cnhasm – Assembly Reference

NOTE - This documentation is an excerpt from the full final report my group and I wrote for the Rose-Hulman CSSE232 final project.

Contents

A.7.1: GENERAL SYNTAX

The assembly dialect uses a simplified syntax and instruction set but shares similarities with other architecture assembly languages. The type of instruction will dictate the generalized syntax form, but exceptions to these rules do exist.

A.7.2: GENERAL INSTRUCTION SYNTAX

All instructions will be of a general form similar to the following:

```
[operation][operand 1](,[operand2](,[operand3]))
```

Each instruction will have one instruction, then up to three comma-delimited operands depending on the instruction. Any amount of whitespace separating the operation and first operand can be used. Lines are differentiated by the assembler with the OS-specific new-line character; therefore, no other terminating character is needed after the last operand.

A.7.3: LABELS

To make procedure-based programming and jump syntax easier for the programmer, standard assembly label syntax can be used like so:

```
[label]:
      [instruction]
      [instruction]
      …
```

The assembler before translating instructions will search the source code (in each respective section) for labels and determine the address in memory they would reside in. Kernel code and program code cannot use labels located in the other memory section, and no two labels should have the same name. Labels can be a string of alphanumeric characters, hyphens, or underscores of any length, that is immediately followed by a colon.

A.7.4: ASSEMBLY DIRECTIVES

The assembler supports only two specific assembler directives: `.ktext` and `.text`. The `.ktext` directive indicates that all instructions below it when assembled reside in the respective section in memory. The `.text` directive serves a similar function as the `.ktext` directive except all instructions below it reside in the `.text` section. The main purpose of these directives is to help separate kernel and program code, which can be useful for giving a common location for commonly used procedures.

A.7.5: COMMENTS

Comments always start with # and will have the assembler ignore all characters after it on the same line. This is the only way to make comments and no multiline or in-line comment syntax explicitly exists.

A.7.6: ASSEMBLER

The assembler program (`chnasm`) is a cross-platform assembler written in C# using the .NET Core 3.1 Framework. While only tested on a Windows target machine (Win-x86), it is most likely possible that the assembler source can be recompiled targeting other platforms (e.g., Linux). The assembler is invoked as follows (assuming a Windows machine):

```
>chnasm.exe <path_to_asm> <path_of_output>
```

If there is a syntax error, the assembler will throw an exception giving some details on what the possible error is. While the regular expressions created for the assembler have been tested under multiple possible scenarios, there may be a rare instance where the program is unable to parse correct code. The best workaround for this is to alter the formatting of the offending line slightly or to write alternative code (if possible).

A.7.7: PROCEDURE CALL CONVENTIONS

A.7.7.1: CALLER RESPONSIBILITIES

Before a procedure call, the caller is expected to save all values in the general-purpose registers (`$r0`-`$r4`) if needed after the procedure call, the return address register (`$ra`), and the stack frame pointer value (`$fp`). The programmer must always save the value of the frame pointer (`$fp`) and it should be the last register saved so that the called procedure at the end of its execution can restore the original value before returning to the caller. The value of the frame pointer (`$fp`) should then be placed at the current stack pointer (`$sp`) value.

If a called procedure requires parameters, they can either be stored in the general-purpose registers before transferring control to the called procedure or added after the frame pointer (`$fp`) has been set for the called procedure. It is the programmer's discretion to determine how they want to transfer parameters depending on the context of the procedure call. The option to use the stack is useful because the scope of stack operations is only within a procedure's stack frame, allowing flexibility in programs needing more parameters to pass-through a procedure that exceeds the number of general-purpose registers.

The only responsibility that the caller has after a procedure call is to restore the value of the return address register as the callee cannot restore it before returning without jumping to the caller's parent procedure. As described in the Callee Responsibilities below, the callee should move this value to the top of the stack to allow for easy "disposal" after restoring the register's value.

A.7.7.2: CALLEE RESPONSIBILITIES

If the caller properly prepares for a procedure call, this focuses the responsibilities of the callee to a proper procedure return. When preparing to return to the caller function, the callee must always restore the old frame pointer (`$fp`) value (which should be at the current frame pointer (`$fp`) address). This should always be done first, and the old frame pointer value should not remain in the stack after this step. If the procedure is to return a value, the same options that the caller has for parameters is available: storing in the general-purpose registers or at the top of the stack. It should be noted that the implication of storing the return values on top of the stack implies that the values are stored in the caller function's stack frame assuming the frame pointer is restored before this stage.

It is important to remember that because the callee needs to know where in the caller code to return, the return address register (`$ra`) is used to store the desired address. However, the callee is using a newer value of the return address register (`$ra`) that is required to properly return to the caller procedure. Because of this issue, it is imperative that the callee still makes restoring the return address register easy for the caller after the return. The best solution is to move the preserved value of `$ra` to the top of the stack such that the caller can execute a `POP $ra` instruction to restore the value while removing it from the stack in one instruction.

A.7.8: INSTRUCTION OVERVIEW

The architecture supports the following operations overall. More specific details can be found in the pages after this list:

- `PUSH` (Push to Stack)
- `POP` (Pop from Stack)
- `LOAD` (Load Word from Memory)
- `STORE` (Store Word into Memory)
- `RDIO` (Read Word from I/O Input Buffer)
- `WRIO` (Store Word to I/O Output Buffer)
- `ADD` (Signed Integer Addition)
- `ADDI` (Signed Integer Addition with Immediate)
- `SUB` (Integer Subtraction)
- `LSL` (Logical Shift Left)
- `LSR` (Logical Shift Right)
- `AND` (Bitwise AND)
- `ANDI` (Bitwise AND with Immediate)
- `OR` (Bitwise OR)
- `ORI` (Bitwise OR with Immediate)
- `XOR` (Bitwise Exclusive OR)
- `XORI` (Bitwise Exclusive OR with Immediate)
- `NOT` (Bitwise NOT)
- `SLT` (Set Less Than)
- `JMP` (Unconditional Branch)
- `JEQ` (Branch if Equal)
- `JNE` (Branch if Not Equal)
- `JPRC` (Procedure Call Branch)
- `JRET` (Procedure Call Return)

A.7.8.1: MEMORY/STACK OPERATIONS

These instructions relate to interaction with the architecture's memory module and stack construct.

A.7.8.1.1: LOAD

Instruction Type: I-Type, Opcode: `0x03`

Syntax: `LOAD [$reg] [offset]`

RTL: `R[$reg] = M [$sp – ZeroExt(offset)]`

This instruction loads the word stored `offset` bytes from the current stack pointer address `$sp` in memory. The value of offset is treated as an unsigned integer to prevent memory leaks, hence the zero-extension of the immediate.

A.7.8.1.2: STORE

Instruction Type: I-Type, Opcode: `0x04`

Syntax: `STORE [$reg] [offset]`

RTL: `M[$sp – ZeroExt(offset)] = R[$reg]`

This instruction writes the word at the memory address defined by `offset` bytes from the current stack pointer address `$sp`. The value of offset is treated as an unsigned integer to prevent memory leaks, hence the zero-extension of the immediate.

A.7.8.1.3: RDIO

Instruction Type: R-Type (Single Register), Opcode: `0x0B`

Syntax: `RDIO [$Reg]`

RTL: `R[$Reg] = M[0xFFFF]`

This is a specialized form of the LOAD instruction that instead reads memory at the address 0xFFFF, which is specifically mapped to the processor's I/O bus. This instruction sets the memory module's read address to 0xFFFF, which will output the value in the processor's input bus. This value written to the register `$[Reg]`.

A.7.8.1.4: WRIO

Instruction Type: R-Type (Single Register), Opcode: `0x0C`

Syntax: `WRIO [$Reg]`

RTL: `M[0xFFFF] = R[$Reg]`

This is a specialized form of the LOAD instruction that instead reads memory at the address 0xFFFF, which is specifically mapped to the processor's I/O bus. This instruction sets the memory module's read address to 0xFFFF, which will output the value in the processor's input bus. This value written to the register `$[Reg]`.

A.7.8.2: ARITHMETIC/LOGIC OPERATIONS

These operations are for performing arithmetic or logical operations on values.

A.7.8.2.1: ADD

Instruction Type: R-Type (Register/Immediate), Opcode: `0x10`
Syntax (Register Form): `ADD $[Reg1], $[Reg2], $[Reg3]`
RTL (Register Form): `R[Reg1] = R[Reg2] + R[Reg3]`
Syntax (Immediate Form): `ADD $[Reg1], $[Reg2], [imm'4b]`
RTL (Immediate Form): `R[Reg1] = R[Reg2] + ZeroExt(imm'4b)`

This instruction performs the signed addition of the values in the two general purpose registers `$[Reg2]` and `$[Reg3]` (Register Form Syntax) or the signed addition between the value in register `$[Reg2]` and a 4-bit zero-extended immediate (Immediate Form Syntax) and places the sum in the specified destination register `$[Reg1]`.

A.7.8.2.2: ADDI

Instruction Type: I-Type, Opcode: `0x07`
Syntax: `ADDI [$Reg] [imm'8b]`
RTL: `R[$Reg] = R[$Reg] + SignExt(imm'8b)`

This instruction performs the signed addition of the value in [`$Reg]` and the sign-extended form of the 8-bit immediate. The result is placed back into the register `[$Reg]`.

A.7.8.2.3: SUB

Instruction Type: R-Type (Register/Immediate), Opcode: `0x11`
Syntax (Register Form): `ADD $[Reg1], $[Reg2], $[Reg3]`
RTL (Register Form): `R[Reg1] = R[Reg2] + R[Reg3]`
Syntax (Immediate Form): `ADD $[Reg1], $[Reg2], [imm'4b]`
RTL (Immediate Form): `R[Reg1] = R[Reg2] + ZeroExt(imm'4b)`

This instruction performs the subtraction of the two general purpose registers (Register Form Syntax) or the subtraction between a register and a 4-bit zero-extended immediate (Immediate Form Syntax) and places the result in the specified destination register.

A.7.8.2.4: LSL

Instruction Type: R-Type (Immediate), Opcode: `0x12`
Syntax (Immediate Form): `LSL $[Reg1], $[Reg2], [imm'4b]`
RTL (Immediate Form): `R[Reg1] = R[Reg2] << ZeroExt(imm'4b)`

This instruction performs the logical left shift operation on the value in `$[Reg2]`. The number of bits that are shifted is specified by the zero-extended immediate. The result of the shift operation is stored in the specified destination register `$[Reg1]`.

A.7.8.2.5: LSR

Instruction Type: R-Type (Immediate), Opcode: `0x13`

Syntax (Immediate Form): `LSR $[Reg1], $[Reg2], [imm’4b]`
RTL (Immediate Form): `R[$Reg1] = R[Reg2] >> ZeroExt(imm’4b)`

This instruction performs the logical right shift operation on the value in `$[Reg2]`. The number of bits that are shifted is specified by the zero-extended immediate. The result of the shift operation is stored in the specified destination register `[$Reg1]`.

A.7.8.2.6: AND

Instruction Type: R-Type (Register/Immediate), Opcode: `0x14`
Syntax (Register Form): `ADD [$Reg1], [$Reg2], [$Reg3]`
RTL (Register Form): `R[$Reg1] = R[$Reg2] & R[$Reg3]`
Syntax (Immediate Form): `ADD [$Reg1], [$Reg2], [imm’4b]`
RTL (Immediate Form): `R[$Reg1] = R[$Reg2] & ZeroExt(imm’4b)`

This instruction performs the bitwise AND operation on the values in the two general purpose registers `[$Reg2]` and `[$Reg3]` (Register Form Syntax) or the value in register `[$Reg2]` and a 4-bit zero-extended immediate (Immediate Form Syntax) and places the result in the specified destination register `[$Reg1]`.

A.7.8.2.7: ANDI

Instruction Type: I-Type, Opcode: `0x08`
Syntax: `ANDI [$Reg] [imm’8b]`
RTL: `R[$Reg] = R[$Reg] & SignExt(imm’8b)`

This instruction performs the bitwise AND of the value in `[$Reg]` and the sign-extended form of the 8-bit immediate. The result is placed back into the register `[$Reg]`.

A.7.8.2.8: OR

Instruction Type: R-Type (Register/Immediate), Opcode: `0x15`
Syntax (Register Form): `OR $[Reg1], $[Reg2], $[Reg3]`
RTL (Register Form): `R[Reg1] = R[Reg2] | R[Reg3]`
Syntax (Immediate Form): `OR $[Reg1], $[Reg2], [imm’4b]`
RTL (Immediate Form): `R[Reg1] = R[Reg2] | ZeroExt(imm’4b)`

This instruction performs the bitwise OR operation on the values in the two general purpose registers `$[Reg2]` and `$[Reg3]` (Register Form Syntax) or the value in register `$[Reg2]` and a 4-bit zero-extended immediate (Immediate Form Syntax) and places the result in the specified destination register `$[Reg1]`.

A.7.8.2.9: ORI

Instruction Type: I-Type, Opcode: `0x09`
Syntax: `ORI [$Reg] [imm’8b]`
RTL: `R[$Reg] = R[$Reg] | SignExt(imm’8b)`

This instruction performs the bitwise OR of the value in `[$Reg]` and the sign-extended form of the 8-bit immediate. The result is placed back into the register `[$Reg]`.

A.7.8.2.10: XOR

Instruction Type: R-Type (Register/Immediate), Opcode: `0x16`

Syntax (Register Form): `XOR $[Reg1], $[Reg2], $[Reg3]`
RTL (Register Form): `R[Reg1] = R[Reg2] ^ R[Reg3]`
Syntax (Immediate Form): `XOR $[Reg1], $[Reg2], [imm'4b]`
RTL (Immediate Form): `R[Reg1] = R[Reg2] ^ ZeroExt(imm'4b)`

This instruction performs the bitwise XOR (Exclusive OR) operation on the values of the two general purpose registers `$[Reg2]` and `$[Reg3]` (Register Form Syntax) or the value in register `$[Reg2]` and a 4-bit zero-extended immediate (Immediate Form Syntax) and places the result in the specified destination register `$[Reg1]`.

A.7.8.2.11: XORI

Instruction Type: I-Type, Opcode: `0x0A`
Syntax: `XORI [$Reg] [imm'8b]`
RTL: `R[$Reg] = R[$Reg] ^ SignExt(imm'8b)`

This instruction performs the bitwise XOR (Exclusive OR) of the value in `[$Reg]` and the sign-extended form of the 8-bit immediate. The result is placed back into the register `[$Reg]`.

A.7.8.2.12: NOT

Instruction Type: R-Type (Single Register), Opcode: `0x17`
Syntax: `NOT [$Reg]`
RTL: `R[$Reg] = ~R[$Reg]`

This instruction performs the bitwise NOT operation on the value in register `[$Reg]` and writes the result back to register `$[Reg]`.

A.7.8.2.13: SLT

Instruction Type: R-Type (Register/Immediate), **Opcode:** `0x00`
Syntax (Register Form): `SLT [$Reg1], [$Reg2], [$Reg3]`
RTL (Register Form): `R[$Reg1] = (R[$Reg2] < R[$Reg3])? 1:0`
Syntax (Immediate Form): `SLT [$Reg1], [$Reg2], [imm'4b]`
RTL (Immediate Form): `R[$Reg1] = (R[$Reg2] < ZeroExt(imm'4b))? 1:0`

This instruction checks if the value in `$[Reg2]` is less than the value in `$[Reg3]` (Register Form Syntax) or the 4-bit zero-extended immediate (Immediate Form Syntax) and places the result (1 if comparison is true, 0 otherwise) in the specified destination register `[$Reg1]`.

While the architecture does not support a unique "set greater than" instruction, the equivalent comparison can be performed by swapping the order of the source operands. For example, if the programmer wants to check if the value in `[$Reg2]` is greater than the value in `[$Reg3]`, the equivalent code would be `SLT [$Reg1], [$Reg3], [$Reg2]`.

A.7.8.3: BRANCH OPERATIONS

These operations are used for function branching and program flow. These are the only way the programmer can interact with the Program Counter.

A.7.8.3.1: JMP

Instruction Type: J-Type, Opcode: `0x00`
Syntax: `JMP [offset]`
RTL: `$PC = $PC + 2 + {5{offset [11]}, offset}`

This instruction performs an unconditional function branch by taking the current Program Counter (PC) value, incrementing it by `0x02`, then modifying the resultant value by the offset via signed integer addition. That resultant sum is then written to the PC.

Because the assembler supports labels, `[offset]` can be replaced with the name of a label in the code. This still requires that the distance from the JMP instruction to the location of the label in code memory is reachable within an 11-bit signed offset.

A.7.8.3.2: JEQ

Instruction Type: J-Type, Opcode: `0x01`
Syntax: `JNE [offset]`
RTL: `if (R[$r0] == R[$r1]) PC = $PC + 2 + {5{offset [11]}, offset}`

This instruction performs a conditional function branch by checking the equality of the values in `$r0` and `$r1`. If `$r0==$r1`, the processor will overwrite the current Program Counter (PC) value to the desired location specified by the `[offset]` from the current JEQ instruction. The math for the location of the branch is detailed in the documentation of the JMP instruction.

Because the assembler supports labels, `[offset]` can be replaced with the name of a label in the code. This still requires that the distance from the JEQ instruction to the location of the label in code memory is reachable within an 11-bit signed offset.

A.7.8.3.3: JNE

Instruction Type: J-Type, **Opcode:** `0x02`
Syntax: `JNE [offset]`
RTL: `if (R[$r0] != R[$r1]) PC = $PC + 2 + {5{offset [11]}, offset}`

This instruction performs a conditional function branch by checking the equality of the values in `$r0` and `$r1`. If `$r0!=$r1`, the processor will overwrite the current Program Counter (PC) value to the desired location specified by the `[offset]` from the current JNE instruction. The math for the location of the branch is detailed in the documentation of the JMP instruction.

Because the assembler supports labels, `[offset]` can be replaced with the name of a label in the code. This still requires that the distance from the JNE instruction to the location of the label in code memory is reachable within an 11-bit signed offset.

A.7.8.3.4: JPRC

Instruction Type: J-Type, **Opcode:** `0x05`
Syntax: `JPRC [offset]`
RTL: `$ra = $PC + 0x02; $PC = $PC + [offset]`

This instruction performs an unconditional function branch that retains the address of the next instruction (after JPRC) in register `$ra` for later use. This procedure is incredibly useful in the context of procedure calls.

Because the assembler supports labels, `[offset]` can be replaced with the name of a label in the code. This still requires that the distance from the JPRC instruction to the location of the label in code memory is reachable within an 11-bit signed offset.

A.7.8.3.4: JRET

Instruction Type: R-Type (Single Register), **Opcode:** `0x05`

Syntax: JRET `[$Reg]`

RTL: `$PC = [$Reg]`

This instruction performs an unconditional function branch by writing the value in the register `[$Reg]` to the PC. This is often used in the context of procedure calls with `$ra`.

A.7.9: COMMON CODE FRAGMENTS

This section details the assembly implementation of common code fragments seen in higher-level programming languages.

A.7.9.1: IF-ELSE CONDITIONAL:

```
#Example Conditional where code should execute if $r0 < $r1
    SLT     $r2, $r0, $r1
    JEQ     $r2, COND_TRUE
#ELSE CODE GOES HERE
    JMP     ALWAYS_EXEC
COND_TRUE:
    #IF CODE GOES HERE
ALWAYS_EXEC:
#Code after Conditional
```

A.7.9.2: FOR LOOP:

```
#Assume $r4 is the loop index, $r3 is loop target
#Values initialized in previously executed code
LOOP_CODE:
    SLT     $r2, $r4, $r3
    JNE     $r2, EXIT_LOOP
#Code that is iterated over in loop
    ADDI    $r4, 0x01
    JMP     LOOP_CODE
EXIT_LOOP:
#Code After Loop Here
```

A.7.9.3: DO-WHILE LOOP:

```
# Code should execute while $r3 < $r4
#Values of $r3, $r4 change inside of DO_CODE
DO_CODE:
# Do Code goes here
    SLT     $r2, $r3, $r4
    JEQ     $r2, DO_CODE
#Non-Loop Code Here
```

A.7.9.4: EXAMPLE CODE (RELATIVE PRIMES VIA EUCLID'S ALGORITHM)

```
.text
main:
    RDIO     $r0
    JPRC     relprime
    WRIO     $r0
    JMP      end

relprime:
    XOR      $r1, $r1, $r1
    ORI      $r1, 1
    XOR      $r2, $r2, $r2
    ORI      $r2, 2
    ADDI     $sp, 8
    STORE    $r0, -8
    STORE    $r1, -6
    STORE    $ra, -2
relloop:
    STORE    $r2, -4
    JPRC     gcd
    LOAD     $r2, -4
    LOAD     $r1, -6
    JEQ      relend
    LOAD     $r0, -8
    ADDI     $r2, 1
    JMP      relloop
relend:
    XOR      $r0, $r0, $r0
    OR       $r0, $r0, $r2
    LOAD     $ra, -2
    ADDI     $sp, 8
    JRET     $ra

gcd:
    # Check a == 0
    XOR      $r1, $r1, $r1
    JNE      gcdcont
    XOR      $r0, $r0, $r0
    OR       $r0, $r0, $r2
    JRET     $ra
gcdcont:
    # Swap a and b
    ADD      $r0, $r2, $r0
    SUB      $r2, $r0, $r2
    SUB      $r0, $r0, $r2
```