KH2AI ISA

v0.1

Kingdom Hearts 2 is a video game developped by Square Enix that also happens to be a very good game. As Square loves to reinvent the wheel they decided to make a custom AI assembler like scripting language for this engine, which also happens to be pretty inconsistent. This document will, in its value of a document, document this language as an Instruction Set Architecture(ISA) with additional information when necessary.

This booklet is separated into parts:

- The Notational Convention, explaining how every instruction is defined
- The Instruction Set, defining every operation in this language
- The System Calls, documenting calls done by the language outside of its own scope
- Known issues, if any
- An appendix for additional documents that might help comprehension

It is also worthy to note that some operations that otherwise do the same thing are given a different mnemonic depending on the context to be easier to write an assembler. An example of this can be seen in the PUSH.V and PUSH.L operations, which, while they both push a value to the stack, one of them is 48bits long and pushes a raw value while the other is 32bits long and does a relocation on the encoded address before pushing it, making the different naming needed.

1. Notational Convention

Instruction Format of Each Instruction

The description of each instruction uses the following format:

Mnemonic

Page headings show the instruction mnemonic and a brief description of the function, and the MIPS architecture level.

Instruction Encoding

This picture illustrates the bit formats of an instruction word.

Format

This section indicates the instruction formats for the assembler. Lower case indicates variables, corresponding to variable fields in the encoding picture.

Description Section

This section describes the instruction function and operation.

Exception Section

This section shows the exceptions that can be caused by the instructions.

Operation Section

This section describes the instruction operations in SLEIGH. You can refer to SLEIGH's own documentation for its notational conventions and refer to the Appendix for the custom SLEIGH notational conventions defined.

Programming Notes Section

This section shows the supplementary information about programming when using the instruction.

2. Instruction Set

PUSH.V: PUSH a Value

Operation Code

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i

i	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i

Format

PUSH.V ri

Description

Pushes a value to the stack.

Operations

```
push(full_ext:4);
```

PUSH..L: PUSH a given relocated Label

Operation Code

1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Format

PUSH.L la

Description

Pushes a relocated label(address pointer) to the stack.

Operations

```
push(LABEL02:4);
```

Programming notes

The relocation formula is $0x10 + (l >> 1)$

PUSH.A: PUSH and Add

Operation Code

r	r	1	1	0	0	0	0	0	0	0	0	0	0	0	0	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i

Format

PUSH.A rn, ri

Description

Pushes to the stack the value (rn+ri).

Operations

PUSH.AP: PUSH and Add to Pointer

Operation Code

r	r	1	0	0	0	0	0	0	0	0	0	0	0	0	0	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i

Format

PUSH.AP rn, ri

Description

Pushed to the stack a pointer toward (rn+ri).

Operations

POP.A: POP and Add

Operation Code

r	r	0	0	0	0	0	1	0	0	0	0	0	0	0	0	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i

Format

POP.A rn, ri

Description

Pops the last value from the stack to the address (rn+ri).

Operations

POP.L: POP at a given relocated Label

Operation Code

1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Format

POP.L la

Description

Pops the latest value from the stack and stores it at the relocated label(address pointer) la.

Operations

```
push(LABEL02:4);
```

Programming notes

The relocation formula is $0x10 + (l >> 1)$. This opcode is never used in practice as the only way to use this opcode is to modify the AI's own ram region, which would create self-modifying code.

CFTI: Convert Float To Int

Operation Code

0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0

Format

CFTI

Description

Retrieves the last value pushed on to the stack and converts it from a signed integer to a floating point value, pushing back the result to the stack.

Operations

```
local tmp:4 = sp;
pop(tmp);
tmp = round(tmp);
push(tmp);
```

NEG: convert to NEGative signed number

Operation Code

1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0

Format

NEG

Description

Retrieves the last value pushed on to the stack and converts it to a negative number, pushing back the result to the stack.

Operations

```
local tmp:4 = sp;
pop(tmp);
tmp=-tmp;
push(tmp);
```

INV: INVert an unsigned value

Operation Code

1	1	0	0	0	1	0	1	0	0	0	0	0	0	0	0

Format

INV

Description

Retrieves the last value pushed on to the stack and inverts it, pushing back the result to the stack.

Operations

```
local tmp:4 = sp;
pop(tmp);
tmp = ~tmp;
push(tmp);
```

EQZ: conditional is EQual Zero

Operation Code

0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1

Format

EQZ

Description

Retrieves the last value pushed on to the stack and compares it to zero, pushing back the result to the stack.

Operations

```
local tmp:4 = sp;
pop(tmp);
local ret = (tmp == 0);
push(ret);
```

ABS: convert to ABSolute value

Operation Code

0	1	0	0	0	1	0	1	0	0	0	0	0	0	0	1

Format

ABS

Description

Retrieves the last value pushed on to the stack and converts it to an absolute value, pushing back the result to the stack.

Operations

```
local tmp:4 = sp;
pop(tmp);
if(tmp s<= 0) goto <min>;
goto <done>;
<min>
    tmp=-tmp;
<done>
    push(tmp);
```

MSB: return Most Significant Bit

Operation Code

1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1

Format

MSB

Description

Retrieves the last value pushed on to the stack and gets back its most significant bit, pushing back the result to the stack.

Operations

```
local tmp:4 = sp;
pop(tmp);
tmp = tmp >> 0x1F;
push(tmp);
```

INFO: conditional INFerior to One

Operation Code

1	1	0	0	0	1	0	1	0	0	0	0	0	0	0	1

Format

INFO

Description

Retrieves the last value pushed on to the stack and compares it to one, pushing back the result to the stack.

Operations

```
local tmp:4 = sp;
pop(tmp);
push((tmp s< 1));
```

NEQZ: conditional Not Equal to Zero

Operation Code

0	1	0	0	0	1	0	1	0	0	0	0	0	0	1	0

Format

NEQZ

Description

Retrieves the last value pushed on to the stack and compares it to zero, pushing back the result to the stack.

Operations

```
local tmp:4 = sp;
pop(tmp);
push((tmp != 0));
```

MSBI: return Most Significant Bit Inverted

Operation Code

1	0	0	0	0	1	0	1	0	0	0	0	0	0	1	0

Format

MSBI

Description

Retrieves the last value pushed on to the stack and gets back its most significant bit and inverts it, pushing back the result to the stack.

Operations

```
local tmp:4 = sp;
pop(tmp);
tmp = tmp >> 0x1F;
push(~tmp);
```

IPOS: Conditional Is POSitive

Operation Code

1	1	0	0	0	1	0	1	0	0	0	0	0	0	1	0

Format

IPOS

Description

Retrieves the last value pushed on to the stack and compares it to zero, pushing back the result to the stack.

Operations

```
local tmp:4 = sp;
pop(tmp);
push((tmp s> 0));
```

CITF: Convert Int To Float

Operation Code

0	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0

Format

CITF

Description

Retrieves the last value pushed on to the stack and converts it from a signed integer to a floating point value, pushing back the result to the stack.

Operations

```
local tmp:4 = sp;
pop(tmp);
tmp=int2float(tmp);
push(tmp);
```

NEGF: convert to NEGative Float

Operation Code

1	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0

Format

NEGF

Description

Retrieves the last value pushed on to the stack and converts it to a negative value, pushing back the result to the stack.

Operations

Programming notes

This function exclusively deals with floating numbers

ABSF: convert to ABSolute Float

Operation Code

0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	1

Format

ABSF

Description

Retrieves the last value pushed on to the stack and converts it to an absolute value, pushing back the result to the stack.

Operations

```
local tmp:4 = sp;
pop(tmp);
tmp=abs(tmp);
push(tmp);
```

Programming notes

This function exclusively deals with floating numbers

INFZF: Conditional INFerior to Zero Float

Operation Code

1	0	0	1	0	1	0	1	0	0	0	0	0	0	0	1

Format

INFZF

Description

Retrieves the last value pushed on to the stack and compares it to zero, pushing back the result to the stack.

Operations

```
local tmp:4 = sp;
pop(tmp);
push((tmp f< 0));
```

Programming notes

This function exclusively deals with floating numbers

INFOEZF: Conditional INFerior Or Equal to Zero Float

Operation Code

1	1	0	1	0	1	0	1	0	0	0	0	0	0	0	1

Format

INFOEZF

Description

Retrieves the last value pushed on to the stack and compares it to zero, pushing back the result to the stack.

Operations

```
local tmp:4 = sp;
pop(tmp);
push((tmp f<= 0));
```

Programming notes

This function exclusively deals with floating numbers

EQZF: conditional is EQual Zero Float

Operation Code

0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	0

Format

EQZF

Description

Retrieves the last value pushed on to the stack and compares it to zero, pushing back the result to the stack.

Operations

```
local tmp:4 = sp;
pop(tmp);
push((tmp f== 0));
```

Programming notes

This function exclusively deals with floating numbers

NEQZF: conditional Not Equal to Zero Float

Operation Code

0	1	0	1	0	1	0	1	0	0	0	0	0	0	1	0

Format

NEQZF

Description

Retrieves the last value pushed on to the stack and compares it to zero, pushing back the result to the stack.

Operations

```
local tmp:4 = sp;
pop(tmp);
push((tmp f!= 0));
```

Programming notes

This function exclusively deals with floating numbers

SUPOEZF: conditional SUPerior Or Equal to Zero Float

Operation Code

1	0	0	1	0	1	0	1	0	0	0	0	0	0	1	0

Format

SUPOEZF

Description

Retrieves the last value pushed on to the stack and compares it to zero, pushing back the result to the stack.

Operations

```
local tmp:4 = sp;
pop(tmp);
push((tmp f>= 0));
```

Programming notes

This function exclusively deals with floating numbers

SUPZF: conditional SUPerior to Zero Float

Operation Code

1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	0

Format

SUPZF

Description

Retrieves the last value pushed on to the stack and compares it to zero, pushing back the result to the stack.

Operations

```
local tmp:4 = sp;
pop(tmp);
push((tmp f> 0));
```

Programming notes

This function exclusively deals with floating numbers

ADD: ADDition

Operation Code

0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0

Format

ADD

Description

Retrieves the last 2 values pushed on to the stack and applies an addition between them, pushing back the result to the stack.

Operations

```
local tmp:4 = sp;
local tmp2:4 = sp;
pop(tmp);
pop(tmp2);
push(tmp2+tmp);
```

SUB: SUBstraction

Operation Code

0	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0

Format

SUB

Description

Retrieves the last 2 values pushed on to the stack and applies a substraction between them, pushing back the result to the stack.

Operations

```
local tmp:4 = sp;
local tmp2:4 = sp;
pop(tmp);
pop(tmp2);
push(tmp2-tmp);
```

MUL: MULtiplication

Operation Code

1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0

Format

MUL

Description

Retrieves the last 2 values pushed on to the stack and applies a multiplication between them, pushing back the result to the stack.

Operations

```
local tmp:4 = sp;
local tmp2:4 = sp;
pop(tmp);
pop(tmp2);
push(tmp2 * tmp);
```

DIV: DIVision

Operation Code

1	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0

Format

DIV

Description

Retrieves the last 2 values pushed on to the stack and applies a division between them, pushing back the result to the stack.

Operations

```
local tmp:4 = sp;
local tmp2:4 = sp;
pop(tmp);
pop(tmp2);
push(tmp2 s/ tmp);
```

MOD: MODulo

Operation Code

0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1

Format

MOD

Description

Retrieves the last 2 values pushed on to the stack and applies a modulo between them, pushing back the result to the stack.

Operations

```
local tmp:4 = sp;
local tmp2:4 = sp;
pop(tmp);
pop(tmp2);
push(tmp2 s% tmp);
```

AND: logical AND

Operation Code

0	1	0	0	0	1	1	0	0	0	0	0	0	0	0	1

Format

AND

Description

Retrieves the last 2 values pushed on to the stack and applies a logical and between them, pushing back the result to the stack.

Operations

```
local tmp:4 = sp;
local tmp2:4 = sp;
pop(tmp);
pop(tmp2);
push(tmp2&tmp);
```

OR: logical OR

Operation Code

1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1

Format

OR

Description

Retrieves the last 2 values pushed on to the stack and applies a logical or between them, pushing back the result to the stack.

Operations

```
local tmp:4 = sp;
local tmp2:4 = sp;
pop(tmp);
pop(tmp2);
push(tmp2|tmp);
```

XOR: logical eXclusive OR

Operation Code

1	1	0	0	0	1	1	0	0	0	0	0	0	0	0	1

Format

XOR

Description

Retrieves the last 2 values pushed on to the stack and applies an exclusive or between them, pushing back the result to the stack.

Operations

```
local tmp:4 = sp;
local tmp2:4 = sp;
pop(tmp);
pop(tmp2);
push(tmp2^tmp);
```

SLL: Shift Logical Left

Operation Code

0	0	0	0	0	1	1	0	0	0	0	0	0	0	1	0

Format

SLL

Description

Retrieves the last 2 values pushed on to the stack and applies a left logical shift between them, pushing back the result to the stack.

Operations

```
local tmp:4 = sp;
local tmp2:4 = sp;
pop(tmp);
pop(tmp2);
push(tmp2<<(tmp&0x1F));
```

SRA: Shift Right Arithmetic

Operation Code

0	1	0	0	0	1	1	0	0	0	0	0	0	0	1	0

Format

SRA

Description

Retrieves the last 2 values pushed on to the stack and applies a right arithmetic shift between them, pushing back the result to the stack.

Operations

```
local tmp:4 = sp;
local tmp2:4 = sp;
pop(tmp);
pop(tmp2);
push(tmp2>>(tmp&0x1F));
```

NEQZV: conditional Not EQual to Zero with stack Values

Operation Code

1	1	0	0	0	1	1	0	0	0	0	0	0	0	1	0

Format

NEQZV

Description

Retrieves the last 2 values pushed on to the stack and verifies if both are equal to zero, pushing back the result to the stack.

Operations

```
 local tmp:4 = sp;
 local tmp2:4 = sp;
 pop(tmp);
 pop(tmp2);
 local ret:4 = 1;
 if(tmp==0) goto <next>;
 goto <end>;
<next>
    if(tmp2!=0) goto <end>;
 ret=0;
<end>
    push(ret);
```

EQZV: conditional EQual to Zero with stack Values

Operation Code

1	0	0	0	0	1	1	0	0	0	0	0	0	0	1	0

Format

EQZV

Description

Retrieves the last 2 values pushed on to the stack and verifies if both are equal to zero, pushing back the result to the stack.

Operations

```
local tmp:4 = sp;
local tmp2:4 = sp;
pop(tmp);
pop(tmp2);
local ret:4 = 1;
if(tmp!=0) goto <next>;
goto <end>;
<next>
     if(tmp2==0) goto <end>;
ret=0;
<end>
     push(ret);
```

ADDF: ADDition with Float values

Operation Code

0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0

Format

ADDF

Description

Retrieves the last 2 values pushed on to the stack and apply an addition onto them, pushing back the result to the stack.

Operations

```
local tmp:4 = sp;
local tmp2:4 = sp;
pop(tmp);
pop(tmp2);
push(tmp2 f+ tmp);
```

Programming notes

This function exclusively deals with floating numbers

SUBF: SUBstraction with Float values

Operation Code

0	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0

Format

SUBF

Description

Retrieves the last 2 values pushed on to the stack and apply a substraction onto them, pushing back the result to the stack.

Operations

```
local tmp:4 = sp;
local tmp2:4 = sp;
pop(tmp);
pop(tmp2);
push(tmp2 f- tmp);
```

Programming notes

This function exclusively deals with floating numbers

MULF: MULtiplication with Float values

Operation Code

1	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0

Format

MULF

Description

Retrieves the last 2 values pushed on to the stack and apply a multiplication onto them, pushing back the result to the stack.

Operations

```
local tmp:4 = sp;
local tmp2:4 = sp;
pop(tmp);
pop(tmp2);
push(tmp2 f* tmp);
```

Programming notes

This function exclusively deals with floating numbers

DIVF: DIVision with Float values

Operation Code

1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0

Format

DIVF

Description

Retrieves the last 2 values pushed on to the stack and apply a division onto them, pushing back the result to the stack.

Operations

```
local tmp:4 = sp;
local tmp2:4 = sp;
pop(tmp);
pop(tmp2);
push(tmp2 f/ tmp);
```

Programming notes

This function exclusively deals with floating numbers

MODF: MODulo with Float values

Operation Code

0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	1

Format

MODF

Description

Retrieves the last 2 values pushed on to the stack and apply a modulo onto them, pushing back the result to the stack.

Operations

```
local tmp:4 = sp;
local tmp2:4 = sp;
pop(tmp);
pop(tmp2);
# primitive doesn't exist, so we do with what we can
local ret:4 = fmod(tmp2, tmp);
push(ret);
```

Programming notes

This function exclusively deals with floating numbers

JMP: JuMP

Operation Code

i	i	0	0	1	0	0	0	i	i	i	i	i	i	i	i	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a

Format

JMP ri, addr

Description

Change the control flow to the given address addr and saves the instruction following it as the return pointer.

Operations

```
call LABEL8;
```

Programming notes

Argument ri is currently unknown. The following address relocation formula is applied when decoding a into addr: $addr = inst\_start + (a * 2) + 4$ where inst_start is the beginning of the instruction.

EXIT: EXIT

Operation Code

0	0	0	0	1	0	0	1	i	i	0	0	0	0	0	0

Format

EXIT ri

Description

Completely stops the execution flow of the AI Parser with return code ri

Operations

```
exit(iarg:1);
```

Programming notes

In the bitwise encoding ri is encoded as $r = ri - 1$

RET: RETurn

Operation Code

1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0

Format

RET

Description

Stops the execution flow and return back to the last saved function call

Operations

```
return[ra];
```

PUSH.CA: PUSH CAched value

Operation Code

1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0

Format

PUSH.CA

Description

Pushes the last cached stack value to the stack

Operations

Programming notes

This seems to have the same effect as PUSH.C but without doing a POP. I have no clue why both of those instructions exist alongisde another.

PUSH.C: PUSH Copy

Operation Code

0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	1

Format

PUSH.C

Description

Pops the latest value from the stack and pushes it back twice

Operations

SIN: SINus

Operation Code

1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1

Format

SIN

Description

Retrieves the latest value pushed to the stack and apply a sinus onto it, pushing it to the stack

Operations

```
local tmp:4 = sp;
pop(tmp);
local ret:4 = sin(tmp);
push(ret);
```

Programming notes

Radians are used as input. Radians used are modulo $[\pi - 2\pi]$

COS: COSinus

Operation Code

1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	1

Format

COS

Description

Retrieves the latest value pushed to the stack and apply a cosinus onto it, pushing it to the stack

Operations

```
local tmp:4 = sp;
pop(tmp);
local ret:4 = cos(tmp);
push(ret);
```

Programming notes

Radians are used as input. Radians used are modulo $[\pi - 2\pi]$

DEGR: DEGrees to Radians

Operation Code

0	0	0	0	1	0	0	1	0	0	0	0	0	0	1	0

Format

DEGR

Description

Retrieves the last element pushed to the stack and converts it to radians, pushing it to the stack

Operations

```
local tmp:4 = sp;
pop(tmp);
local ret:4 = degrees_to_radians(tmp);
push(ret);
```

Programming notes

Radians used are modulo [$\pi - 2\pi$]

RADD: RADians to Degrees

Operation Code

0	1	0	0	1	0	0	1	0	0	0	0	0	0	1	0

Format

RADD

Description

Retrieves the last element pushed to the stack and converts it to degrees, pushing it to the stack

Operations

```
local tmp:4 = sp;
pop(tmp);
local ret:4 = radians_to_degrees(tmp);
push(ret);
```

Programming notes

Radians used are modulo $[\pi - 2\pi]$

SYSCALL: SYStem CALL

Operation Code

i	i	0	0	1	0	1	0	0	0	0	0	0	0	0	0	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a

Format

syscall ri, ra

Description

Executes a System Call, using the stack as arguments

Operations

```
system_call(opesub:4,ope2:4);
```

Programming notes

Refer to the syscall own documentation chapter for more information about this instruction.

3. System Calls

Introduction

KH2AI has an instruction used to call some functions into the base game, which we call syscall, short for System Call. None of them are currently documented, they are available at address 0x0034dd00 of SLPM_666.75, which is Kingdom Hearts 2 Final Mix ELF file. If you want to contribute you can submit your syscall findings at `https://framaforms.org/kh2ai-report-errata-1577102965` for them to be incorporated into the next release of the ISA.

4. Known issues

As this is very much a work-in-progress project, much of the ISA has yet to stabilize before getting a stable documentation and some issues still exist. You will find below some of those.

Notable amount of undocumented instructions

While the disassemblers knows the size of all instructions and is able to get a complete unbroken output, some functions are still partly or fully unknown and as such cannot be assembled yet, nor are they understood by the decompiler. Such instructions will most likely have "unk" in their name.

syscalls function pointers breaks X-Refs

Sometimes, syscalls take for arguments function pointers. An analyzer has been created to be able to analyze this specific case but I have been unable to find a way to get a similar instruction but resolving the relocation without breaking the assembler. As such pointers are written down as comments next to the instruction. You would have to use those to verify X-Refs until a better solution is found.

5. Appendix

SLEIGH additional notational convention

```
define space ram type=ram_space size=4 wordsize=1
    default;
define space register type=register_space size=4;

# this is obviously wrong and will need to be edited
    when i understand how
# internal regs are used besides stack
define register offset=0 size=4 [
    r0 r1 r2 r3 r4 r5 r6
    r7 r8 pc sp ra broken
];

define token instr(16)
    opcode = (0, 3)
    ssub_opc = (6, 15)
    sub_opc = (4, 5)
    iarg = (14, 15);

define token instr_ext(32)
    opcode_ext = (0, 3)
    sub_opc_ext = (4, 5)
    opesub = (6, 7)
    _opesub = (6, 7)
    rn = (6, 7)
    ope3 = (6, 15)
    full_ext = (0, 31)
    full_rel = (0, 31) signed
# the label thingy
    ope2 = (16, 31)
    _ope2 = (16, 31)
    ope2s = (16, 31) signed;

# relocated labels
LABEL8: reloc is ope2s  [ reloc = inst_start+(ope2s*2)
    +4; ]      { export *:4 reloc; }
LABEL02: reloc is ope2s [ reloc = 0x10+(ope2s*2); ]
    { export *:4 reloc; }
#LABELV: reloc is full_rel      [ reloc = 0x10+(full_rel
    *2); ]  { export *:4 reloc; }
NOT_LABEL03: reloc is ope2s     [ reloc = 0x10+(ope2s*2)
    ; ]      { tmp:4 = reloc:4; export tmp; }
#CLABEL: reloc is full_ext      [ reloc = 0x10+(full_ext
    *2); ]  { export *:4 reloc; }
```

```
# if i'm not mistaken 0x1da4d8 1 and 2 uses two regs
# one of them or more is a status reg so i'll have to double check how it's used
attach variables [ rn ] [ r0 r1 r2 r3 ];

# exit values
attach values [ iarg ] [ 1 2 _ _ ];


define pcodeop system_call;
define pcodeop fmod;
define pcodeop exit;
define pcodeop cos;
define pcodeop sin;
define pcodeop radians_to_degrees;
define pcodeop degrees_to_radians;


macro push(v) {
    *[ram]:4 sp = v;
    sp = sp + 4;
}

macro pop(v) {
    sp = sp - 4;
    v = *[ram]:4 sp;
}

macro to_address(v) {
    if(v!=0) goto <address>;
        v=0;
        goto <end>;
    <address>
        v=0x10+(v*2);
    <end>
}
```