SICStus Prolog JIT White Paper

Mats Carlsson et al.

RISE Research Institutes of Sweden AB
PO Box 1263
SE-164 29 Kista, Sweden

Release 4.6.0
April 2020

RISE Research Institutes of Sweden AB
sicstus-request@sics.se https://sicstus.sics.se/

Table of Contents

1 Introduction

The SICStus Prolog Just-In-Time Compiler (*SPJIT*) is currently operational on the `x86` and `x86_64` architectures under Windows, Mac OS X and Linux, and is being ported to the `PPC64` (Power8) architecture under Linux. SPJIT works entirely in-memory; generated native code is never written to any files. The unit of compilation is a Prolog predicate. The compilation is performed in two steps: (a) from WAM (Warren Abstract Machine) to IR (intermediate representation), (b) from IR to native code. SPJIT thus consists of three modules:

1. A WAM to IR translator, written in Prolog. Goals of this module include to be target independent and to run in time linear in the size of the input.
2. An IR to native code translator, written in C. This module obviously needs to be adapted to the specific target. One source code version covers `x86` and `x86_64`, whereas another one is being developed for `PPC64`.
3. A runtime system to support the native code, written in assembly language. It also contains entry points when emulated code wants to call, continue to, or fail back to native code. Conversely, it contains return points when native code wants to call, continue to, or fail back to non-native code, or for all kinds of exception handling. This module also calls other parts of the runtime system as needed. This module too obviously needs to be adapted to the specific target. One source code version covers `x86` and `x86_64`, whereas another one is being developed for `PPC64`.

2 Intermediate Representation

The intermediate representation can be thought of as a universal assembly language, or at least a language general enough to execute the Prolog virtual machine if assisted by a runtime system. In this chapter, we list its instructions as well as their expansion into native instructions.

[**PERM:** It would be nice with some description of the abstract CPU. In particular the condition codes and how they are maintained.]

2.1 IR Instruction Set

```
Insn      ::= move(Src,Dest)
            | cmove(Cond,Src,Dest)
            | cmps(Dest,Src)
            | cmpu(Dest,Src)
            | test(Dest,Src)
            | jump(Target)
            | call(Target)
            | ccall(Cond,Target)
            | branch(Cond,Target)
            | add(Src,Src,Dest)
            | addo(Src,Src,Dest)
            | sub(Src,Src,Dest)
            | subo(Src,Src,Dest)
            | mulo(Src,Src,Dest)
            | sh(Src,Src,Dest)
            | and(Src,Src,Dest)
            | or(Src,Src,Dest)
            | xor(Src,Src,Dest)
            | int2cp(Src,Dest)
            | cp2int(Src,Dest)
            | init(Dest,Dest)
            | pop
            | context(Target)
            | half(Constant)
            | word(Constant)
            | label(_)
            | align(0 | 1 | 2 | 3 | 4)
            | try_chain(list of (label(_)-Int),Int)
            | switch(list of (Immediate-Target),Target)

Cond      ::= gu | geu | lu | leu | g | ge | l | le | e | ne | no | o

Target    ::= kernel(Atom)
            | label(_)
            | native_entry(Atom:Atom/Int)
```

```
                | native_entry(Int)
                | cp(Offset)
                | Int

Src             ::= Immediate | Reg | Mem

Dest            ::= Reg | Mem

Immediate       ::= functor(Atom/Int)
                | constant(Atomic)
                | nil
                | label(Cont)
                | Offset

Reg             ::= val | h | s | ac0 | ac1 | ab | a | e | cp | b
                | gpr(Int) // general purpose register
                | fpr(Int) // floating-point register
                | arg0
                | arg1
                | arg2
                | w_insn
                | w_heap_warn_soft
                | w_next_node
                | w_numstack_end
                | w_stack_start
                | w_stack_warn
                | w_fli_stack_start

Mem             ::= x(Int) | x(Int,_) | y(Int) | y(Int,_)
                | a(Offset)
                | e(Offset)
                | cp(Offset)
                | h(Offset)
                | s(Offset)
                | val(Offset)

Offset          ::= half(Constant)
                | word(Constant)
                | Constant

Constant        ::= native_op
                | kontinue
                | itoy(Int)
                | Int
                | native_entry(Atom:Atom/Int)
                | native_entry(Int)
```

2.2 Targets

`kernel(`*Atom*`)`
: **FIXME: Document**

`label(_)`
: **FIXME: Document**

`native_entry(`*Atom*`:`*Atom*`/`*Int*`)`
: **FIXME: Document**

`native_entry(`*Int*`)`
: **FIXME: Document**

`cp(`*Int*`)`
: **FIXME: Document**

2.3 Offsets

`half(`*Constant*`)`
: Denotes *Constant* multiplied by the number of bytes per half machine word.

`word(`*Constant*`)`
: Denotes *Constant* multiplied by the number of bytes per machine word.

Constant
: Denotes *Constant*.

2.4 Constants

`native_op`
: Denotes the value of the C expression `Wmode(NATIVE_OP)`.

Int
: Denotes *Int*.

`itoy(`*Constant*`)`
: Denotes *Constant* added by two and multiplied by the number of bytes per machine word.

2.5 Instructions

In this report, the term *condition codes* denotes conditions used by conditional branches, including the overflow condition.

On the x86/x86_64 architectures, operations such as `add` set overflow set iff the signed add yields an arithmetic overflow, and clear it otherwise. To achieve the same on PPC64, one must use the technique:

```
<<clear XER>>
addo. Dest,Src1,Src2
```

which first clears the XER register (see below), and `addo.`, in case of an overflow, sets the SO flag of the XER. In either case, the overflow condition is set to the resulting SO flag, reflecting the outcome of the operation. The SO flag can then be used for conditional branching and the like.

Clearing (the SO-bit of) the XER register can be achieved in many ways. We will clear the entire XER register, using the sequence:

```
li 0,0
mtxer 0
```

which first clears R0 and then moves that into the XER register.[1]

Static branch prediction has not been exploited in this report, but should.

The following table shows the correspondence between IR condition codes and conditional branch instructions.

IR	**x86/x86_64**	**PPC64**
gu	ja	bgt
geu	jae	bge
lu	jb	blt
leu	jbe	ble
g	jg	bgt
ge	jge	bge
l	jl	blt
le	jle	ble
e	je	beq
ne	jne	bne
o	jo	bso
no	jno	bns

We now list each IR instructions with its purpose and back-end specific translation.

2.5.1 move(*Src*,*Dest*)

Purpose To copy the value of source *Src* into destination *Dest*.

Condition Codes
Undefined.

`x86`
`x86_64` If the operands are identical, then

```
/* nothing */
```

Else if *Src* is the constant 0 and *Dest* is a register,

```
xor Dest,Dest
```

Else, for `x86_64`, if *Src* is a local label and *Dest* is a register, then

```
lea OFFSET(%rip),Dest
```

Else if *Src* is a floating-point register and *Dest* is in memory, then

```
// if x86
fstpl Dest
```

[1] The `mcrxr 0` instruction would be shorter, but it is not available on server class Power CPUs.

// else if x86_64
```
movsd Src,Dest
```
Else if *Src* is in memory and *Dest* is a floating-point register, then

// if x86
```
fldl Src
```
// else if x86_64
```
movsd Src,Dest
```
Else if one operand is a register and the other one is a register or in memory, then
```
mov Src,Dest
```
Else if both operands are in memory, then
```
mov Src,%rax
mov %rax,Dest
```
Else if *Src* is a 32-bit signed integer, then
```
mov $Src,Dest
```
Else if *Dest* is a register, then
```
movabs $Src,Dest
```
Else let `r` be `%rdx` if *Dest* uses `%rax` and `%rax` otherwise, and
```
movabs $Src,r
mov r,Dest
```

`PPC64` [**PERM:** Note: `std` and `ld` treat base register R0 as zero, so this must be forbidden here.]

If *Src* is a floating-point register and *Dest* is in memory, then
```
stfd Src,Dest
```
Else if *Src* is in memory and *Dest* is a floating-point register, then
```
lfd Dest,Src
```
Else if *Src* is in a register and *Dest* is in memory, then
```
std Src,Dest
```
Else if *Src* is in memory and *Dest* is in a register, then
```
ld Dest,Src
```
Else if *Dest* is in memory, then reduce to [**PERM:** FIXME: `arg0..arg2` must be preserved, use something else.]
```
move(Src,arg1)
std arg1,Dest
```
Else if *Src* is a register, then
```
mr Dest,Src
```
Else if *Src* is a signed 16-bit integer *SI*, then
```
li Dest,SI
```
Else if *Src* equals *(HI<<16)+LO*, where *HI* is a signed 16-bit integer and *LO* is an unsigned 16-bit integer, then
```
lis Dest,HI
```

```
ori Dest,Dest,LO // omit if LO = 0
```

Else if *Src* is a local label at offset *OFF* from [**PERM:** This could (and naturally will) be done for any (32bit-aligned) immediate that happens to have the value `toc+OFF`, with OFF a signed, multiple-of-4, 16-bit integer.] the TOC, then reduce to

[**PERM:** This may clobber `arg0`. Can *Dest* be `arg0`?].

```
add(toc,OFF,Dest)
```

Else, *Src* must be preallocated at offset *OFF* in the TOC, and [**PERM:** Discuss TOC allocation and toc-register handling, somewhere.]

- if *OFF* is a signed 16-bit integer, then

  ```
  ld Dest,OFF(toc)
  ```

- if *OFF* equals *(HI<<16)+LO*, where *HI* is a signed 16-bit integer and *LO* is an unsigned 16-bit integer and *LO* >= *0x8000*, then [**PERM:** Can do better if HI+1 = 0?]

  [**PERM:** `NOTE: pretty sure this is wrong i HI is 0x7FFF, i.e. HI+1 over`

  ```
  addis arg5,toc,HI+1 // Dest can be r0
  ld Dest,LO(arg5)
  ```

- if *OFF* equals *(HI<<16)+LO*, where *HI* is a signed 16-bit integer and *LO* is an unsigned 16-bit integer and *LO* < *0x8000*, then

  ```
  addis arg5,toc,HI // Dest can be r0
  ld Dest,LO(arg5)
  ```

2.5.2 cmps(*Dest*,*Src*)

Purpose To compare *Dest* and *Src* as signed values. *Dest* must be a general purpose register or in memory.

Condition Codes
Overflow is undefined, the others are set.

`x86`
`x86_64` If both operands are in memory, then reduce to

```
move(Src,val)
cmps(Dest,val)
```

Else if *Src* is an immediate and *Dest* is of the form `cp(0)`, then

```
cmpw $Src,(%rcx)
```

Else if one operand is a register and the other one is a register or in memory, then

```
cmp Src,Dest
```

Else if *Src* is a 32-bit signed integer, then

```
cmp $Src,Dest
```

Else, for `x86_64`

```
movabs $Src,%r11
cmp %r11,Dest
```

`PPC64` If *Dest* is of the form `cp(0)`, then reduce to [**PERM:** FIXME: `arg0..arg2` must be preserved, use something else.]

```
lwz arg0,Dest
cmps(arg0,Src)
```

Else if *Dest* is in memory, then reduce to [**PERM:** FIXME: `arg0..arg2` must be preserved, use something else.]

```
ld arg0,Dest
cmps(arg0,Src)
```

Else if *Src* is in memory, then reduce to [**PERM:** FIXME: `arg0..arg2` must be preserved, use something else.]

```
ld arg1,Src
cmps(Dest,arg1)
```

Else if *Src* is a register, then

```
cmpd Dest,Src
```

Else if *Src* is a signed 16-bit integer *SI*, then

```
cmpdi Dest,SI
```

Else, reduce to [**PERM:** FIXME: `arg0..arg2` must be preserved, use something else.]

```
move(Src,arg1)
cmpd Dest,arg1
```

2.5.3 cmpu(*Dest*,*Src*)

Purpose To compare *Dest* and *Src* as unsigned values. *Dest* must be a general purpose register or in memory.

Condition Codes
Overflow is undefined, the others are set.

`x86`
`x86_64` If both operands are in memory, then reduce to

```
move(Src,val)
cmpu(Dest,val)
```

Else if *Src* is an immediate and *Dest* is of the form `cp(0)`, then

```
cmpw $Src,(%rcx)
```

Else if one operand is a register and the other one is a register or in memory, then

```
cmp Src,Dest
```

Else if *Src* is a 32-bit signed integer, then

```
cmp $Src,Dest
```

Else, for `x86_64`

```
movabs $Src,%r11
cmp %r11,Dest
```

PPC64 If *Dest* is of the form `cp(0)`, then reduce to [**PERM:** FIXME: `arg0..arg2` must be preserved, use something else.]

```
lwz arg0,Dest
cmpu(arg0,Src)
```

Else if *Dest* is in memory, then reduce to [**PERM:** FIXME: `arg0..arg2` must be preserved, use something else.]

```
ld arg0,Dest
cmpu(arg0,Src)
```

Else if *Src* is in memory, then reduce to [**PERM:** FIXME: `arg0..arg2` must be preserved, use something else.]

```
ld arg1,Src
cmpu(Dest,arg1)
```

Else if *Src* is a register, then

```
cmpld Dest,Src
```

Else if *Src* is an unsigned 16-bit integer *UI*, then

```
cmpldi Dest,UI
```

Else, reduce to [**PERM:** FIXME: `arg0..arg2` must be preserved, use something else.]

```
move(Src,arg1)
cmpld Dest,arg1
```

2.5.4 test(*Dest*,*Src*)

Purpose Compute `(Dest /\ Src)`. *Src* must be an immediate or `ac1`.

Condition Codes
Set `e` if the result is zero, and `ne` otherwise. Other condition codes are undefined.

x86
x86_64 If *Dest* is a register and *Src* is an 8-bit unsigned integer, then

```
testb $Src,Dest
```

Else if *Dest* translates to a memory operand `r(OFFSET)` and *Src* can be obtained by shifting an 8-bit unsigned integer *c* left by *8*n* bits, then

```
testb $c,(OFFSET+n)(r)
```

Else [**PERM:** This is incorrect if *Src* is `ac1` (i.e. in memory)]

```
test $Src,Dest
```

PPC64 If *Dest* is in memory, then reduce to [**PERM:** FIXME: `arg0..arg2` must be preserved, use something else.]

```
ld arg0,Dest
test(arg0,Src)
```

Else if *Src* is a register (i.e. ac1), then [**PERM:** FIXME: `arg0..arg2` must be preserved, use something else.]

```
and. arg0,Dest,Src
```

Else if *Src* is a 16-bit unsigned integer, then [**PERM:** FIXME: `arg0..arg2` must be preserved, use something else.]

```
andi. arg0,Dest,Src
```

Else if *Src* is a 16-bit unsigned integer *UI* shifted 16 bits, then [**PERM:** FIXME: `arg0..arg2` must be preserved, use something else.]

```
andis. arg0,Dest,UI
```

Else if *Src* is a stretch of *N* 1-bits followed by *M* least significant 0-bits, then [**PERM:** FIXME: `arg0..arg2` must be preserved, use something else.]

```
rldicr. arg0,Dest,64-N-M,N-1
```

Else, reduce to [**PERM:** FIXME: `arg0..arg2` must be preserved, use something else.]

```
move(Src,arg1)
and. arg0,Dest,arg1
```

2.5.5 jump(*Target*)

Purpose To transfer program control to *Target*.

Condition Codes
Undefined.

`x86`
`x86_64` If *Target* is of the form `cp(OFF)`, then

```
lea OFF(cp), %rax
jmp *%rax
```

Else for `x86_64`, if *Target* is not reachable with a 32-bit offset

```
jmp Trampoline
[...]
Trampoline: jmp *0(%rsi)
.quad Target
```

Else

```
jmp Target
```

`PPC64` If *Target* is of the form `cp(OFF)`, then `OFF` is nonzero, and the transfer *must* use the CTR register:

```
addi 0, cp, OFF
mtctr 0
bctr
```

Else if *Target* is a local label, then

```
b Target
```

Else, reduce to: [**PERM:** Do we *need to* use the CTR register here (e.g. can the callee be relying on CTR being set?)]

```
b Trampoline
[...]
Trampoline:
```

```
move(Target,0)
mtctr 0
bctr
```

2.5.6 call(*Target*)

Purpose
: To transfer program control to *Target*, with the return address pushed on the stack or saved in a register.

Condition Codes
: Undefined.

`x86`
`x86_64`
: For `x86_64`, if *Target* is not reachable with a 32-bit offset

```
call Trampoline
[...]
Trampoline: jmp *0(%rsi)
.quad Target
```

Else

```
call Target
```

`PPC64`

There is a bug here:

IR instruction: call(label(G))

Power code: bl 0x3fffb035c8e8

Problem 1: Callee expects CTR initialized.

Problem 2: Callee can escape to native_nonjit, which will access TOC[arg5].

Conclusion: call(label(_)) must emit the same sequence as call(native_entry(_))!

If *Target* is a local label, then

```
bl Target
```

[PERM: NOTE: using `bl` is sub-optimal if we will not return (via the link register) to the following instruction. See p. 36 "Use Branch instructions for which LK=1 only as subroutine calls"]

Else, reduce to the following, where the transfer *must* use the CTR register.

```
bl Trampoline
[...]
Trampoline:
move(Target,0)
mtctr 0
bctr
```

[PERM: NOTE: this move must be encoded in a way that CALLEE_TOC_OFFSET in ppc64le_kernel.s4 understands! Document the requirements! We could simplify initial implementation by always putting the toc offset in a fixed register `the_reg` (e.g. `arg5`)

(so CALLEE_TOC_OFFSET can just patch TOC+the_reg. We can optimize this later. Question: Presumably Target will be an immediate in these cases?]

[**PERM: NOTE: the jitter must not blindly re-use same-valued TOC entries, since some entries may be changed, post-jit, by CALLEE_TOC_OFFSET users.**]

2.5.7 ccall(*Cond*,*Target*)

Purpose
: If *Cond* is true, then transfer program control to *Target*, with the return address pushed on the stack or saved in a register. *Cond* is most likely false.

Condition Codes
: Undefined.

`x86`
`x86_64`
: Let *NCond* be the negation of *Cond*, and

```
jcc NCond,1f
call(Target)
1:
```

`PPC64`
: [**PERM:** BUG: Does this have the same problem as call to local label? (must go via CTR+TOC)]

If *Target* is a local label, then

```
bcl Cond,Target
```

[**PERM:** Is it true here, as for the `call` instruction, that "the transfer *must* use the CTR register." (and the CALLEE_TOC_OFFSET issues)?]

Else if `Trampoline` is within 32764 bytes, reduce to:

```
bcl Cond,Trampoline
[...]
Trampoline:
move(Target,0)
mtctr 0
bctr
```

Else, let *NCond* be the negation of *Cond*, and reduce to:

```
bc NCond, 1f
bl Trampoline
1: [...]
Trampoline:
move(Target,0)
mtctr 0
bctr
```

2.5.8 branch(*Cond*,*Target*)

Purpose
: To conditionally transfer program control to *Target*.

Condition Codes
: Must preserve all condition codes except overflow, which is left undefined.

x86
x86_64 For `x86_64`, if *Target* is not reachable with a 32-bit offset

```
jcc Cond,Trampoline
[...]
Trampoline: jmp *0(%rsi)
.quad Target
```

Else

```
jcc Cond,Target
```

PPC64 Let *NCond* be the negation of *Cond*. If *Target* is a local label, then

```
// if Target is within 32764 bytes
bc Cond,Target
// else Target is not within 32764 bytes
bc NCond, 1f
b Target
1:
```

Else the explicit branch instruction must [**PERM:** is a trampoline really strictly necessary, or just desirable?] go via a trampoline:

```
// if Trampoline is within 32764 bytes
bc Cond,Trampoline
// else Trampoline is not within 32764 bytes
bc NCond, 1f
b Trampoline
1: [...]
Trampoline:
move(Target,0)
mtctr 0
bctr
```

2.5.9 cmove(*Cond*,*Src*,*Dest*)

Purpose To conditionally copy the value of source *Src* into destination *Dest*.

Condition Codes
Undefined.

x86
x86_64 If both operands are in registers, then

```
cmove Cond,Src,Dest
```

Else, let *NCond* be the negation of *Cond*, and

```
jcc NCond,1f
move(Src,Dest)
1:
```

PPC64 If both operands are in registers, then note that neither *Src* nor *Dest* can be R0 (which would be treated as constant zero), and:

```
// if Cond is l or lu
isel Dest,Src,Dest,0
```

```
// else if Cond is g or gu
isel Dest,Src,Dest,1
// else if Cond is e
isel Dest,Src,Dest,2
// else if Cond is o
isel Dest,Src,Dest,3
// else if Cond is le or leu
isel Dest,Dest,Src,1
// else if Cond is ge or geu
isel Dest,Dest,Src,0
// else if Cond is ne
isel Dest,Dest,Src,2
// else if Cond is no
isel Dest,Dest,Src,3
```

Else, let *NCond* be the negation of *Cond*, and

```
bc NCond,1f
move(Src,Dest)
1:
```

2.5.10 add(*Src1*,*Src2*,*Dest*)

Purpose To store the value of the expression *(Src1+Src2)* in *Dest*.

Condition Codes
Undefined.

`x86`
`x86_64` If *Src1* and *Dest* are the same memory operand and *Src2* is the constant 0, then

```
/* nothing */
```

Else if *Src2* is the constant 0, then the instruction reduces to

```
move(Src1,Dest)
```

Else if *Src1* and *Dest* are the same memory operand and *Src2* is a 32-bit signed integer, then

```
add $Src2,Dest
```

Else if *Src1* is a register, *Src2* is the 32-bit signed integer `OFFSET`, and *Dest* is a register, then

```
lea OFFSET(Src1),Dest
```

Else for `x86_64`, if *Src1* and *Dest* are the same memory operand and *Src2* is not a 32-bit signed integer, then

```
movabs $Src2,%r11
add %r11,Dest
```

Else if *Dest* is in memory, the instruction reduces to

```
add(Src1,Src2,val)
move(val,Dest)
```

Else, the instruction reduces to

```
move(Src1,Dest)
add(Dest,Src2,Dest)
```

`PPC64` [**PERM:** An unstated assumption seems to be that *Src1* is a register or in memory.]

If *Dest* is in memory, then reduce to [**PERM:** FIXME: `arg0..arg2` must be preserved, use something else.]

```
add(Src1,Src2,arg0)
std arg0,Dest
```

Else if *Src1* is in memory, then reduce to [**PERM:** FIXME: `arg0..arg2` must be preserved, use something else.]

```
ld arg1,Src1
add(arg1,Src2,Dest)
```

Else if *Src2* is in memory, then reduce to [**PERM:** FIXME: `arg0..arg2` must be preserved, use something else.]

```
ld arg2,Src2
add(Src1,arg2,Dest)
```

Else if *Src2* is a signed 16-bit integer *SI*, then note that *Src1* cannot be R0, which would mean the constant zero, and

```
addi Dest,Src1,SI
```

Else if *Src2* equals *(HI<<16)+LO*, where *HI* is a signed 16-bit integer and *LO* is an unsigned 16-bit integer, then note that neither register operand can be R0, which would mean the constant zero, and

```
addis Dest,Src1,HI
ori Dest,Dest,LO // omit if LO = 0   [PERM: NO! this is wrong for addition]
```

Else, reduce to [**PERM:** FIXME: `arg0..arg2` must be preserved, use something else.]

```
move(Src2,arg2)
add Dest,Src1,arg2
```

2.5.11 addo(*Src1*,*Src2*,*Dest*)

Purpose To store the value of the expression *(Src1+Src2)* in *Dest*.

Condition Codes

Overflow set iff the signed add yields an arithmetic overflow, and cleared otherwise. Other condition codes undefined.

`x86`
`x86_64` *Src2* is an immediate.

If *Src1* and *Dest* are the same memory operand and *Src2* is a 32-bit signed integer, then

```
add $Src2,Dest
```

Else for `x86_64`, if *Src1* and *Dest* are the same memory operand and *Src2* is not a 32-bit signed integer, then

```
movabs $Src2,%r11
```

```
add %r11,Dest
```

Else if *Dest* is in memory, the instruction reduces to

```
addo(Src1,Src2,val)
mov val,Dest
```

Else, the instruction reduces to

```
move(Src1,Dest)
addo(Dest,Src2,Dest)
```

`PPC64` *Src1* is a register and *Src2* is an immediate[2].

[**PERM:** BUG: the arguments can be, e.g. `addo(ac0, ac1, ac0)`, i.e. *Src2* may not be an immediate.]

Reduce to [**PERM:** FIXME: `arg0..arg2` must be preserved, use something else.]

```
li 0,0
mtxer 0
move(Src2,arg2) [PERM: Wrong. Move does not preserve condition codes (so c
addo. Dest,Src1,arg2
```

2.5.12 sub(*Src1*,*Src2*,*Dest*)

Purpose To store the value of the expression *(Src1-Src2)* in *Dest*.

Condition Codes
Undefined.

`x86`
`x86_64` If *Src1* and *Dest* are the same memory operand and *Src2* is a 32-bit signed integer, then

```
sub $Src2,Dest
```

Else if *Src1* is a register, *Src2* is the 32-bit signed integer `OFFSET`, and *Dest* is a register, then

```
lea -OFFSET(Src1),Dest
```

Else for `x86_64`, if *Src1* and *Dest* are the same memory operand and *Src2* is not a 32-bit signed integer, then

```
movabs $Src2,%r11
sub %r11,Dest
```

Else if *Dest* is in memory, the instruction reduces to

```
sub(Src1,Src2,val)
move(val,Dest)
```

Else, the instruction reduces to

```
move(Src1,Dest)
sub(Dest,Src2,Dest)
```

---

[2] Unlike the case for x86/x86_64

PPC64 If *Dest* is in memory, then reduce to [**PERM:** FIXME: `arg0..arg2` must be preserved, use something else.]

```
sub(Src1,Src2,arg0)
std arg0,Dest
```

Else if *Src1* is in memory, then reduce to [**PERM:** FIXME: `arg0..arg2` must be preserved, use something else.]

```
ld arg1,Src1
sub(arg1,Src2,Dest)
```

Else if *Src2* is in memory, then reduce to [**PERM:** FIXME: `arg0..arg2` must be preserved, use something else.]

```
ld arg2,Src2
sub(Src1,arg2,Dest)
```

Else if -*Src2* is a signed 32-bit integer, then reduce to

```
add(Src1, -Src2, Dest)
```

Else, reduce to [**PERM:** FIXME: `arg0..arg2` must be preserved, use something else.]

```
move(Src2,arg2)
subf Dest,arg2,Src1
```

2.5.13 subo(*Src1*,*Src2*,*Dest*)

Purpose To store the value of the expression *(Src1-Src2)* in *Dest*. *Src2* need not be an immediate.

Condition Codes
Overflow set if the signed subtract yields an arithmetic overflow, and cleared otherwise. Other condition codes undefined.

`x86`
`x86_64`

If *Src2* and *Dest* are the same and *Src1* is the constant 0, then

```
neg Dest
```

Else if *Src1* and *Dest* are the same memory operand and *Src2* is a 32-bit signed integer, then

```
sub $Src2,Dest
```

Else for `x86_64`, if *Src1* and *Dest* are the same memory operand and *Src2* is not a 32-bit signed integer, then

```
movabs $Src2,%r11
sub %r11,Dest
```

Else if *Dest* is in memory, the instruction reduces to

```
subo(Src1,Src2,val)
mov val, Dest
```

Else, the instruction reduces to

```
move(Src1,Dest)
subo(Dest,Src2,Dest)
```

`PPC64` No operand can be in memory. [**PERM:** Does not the same hold for the operands also for x86/x86_64? If not, why? Because registers are scarce on x86/x86_64, operands can be in memory there. –Mats]

[**PERM:** BUG: the arguments can be, e.g. `subo(val,y(1),val)`, i.e. operands can be in memory.]

If *Src1* is 0, then

```
li 0,0
mtxer 0
nego. Dest,Src2
```

Else if *Src1* and *Src2* are in registers, then

```
li 0,0 [PERM:  What if  Src2 or  Src1 is R0? xref addo.]
mtxer 0
subfo. Dest,Src2,Src1
```

Else, reduce to, *to be completed* [**PERM:** What if Src2 is the most negative value, will overflow condition be set correctly?]

```
addo(Src1,-Src2,Dest)
```

2.5.14 mulo(*Src1*,*Src2*,*Dest*)

Purpose To store the value of the expression *(Src1*Src2)* in *Dest*. *Dest* must be a register and *Src2* must be an immediate.

[**PERM:** BUG: the arguments can be, e.g. `mulo(ac0,ac1,val)`, i.e. *Src2* may not be an immediate.]

Condition Codes

Overflow set if the signed multiply yields an arithmetic overflow, and cleared otherwise. Other condition codes are undefined.

`x86`
`x86_64` For `x86_64`, if *Src2* is not a 32-bit signed integer, then

```
mov Src1,Dest
movabs $Src2,%r11
mul %r11,Dest
```

Else

```
mov Src1,Dest
mul $Src2,Dest
```

`PPC64` [**PERM:** FIXME: `arg0..arg2` must be preserved, use something else.]

```
li 0,0
mtxer 0
move(Src2,arg2) [PERM:  Wrong. Move does not preserve condition codes (so c
mulldo. Dest,Src1,arg2
```

2.5.15 sh(*Src1*,*Src2*,*Dest*)

Purpose To store the value of the expression *(Src1<<Src2)* in *Dest*. *Dest* must be a register and *Src2* must be an immediate in the range [-4,4].

Condition Codes
: Undefined.

x86
x86_64
: If *Src1* is different from *Dest*, then reduce to

```
mov Src1,Dest
sh(Dest,Src2,Dest)
```

Else if *Src2* > *0* then

```
shl $Src2,Dest
```

Else

```
shr $-Src2,Dest
```

PPC64
: If *Src1* is in memory, then reduce to [**PERM:** FIXME: `arg0..arg2` must be preserved, use something else.]

```
ld arg1,Src1
sh(arg1,Src2,Dest)
```

Else if *Src2* > *0* then

```
sldi Dest,Src1,Src2
```

Else

```
srdi Dest,Src1,-Src2
```

2.5.16 and(*Src1*,*Src2*,*Dest*)

Purpose
: To store the value of the expression *(Src1*$\wedge$*Src2)* in *Dest*. *Src1* and *Dest* must be the same operand and *Src2* must be an immediate.

[**PERM:** BUG: the arguments can be, e.g. `and(x(3),x(2),val)`, i.e. *Src1* and *Dest* may differ.] [**PERM:** BUG: the arguments can be, e.g. `and(ac0,ac1,ac0)`, i.e. *Src2* may not be an immediate.]

Condition Codes
: Undefined.

x86
x86_64
: For `x86_64`, if *Src2* is not a 32-bit signed integer, then

```
movabs $Src2,%r11
and %r11,Dest
```

Else

```
and $Src2,Dest
```

PPC64
: If *Src2* is a 16-bit unsigned integer *UI*, then

```
andi. Dest,Src1,UI
```

Else if *Src2* equals *(HI<<16)*, where *HI* is an unsigned 16-bit integer, then

```
andis. Dest,Src1,HI
```

Else if *Src2* is a stretch of *N* 1-bits, extending through the least significant bit, then

```
rldicl Dest,Src1,0,64-N
```

Else if *Src2* is a stretch of *N* 1-bits, extending through the most significant bit, then

```
rldicr Dest,Src1,0,N-1
```

Else, reduce to [**PERM:** FIXME: `arg0..arg2` must be preserved, use something else.]

```
move(Src2,arg2)
and Dest,Src1,arg2
```

2.5.17 or(*Src1*,*Src2*,*Dest*)

Purpose
: To store the value of the expression `(Src1\/Src2)` in *Dest*. *Src1* and *Dest* must be the same operand and *Src2* must be an immediate.

  [**PERM:** BUG: the arguments can be, e.g. `or(val,11,x(3,0))`, i.e. *Src1* and *Dest* may differ.] [**PERM:** BUG: the arguments can be, e.g. `or(val,y(6),val)` and `or(ac0,ac1,ac0)`, i.e. *Src2* may not be an immediate.]

Condition Codes
: Undefined.

`x86`
`x86_64`
: For `x86_64`, if *Src2* is not a 32-bit signed integer, then

```
movabs $Src2,%r11
or %r11,Dest
```

Else

```
or $Src2,Dest
```

`PPC64`
: If *Src2* is a 16-bit unsigned integer *UI*, then

```
ori Dest,Src1,UI
```

Else if *Src2* equals *(HI<<16)+LO*, where *HI* is an unsigned 16-bit integer and *LO* is an unsigned 16-bit integer, then

```
oris Dest,Src1,HI
ori  Dest,Dest,LO // omit if LO = 0
```

Else, reduce to [**PERM:** FIXME: `arg0..arg2` must be preserved, use something else.]

```
move(Src2,arg2)
or Dest,Src1,arg2
```

2.5.18 xor(*Src1*,*Src2*,*Dest*)

Purpose
: To store the value of the expression *(Src1 \ Src2)* in *Dest*. *Src1* and *Dest* must be the same operand and *Src2* must be an immediate.

  [**PERM:**
  BUG: the arguments can be, e.g. `xor(val,-5,arg1)` or `xor(ac0,ac1,ac0)`, i.e. *Src1* and *Dest* may differ.]

Condition Codes
: Undefined.

x86
x86_64 For `x86_64`, if *Src2* is not a 32-bit signed integer, then

```
movabs $Src2,%r11
xor %r11,Dest
```

Else

```
xor $Src2,Dest
```

PPC64 If *Src2* is a 16-bit unsigned integer *UI*, then

```
xori Dest,Src1,UI
```

Else if *Src2* equals *(HI<<16)+LO*, where *HI* is an unsigned 16-bit integer and *LO* is an unsigned 16-bit integer, then

```
xoris Dest,Src1,HI
xori  Dest,Dest,LO // omit if LO = 0
```

Else, reduce to [**PERM:** FIXME: `arg0..arg2` must be preserved, use something else.]

```
move(Src2,arg2)
xor Dest,Src1,arg2
```

2.5.19 int2cp(*Src*,*Dest*)

Purpose To convert a tagged integer to a choicepoint pointer. *Dest* must be `val`.

Condition Codes
Undefined.

x86

```
mov Src,%eax
sar $1,%eax
dec %eax
add w_choice_start,%eax
```

note that `val` is `%eax` on x86.

x86_64

```
mov Src,%rax
sub $3,%rax
add w_choice_start,%rax
```

note that `val` is `%rax` on x86.

PPC64 If *Src* is in memory, then reduce to [**PERM:** FIXME: `arg0..arg2` must be preserved, use something else.]

```
ld arg1,Src
int2cp(arg1,Dest)
```

Else,

```
ld val,w_choice_start
addi val,val,-3
add val,Src,val
```

2.5.20 cp2int(*Src*,*Dest*)

Purpose
: To convert a choicepoint pointer to a tagged integer. Dest cannot be `val`.

Condition Codes
: Undefined.

`x86`

```
mov Src,%eax
sub w_choice_start,%eax
lea 3(,%eax,2),%eax
mov %eax,Dest [PERM: Can do better if Dest is a register]
```

`x86_64`

```
mov Src,%rax
sub w_choice_start,%rax
add $3,%rax
mov %rax,Dest [PERM: Can do better if Dest is a register]
```

`PPC64`
: If *Src* is in memory, then reduce to [**PERM:** FIXME: `arg0..arg2` must be preserved, use something else.]

```
ld arg1,Src
cp2int(arg1,Dest)
```

Else if *Dest* is in memory, then reduce to [**PERM:** FIXME: `arg0..arg2` must be preserved, use something else.]

```
cp2int(Src,arg0)
std arg0,Dest
```

Else,

```
ld Dest,w_choice_start
subf Dest,Dest,Src
addi Dest,Dest,3
```

2.5.21 init(*Dest1*,*Dest2*)

Purpose
: To create a brand new variable in the first destination, making the second destination a variable bound to the first. *Dest1* must be in memory.

Condition Codes
: Undefined.

`x86`
`x86_64`
: If *Dest1* is on the form `r(0)`, then

```
mov r,Dest1
mov r,Dest2
```

Else if *Dest2* is the register `r`, then

```
lea Dest1,r
mov r,Dest1
```

Else

```
lea Dest1,%rax
```

```
mov %rax,Dest1
mov %rax,Dest2
```

PPC64

Both *Dest1* and *Dest2* must not be based on R0 (which would mean zero in the instructions `la` and `std`).

If *Dest1* is on the form `r(0)`, then

```
std r,Dest1
std r,Dest2
```

Else if *Dest2* is the register `r`, then

```
la r,Dest1
std r,Dest1 [PERM: // saner as ''std r,r'' I think]
```

Else [**PERM:** FIXME: `arg0..arg2` must be preserved, use something else.]

```
la arg0,Dest1
std arg0,Dest1 [PERM: // saner as ''std arg0,arg0'' I think (since arg0 co
std arg0,Dest2
```

2.5.22 pop

Purpose To discard the top of the stack.

Condition Codes
Undefined.

x86
x86_64

```
pop %rax
```

PPC64

```
/* nothing */
```

2.5.23 context(*Target*)

Target is a local label.

Purpose To refresh the TOC pointer.

Condition Codes
Undefined.

x86
x86_64

```
/* nothing */
```

PPC64 The `CTR` is assumed to contain the address of the local label (this is ensured by the caller, typically by jumping to the label using `bctr` or the like).

Let *OFF* be the offset to the TOC from *Target*. Reduce to

```
mfctr toc
add(toc,OFF,toc)
```

2.5.24 half(*Constant*)

Purpose To lay out an aligned constant occupying half a machine word.

Condition Codes
Undefined.

`x86`

```
[possible padding]
.value Constant
```

`x86_64`[**PERM:** No padding for x86_64? jit.c does padding for all Intel]
`PPC64`

```
.long Constant
```

2.5.25 word(*Constant*)

Purpose To lay out an aligned constant occupying one machine word.

Condition Codes
Undefined.

`x86`

```
[possible padding]
.long Constant
```

`x86_64`
`PPC64`

```
[possible padding]
.quad Constant
```

2.5.26 label(L)

Purpose A label indicating a code point that can be referred to by other instructions. `L` is on the form `'$VAR'(`*`Int`*`)`.

Condition Codes
Undefined.

2.5.27 align(*Int*)

Purpose To enforce code alignment. Let *pc16* denote “program counter modulo 16”.

Condition Codes
Undefined.

`x86` Depending on *Int*:

0 If pc16 in [1,8], bump pc until pc16=8. Else if pc16 in [9,12], bump pc until pc16=12. Else, bump pc until pc16=0.

1 If pc16 in [3,10], bump pc until pc16=10. Else if pc16 in [11,14], bump pc until pc16=14. Else, bump pc until pc16=2.

2 If pc16 in [9,15], bump pc until pc16=0.

3 Bump pc until pc16=12.

4 Bump pc until pc16 in {0,4,8,12}.

`x86_64` Depending on *Int*:

0 If pc16 in [1,8], bump pc until pc16=8. Else, bump pc until pc16=0.

1 Else if pc16 in [5,12], bump pc until pc16=12. Else, bump pc until pc16=4.

2 If pc16 in [9,15], bump pc until pc16=0.

3 Bump pc until pc16=8.

4 Bump pc until pc16 in {0,8}.

`PPC64` Depending on *Int*:

1 Bump pc until pc16 in {4,12}.

2 No alignment needed.

0
3
4 Bump pc until pc16 in {0,8}.

2.5.28 try_chain(*list of* (*Label-Alternative*),*Arity*)

Purpose To lay out a data structure for backtracking purposes.

Condition Codes
Undefined.

`x86`
`x86_64`
`PPC64` Every element of the list of pairs corresponds to a block of three machine words followed by two half machine words, laid out as follows, where `b+o` denotes an address at `o` machine words after the start of the block:

```
b+0  : Pointer to the next block, or NULL if it is the last block.
b+1  : Label, i.e. code address.
b+2  : Alternative, i.e. struct try_node pointer.
b+3  : offsetof(struct node,term[Arity])
b+3.5: Wmode(TRY)
```

2.5.29 switch(*list of* (*Key-Target*),*Default*)

Purpose To perform a switch on the principal functor of register `x0`. *Target* is the jump target when `x0` matches *Key*. *Default* is the default jump target.

Condition Codes
Undefined.

`x86`
`x86_64`
`PPC64` This is laid out as a regular `struct sw_on_key`, machine-word aligned.

2.5.30 trampolines(*Base*)

Base is a local label that must be thre preceeding instruction.

The trampolines, if any, are emitted here.

2.5.31 toc(*Base*)

Base is a local label that must be thre preceeding instruction.

The TOC entries, if any, are emitted here.

3 Predicate Linkage

For the purposes of SPJIT, it is useful to think of three modes in which a predicate *p* can be:

jitex — *p* has been JIT compiled and does not have a breakpoint, block declaration or the like. Calls from other *jitex* predicates to *p* stay in native code.

cex — *p* is implemented by a C function and does not have a breakpoint, block declaration or the like. Calls from *jitex* predicates to *p* go to the C function[**PERM:** "go to the C function" means what? Direcly (not via `native_c`? If so, how are breakpoints/redefinitions to C functions triggered when called from jitted code_]. Such predicates are never subject to JIT compilation.

wamex — All other cases. Calls from *jitex* predicates to *p* are routed via the WAM emulator. The transfer of control is implemented by returning from the JIT runtime system with the value 2.

When SICStus starts, no *jitex* predicates exist, but start to appear as emulated predicates get JIT compiled.

Note that setting a breakpoint on a *jitex* predicate changes its state to *wamex*. Removing the breakpoint changes the state back to *jitex*. Redefining a *jitex* predicate also changes its state to *wamex*.

For a predicate *p* whose `struct definition *` pointer is `def`, `def->jit` is either NULL or points at the JIT code generated for *p*, whereas `def->proc.native` contains a lead-in sequence of machine instructions. The JIT compiler translates a call from *p* to *q* into a call to *q*'s lead-in sequence, no matter what type of predicate *q* is or whether it is even defined.

[**PERM:** Do `mod_def.proxies` definitions need some special treatment?]

If *p* is *jitex*, then the lead-in sequence calls the kernel subroutine `native_shunt_link`, which patches the caller to directly call the JIT code the next time around. If *p* is *cex*, then the lead-in sequence calls the kernel subroutine `native_c`, which routes the call to the C function. If *p* is *wamex*, then the lead-in sequence calls the kernel subroutine `native_nonjit`, which arranges for the call to be handled by the WAM emulator.

If the state of *p* changes from *jitex* to *wamex*, then a prefix of `def->jit` is modified to an instruction sequence that calls the kernel subroutine `native_restore_link`, which patches the caller to call the lead-in sequence the next time around.

If the state of *p* changes from *cex* to *wamex* because a breakpoint was set, then `def->jit` is not relevant (because such predicates are not jitted).

If the state of *p* changes from *wamex* back to *jitex* because a breakpoint was removed, then the prefix of `def->jit` is repaired to contain the original JIT instructions for *p*.

If *p* was first JIT compiled and then redefined, then `def->jit` cannot be freed entirely, because there may be dangling references to it created by `native_shunt_link`. Thus, its prefix, which calls `native_restore_link`, must be preserved. This small memory leak is not expected to be noticeable in a production setting.

The exact layout of these code sequences is back-end dependent and is explained in the following sections.

3.1 Code Outline, Lead-In and Prefix Sequences for x86

For all modes, the prefix sequence is preceded by a single word containing a pointer to the current predicate. The prefix sequence is followed by a single section of code.

Mode	**Lead-In**	**Prefix**

jitex	`jmp native_shunt_link`	`cmp h,w_heap_warn_soft`
		`jae native_nonjit`
		`pop %eax`

wamex	`jmp native_nonjit`	`jmp native_restore_link`
cex	`jmp native_c`	—

3.2 Code Outline, Lead-In and Prefix Sequences for x86_64

For all modes, the prefix sequence is preceded by a single word containing a pointer to the current predicate. The prefix sequence is followed by two sections of code.

Main Body
: The main body of the generated JIT code.

Trampolines
: Several small help routines for branching to kernel subroutines and other predicates.

Mode	**Lead-In**	**Prefix**

jitex	`jmp native_shunt_link`
	
	...or...
	`jmp *0(%rsi)`
	`.quad native_shunt_link`
		`cmp h,w_heap_warn_soft`
		`jae native_nonjit`
		`pop %rax`
		...or...
		`cmp h,w_heap_warn_soft`
		`jae Trampoline`
		`pop`
		...
		`Trampoline: jmp *0(%rsi)`
		`.quad native_nonjit`

wamex	`jmp native_nonjit`
	...or...
	`jmp *0(%rsi)`
		`jmp native_restore_link`
		...or...
		`jmp *0(%rsi)`

	`.quad native_nonjit`	`.quad native_restore_link`
cex	`jmp native_c`	—
	...or...	...or...
	`jmp *0(%rsi)`	—
	`.quad native_c`	—

3.3 Code Outline, Lead-In and Prefix Sequences for PPC64

For all modes, the prefix sequence is preceded by a single word containing a pointer to the current predicate. The prefix sequence is followed by three sections of code and data.

Main Body
: The main body of the generated JIT code.

Trampolines
: Several small help routines for branching to kernel subroutines and other predicates.

TOC
: An array of constants for loading instead of synthesizing, for cases where loading is faster. A pointer to the TOC is maintained in `toc` and is refreshed by the `context(_)` IR instruction. Every TOC must begin with:

```
toc+0  : native_shunt_link
toc+8  : native_restore_link
toc+16 : native_nonjit
toc+24 : native_c
```

In the lead-in sequence, the toc register is guaranteed to point at *some* valid JIT-TOC, and thus contain the above four entries.

Mode	**Lead-In**	**Prefix**
jitex	`ld 0,0(toc)`	`ld 0,w_heap_warn_soft`
	`mtctr 0`	`cmpld 7,h,0`[**PERM:** Is this `cmpl cr7,1,h,0`? Why CR7 and not the default CR0?]
	`bctr`	`blt 7,1f`[**PERM:** Is this `blt cr7,1f`? Why CR7 and not the default CR0?]
		`ld 0,16(toc)`
		`mtctr 0`
		`bctr`
		`1:`
wamex	`ld 0,16(toc)`	`ld 0,8(toc)`
	`mtctr 0`	`mtctr 0`
	`bctr`	`bctr`
cex	`ld 0,24(toc)`	—
	`mtctr 0`	—

```
bctr                              —
```

3.4 WAM-JIT Interface

In terms of the C call stack, the WAM emulator calls the JIT runtime system, but the latter never calls the WAM emulator. Recursive nesting can only happen in the foreign language interface, if the foreign function calls Prolog, and similarly in a predicate implemented as a C function, if the C function calls Prolog.

The WAM emulator has a general mechanism to dispatch on the predicate type. When it sees a *jitex* predicate, it routes the call with call site `w->insn` and callee `w->predicate` to the ABI function:

```
int call_native(struct worker *w);
```

The WAM instruction set has been extended by the special instruction `NATIVE_OP`, and it is legal for `w->next_insn` to point to it, i.e., it is a legal continuation. As for all continuations, the half word preceding it is the environment size field. The word following it points to the WAM code equivalent of the continuation, immediately followed by the native code of the continuation. When the WAM emulator sees it, it routes the call with `w->insn` pointing to it to the ABI function:

```
int proceed_native(struct worker *w);
```

Both ABI function return the values:

0 *jitex* code backtracks into *wamex* code.

1 *jitex* code proceeds to *wamex* code at address `w->insn`, in read mode if `x(0)` is nonvar and in write mode otherwise.

2 *jitex* code calls *wamex* code with call site `w->insn` and callee `w->predicate`.

3 *jitex* code proceeds to the WAM instruction `PROGRESS`.

4 Register Allocation

4.1 Placement of WAM and IR Registers

The “WAM registers” `arg0..arg2` are for passing parameters from the JIT code to the runtime system. These “WAM registers” must be preserved by the machine code that implements the IR instructions (i.e. the generated machine code must not used any of `arg0..arg2` as scratchpad registers).

The “WAM registers” `arg3..arg5` are scratchpad registers of the runtime system and may also be freely used by the machine code that implements the IR instructions.

For x86_64, the exact offsets of `ac0` and `ac1` are ABI dependent (Windows vs. non-Windows).

For PPC64, the `CTR` register is used by `context(_)` instructions, in predicate-to-predicate calls, and for jumping to continuations. The link register is used in `call` and `ccall` instructions. Otherwise, `CTR` can be used freely, and so can the link register. Additionally `R0` and `arg3..arg5` can be used freely by the machine code that implements the IR instructions.

WAM	**x86**	**x86_64**	**PPC64**
`sp`	`%esp`	`%rsp`	`r1`
`toc`	—	—	`r2`
`val`	`%eax`	`%rax`	`r3`
`arg0`	`0(%esp)`	`%rax`	`r3`
`arg1`	`4(%esp)`	`%r10`	`r4`
`arg2`	`8(%esp)`	`%r11`	`r5`
`arg3`	—	—	`r6`
`arg4`	—	—	`r7`
`arg5`	—	—	`r8`
`s`	`%edx`	`%rdx`	`r9`
`ac0`	`28(%esp)`	`OFF(%rsp)`	`r9`
`ac1`	`32(%esp)`	`OFF(%rsp)`	`r10`
`ab`	`W_LOCAL_UNCOND(w)`	`NODE_LOCAL_TOP(b)`	`r11`
`hb`	`W_GLOBAL_UNCOND(w)`	`NODE_GLOBAL_TOP(b)`	`r12`
`b`	`W_NODE(w)`	`r8`	`r14`
`a`	`%ebp`	`%rbp`	`r15`
`h`	`%esi`	`%rsi`	`r16`
`tr`	`W_TRAIL_TOP(w)`	`r9`	`r17`
`e`	`%edi`	`%rdi`	`r18`
`cp`	`%ecx`	`%rcx`	`r19`
`w`	`%ebx`	`%rbx`	`r20`
`w_insn`	`W_INSN(w)`	`W_INSN(w)`	`r21`
`w_heap_warn_soft`	`W_HEAP_WARN_SOFT(w)`	`W_HEAP_WARN_SOFT(w)`	`W_HEAP_WARN_SOFT(w)`
`w_next_node`	`W_NEXT_NODE(w)`	`W_NEXT_NODE(w)`	`W_NEXT_NODE(w)`
`w_numstack_end`	`W_NUMSTACK_END(w)`	`W_NUMSTACK_END(w)`	`W_NUMSTACK_END(w)`

w_stack_start	W_STACK_START(w)	W_STACK_START(w)	W_STACK_START(w)
w_stack_warn	W_STACK_WARN(w)	W_STACK_WARN(w)	W_STACK_WARN(w)
w_fli_stack_ start	W_FLI_STACK_ START(w)	W_FLI_STACK_ START(w)	W_FLI_STACK_ START(w)
x(0)	W_TERM0(w)	%r12	r22
x(1)	W_TERM1(w)	%r13	r23
x(2)	W_TERM2(w)	%r14	r24
x(3)	W_TERM3(w)	%r15	r25
x(4)	W_TERM4(w)	W_TERM4(w)	r26
x(5)	W_TERM5(w)	W_TERM5(w)	r27
x(6)	W_TERM6(w)	W_TERM6(w)	r28
x(7)	W_TERM7(w)	W_TERM7(w)	r29
x(8)	W_TERM8(w)	W_TERM8(w)	r30
x(9)	W_TERM9(w)	W_TERM9(w)	r31

4.2 Use of Machine Registers and Stack Frame Slots for x86

%eax	gpr(0)	val
%ecx	gpr(1)	cp
%edx	gpr(2)	s
%ebx	gpr(3)	w
%esp	gpr(4)	SP
%ebp	gpr(5)	a
%esi	gpr(6)	h
%edi	gpr(7)	e
0(%esp)		arg0
4(%esp)		arg1
8(%esp)		arg2
12(%esp)		%ebx callee save
16(%esp)		%edi callee save
20(%esp)		%esi callee save
24(%esp)		%ebp callee save
28(%esp)		ac0
32(%esp)		ac1
36(%esp)		pad
40(%esp)		pad
44(%esp)		pad
48(%esp)		ret address
52(%esp)		w

4.3 Use of Machine Registers and Stack Frame Slots for x86_64 (non-Windows)

`%rax`	`gpr(0)`	`val, arg0`
`%rcx`	`gpr(1)`	`cp`
`%rdx`	`gpr(2)`	`s`
`%rbx`	`gpr(3)`	`w`
`%rsp`	`gpr(4)`	`SP`
`%rbp`	`gpr(5)`	`a`
`%rsi`	`gpr(6)`	`h`
`%rdi`	`gpr(7)`	`e`
`%r8`	`gpr(8)`	`b`
`%r9`	`gpr(9)`	`tr`
`%r10`	`gpr(10)`	`arg1`
`%r11`	`gpr(11)`	`arg2`
`%r12`	`gpr(12)`	`x(0)`
`%r13`	`gpr(13)`	`x(1)`
`%r14`	`gpr(14)`	`x(2)`
`%r15`	`gpr(15)`	`x(3)`
`0(%rsp)`		`%rbx` callee save
`8(%rsp)`		`%rbp` callee save
`16(%rsp)`		`%r12` callee save
`24(%rsp)`		`%r13` callee save
`32(%rsp)`		`%r14` callee save
`40(%rsp)`		`%r15` callee save
`48(%rsp)`		`ac0`
`56(%rsp)`		`ac1`
`64(%rsp)`		`arg0` spill slot
`72(%rsp)`		`pad`
`80(%rsp)`		ret address

4.4 Use of Machine Registers and Stack Frame Slots for x86_64 (Windows)

%rax	gpr(0)	val, arg0
%rcx	gpr(1)	cp
%rdx	gpr(2)	s
%rbx	gpr(3)	w
%rsp	gpr(4)	SP
%rbp	gpr(5)	a
%rsi	gpr(6)	h
%rdi	gpr(7)	e
%r8	gpr(8)	b
%r9	gpr(9)	tr
%r10	gpr(10)	arg1
%r11	gpr(11)	arg2
%r12	gpr(12)	x(0)
%r13	gpr(13)	x(1)
%r14	gpr(14)	x(2)
%r15	gpr(15)	x(3)
0(%rsp)		%rbx callee save
8(%rsp)		%rbp callee save
16(%rsp)		%rsi callee save
24(%rsp)		%rdi callee save
32(%rsp)		%r12 callee save
40(%rsp)		%r13 callee save
48(%rsp)		%r14 callee save
56(%rsp)		%r15 callee save
64(%rsp)		ac0
72(%rsp)		ac1
80(%rsp)		arg0 spill slot
88(%rsp)		pad
96(%rsp)		ret address

4.5 Use of Machine Registers and Stack Frame Slots for PPC64

[**PERM:** Would it be better to have the four special TOC-entries on the stack (like '$ref'/2 functor) so not all predicates would need to allocate/maintain a TOC.]

`r0`	`gpr(0)`	`scratch`
`r1`	`gpr(1)`	`sp` stack ptr
`r2`	`gpr(2)`	`toc` JIT-TOC ptr callee save
`r3`	`gpr(3)`	`arg0/val`
`r4`	`gpr(4)`	`arg1`
`r5`	`gpr(5)`	`arg2`
`r6`	`gpr(6)`	`arg3`
`r7`	`gpr(7)`	`arg4`
`r8`	`gpr(8)`	`arg5`
`r9`	`gpr(9)`	`ac0/s`
`r10`	`gpr(10)`	`ac1`
`r11`	`gpr(11)`	`ab`
`r12`	`gpr(12)`	`hb`
`r13`	`gpr(13)`	thread ptr
`r14`	`gpr(14)`	`b` callee save
`r15`	`gpr(15)`	`a` callee save
`r16`	`gpr(16)`	`h` callee save
`r17`	`gpr(17)`	`tr` callee save
`r18`	`gpr(18)`	`e` callee save
`r19`	`gpr(19)`	`cp` callee save
`r20`	`gpr(20)`	`w` callee save
`r21`	`gpr(21)`	`insn` callee save
`r22`	`gpr(22)`	`x(0)` callee save
`r23`	`gpr(23)`	`x(1)` callee save
`r24`	`gpr(24)`	`x(2)` callee save
`r25`	`gpr(25)`	`x(3)` callee save
`r26`	`gpr(26)`	`x(4)` callee save
`r27`	`gpr(27)`	`x(5)` callee save
`r28`	`gpr(28)`	`x(6)` callee save
`r29`	`gpr(29)`	`x(7)` callee save
`r30`	`gpr(30)`	`x(8)` callee save
`r31`	`gpr(31)`	`x(9)` callee save
`32(sp)`		`'$mutable'/2`
`40(sp)`		`'$ref'/2`
`48(sp)`		`ld 0, 16(toc)` for case analysis in `native_nonjit`

5 Runtime System

The runtime system contains 140 subroutines, each of which is briefly described in the following table. The arguments and return values are “typed” by the registers in which they are passed. Many arithmetic subroutines act on the accumulators `ac0` and `ac1`, each of which can be *unboxed*, i.e. contain a raw integer, or *boxed*, i.e. contain a tagged pointer to a big integer or float, either on the global stack or on a scratchpad area. If both accumulators are live, then either both are boxed or both are unboxed.

The type `cc` denotes a return value passed as a condition code, with the following conventions:

`o`
: Signals an arithmetic overflow or other error. Other condition codes are undefined.

`e` vs. `ne`
: Continue in write mode vs. read mode. Other condition codes are undefined.

`e` vs. `ne`
: Continue with unboxed accumulators vs. boxed accumulators. Other condition codes are undefined.

`e` vs. `ne`
: Reflects the outcome of `native_test_numbers()`; see below. Other condition codes are undefined.

`e` vs. `ne`
: Failure vs. success of a type test. Other condition codes are undefined.

`e`, `ne`, `l`, `le`, `g`, `ge`
: Reflects the outcome of a comparison. Other condition codes are undefined.

Following are the subroutines:

`void native_nonjit()`
: Handle general events as well as calls to non-*jitex* predicates.

`void native_restore_link()`
: Patch the caller, which corresponds to an IR instructions of the form `call(native_entry(M:F/A))`, to call the lead-in sequence, and remake the call. For x86/x86_64, this affects a `call` machine instruction, in the main body or in a trampoline. For PPC64, this never affects any machine instructions. Only TOC slots are affected.

`void native_shunt_link()`
: Patch the caller, which corresponds to an IR instructions of the form `call(native_entry(M:F/A))`, to call the prefix sequence, and jump there. For x86/x86_64, this affects a `call` machine instruction, in the main body or in a trampoline. For PPC64, this never affects any machine instructions. Only TOC slots are affected.

`void native_get_constant(val Xj, arg1 C)`
: Unify `Xj` with the constant `C`.

`cc native_get_list(val Xj)`
: Unify `Xj` with a list, setting `s` if read mode. Condition

`void native_get_nil(val Xj)`
: Unify `Xj` with the constant `[]`.

`cc native_get_structure(val Xj, arg1 F)`
: Unify `Xj` with a structure with principal functor `F`, setting `s` if read mode.

`void native_get_subconstant(val Xj, arg1 C)`
: Unify `Xj` with the constant `C`, where Xj occurs in compound term.

`cc native_get_sublist(val Xj)`
: Unify `Xj` with a list, setting `s` if read mode, where Xj occurs in compound term.

`void native_get_subnil(val Xj)`
: Unify `Xj` with the constant `[]`, where Xj occurs in compound term.

`cc native_get_substructure(val Xj, arg1 F)`
: Unify `Xj` with a structure with principal functor `F`, setting `s` if read mode, where Xj occurs in compound term.

`void native_get_value(val X, arg1 Y)`
: Unify `X` and `Y`.

`void native_bind(val X)`
: Trail the binding of X that just took place if necessary.

`void native_trail_unsafe(val X)`
: Trail local variable `X` if needed, in the context of `*_unsafe_variable`.

`void native_make_global(val X)`
: Globalize variable X if needed.

`cc native_compareop(arg0 X, arg1 Y)`
: Term compare `X` and `Y` with the condition code reflecting the output.

`void native_cut(val B)`
: Execute a cut (!) back to the choicepoint `B`.

`void native_fail()`
: Backtrack.

`void native_if()`
: Support for `ANOP_IF`.

`void native_metacall(val Callee)`
: Support for a metacall to `Callee`.

`void native_proceed()`
: Handle `PROCEED`, continuing into native code for `NATIVE_OP` continuations.

`void native_progress()`
: A general event has occurred; fall back on the WAM emulator to handle it and to proceed with a `PROGRESS` operation.

`void native_subproceed()`
: Tell the WAM emulator to proceed at address `w->insn`.

`void native_switch(val key, arg1 sw)`
: Dispatch on `key`, the principal functor of `x(0)`. `arg1` points at possible padding followed by an aligned `switch_on_key` struct.

`void native_try(val Label)`
: Push a choicepoint with a chain of alternatives at `Label`, and branch to the first alternative.

`void native_spill(val V, arg1 Xi)`
: Support `SPILL`.

`val native_unspill(val V)`
: Support `UNSPILL`.

`void native_first_float()`
: Support for converting unboxed `ac0` to a boxed float, allocated on the numstack.

`void native_first_long()`
: Support for boxing unboxed `ac0`.

`cc native_first_value(val X)`
: Load `ac0` with the value of `X`. `cc` reflects read/write mode.

`void native_fli_close()`
: Close the foreign call: restore C and SP_term_ref stacks, reset FLI exception flag, free any mems for `+codes` arguments, and `proceed`.

`val native_fli_get_atom(val X)`
: Check a `+atom` foreign argument. Escape to the emulator in case of error.

`void native_fli_get_codes(val X, val arg1)`
: Check a `+codes` foreign argument. Escape to the emulator in case of error. Otherwise, convert it to a string, allocate a mem, and add it to the mem ring in `arg1`. Returns the augmented mem ring.

`fpr(8) native_fli_get_float(val X)`
: Check a `+float` foreign argument. Escape to the emulator in case of error. Otherwise, convert it and return as a float.

`val native_fli_get_integer(val X)`
: Check a `+integer` foreign argument. Escape to the emulator in case of error. Otherwise, convert it and return as an integer.

`val native_fli_get_string(val X)`
: Check a `+string` foreign argument. Escape to the emulator in case of error. Otherwise, convert it and return as a string.

`void native_fli_open(inline Pred, inline Size, inline Arity)`
: Open a foreign call, with `w_insn` pointing to the corresponding WAM instruction. Push a C stack frame of size `Size`. Save SP_term_ref stack index and FLI exception flag. Push a WAM stack frame with the dereferenced argument registers of size `Arity`. Point `cp` to an inline `KONTINUE` instruction just after `Arity`.

`val native_fli_refresh(val X)`
Check FLI exception flag, and if set, close the foreign call and fail. call `heap_overflow()` if necessary. Must preserve `val` and `fpr(0)`.

`void native_fli_unify_atom(val X, arg1 Y)`
Unify a foreign `-atom` or `[-atom]` argument with `X`.

`void native_fli_unify_codes(val X, arg1 Y)`
Unify a foreign `-codes` or `[-codes]` argument with `X`. If the received value is misencoded, close the call and raise an error.

`void native_fli_unify_float(val X, arg1 Y)`
Unify a foreign `-float` or `[-float]` argument with `X`. If the received value is not a proper float, close the call and raise an error.

`void native_fli_unify_integer(val X, arg1 Y)`
Unify a foreign `-integer` or `[-integer]` argument with `X`.

`void native_fli_unify_string(val X, arg1 Y)`
Unify a foreign `-string` or `[-string]` argument with `X`. If the received value is misencoded, close the call and raise an error.

`void native_fli_unify_term(val X, arg1 Y)`
Unify a foreign `-term` or `[-term]` argument with `X`.

`void native_later_float()`
Convert unboxed `ac1` to a boxed float, allocated on the numstack.

`void native_later_long()`
Box unboxed `ac1`.

`void native_later_value_boxed(val X)`
Load `ac1` with the value of `X` where `ac0` is boxed.

`cc native_later_value_unboxed(val X)`
Load `ac1` with the value of `X` where `ac0` is unboxed. `cc` reflects read/write mode.

`void native_store_value_boxed(val X)`
Support for unifying boxed `ac0` with the value of `X`.

`void native_store_value_unboxed(val X)`
Support for unifying unboxed `ac0` with the value of `X`.

`val native_store_variable_boxed()`
Support for storing the value of boxed `ac0` in `val`.

`val native_store_variable_unboxed()`
Support for storing the value of unboxed `ac0` in `val`.

`cc native_compare_numbers()`
Compare the numbers in the accumulators with the condition code reflecting the output. Overflow reflects an error. [**PERM:** Who clears Overflow on non-error? Not `native_compare_numbers()`, it seems.]

void native_test_numbers()
: Perform a logical and of the boxed accumulators. The condition code reflects whether the result is zero.

cc native_fdivide_unboxed()
cc native_gcd_unboxed()
cc native_idivide_unboxed()
cc native_ipower2_unboxed()
cc native_lsh_unboxed()
cc native_modulus_unboxed()
cc native_msb_unboxed()
cc native_remainder_unboxed()
cc native_rsh_unboxed()
: Support for binary operations on unboxed accumulators.

void native_float1()
cc native_integer1()
cc native_left_shift()
cc native_minus()
cc native_right_shift()
cc native_sign()
: Support for unary and binary operations on boxed accumulators.

cc native_atom(val X)
cc native_atomic(val X)
cc native_float(val X)
cc native_integer(val X)
cc native_number(val X)
cc native_nonvar(val X)
cc native_var(val X)
cc native_simple(val X)
cc native_compound(val X)
cc native_callable(val X)
cc native_ground(val X)
cc native_mutable(val X)
cc native_db_reference(val X)
: Support for type-test instructions. Condition code e signals failure.

void native_append(arg0 X, arg1 Y, arg2 Z)
void native_arg(arg0 X, arg1 Y, arg2 Z)
void native_compare(arg0 X, arg1 Y, arg2 Z)
void native_create_mutable(arg0 X, arg1 Y)
void native_get_mutable(arg0 X, arg1 Y)
void native_update_mutable(arg0 X, arg1 Y)
void native_functor(arg0 X, arg1 Y, arg2 Z)
void native_length(arg0 X, arg1 Y)
void native_univ(arg0 X, arg1 Y)
: Support for the corresponding built-in predicates, which all compile inline.

cc native_abs()
cc native_acos()
cc native_acosh()
cc native_acot()
cc native_acot2()
cc native_acoth()
cc native_add()
cc native_and()
cc native_asin()
cc native_asinh()
cc native_atan()
cc native_atan2()
cc native_atanh()
cc native_ceiling()
cc native_complement()
cc native_cos()
cc native_cosh()
cc native_cot()
cc native_coth()
cc native_divide()
cc native_exp()
cc native_exp2()
cc native_fdivide()
cc native_float_fractional_part()
cc native_float_integer_part()
cc native_floor()
cc native_gcd()
cc native_idivide()
cc native_ipower2()
cc native_log()
cc native_log2()
cc native_maximum()
cc native_minimum()
cc native_modulus()
cc native_msb()
cc native_multiply()
cc native_or()
cc native_power2()
cc native_remainder()
cc native_round()
cc native_sin()
cc native_sinh()
cc native_sqrt()
cc native_subtract()
cc native_tan()
cc native_tanh()
cc native_truncate()
cc native_xor()

Arithmetic support acting on boxed accumulators.

6 Misc

6.1 Options Affecting Jitting

Description of some setting that affect JIT compilation and related things.

6.1.1 System Properties Affecting the JIT Compilation

SP_USE_SHADOW_KERNEL (default `yes`)
: `sicstus -DSP_USE_SHADOW_KERNEL=no` turns off the use of “shadow” kernel, i.e. the copies of the real kernel. Turning it off is useful if you want to set breakpoints in gdb etc. POWER only.

`SP_USE_XER` (default `no`)
: The default value for , and . POWER only.

`SP_USE_XER_ADDO`
: Whether the XER register should be used for overflow detection of `addo` IR-instruction on POWER.

`SP_USE_XER_SUBO`
: Whether the XER register should be used for overflow detection of `subo` IR-instruction on POWER.

`SP_USE_XER_MULO`
: Whether the XER register should be used for overflow detection of `mulo` IR-instruction on POWER.

`SP_JIT_HUGE_BLOCK` (default `yes`)
: Whether a huge block, with a shadow kernel in the middle, should be preallocated for jitted code. This is so that kernel calls from jitted code can use direct branches.

  The shadow kernel in the huge block will not be used if `SP_USE_SHADOW_KERNEL` is off.

`SP_JIT_STATS` (default `no`)
: Whether to ensure that `prolog:'$jit_print_stats'/0` prints accurate statistics about emitted IR instructions.

  Turning it on will prevent re-use of jitted code between iterations. This is why it is not enabled by default for debug builds.

`SP_JIT_ALIGN2` (default `yes`)
: Whether `align 2` should align to a multiple of 32 (instead of being a no-op). POWER only.

`SP_JIT_ALIGN3` (default `no`)
: Whether `align 3` should align to 24 (modulo 32) (instead of aligning to 0 (modulo 8)). POWER only.

  Do **not** turn it on, it will crash the system.

`SP_QUIET_JIT_FAIL` (default `yes`)
: `sicstus -DSP_QUIET_JIT_FAIL=no` will cause an assertion to trigger if jitting needs too many iterations (e.g. it would not terminate).

`SP_SPTI_PATH=OPTION`
: Whether to load code that gets informed about jitting events.

  `sicstus -DSP_SPTI_PATH=perf` is allowed if `--enable-perf` was specified when configuring. It will cause `perf` data to be emitted. This is enabled by default if `sicstus` detects that it is started under `perf`.

  The automatic enablement can be turned off `sicstus -DSP_SPTI_PATH=none`.

  `sicstus -DSP_SPTI_PATH=opdis` is also possible, if `--enable-oprofile` was specified when configuring.

6.1.2 Configuration Options Affecting the JIT Compilation

`--enable-jit-lq-stq` (default disabled)
: Whether to use quad-word load and store instructions (`lq` and `stq`) in the kernel and in jitted code. POWER only.

`--enable-jit-preload-fail` (default disabled)
: Whether to preload JIT failure continuation.

`--enable-jit-fli` (default enabled on supported platforms)
: Whether to use JIT compilation.

`--enable-jit-fli` (default enabled on supported platforms)
: Whether to use JIT compilation of FLI predicates. Ignored if JIT compilation is not enabled.

  This feature has not yet been implemented on POWER.

`--enable-jit-plcall-pass-cp-in-link` (default disabled)
: Whether to pass the Caller information in the link register. If it is disabled then the information is passed in a register, or not at all. POWER only.

`--with-opdis=PATH`
: The path to a OPDIS installation, e.g. `/usr/local/opdis`. This is needed in order to get machine code disassembly while dumping IR-code. Ignored unless `--enable-opdis` is also passed.

`--enable-opdis` (default disabled)
: Whether OPDIS should be used for disassembling machine code in debug output.

  OPDIS is supported on Linus, OS X and POWER. OPDIS itself needs to be modified to build on POWER.

  On OS X the path to binutils must be specified with `--with-binutils=PATH` on order to use OPDIS.

  Note: OPDIS must never be included in a released build. Licencing issues.

`--enable-perf` (default disabled)
: Whether `perf` should be supported, i.e. so that jitted code can be disassembled and annotated by `perf`.

  Supported on 64-bit Intel Linux and on (64-bit) POWER Linux.

`--enable-oprofile` (default enabled on Linux)
: Whether `oprofile` should be supported. Nowadays `perf` is preferred.

7 References

[PowerISA]

Power ISA Version 2.07 B. Downloaded from `https://www.power.org/`. Available in `/src/sicstus/docs/POWER/PowerISA_V2.07B.pdf`. Describes the POWER instruction set and its encoding.

[PowerABIELFv2]

Power Architecture 64-Bit ELF V2 ABI Specification. Downloaded from `www.ibm.com` (A newer version is available at Open Power Foundation `http://openpowerfoundation.org/technical/technical-resources/technical-specifications/` as `https://members.openpowerfoundation.org/document/dl/576`. Available in `/src/sicstus/docs/POWER/ABI64BitOpenPOWER_21July2014_pub.pdf`. Describes calling conventions etc. for Linux on (little-endian) POWER.