constant-manipulating instructions

The form of the constant-manipulating instructions is `iRegi`.

Machine operation codes[18]:

```
liOp  -> Pname={"li"};
luiOp -> Pname={"lui"};
```

This macro is defined in definitions 11, 18, 24, 28, 39, 44, and 52.
This macro is invoked in definition 72.

Factory methods[19]:

Construct an iRegi node for[15](‘`li`’)
Construct an iRegi node for[15](‘`lui`’)

This macro is defined in definitions 7, 9, 16, 19, 22, 25, 36, 41, 46, 49, 58, 60, 63, 65, 67, and 69.
This macro is invoked in definition 74.

Tree parser rules[20]:

```
IntVal: int;

IntReg ::= IntVal :  li_iRegi;
```

This macro is defined in definitions 17, 20, 23, 26, 37, 42, 47, 50, 59, 61, and 70.
This macro is invoked in definition 73.

The MIPS does not allow floating-point constants in instructions. Thus floating-point constants used as operands must be stored in memory. While the translator could reserve storage for all of the floating-point constants appearing in the source program, it would not be able to take advantage of constant folding during target program tree construction. Such storage reservation also conceals the fact that the values themselves are constant, a fact that might be useful in later optimization. These objections can be overcome by defining a floating-point analog of the `li` instruction:

Floating Point Constants[21]:

```
RULE LoadFloat: FltReg ::= Denoter END;
```

This macro is invoked in definition 71.

Factory methods[22]:

```
#define FltVal(x) MkLoadFloat(NoPosition,x)
```

This macro is defined in definitions 7, 9, 16, 19, 22, 25, 36, 41, 46, 49, 58, 60, 63, 65, 67, and 69.
This macro is invoked in definition 74.

Tree parser rules[23]:

```
FltReg: NODEPTR;
FltVal: int;

FltReg ::= FltVal       :  FltVal;
```

This macro is defined in definitions 17, 20, 23, 26, 37, 42, 47, 50, 59, 61, and 70.
This macro is invoked in definition 73.

An assembly language translation of `LoadFloat` must establish a memory location containing the constant specified by `Denoter`, and generate a floating-point load from that memory location.

1.2.4 Comparison instructions

The form of the comparison operations is either `iRegrr` or `iRegri`.

Machine operation codes[24]:

```
seqOp  -> Pname={"seq"};
sgeOp  -> Pname={"sge"};
sgeuOp -> Pname={"sgeu"};
sgtOp  -> Pname={"sgt"};
sgtuOp -> Pname={"sgtu"};
sleOp  -> Pname={"sle"};
sleuOp -> Pname={"sleu"};
sltOp  -> Pname={"slt"};
sltuOp -> Pname={"sltu"};
sneOp  -> Pname={"sne"};
```

This macro is defined in definitions 11, 18, 24, 28, 39, 44, and 52.
This macro is invoked in definition 72.

Factory methods[25]:

Construct an iRegrr node for[12](‘`seq`’)
Construct an iRegrr node for[12](‘`sge`’)
Construct an iRegrr node for[12](‘`sgeu`’)
Construct an iRegrr node for[12](‘`sgt`’)
Construct an iRegrr node for[12](‘`sgtu`’)
Construct an iRegrr node for[12](‘`sle`’)
Construct an iRegrr node for[12](‘`sleu`’)
Construct an iRegrr node for[12](‘`slt`’)
Construct an iRegrr node for[12](‘`sltu`’)
Construct an iRegrr node for[12](‘`sne`’)

Construct an iRegri node for[13](‘`seq`’)
Construct an iRegri node for[13](‘`sge`’)
Construct an iRegri node for[13](‘`sgeu`’)
Construct an iRegri node for[13](‘`sgt`’)
Construct an iRegri node for[13](‘`sgtu`’)
Construct an iRegri node for[13](‘`sle`’)
Construct an iRegri node for[13](‘`sleu`’)

Construct an iRegri node for[13](‘`slt`’)
Construct an iRegri node for[13](‘`sltu`’)
Construct an iRegri node for[13](‘`sne`’)

This macro is defined in definitions 7, 9, 16, 19, 22, 25, 36, 41, 46, 49, 58, 60, 63, 65, 67, and 69.
This macro is invoked in definition 74.

Tree parser rules[26]:

```
IntReg ::= seqTp(IntReg,IntReg) :  seq_iRegrr;
IntReg ::= sgeTp(IntReg,IntReg) :  sge_iRegrr;
IntReg ::= sgeuTp(IntReg,IntReg)        :  sgeu_iRegrr;
IntReg ::= sgtTp(IntReg,IntReg) :  sgt_iRegrr;
IntReg ::= sgtuTp(IntReg,IntReg)        :  sgtu_iRegrr;
IntReg ::= sleTp(IntReg,IntReg) :  sle_iRegrr;
IntReg ::= sleuTp(IntReg,IntReg)        :  sleu_iRegrr;
IntReg ::= sltTp(IntReg,IntReg) :  slt_iRegrr;
IntReg ::= sltuTp(IntReg,IntReg)        :  sltu_iRegrr;
IntReg ::= sneTp(IntReg,IntReg) :  sne_iRegrr;

IntReg ::= seqTp(IntReg,IntVal) :: seq_iRegri;
IntReg ::= sgeTp(IntReg,IntVal) :  sge_iRegri;
IntReg ::= sgeuTp(IntReg,IntVal)        :  sgeu_iRegri;
IntReg ::= sgtTp(IntReg,IntVal) :  sgt_iRegri;
IntReg ::= sgtuTp(IntReg,IntVal)        :  sgtu_iRegri;
IntReg ::= sleTp(IntReg,IntVal) :  sle_iRegri;
IntReg ::= sleuTp(IntReg,IntVal)        :  sleu_iRegri;
IntReg ::= sltTp(IntReg,IntVal) :  slt_iRegri;
IntReg ::= sltuTp(IntReg,IntVal)        :  sltu_iRegri;
IntReg ::= sneTp(IntReg,IntVal) :: sne_iRegri;

IntReg ::= eqTp(IntReg,IntReg)  :  seq_iRegrr;
IntReg ::= geTp(IntReg,IntReg)  :  sge_iRegrr;
IntReg ::= geuTp(IntReg,IntReg) :  sgeu_iRegrr;
IntReg ::= gtTp(IntReg,IntReg)  :  sgt_iRegrr;
IntReg ::= gtuTp(IntReg,IntReg) :  sgtu_iRegrr;
IntReg ::= leTp(IntReg,IntReg)  :  sle_iRegrr;
IntReg ::= leuTp(IntReg,IntReg) :  sleu_iRegrr;
IntReg ::= ltTp(IntReg,IntReg)  :  slt_iRegrr;
IntReg ::= ltuTp(IntReg,IntReg) :  sltu_iRegrr;
IntReg ::= neTp(IntReg,IntReg)  :  sne_iRegrr;

IntReg ::= eqTp(IntReg,IntVal)  :: seq_iRegri;
IntReg ::= geTp(IntReg,IntVal)  :  sge_iRegri;
IntReg ::= geuTp(IntReg,IntVal) :  sgeu_iRegri;
IntReg ::= gtTp(IntReg,IntVal)  :  sgt_iRegri;
IntReg ::= gtuTp(IntReg,IntVal) :  sgtu_iRegri;
IntReg ::= leTp(IntReg,IntVal)  :  sle_iRegri;
IntReg ::= leuTp(IntReg,IntVal) :  sleu_iRegri;
IntReg ::= ltTp(IntReg,IntVal)  :  slt_iRegri;
IntReg ::= ltuTp(IntReg,IntVal) :  sltu_iRegri;
IntReg ::= neTp(IntReg,IntVal)  :: sne_iRegri;
```

This macro is defined in definitions 17, 20, 23, 26, 37, 42, 47, 50, 59, 61, and 70.
This macro is invoked in definition 73.

1.2.5 Branch and jump instructions

Branch and jump instructions do not deliver a result value. Instead, their "result" is a transfer of control. For the purposes of the target program tree abstraction, the distinction between branch and jump constructs is that the destination of the control transfer for a branch construct is determined by computation over the target program tree, whereas the destination of a jump construct is either specified explicitly by the translator or given by the content of a register.

The target label for the instruction of a branch construct will be computed as an attribute of the `Branch` symbol during target attribution. Since all conditional transfers of control are SPIM branch instructions, `JumpCond` is provided to allow a conditional transfer whose destination is determined by the translator.

Branch and jump instructions[27]:

```
RULE Branchn:  Branch ::= Mop                END;
RULE Branchcf: Branch ::= Mop CndFlg         END;
RULE Branchrr: Branch ::= Mop IntReg IntReg  END;
RULE Branchri: Branch ::= Mop IntReg Denoter END;
RULE Branchr:  Branch ::= Mop IntReg         END;

RULE Jump:     Item   ::= Mop        Label   END;
RULE Jumpr:    Item   ::= Mop IntReg         END;
RULE JumpCond: Item   ::= Branch     Label   END;
```

This macro is invoked in definition 71.

Machine operation codes[28]:

```
bOp      -> Pname={"b"};
bc1tOp   -> Pname={"bc1t"};
bc1fOp   -> Pname={"bc1f"};
beqOp    -> Pname={"beq"};
beqzOp   -> Pname={"beqz"};
bgeOp    -> Pname={"bge"};
bgeuOp   -> Pname={"bgeu"};
bgezOp   -> Pname={"bgez"};
bgezalOp -> Pname={"bgezal"};
bgtOp    -> Pname={"bgt"};
bgtuOp   -> Pname={"bgtu"};
bgtzOp   -> Pname={"bgtz"};
bleOp    -> Pname={"ble"};
bleuOp   -> Pname={"bleu"};
blezOp   -> Pname={"blez"};
bltOp    -> Pname={"blt"};
bltuOp   -> Pname={"bltu"};
bltzOp   -> Pname={"bltz"};
bltzalOp -> Pname={"bltzal"};
bneOp    -> Pname={"bne"};
bnezOp   -> Pname={"bnez"};
```

```
jOp      -> Pname={"j"};
jalOp    -> Pname={"jal"};
jalrOp   -> Pname={"jalr"};
jrOp     -> Pname={"jr"};
```

This macro is defined in definitions 11, 18, 24, 28, 39, 44, and 52.
This macro is invoked in definition 72.

The factory methods for constructing nodes representing branch and jump instructions are:

Construct a Branchn node for[29](⋄1):

```
#define ⋄1_Branchn() MkBranchn(NoPosition,⋄1Op)
```

This macro is invoked in definition 36.

Construct a Branchcf node for[30](⋄1):

```
#define ⋄1_Branchcf(x) MkBranchcf(NoPosition,⋄1Op,x)
```

This macro is invoked in definition 36.

Construct a Branchrr node for[31](⋄1):

```
#define ⋄1_Branchrr(x,y) MkBranchrr(NoPosition,⋄1Op,x,y)
```

This macro is invoked in definition 36.

Construct a Branchri node for[32](⋄1):

```
#define ⋄1_Branchri(x,y) MkBranchri(NoPosition,⋄1Op,x,y)
```

This macro is invoked in definition 36.

Construct a Branchr node for[33](⋄1):

```
#define ⋄1_Branchr(x) MkBranchr(NoPosition,⋄1Op,x)
```

This macro is invoked in definition 36.

Construct a Jump node for[34](⋄1):

```
#define ⋄1_Jump(x) MkJump(NoPosition,⋄1Op,x)
```

This macro is invoked in definition 36.

Construct a Jumpr node for[35](⋄1):

```
#define ⋄1_Jumpr(x) MkJumpr(NoPosition,⋄1Op,x)
```

This macro is invoked in definition 36.

Factory methods[36]:

Construct a Branchn node for[29](‘`b`’)
Construct a Branchcf node for[30](‘`bc1t`’)
Construct a Branchcf node for[30](‘`bc1f`’)
Construct a Branchrr node for[31](‘`beq`’)
Construct a Branchr node for[33](‘`beqz`’)
Construct a Branchrr node for[31](‘`bge`’)
Construct a Branchrr node for[31](‘`bgeu`’)
Construct a Branchr node for[33](‘`bgez`’)
Construct a Branchr node for[33](‘`bgezal`’)
Construct a Branchrr node for[31](‘`bgt`’)
Construct a Branchrr node for[31](‘`bgtu`’)
Construct a Branchr node for[33](‘`bgtz`’)
Construct a Branchrr node for[31](‘`ble`’)
Construct a Branchrr node for[31](‘`bleu`’)
Construct a Branchr node for[33](‘`blez`’)
Construct a Branchrr node for[31](‘`blt`’)
Construct a Branchrr node for[31](‘`bltu`’)
Construct a Branchr node for[33](‘`bltz`’)
Construct a Branchr node for[33](‘`bltzal`’)
Construct a Branchrr node for[31](‘`bne`’)
Construct a Branchr node for[33](‘`bnez`’)

Construct a Branchri node for[32](‘`beq`’)
Construct a Branchri node for[32](‘`bge`’)
Construct a Branchri node for[32](‘`bgeu`’)
Construct a Branchri node for[32](‘`bgt`’)
Construct a Branchri node for[32](‘`bgtu`’)
Construct a Branchri node for[32](‘`ble`’)
Construct a Branchri node for[32](‘`bleu`’)
Construct a Branchri node for[32](‘`blt`’)
Construct a Branchri node for[32](‘`bltu`’)
Construct a Branchri node for[32](‘`bne`’)

Construct a Jump node for[34](‘`j`’)
Construct a Jump node for[34](‘`jal`’)

Construct a Jumpr node for[35](‘`jalr`’)
Construct a Jumpr node for[35](‘`jr`’)

```
#define do_branch(x,y) MkJumpCond(NoPosition,x,y)
```

This macro is defined in definitions 7, 9, 16, 19, 22, 25, 36, 41, 46, 49, 58, 60, 63, 65, 67, and 69.
This macro is invoked in definition 74.

Tree parser rules[37]:

```
Branch, BrIfTr, BrIfFl: NODEPTR;

Branch ::= beqTp(IntReg,IntReg) :  beq_Branchrr;
Branch ::= beqzTp(IntReg)       :  beqz_Branchr;
```

```
Branch ::= bgeTp(IntReg,IntReg) :  bge_Branchrr;
Branch ::= bgeuTp(IntReg,IntReg)        :  bgeu_Branchrr;
Branch ::= bgezTp(IntReg)       :  bgez_Branchr;
Branch ::= bgezalTp(IntReg)     :  bgezal_Branchr;
Branch ::= bgtTp(IntReg,IntReg) :  bgt_Branchrr;
Branch ::= bgtuTp(IntReg,IntReg)        :  bgtu_Branchrr;
Branch ::= bgtzTp(IntReg)       :  bgtz_Branchr;
Branch ::= bleTp(IntReg,IntReg) :  ble_Branchrr;
Branch ::= bleuTp(IntReg,IntReg)        :  bleu_Branchrr;
Branch ::= blezTp(IntReg)       :  blez_Branchr;
Branch ::= bltTp(IntReg,IntReg) :  blt_Branchrr;
Branch ::= bltuTp(IntReg,IntReg)        :  bltu_Branchrr;
Branch ::= bltzTp(IntReg)       :  bltz_Branchr;
Branch ::= bltzalTp(IntReg)     :  bltzal_Branchr;
Branch ::= bneTp(IntReg,IntReg) :  bne_Branchrr;
Branch ::= bnezTp(IntReg)       :  bnez_Branchr;

Branch ::= beqTp(IntReg,IntVal) :: beq_Branchri;
Branch ::= bgeTp(IntReg,IntVal) :  bge_Branchri;
Branch ::= bgeuTp(IntReg,IntVal)        :  bgeu_Branchri;
Branch ::= bgtTp(IntReg,IntVal) :  bgt_Branchri;
Branch ::= bgtuTp(IntReg,IntVal)        :  bgtu_Branchri;
Branch ::= bleTp(IntReg,IntVal) :  ble_Branchri;
Branch ::= bleuTp(IntReg,IntVal)        :  bleu_Branchri;
Branch ::= bltTp(IntReg,IntVal) :  blt_Branchri;
Branch ::= bltuTp(IntReg,IntVal)        :  bltu_Branchri;
Branch ::= bneTp(IntReg,IntVal) :: bne_Branchri;

BrIfTr ::= IntReg               :  bnez_Branchr;
BrIfTr ::= eqTp(IntReg,IntReg)  :  beq_Branchrr;
BrIfTr ::= eqzTp(IntReg)        :  beqz_Branchr;
BrIfTr ::= geTp(IntReg,IntReg)  :  bge_Branchrr;
BrIfTr ::= geuTp(IntReg,IntReg) :  bgeu_Branchrr;
BrIfTr ::= gezTp(IntReg)        :  bgez_Branchr;
BrIfTr ::= gezalTp(IntReg)      :  bgezal_Branchr;
BrIfTr ::= gtTp(IntReg,IntReg)  :  bgt_Branchrr;
BrIfTr ::= gtuTp(IntReg,IntReg) :  bgtu_Branchrr;
BrIfTr ::= gtzTp(IntReg)        :  bgtz_Branchr;
BrIfTr ::= leTp(IntReg,IntReg)  :  ble_Branchrr;
BrIfTr ::= leuTp(IntReg,IntReg) :  bleu_Branchrr;
BrIfTr ::= lezTp(IntReg)        :  blez_Branchr;
BrIfTr ::= ltTp(IntReg,IntReg)  :  blt_Branchrr;
BrIfTr ::= ltuTp(IntReg,IntReg) :  bltu_Branchrr;
BrIfTr ::= ltzTp(IntReg)        :  bltz_Branchr;
BrIfTr ::= ltzalTp(IntReg)      :  bltzal_Branchr;
BrIfTr ::= neTp(IntReg,IntReg)  :  bne_Branchrr;
BrIfTr ::= nezTp(IntReg)        :  bnez_Branchr;

BrIfTr ::= eqTp(IntReg,IntVal)  :: beq_Branchri;
BrIfTr ::= geTp(IntReg,IntVal)  :  bge_Branchri;
BrIfTr ::= geuTp(IntReg,IntVal) :  bgeu_Branchri;
```

```
BrIfTr ::= gtTp(IntReg,IntVal)  :  bgt_Branchri;
BrIfTr ::= gtuTp(IntReg,IntVal) :  bgtu_Branchri;
BrIfTr ::= leTp(IntReg,IntVal)  :  ble_Branchri;
BrIfTr ::= leuTp(IntReg,IntVal) :  bleu_Branchri;
BrIfTr ::= ltTp(IntReg,IntVal)  :  blt_Branchri;
BrIfTr ::= ltuTp(IntReg,IntVal) :  bltu_Branchri;
BrIfTr ::= neTp(IntReg,IntVal)  :: bne_Branchri;

BrIfTr ::= notTp(BrIfFl)        :  copy;

BrIfFl ::= IntReg               :  beqz_Branchr;
BrIfFl ::= eqTp(IntReg,IntReg)  :  bne_Branchrr;
BrIfFl ::= eqzTp(IntReg)        :  bnez_Branchr;
BrIfFl ::= geTp(IntReg,IntReg)  :  blt_Branchrr;
BrIfFl ::= geuTp(IntReg,IntReg) :  bltu_Branchrr;
BrIfFl ::= gezTp(IntReg)        :  bltz_Branchr;
BrIfFl ::= gezalTp(IntReg)      :  bltzal_Branchr;
BrIfFl ::= gtTp(IntReg,IntReg)  :  ble_Branchrr;
BrIfFl ::= gtuTp(IntReg,IntReg) :  bleu_Branchrr;
BrIfFl ::= gtzTp(IntReg)        :  blez_Branchr;
BrIfFl ::= leTp(IntReg,IntReg)  :  bgt_Branchrr;
BrIfFl ::= leuTp(IntReg,IntReg) :  bgtu_Branchrr;
BrIfFl ::= lezTp(IntReg)        :  bgtz_Branchr;
BrIfFl ::= ltTp(IntReg,IntReg)  :  bge_Branchrr;
BrIfFl ::= ltuTp(IntReg,IntReg) :  bgeu_Branchrr;
BrIfFl ::= ltzTp(IntReg)        :  bgez_Branchr;
BrIfFl ::= ltzalTp(IntReg)      :  bgezal_Branchr;
BrIfFl ::= neTp(IntReg,IntReg)  :  beq_Branchrr;
BrIfFl ::= nezTp(IntReg)        :  beqz_Branchr;

BrIfFl ::= eqTp(IntReg,IntVal)  :: bne_Branchri;
BrIfFl ::= geTp(IntReg,IntVal)  :  blt_Branchri;
BrIfFl ::= geuTp(IntReg,IntVal) :  bltu_Branchri;
BrIfFl ::= gtTp(IntReg,IntVal)  :  ble_Branchri;
BrIfFl ::= gtuTp(IntReg,IntVal) :  bleu_Branchri;
BrIfFl ::= leTp(IntReg,IntVal)  :  bgt_Branchri;
BrIfFl ::= leuTp(IntReg,IntVal) :  bgtu_Branchri;
BrIfFl ::= ltTp(IntReg,IntVal)  :  bge_Branchri;
BrIfFl ::= ltuTp(IntReg,IntVal) :  bgeu_Branchri;
BrIfFl ::= neTp(IntReg,IntVal)  :: beq_Branchri;

BrIfFl ::= notTp(BrIfTr)        :  copy;
```

This macro is defined in definitions 17, 20, 23, 26, 37, 42, 47, 50, 59, 61, and 70.
This macro is invoked in definition 73.

1.2.6 Load instructions

SPIM load instructions move values from memory to registers. Only the operations that move values to CPU registers are described here. The extended SPIM instructions for loading values into floating-point coprocessor registers are described with the floating-point instructions.

Load instructions[38]:

```
RULE iLoad: IntReg ::= Mop Memory END;
```

This macro is invoked in definition 71.

Machine operation codes[39]:

```
laOp   -> Pname={"la"};
lbOp   -> Pname={"lb"};
lbuOp  -> Pname={"lbu"};
lhOp   -> Pname={"lh"};
lhuOp  -> Pname={"lhu"};
lwOp   -> Pname={"lw"};
lwlOp  -> Pname={"lwl"};
lwrOp  -> Pname={"lwr"};
ulhOp  -> Pname={"ulh"};
ulhuOp -> Pname={"ulhu"};
ulwOp  -> Pname={"ulw"};
```

This macro is defined in definitions 11, 18, 24, 28, 39, 44, and 52.
This macro is invoked in definition 72.

Construct an iLoad node for[40](◇1):

```
#define ◇1_iLoad(x) MkiLoad(NoPosition,◇1Op,x)
```

This macro is invoked in definition 41.

Factory methods[41]:

Construct an iLoad node for[40](‘`la`’)
Construct an iLoad node for[40](‘`lb`’)
Construct an iLoad node for[40](‘`lbu`’)
Construct an iLoad node for[40](‘`ld`’)
Construct an iLoad node for[40](‘`lh`’)
Construct an iLoad node for[40](‘`lhu`’)
Construct an iLoad node for[40](‘`lw`’)
Construct an iLoad node for[40](‘`lwl`’)
Construct an iLoad node for[40](‘`lwr`’)
Construct an iLoad node for[40](‘`ulh`’)
Construct an iLoad node for[40](‘`ulhu`’)
Construct an iLoad node for[40](‘`ulw`’)

This macro is defined in definitions 7, 9, 16, 19, 22, 25, 36, 41, 46, 49, 58, 60, 63, 65, 67, and 69.
This macro is invoked in definition 74.

Tree parser rules[42]:

```
IntReg ::= laTp(IntMem) : la_iLoad;
IntReg ::= lbTp(IntMem) : lb_iLoad;
```

```
IntReg ::= lbuTp(IntMem)        : lbu_iLoad;
IntReg ::= ldTp(IntMem) : ld_iLoad;
IntReg ::= lhTp(IntMem) : lh_iLoad;
IntReg ::= lhuTp(IntMem)        : lhu_iLoad;
IntReg ::= lwTp(IntMem) : lw_iLoad;
IntReg ::= lwlTp(IntMem)        : lwl_iLoad;
IntReg ::= lwrTp(IntMem)        : lwr_iLoad;
IntReg ::= ulhTp(IntMem)        : ulh_iLoad;
IntReg ::= ulhuTp(IntMem)       : ulhu_iLoad;
IntReg ::= ulwTp(IntMem)        : ulw_iLoad;

IntReg ::= IntMem : lw_iLoad;
```

This macro is defined in definitions 17, 20, 23, 26, 37, 42, 47, 50, 59, 61, and 70.
This macro is invoked in definition 73.

1.2.7 Store instructions

SPIM store instructions move values from registers to memory. Only the operations that move values from CPU registers are described here. The extended SPIM instructions for storing values from floating-point coprocessor registers are described with the floating-point instructions.

Store instructions[43]:

```
RULE iStore: IntReg ::= Mop IntReg Memory END;
```

This macro is invoked in definition 71.

Machine operation codes[44]:

```
sbOp   -> Pname={"sb"};
shOp   -> Pname={"sh"};
swOp   -> Pname={"sw"};
swlOp  -> Pname={"swl"};
swrOp  -> Pname={"swr"};
ushOp  -> Pname={"ush"};
uswOp  -> Pname={"usw"};
```

This macro is defined in definitions 11, 18, 24, 28, 39, 44, and 52.
This macro is invoked in definition 72.

Construct an iStore node for[45](⋄1):

```
#define ⋄1_iStore(x,y) MkiStore(NoPosition,⋄1Op,x,y)
```

This macro is invoked in definition 46.

Factory methods[46]:

Construct an iStore node for[45](‘`sb`’)
Construct an iStore node for[45](‘`sd`’)
Construct an iStore node for[45](‘`sh`’)

Construct an iStore node for[45](‘`sw`’)
Construct an iStore node for[45](‘`swl`’)
Construct an iStore node for[45](‘`swr`’)
Construct an iStore node for[45](‘`ush`’)
Construct an iStore node for[45](‘`usw`’)

This macro is defined in definitions 7, 9, 16, 19, 22, 25, 36, 41, 46, 49, 58, 60, 63, 65, 67, and 69.
This macro is invoked in definition 74.

Tree parser rules[47]:

```
IntReg ::= sbTp(IntReg,IntMem)  :  sb_iStore;
IntReg ::= sdTp(IntReg,IntMem)  :  sd_iStore;
IntReg ::= shTp(IntReg,IntMem)  :  sh_iStore;
IntReg ::= swTp(IntReg,IntMem)  :  sw_iStore;
IntReg ::= swlTp(IntReg,IntMem) :  swl_iStore;
IntReg ::= swrTp(IntReg,IntMem) :  swr_iStore;
IntReg ::= ushTp(IntReg,IntMem) :  ush_iStore;
IntReg ::= uswTp(IntReg,IntMem) :  usw_iStore;

IntReg ::= stoTp(IntReg,IntMem) :  sw_iStore;
```

This macro is defined in definitions 17, 20, 23, 26, 37, 42, 47, 50, 59, 61, and 70.
This macro is invoked in definition 73.

1.2.8 Data movement instructions

The SPIM data movement instructions copy information from one register to another. Copying from one register to another in the same class is not appropriate in an abstraction that defers register allocation: Because register allocation is deferred, the translator does not know which registers contain specific values and therefore has no basis for creating such instructions. Copying between classes is, however, important.

Data movement instructions[48]:

```
RULE mtMover: FltReg ::= IntReg END;
RULE mfMover: IntReg ::= FltReg END;
```

This macro is invoked in definition 71.

Factory methods[49]:

```
#define mt_move(x) MkmtMover(NoPosition,x)
#define mf_move(x) MkmfMover(NoPosition,x)
```

This macro is defined in definitions 7, 9, 16, 19, 22, 25, 36, 41, 46, 49, 58, 60, 63, 65, 67, and 69.
This macro is invoked in definition 74.

Tree parser rules[50]:

```
FltReg ::= IntReg : mt_move;
IntReg ::= FltReg : mf_move;
```

This macro is defined in definitions 17, 20, 23, 26, 37, 42, 47, 50, 59, 61, and 70.
This macro is invoked in definition 73.

1.2.9 Floating point instructions

Floating point instructions[51]:

```
RULE fRegrr: FltReg ::= Mop FltReg FltReg END;
RULE fRegr:  FltReg ::= Mop FltReg        END;
RULE fLoad:  FltReg ::= Mop        Memory END;
RULE fStore: FltReg ::= Mop FltReg Memory END;
RULE fCmprr: CndFlg ::= Mop FltReg FltReg END;
```

This macro is invoked in definition 71.

Machine operation codes[52]:

```
absdOp  -> Pname={"abs.d"};
abssOp  -> Pname={"abs.s"};
adddOp  -> Pname={"add.d"};
addsOp  -> Pname={"add.s"};
ceqdOp  -> Pname={"c.eq.d"};
ceqsOp  -> Pname={"c.eq.s"};
cledOp  -> Pname={"c.le.d"};
clesOp  -> Pname={"c.le.s"};
cltdOp  -> Pname={"c.lt.d"};
cltsOp  -> Pname={"c.lt.s"};
cvtdsOp -> Pname={"cvt.d.s"};
cvtdwOp -> Pname={"cvt.d.w"};
cvtsdOp -> Pname={"cvt.s.d"};
cvtswOp -> Pname={"cvt.s.w"};
cvtwdOp -> Pname={"cvt.w.d"};
cvtwsOp -> Pname={"cvt.w.s"};
divdOp  -> Pname={"div.d"};
divsOp  -> Pname={"div.s"};
ldOp    -> Pname={"l.d"};
lsOp    -> Pname={"l.s"};
muldOp  -> Pname={"mul.d"};
mulsOp  -> Pname={"mul.s"};
negdOp  -> Pname={"neg.d"};
negsOp  -> Pname={"neg.s"};
sdOp    -> Pname={"s.d"};
ssOp    -> Pname={"s.s"};
subdOp  -> Pname={"sub.d"};
subsOp  -> Pname={"sub.s"};
```

This macro is defined in definitions 11, 18, 24, 28, 39, 44, and 52.
This macro is invoked in definition 72.

The factory methods for constructing nodes representing floating point instructions are:

Construct an fRegrr node for[53](◇1):

```
#define ◇1_fRegrr(x,y) MkfRegrr(NoPosition,◇1Op,x,y)
```

This macro is invoked in definition 58.

Construct an fRegr node for[54](◇1):

```
#define ◇1_fRegr(x) MkfRegr(NoPosition,◇1Op,x)
```

This macro is invoked in definition 58.

Construct an fLoad node for[55](◇1):

```
#define ◇1_fLoad(x) MkfLoad(NoPosition,◇1Op,x)
```

This macro is invoked in definition 58.

Construct an fStore node for[56](◇1):

```
#define ◇1_fStore(x,y) MkfStore(NoPosition,◇1Op,x,y)
```

This macro is invoked in definition 58.

Construct an fCmprr node for[57](◇1):

```
#define ◇1_fCmprr(x,y) MkfCmprr(NoPosition,◇1Op,x,y)
```

This macro is invoked in definition 58.

Factory methods[58]:

Construct an fRegr node for[54](‘`absd`’)
Construct an fRegr node for[54](‘`abss`’)
Construct an fRegrr node for[53](‘`addd`’)
Construct an fRegrr node for[53](‘`adds`’)
Construct an fCmprr node for[57](‘`ceqd`’)
Construct an fCmprr node for[57](‘`ceqs`’)
Construct an fCmprr node for[57](‘`cled`’)
Construct an fCmprr node for[57](‘`cles`’)
Construct an fCmprr node for[57](‘`cltd`’)
Construct an fCmprr node for[57](‘`clts`’)
Construct an fRegr node for[54](‘`cvtds`’)
Construct an fRegr node for[54](‘`cvtdw`’)
Construct an fRegr node for[54](‘`cvtsd`’)
Construct an fRegr node for[54](‘`cvtsw`’)
Construct an fRegr node for[54](‘`cvtwd`’)
Construct an fRegr node for[54](‘`cvtws`’)
Construct an fRegrr node for[53](‘`divd`’)
Construct an fRegrr node for[53](‘`divs`’)
Construct an fLoad node for[55](‘`ld`’)
Construct an fLoad node for[55](‘`ls`’)
Construct an fRegrr node for[53](‘`muld`’)
Construct an fRegrr node for[53](‘`muls`’)
Construct an fRegr node for[54](‘`negd`’)
Construct an fRegr node for[54](‘`negs`’)
Construct an fStore node for[56](‘`sd`’)
Construct an fStore node for[56](‘`ss`’)
Construct an fRegrr node for[53](‘`subd`’)
Construct an fRegrr node for[53](‘`subs`’)

This macro is defined in definitions 7, 9, 16, 19, 22, 25, 36, 41, 46, 49, 58, 60, 63, 65, 67, and 69.
This macro is invoked in definition 74.

Tree parser rules[59]:

```
FltReg ::= absdTp(FltReg)              : absd_fRegr;
FltReg ::= abssTp(FltReg)              : abss_fRegr;
FltReg ::= adddTp(FltReg,FltReg)       : addd_fRegrr;
FltReg ::= addsTp(FltReg,FltReg)       : adds_fRegrr;
FltReg ::= cvtdsTp(FltReg)             : cvtds_fRegr;
FltReg ::= cvtdwTp(FltReg)             : cvtdw_fRegr;
FltReg ::= cvtsdTp(FltReg)             : cvtsd_fRegr;
FltReg ::= cvtswTp(FltReg)             : cvtsw_fRegr;
FltReg ::= cvtwdTp(FltReg)             : cvtwd_fRegr;
FltReg ::= cvtwsTp(FltReg)             : cvtws_fRegr;
FltReg ::= divdTp(FltReg,FltReg)       : divd_fRegrr;
FltReg ::= divsTp(FltReg,FltReg)       : divs_fRegrr;
FltReg ::= ldTp(FltMem)                : ld_fLoad;
FltReg ::= lsTp(FltMem)                : ls_fLoad;
FltReg ::= muldTp(FltReg,FltReg)       : muld_fRegrr;
FltReg ::= mulsTp(FltReg,FltReg)       : muls_fRegrr;
FltReg ::= negdTp(FltReg)              : negd_fRegr;
FltReg ::= negsTp(FltReg)              : negs_fRegr;
FltReg ::= sdTp(FltReg,FltMem)         : sd_fStore;
FltReg ::= ssTp(FltReg,FltMem)         : ss_fStore;
FltReg ::= subdTp(FltReg,FltReg)       : subd_fRegrr;
FltReg ::= subsTp(FltReg,FltReg)       : subs_fRegrr;

FltReg ::= absTp(FltReg)               : abss_fRegr;
FltReg ::= addTp(FltReg,FltReg)        : adds_fRegrr;
FltReg ::= divTp(FltReg,FltReg)        : divs_fRegrr;
FltReg ::= FltMem                      : ls_fLoad;
FltReg ::= mulTp(FltReg,FltReg)        : muls_fRegrr;
FltReg ::= negTp(FltReg)               : negs_fRegr;
FltReg ::= stoTp(FltReg,FltMem)        : ss_fStore;
FltReg ::= subTp(FltReg,FltReg)        : subs_fRegrr;
```

This macro is defined in definitions 17, 20, 23, 26, 37, 42, 47, 50, 59, 61, and 70.
This macro is invoked in definition 73.

Floating-point comparison operations set the condition flag of the floating-point coprocessor. That result is converted into a `Branch` result by executing a branch on coprocessor condition flag. It is convenient to lump these two operations together, producing analogs of the integer branch operations.

Factory methods[60]:

```
#define beqs_fRegrr(x,y) bc1t_Branchcf(ceqs_fCmprr(x,y))
#define bges_fRegrr(x,y) bc1f_Branchcf(clts_fCmprr(x,y))
#define bgts_fRegrr(x,y) bc1f_Branchcf(cles_fCmprr(x,y))
#define bles_fRegrr(x,y) bc1t_Branchcf(cles_fCmprr(x,y))
#define blts_fRegrr(x,y) bc1t_Branchcf(clts_fCmprr(x,y))
#define bnes_fRegrr(x,y) bc1f_Branchcf(ceqs_fCmprr(x,y))
```

This macro is defined in definitions 7, 9, 16, 19, 22, 25, 36, 41, 46, 49, 58, 60, 63, 65, 67, and 69.
This macro is invoked in definition 74.

Tree parser rules[61]:

```
Branch ::= beqsTp(FltReg,FltReg)        : beqs_fRegrr;
Branch ::= bgesTp(FltReg,FltReg)        : bges_fRegrr;
Branch ::= bgtsTp(FltReg,FltReg)        : bgts_fRegrr;
Branch ::= blesTp(FltReg,FltReg)        : bles_fRegrr;
Branch ::= bltsTp(FltReg,FltReg)        : blts_fRegrr;
Branch ::= bnesTp(FltReg,FltReg)        : bnes_fRegrr;

BrIfTr ::= eqTp(FltReg,FltReg)          : beqs_fRegrr;
BrIfTr ::= geTp(FltReg,FltReg)          : bges_fRegrr;
BrIfTr ::= gtTp(FltReg,FltReg)          : bgts_fRegrr;
BrIfTr ::= leTp(FltReg,FltReg)          : bles_fRegrr;
BrIfTr ::= ltTp(FltReg,FltReg)          : blts_fRegrr;
BrIfTr ::= neTp(FltReg,FltReg)          : bnes_fRegrr;

BrIfFl ::= eqTp(FltReg,FltReg)          : bnes_fRegrr;
BrIfFl ::= geTp(FltReg,FltReg)          : blts_fRegrr;
BrIfFl ::= gtTp(FltReg,FltReg)          : bles_fRegrr;
BrIfFl ::= leTp(FltReg,FltReg)          : bgts_fRegrr;
BrIfFl ::= ltTp(FltReg,FltReg)          : bges_fRegrr;
BrIfFl ::= neTp(FltReg,FltReg)          : beqs_fRegrr;
```

This macro is defined in definitions 17, 20, 23, 26, 37, 42, 47, 50, 59, 61, and 70.
This macro is invoked in definition 73.

1.3 The Abstract SPIM Program

An abstract SPIM program consists of a sequence of trees, each representing a routine:

The Abstract SPIM Program[62]:

```
RULE SPIM: Target   ::=    Routines END;
RULE Body: Routines LISTOF Routine  END;
```

This macro is invoked in definition 71.

There are two kinds of factory methods used in building this sequence of trees – one kind combines `Item` nodes and the other builds the root node from a given sequence:

Factory methods[63]:

```
#define Spim(x) MkSPIM(NoPosition,MkBody(NoPosition,x))
#define TwoRoutines(x,y) Mk2Body(NoPosition,x,y)
```

This macro is defined in definitions 7, 9, 16, 19, 22, 25, 36, 41, 46, 49, 58, 60, 63, 65, 67, and 69.
This macro is invoked in definition 74.

1.3.1 Routine definitions

Each `Routine` is a sequence of items representing instructions and directives:

Routine definitions[64]:

```
RULE routine: Routine ::=    Label Items END;
RULE items:   Items   LISTOF       Item  END;
```

This macro is invoked in definition 71.

Factory methods[65]:

```
#define define_routine(x,y)  Mkroutine(NoPosition,x,Mkitems(NoPosition,y))
#define NoItems()            Mk0items(NoPosition)
#define OneItem(i)           Mk1items(NoPosition,i)
#define AddOne(s,i1)         Mk2items(NoPosition,s,i1)
#define AddTwo(s,i1,i2)      Mk2items(NoPosition,AddOne(s,i1),i2)
#define AddThree(s,i1,i2,i3) Mk2items(NoPosition,AddTwo(s,i1,i2),i3)
```

This macro is defined in definitions 7, 9, 16, 19, 22, 25, 36, 41, 46, 49, 58, 60, 63, 65, 67, and 69.
This macro is invoked in definition 74.

Clearly one could define additional factory methods for adding more than three `Item` nodes to the list.

1.3.2 Label Definitions

Conditional and iteration constructs in source languages involve conditional execution of parts of a sequence of items. This requires the ability to mark locations in the item sequence.

Label Definitions[66]:

```
RULE LabelDef: Item ::= Label END;
```

This macro is invoked in definition 71.

Use of a `LabelDef` construct marks the location of the next action in the item sequence with the `Label` appearing on the right-hand side. A sequence of `LabelDef` constructs can be used to mark an action with a set of label values. All `Label` values are determined by the translator, and thus these constructs represent explicit execution-order decisions that are *not* being deferred.

Factory methods[67]:

```
#define define_label(x) MkLabelDef(NoPosition,x)
```

This macro is defined in definitions 7, 9, 16, 19, 22, 25, 36, 41, 46, 49, 58, 60, 63, 65, 67, and 69.
This macro is invoked in definition 74.

1.3.3 Computations

Computations always result in a value stored in a register. The register content might be used in further computations, but it might also be ignored (say, after being stored in memory). When a value in a register is ignored, no instructions need be generated. Nevertheless, the fact that the value will be ignored is important for the task of register allocation and therefore must be represented by a node in the target program tree.

Computations[68]:

```
RULE IgnoreInt: Item ::= IntReg END;
RULE IgnoreFlt: Item ::= FltReg END;
```

This macro is invoked in definition 71.

Factory methods[69]:

```
#define ignore_int(x) MkIgnoreInt(NoPosition,x)
#define ignore_float(x) MkIgnoreFlt(NoPosition,x)
```

This macro is defined in definitions 7, 9, 16, 19, 22, 25, 36, 41, 46, 49, 58, 60, 63, 65, 67, and 69.
This macro is invoked in definition 74.

Tree parser rules[70]:

```
Item: NODEPTR;

Item ::= IntReg : ignore_int;
Item ::= FltReg : ignore_float;
```

This macro is defined in definitions 17, 20, 23, 26, 37, 42, 47, 50, 59, 61, and 70.
This macro is invoked in definition 73.

1.4 Specification Files

This specification can be expanded to yield a number of files that together define the target program tree abstraction and its factory methods.

1.4.1 spim.lido

The type-`lido` file describes the set of possible tree nodes.

spim.lido[71]:

Branch and jump instructions[27]
The Abstract SPIM Program[62]
Routine definitions[64]
Computations[68]
Label Definitions[66]

This macro is attached to a product file.

1.4.2 spim.pdl

The type-`pdl` file defines the machine operation codes and their assembly language representations.

spim.pdl[72]:

```
Pname: CharPtr; "Strings.h"
```

Machine operation codes[11]

This macro is attached to a product file.

1.4.3 spim.tp

The type-`tp` file defines the tree parser rules.

spim.tp[73]:

```
"treecon.h"

IntMem, FltMem: NODEPTR;

IntVal ::= ivalTp(int) : copy;
FltVal ::= fvalTp(int) : copy;

IntMem ::= imemTp(NODEPTR) : copy;
FltMem ::= fmemTp(NODEPTR) : copy;
```

Tree parser rules[17]

This macro is attached to a product file.

1.4.4 spim.h

The type-`h` file defines the macros providing the factory methods. Any C program using those factory methods must include this file.

spim.h[74]:

```
#ifndef SPIM_H
#define SPIM_H

#include "treecon.h"
```

```
#define copy(x) (x)
```

Factory methods[7]

```
#endif
```

This macro is attached to a product file.

1.4.5 spim.head

The type-`head` file makes the factory methods available to LIDO computations in any specification supplied to Eli with this one.

spim.head[75]:

```
#include "spim.h"
```

This macro is attached to a product file.

1.4.6 spim.reqmod

The type-`reqmod` file ensures that there will be a definition of the `CharPtr` abstract data type available. By using a type-`reqmod` file instead of a type-`specs` file, the specification allows a user to override the module definitions.

spim.reqmod[76]:

```
$/Tech/Strings.specs
```

This macro is attached to a product file.

2 The SPIM Assembly Language

An abstract target program tree is a description of an assembly language program. Each construct represented by a node in the tree corresponds to a sequence of assembly language instructions. To output those instructions, the compiler must invoke appropriate text generation functions. Those functions are constructed from patterns that specify the required layout. This specification defines appropriate patterns for the SPIM instruction set and expresses the constructs of the abstract target program tree in terms of those patterns.

2.1 Assembler syntax

An assembly code instruction consists of an operation code followed by operands. This section defines patterns that are direct encodings of the instruction formats from the SPIM manual. PTG, the Pattern Text Generator, compiles a text generation routine from each pattern. The name of the generation routine consists of the characters `PTG` followed by the pattern name (e.g. `PTGLabelDef`). Each `$` character in the pattern marks a "hole" where text specified by an argument will be inserted.

2.1.1 Label definition format

Labels are declared by putting them at the beginning of a line followed by a colon. This specification encodes all label definitions as distinct lines, a strategy that is allowed but not required by SPIM:

Label definition format[1]:

```
LabelDef:       $ string ":\n"
```

This macro is invoked in definition 24.

2.1.2 Assembler directives

SPIM supports a subset of the assembler directives provided by the MIPS assembler. Many of the assembler directives have variable-format operand fields, so this specification provides a single pattern taking the directive (as a string) and the result of a text generation function:

Assembler directives[2]:

```
Directive:      "\t" $ string "\t" $ "\n"
```

This macro is invoked in definition 24.

This format allows maximum flexibility: the directive's operand can be constructed using text generation routines of arbitrary complexity. The only drawback is that the second argument of `PTGDirective` *must* be the result of a text-generation function, even when it is a simple string.

2.1.3 Assembly program structure

A complete program consists of a set of segments. Its entry point is specified by the symbol `main`:

Assembly program structure[3]:

```
Spim:
  "\t.globl\tmain\n"
  $
```

This macro is invoked in definition 24.

2.1.4 Memory reference formats

SPIM instructions generally operate on the content of registers and leave their results in registers. Data can be transferred between registers and memory only through the use of load and store instructions. The assembly language addressing modes used in those instructions to reference memory all express the sum of a constant and the contents of a CPU register:

Memory reference formats[4]:

```
memsr:  $ "($" $ int ")"
memsps: $ string "+" $ string
mems:   $ string
memss:  $ string     $ string
```

This macro is invoked in definition 24.

When two string arguments are given, the first must be a symbol and the second must be an integer. Moreover, they must be separated by either `+` or `-`. If the integer is negative, then its sign can play the role of the separator; when the integer is positive, a `+` must be inserted explicitly.

2.1.5 CPU instruction formats

The integer processing unit (the CPU) handles integer operations. Every CPU instruction has one of the following formats:

CPU instruction formats[5]:

```
cpu3:   "\t" $ string "\t$"  $ int ",$"  $ int ",$"   $ int    "\n"
cpu2s:  "\t" $ string "\t$"  $ int ",$"  $ int ","    $ string "\n"
cpu2:   "\t" $ string "\t$"  $ int ",$"  $ int                 "\n"
cpu1ss: "\t" $ string "\t$"  $ int ","   $ string "," $ string "\n"
cpu1s:  "\t" $ string "\t$"  $ int ","                 $ string "\n"
cpu1m:  "\t" $ string "\t$"  $ int ","                 $        "\n"
cpu1:   "\t" $ string "\t$"  $ int                             "\n"
cpu0s:  "\t" $ string "\t"                             $ string "\n"
```

This macro is defined in definitions 5.
This macro is invoked in definition 24.

Each instruction format name specifies that the instruction is a CPU instruction, gives the number of registers it specifies, and the pattern of any additional operands. There are two kinds of additional operands, strings (indicated by `s`) and memory addresses (indicated by `m`). Memory address arguments *must* be the result of a text-generation function.

Register arguments to the text generation functions created from the patterns described in this section must be integer values in the range `0-31`; assembly language register names are not allowed.

2.1.6 Floating point instruction formats

Coprocessor 1 handles floating-point operations. All but one of the formats for coprocessor 1 instructions involve only registers. The exception, `fpu1m`, accepts a memory address and implements the coprocessor load and store instructions:

Floating point instruction formats[6]:

```
fpu3:   "\t" $ string "\t$f" $ int ",$f" $ int ",$f"  $ int    "\n"
fpu2:   "\t" $ string "\t$f" $ int ",$f" $ int                 "\n"
fpu1m:  "\t" $ string "\t$f" $ int ","                 $       "\n"
```

This macro is defined in definitions 6 and 7.
This macro is invoked in definition 24.

Register arguments to the text generation functions created from the patterns described above must be *even* integer values in the range `0-31`. Although there are 32 coprocessor registers, even/odd register pairs are required to hold double-precision floating-point values and instructions manipulating those values must specify the even member of the pair. In order to simplify instruction processing, instructions manipulating single-precision values must adhere to the same restriction.

Finally, there is one instruction format specifying one cpu register and one coprocessor register. Instructions having this format are used to move information in both directions between the cpu and coprocessor:

Floating point instruction formats[7]:

```
regcf:  "\t" $ string "\t$"  $ int ",$f" $ int                 "\n"
```

This macro is defined in definitions 6 and 7.
This macro is invoked in definition 24.

Both register arguments to the text generation function `PTGregcf` must be integer values in the range `0-31`. There is no restriction to even integers for the floating-point register number, which means that any of the 32 coprocessor registers can be specified.

2.2 Register allocation

Both `IntReg` and `FltReg` have `reg` attributes. The value of a `reg` attribute is the number of the register holding the value defined by the subtree rooted in the `IntReg` or `FltReg` node. Values of the `reg` attributes are integers in the range `0-31`, and represent hardware register numbers rather than assembly language register symbols.

The `reg` attribute is assigned in the upper context of the `IntReg` or `FltReg` node:

Register allocation[8]:

```
ATTR reg: int;

SYMBOL Assigned COMPUTE INH.reg=0; END;
SYMBOL IntReg INHERITS Assigned END;
SYMBOL FltReg INHERITS Assigned END;
```

This macro is invoked in definition 26.

Register allocation is omitted from this specification. The computations given above are included only to make the specification complete as far as attribution is concerned. They must be overridden in either symbol computations for `IntReg` or `FltReg`, or in rule computations having either of these symbols on the right-hand side. Even if no register allocation is supplied, however, this specification can be used to verify correct instruction selection.

2.3 Encoding the SPIM abstraction

The complete program is encoded by obtaining the code implementing the top-level `Item` nodes. Each abstract target program tree node has a `Code` attribute whose value is the complete assembly language sequence corresponding to that node:

Encoding the SPIM abstraction[9]:

```
ATTR Code: PTGNode;

SYMBOL Target COMPUTE
  SYNT.Code=
    PTGSpim(
      CONSTITUENTS Item.Code SHIELD Item
        WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull));
END;
```

This macro is defined in definitions 9, 10, and 11.
This macro is invoked in definition 26.

In general, the complete assembly language sequence corresponding to a node is constructed by combining the sequences from the children of the node in left-to-right order and then appending the instruction for the node itself:

Encoding the SPIM abstraction[10]:

```
ATTR Instr: PTGNode;

SYMBOL Coded COMPUTE
  SYNT.Instr=PTGNULL;
  SYNT.Code=
    PTGSeq(
      CONSTITUENTS Coded.Code SHIELD Coded
        WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull),
      THIS.Instr);
END;

SYMBOL Item   INHERITS Coded END;
SYMBOL Branch INHERITS Coded END;
SYMBOL IntReg INHERITS Coded END;
SYMBOL FltReg INHERITS Coded END;
SYMBOL CndFlg INHERITS Coded END;
```

This macro is defined in definitions 9, 10, and 11.
This macro is invoked in definition 26.

These computations for the `Instr` and `Code` attributes of a node can be overridden either in symbol computations for `Item`, `Branch`, `IntReg` or `FltReg`, or computations for rules having any of these symbols on the left-hand side. The label definition abstraction, for example, overrides the default empty instruction, but inherits the computation of the complete assembly language sequence:

Encoding the SPIM abstraction[11]:

```
RULE LabelDef: Item ::= Label
COMPUTE
  Item.Instr=PTGLabelDef(StringTable(GenSym(Label)));
END;
```

This macro is defined in definitions 9, 10, and 11.
This macro is invoked in definition 26.

The `Sym` property of the `Label` specifies the string representing that label. If the `Sym` property has not been set when the string representation is first needed, a representation is generated and becomes the value of the `Sym` property. The generation and delivery of the string representation is handled by an access function for the `Sym` property:

Label string generation[12]:

```
Sym: int [Gen];

int Gen(DefTableKey key)
{ if (key == NoKey) return 0;
  if (!ACCESS) VALUE = GenerateName("L");
  return VALUE;
}
```

This macro is invoked in definition 25.

`GenerateName` is exported by a library module; it is guaranteed to produce symbols that differ from *all* others, both generated and found in the source text:

Instantiate required modules[13]:

```
$/Tech/MakeName.gnrc :inst
```

This macro is defined in definitions 13 and 19.
This macro is invoked in definition 27.

2.3.1 Addressing mode abstractions

Addressing mode abstractions are encoded separately, and the result of the encoding is used in constructing load and store instructions.

Addressing mode abstractions[14]:

```
RULE Indexed: Memory ::= Address IntReg
COMPUTE
  Memory.Code=PTGmemsr(Address.Code,IntReg.reg);
END;

RULE Based:   Memory ::= Address Register
COMPUTE
  Memory.Code=PTGmemsr(Address.Code,Register);
END;
```

```
RULE RelAddr: Address ::= Label Denoter
COMPUTE
  Address.Code=
    IF(EQ(INDEX(StringTable(Denoter),0),'-'),
      PTGmemss(StringTable(GenSym(Label)),StringTable(Denoter)),
      PTGmemsps(StringTable(GenSym(Label)),StringTable(Denoter)));
END;

RULE SymAddr: Address ::= Label
COMPUTE
  Address.Code=PTGmems(StringTable(GenSym(Label)));
END;

RULE AbsAddr: Address ::= Denoter
COMPUTE
  Address.Code=PTGmems(StringTable(Denoter));
END;
```

This macro is invoked in definition 26.

2.3.2 Arithmetic and logical abstractions

All of the arithmetic and logical instructions operate on CPU registers and/or integer values:

Arithmetic and logical abstractions[15]:

```
RULE iRegrr: IntReg ::= Mop IntReg IntReg
COMPUTE
  IntReg[1].Instr=
    PTGcpu3(GetPname(Mop,"BAD"),IntReg[1].reg,IntReg[2].reg,IntReg[3].reg);
END;

RULE iRegri: IntReg ::= Mop IntReg Denoter
COMPUTE
  IntReg[1].Instr=
    PTGcpu2s(
      GetPname(Mop,"BAD"),
      IntReg[1].reg,
      IntReg[2].reg,
      StringTable(Denoter));
END;

RULE iRegr: IntReg ::= Mop IntReg
COMPUTE
  IntReg[1].Instr=PTGcpu2(GetPname(Mop,"BAD"),IntReg[1].reg,IntReg[2].reg);
END;

RULE iRegi: IntReg ::= Mop Denoter
COMPUTE
  IntReg.Instr=
```

```
    PTGcpu1s(GetPname(Mop,"BAD"),IntReg[1].reg,StringTable(Denoter));
END;
```

This macro is invoked in definition 26.

2.3.3 Constant-manipulating abstractions

The abstraction for the constant-manipulating instructions is `iRegi`.

Floating-point constants cannot appear in instructions, but must be stored in the data segment.

The floating-point constant pool[16]:

```
ATTR Key: DefTableKey;
ATTR lds: PTGNode;

RULE LoadFloat: FltReg ::= Denoter
COMPUTE
  .Key=DefineIdn(INCLUDING FpConstAnyScope.FpConstEnv,Denoter);
  .lds=
    PTGfpu1m(
      GetPname(lsOp,"BAD"),
      FltReg.reg,
      PTGmems(StringTable(GenSym(.Key))));
  FltReg.Instr=
    IF(Undeclared(.Key),
      PTGSeq(
        .lds,
        PTGFloatConstant(StringTable(GenSym(.Key)),StringTable(Denoter))),
      .lds);
END;
```

This macro is invoked in definition 26.

Floating point constant value[17]:

```
FloatConstant:
  "\t.data\n"
  $1 string "\t.float\t" $2 string "\n"
  "\t.text\n"
```

This macro is invoked in definition 24.

The access function `Undeclared` yields the value `1` on some arbitrary invocation with a specific key, and yields `0` on all other invocations. Note that there is no relationship between the position in the tree and the value returned:

Guarantee only a single declaration[18]:

```
lared: int [Undec];

int Undec(DefTableKey key)
{ if (key == NoKey) return 0;
  return !ACCESS;
}
```

This macro is invoked in definition 25.

A name analysis module must be instantiated to support this computation:

Instantiate required modules[19]:

```
$/Name/AlgScope.gnrc +instance=FpConst :inst
```

This macro is defined in definitions 13 and 19.
This macro is invoked in definition 27.

2.3.4 Comparison abstractions

The abstractions for the comparison instructions are `iRegrr` and `iRegri`.

2.3.5 Branch and jump abstractions

The `JumpCond` abstraction has no instruction of its own, but it sets the `label` attribute of its child.

Branch and jump abstractions[20]:

```
ATTR label: DefTableKey;

RULE JumpCond: Item ::= Branch Label
COMPUTE
  Branch.label=Label;
END;

RULE Branchn: Branch ::= Mop
COMPUTE
  Branch.Instr=
    PTGcpu0s(GetPname(Mop,"BAD"),StringTable(GenSym(Branch.label)));
END;

RULE Branchcf: Branch ::= Mop CndFlg
COMPUTE
  Branch.Instr=
    PTGcpu0s(GetPname(Mop,"BAD"),StringTable(GenSym(Branch.label)));
END;

RULE Branchrr: Branch ::= Mop IntReg IntReg
COMPUTE
  Branch.Instr=
    PTGcpu2s(
      GetPname(Mop,"BAD"),
      IntReg[1].reg,
      IntReg[2].reg,
      StringTable(GenSym(Branch.label)));
END;

RULE Branchri: Branch ::= Mop IntReg Denoter
```

```
COMPUTE
  Branch.Instr=
    PTGcpu1ss(
      GetPname(Mop,"BAD"),
      IntReg.reg,
      StringTable(Denoter),
      StringTable(GenSym(Branch.label)));
END;

RULE Branchr: Branch ::= Mop IntReg
COMPUTE
  Branch.Instr=
    PTGcpu1s(GetPname(Mop,"BAD"),IntReg.reg,StringTable(GenSym(Branch.label)));
END;

RULE Jump: Item ::= Mop Label
COMPUTE
  Item.Instr=PTGcpu0s(GetPname(Mop,"BAD"),StringTable(GenSym(Label)));
END;

RULE Jumpr: Item ::= Mop IntReg
COMPUTE
  Item.Instr=PTGcpu1(GetPname(Mop,"BAD"),IntReg.reg);
END;
```

This macro is invoked in definition 26.

2.3.6 Load abstractions

Load abstractions[21]:

```
RULE iLoad: IntReg ::= Mop Memory
COMPUTE
  IntReg.Instr=PTGcpu1m(GetPname(Mop,"BAD"),IntReg.reg,Memory.Code);
END;
```

This macro is invoked in definition 26.

2.3.7 Store abstractions

Store abstractions[22]:

```
RULE iStore: IntReg ::= Mop IntReg Memory
COMPUTE
  IntReg[1].Instr=PTGcpu1m(GetPname(Mop,"BAD"),IntReg[2].reg,Memory.Code);
END;
```

This macro is invoked in definition 26.

2.3.8 Floating point abstractions

Floating point abstractions[23]:

```
RULE fRegrr: FltReg ::= Mop FltReg FltReg
COMPUTE
  FltReg[1].Instr=
    PTGfpu3(GetPname(Mop,"BAD"),FltReg[1].reg,FltReg[2].reg,FltReg[3].reg);
END;

RULE fRegr: FltReg ::= Mop FltReg
COMPUTE
  FltReg[1].Instr=PTGfpu2(GetPname(Mop,"BAD"),FltReg[1].reg,FltReg[2].reg);
END;

RULE fLoad: FltReg ::= Mop Memory
COMPUTE
  FltReg.Instr=PTGfpu1m(GetPname(Mop,"BAD"),FltReg.reg,Memory.Code);
END;

RULE fStore: FltReg ::= Mop FltReg Memory
COMPUTE
  FltReg[1].Instr=PTGfpu1m(GetPname(Mop,"BAD"),FltReg[2].reg,Memory.Code);
END;

RULE fCmprr: CndFlg ::= Mop FltReg FltReg
COMPUTE
  CndFlg.Instr=PTGfpu2(GetPname(Mop,"BAD"),FltReg[1].reg,FltReg[2].reg);
END;
```

This macro is invoked in definition 26.

2.4 Specification files

spimasm.ptg[24]:

Label definition format[1]
Assembler directives[2]
Assembly program structure[3]
Memory reference formats[4]
CPU instruction formats[5]
Floating point instruction formats[6]
Floating point constant value[17]

This macro is attached to a product file.

spimasm.pdl[25]:

Guarantee only a single declaration[18]
Label string generation[12]

This macro is attached to a product file.

spimasm.lido[26]:

Register allocation[8]
Encoding the SPIM abstraction[9]
Addressing mode abstractions[14]
Arithmetic and logical abstractions[15]
Branch and jump abstractions[20]
Load abstractions[21]
Store abstractions[22]
Floating point abstractions[23]
The floating-point constant pool[16]

This macro is attached to a product file.

spimasm.specs[27]:

Instantiate required modules[13]
`$/Output/PtgCommon.fw`

This macro is attached to a product file.

3 Code Generation for SPIM

Code generation produces a sequence of target machine instructions to implement the algorithm described by a target program tree.

Three major decisions must be made during code generation: The order in which to evaluate components of target expressions, where to hold the intermediate values, and which instructions to use to implement target operations.

3.1 Operand evaluation order

The number of intermediate values that exist simultaneously during the evaluation of a target expression may depend upon the order in which the components of that expression are evaluated: If an expression has two subexpressions as operands, and if the subexpression requiring the most simultaneous intermediate values is evaluated first, then the total number of simultaneous intermediate values is minimized.

The maximum number of intermediate values that must exist simultaneously during evaluation of an expression is often called the Sethi-Ullman number of the expression. In the following computation, the Sethi-Ullman number of the expression is represented by the `su` attribute:

Determine the number of simultaneous values[1]:

```
ATTR su, fsu: int;

SYMBOL Datum COMPUTE
  SYNT.su =CONSTITUENTS Datum.su  SHIELD Datum WITH (int, Max, IDENTICAL, ONE);
  SYNT.fsu=CONSTITUENTS Datum.fsu SHIELD Datum WITH (int, Max, IDENTICAL, ONE);
END;

SYMBOL Branch INHERITS Datum END;
SYMBOL IntReg INHERITS Datum END;
SYMBOL FltReg INHERITS Datum END;
SYMBOL CndFlg INHERITS Datum END;
```

This macro is invoked in definition 10.

This computation says that the Sethi-Ullman number of an expression is the maximum of the Sethi-Ullman numbers of that expression's children. If the expression is a leaf, then its Sethi-Ullman number is 1.

The default computation shown above works *only* when evaluation of an expression does not require that the values of any of its children be retained while other children are evaluated. Computational operations like addition and subtraction do not fit this model: one operand must be computed and retained while the other is computed. Determination of the Sethi-Ullman number is more complex in that case:

Sethi-Ullman numbering[2](◇3):

```
◇1=
  IF(GT(◇2,◇3),◇2,
  IF(LT(◇2,◇3),◇3,
  ADD(◇2,1)));
```

This macro is invoked in definitions 8 and 9.

Here the three parameters are the Sethi-Ullman numbers of the result expression and the left and right operands respectively. When the Sethi-Ullman numbers of the operands differ, the Sethi-Ullman number of the result is the larger of the two. When the Sethi-Ullman numbers of the operands are the same, the Sethi-Ullman number of the result is one larger because one of the operands must exist simultaneously with the computation of the other.

Once the Sethi-Ullman numbers of the expressions are available, one can determine the order in which to compute the children of a binary expression:

Evaluate left first[3](◇2):

```
GT(◇1,◇2)
```

This macro is invoked in definitions 6 and 7.

Evaluate right first[4](◇2):

```
LE(◇1,◇2)
```

This macro is invoked in definition 6.

Here the two parameters are the Sethi-Ullman numbers of the left and right operands respectively. If the Sethi-Ullman number of the left operand is greater, then the left operand must be evaluated first. Otherwise either the right operand must be evaluated first or the order is irrelevant. When the order is irrelevant, it is better on some machines to evaluate the right operand first.

3.2 Register allocation

Register allocation is based on sets of registers. It must be possible to request an element of a set, and to create a new set from an old one by deleting an element. (Note that the operation requesting an element does *not* remove any elements from the set.)

On the MIPS, there are no restrictions on the registers used as operands of any instruction. Therefore we can consider the register set to be a compact sequence of registers, and represent it by the lowest register number in the sequence. A new set is created from an old one simply by incrementing the lowest register number.

If we choose to use registers 4 through 15 to store intermediate values in target expressions, then the set of registers available for each top-level expression is characterized by the number 4 and the maximum number of intermediate values that can be in registers simultaneously is 12. The `num` attribute of a target expression both characterizes the set of registers available for that expression and specifies the register in which the result is held:

Register allocation[5]:

```
#define MaxReg 12
ATTR num,fnum: int;

SYMBOL Target COMPUTE
  SYNT.num=0;
  SYNT.fnum=0;
END;

SYMBOL Datum COMPUTE
  INH.num=INCLUDING (Datum.num, Target.num);
  INH.fnum=INCLUDING (Datum.fnum, Target.fnum);
```

```
END;

SYMBOL IntReg INHERITS Datum COMPUTE
  INH.reg=ADD(THIS.num,4);
END;

SYMBOL FltReg INHERITS Datum COMPUTE
  INH.reg=MOD(ADD(MUL(THIS.fnum,2),12),16);
END;
```

This macro is invoked in definition 10.

This computation says that target expressions inherit their result register and the set of available registers from their parent. Again, the default computation shown above works *only* when evaluation of an expression does not require that the values of any of its children be retained while other children are evaluated. Computational expressions require a more complex computation:

Registers for binary operators[6](◇4):

```
◇2.◇4num=
  IF(OR(GT(◇1.◇4su,MaxReg),Evaluate left first[3]('◇2.◇4su','◇3.◇4su')),
    ◇1.◇4num,
    ADD(◇1.◇4num,1));
◇3.◇4num=
  IF(OR(GT(◇1.◇4su,MaxReg),Evaluate right first[4]('◇2.◇4su','◇3.◇4su')),
    ◇1.◇4num,
    ADD(◇1.◇4num,1));
```

This macro is invoked in definitions 8 and 9.

Here the three parameters are the result expression and the left and right operands respectively. If the Sethi-Ullman number of the result indicates that there are more intermediate results than there are registers, then one of the operands must be stored in memory while the other is computed. That means that *all* of the registers are available for the computations in *both* subtrees, so the set is passed unchanged to both subtrees. If the Sethi-Ullman number of the result indicates that enough registers are available for the intermediate results, then the set is passed unchanged to the operand being evaluated first. The result register for that operand is deleted from the set passed to the other subtree.

Left-to-right order is, of course, not always correct for the operands of a binary expression. There the determination depends on the Sethi-Ullman numbers:

Composition for binary operators[7](◇4):

```
◇1.Code=
  PTGSeq(
    IF(Evaluate left first[3]('◇2.◇4su','◇3.◇4su'),
      PTGSeq(◇2.Code,◇3.Code),
      PTGSeq(◇3.Code,◇2.Code)),
    ◇1.Instr);
```

This macro is invoked in definitions 8 and 9.

Computational operators with two register operands must have their children composed according to the Sethi-Ullman numbers:

Code generation for computational operators[8]:

```
RULE iRegrr: IntReg ::= Mop IntReg IntReg
COMPUTE
  Sethi-Ullman numbering[2]('IntReg[1].su','IntReg[2].su','IntReg[3].su')
  Registers for binary operators[6]('IntReg[1]','IntReg[2]','IntReg[3]','')
  Composition for binary operators[7]('IntReg[1]','IntReg[2]','IntReg[3]','')
END;

RULE fRegrr: FltReg ::= Mop FltReg FltReg
COMPUTE
  Sethi-Ullman numbering[2]('FltReg[1].fsu','FltReg[2].fsu','FltReg[3].fsu')
  Registers for binary operators[6]('FltReg[1]','FltReg[2]','FltReg[3]','f')
  Composition for binary operators[7]('FltReg[1]','FltReg[2]','FltReg[3]','f')
END;
```

This macro is invoked in definition 10.

Conditional jump operations with two register operands must be processed in the same way as normal computation operations:

Code generation for conditionals[9]:

```
RULE Branchrr: Branch ::= Mop IntReg IntReg
COMPUTE
  Sethi-Ullman numbering[2]('Branch.su','IntReg[1].su','IntReg[2].su')
  Registers for binary operators[6]('Branch','IntReg[1]','IntReg[2]','')
  Composition for binary operators[7]('Branch','IntReg[1]','IntReg[2]','')
END;

RULE fCmprr: CndFlg ::= Mop FltReg FltReg
COMPUTE
  Sethi-Ullman numbering[2]('CndFlg.fsu','FltReg[1].fsu','FltReg[2].fsu')
  Registers for binary operators[6]('CndFlg','FltReg[1]','FltReg[2]','f')
  Composition for binary operators[7]('CndFlg','FltReg[1]','FltReg[2]','f')
END;
```

This macro is invoked in definition 10.

3.3 Specification files

SpimCode.lido[10]:

Register allocation[5]
Determine the number of simultaneous values[1]
Code generation for computational operators[8]
Code generation for conditionals[9]

This macro is attached to a product file.

SpimCode.h[11]:

```
#ifndef SPIMCODE_H
#define SPIMCODE_H

#include "eliproto.h"

extern int Max ELI_ARG((int, int));

#endif
```

This macro is attached to a product file.

SpimCode.c[12]:

```
#include "SpimCode.h"

int
Max(a,b) int a,b;
{ return (a > b ? a : b); }
```

This macro is attached to a product file.

SpimCode.HEAD.phi[13]:

```
#include "SpimCode.h"
```

This macro is attached to a product file.

A Print the Abstract SPIM Tree

A printer for the abstract SPIM tree can be generated by Eli from the specification of the SPIM abstract syntax and tree additional pieces of information.

The abstract syntax does not specify the form of a terminal symbol. A printing function for each kind of terminal symbol is assumed to be available, and these must be defined:

PrintSpimTree.HEAD.phi[1]:

```
Idem_Register:  $ int
#define PTGLabel(x)     PTGId(GenSym(x))
#define PTGMop(x)       PTGAsIs(GetPname(x,"UNKNOWN"))
```

This macro is attached to a product file.

`PTGNum`, `PTGId`, and `PTGAsIs` are all provided by the `PtgCommon` library module, which must therefore be instantiated:

PrintSpimTree.specs[2]:

```
$/Output/PtgCommon.fw
```

This macro is attached to a product file.

Using a block output function with the IdemPtg attribute of any nonterminal symbol in the tree as argument will cause the subtree rooted by that node to be displayed. The following symbol computation displays the entire tree if the attribute `DoSPIMPrint` is nonzero:

PrintSpimTree.lido[3]:

```
CLASS SYMBOL ControlSPIMPrint: DoSPIMPrint: int;

CLASS SYMBOL ControlSPIMPrint COMPUTE
  SYNT.DoSPIMPrint=1;
END;

SYMBOL Target INHERITS ControlSPIMPrint COMPUTE
  IF(THIS.DoSPIMPrint,BP_Out(THIS.IdemPtg));
END;
```

This macro is attached to a product file.

The computation of `DoSPIMPrint` can be overridden in either a symbol or a rule computation for `Target` to suppress printing. Often a program using this specification will have a command line option that selects a "test version" by setting a Boolean variable. That variable can be assigned to `DoSPIMPrint`.