# IOPASM Manual

Johan Jörgensen
Johan.Jorgensen@axis.com

May 16, 2006

# Contents

# 1 Introduction to IOPASM

This manual contains a brief introduction to IOPASM, the assembler for the I/O Processor SPUs and MPU. The following issues are covered:

- *IOPASM command line* Describes various command line options and their meaning
- *IOPASM syntax* The assembler syntax
- *Extending IOPASM* A short how-to describing the steps required to add a new processor architecture to IOPASM

# 2 IOPASM usage

IOPASM is the primary CUI[1] based development tool for the I/O Processor. It has the following features

- Fully integrated and seamless C-preprocessing support
- Supports all SPU & MPU instructions
- Fully integrated linker
- FSM memory utilization optimizer
- Rule-based code generators

## 2.1 Command-line switches

The following switches are recognized by the assembler:

−− ***(C preprocessor directives)*** Pass all command line options after switch this to the C preprocessor. Should be specified last on the command line

−−**allow-undefined-symbols** Assign the value 0 to all symbols that is left undefined in the current program

−−**arch (-M)** ***arch*** Select architecture. ***Arch*** is the name of the architecture and can be either *MPU* or *SPU*

−−**assemble-only (-c)** Instructs the assembler to only run through the parser and intermediate code-generation stages. The output result will always be an object file *(Implies: -t object)*

−**e** ***error-reference*** Causes the configurable error referenced by *error-reference* to generate an error i.e. *-eunaligned-64bit-inst*

[1] *C*haracter based *U*ser *I*nterface

−−**generate-c-macros** ***file*** Generate C-macros for the specified architecture. These C-macros can be used to generate op-codes in c-programs for a specific architecture (see section 2.4 for further details)

−−**include-path (-I)** IOPASM does not use any-predefined paths to include-files per default. Use this switch to add a path to the cpp directives

−−**list-file** ***file*** Generate a program listing and store it in ***file***. The list file shows the code generated by the assembler for a specific input-file and architecture

−−**link-only (-l)** Invoke the integrated linker and skip the parser. This is used to link several object files into one binary file

−**m** ***machine−option*** Pass architecture specific option to the back-end and switch it on i.e. *-mauto-align-64bit-inst*

−**n** ***error−reference*** The configurable error referenced by *error-reference* will be ignored and neither an error nor a warning is emitted (please see −e and −w)

−−**namespace** ***namespace*** Set the name space for symbols. This is done by appending *namespace* to all symbols defined in the program

−−**no-cpp** Do not use the C pre-processor

−−**optimize-fsm** Invoke the FSM memory optimizer

−−**output-file (-o)** ***file*** Save output in ***file*** *(required)*

−−**output-format (-t)** ***format*** Specify output-format. One of the following possible formats may be specified:

**binary** Plain binary format, for distribution use

**object** Save output as object-file.

**vlog64** Save output in VCS-readable format (64-bits/row)

**vlog256** Save output in VCS-readable format (256-bits/row)

−−**preload-object** ***file*** Load object file ***file*** prior to running source through parser

−−**preload-symbol-file** ***file*** Load the symbol file named ***file*** prior to parsing the source file

−−**symbol-file (-s)** ***file*** Specify name of symbol-file

−−**symbol-file-format** ***format*** Specify output format for symbol file specified with −−*symbol-file*.. The following formats are supported:

**plain** Simple row-based format used by IOPASM

**header** Save symbol definitions in C-header file

−−**verbose (-v)** Be a little verbose. The assembler will emit additional information about assembler stages, optimization information etc

−**w** ***error-reference*** The configurable error/warning referenced by *error-reference* will cause a warning instead of an error (eg. *-wunaligned-64bit-inst*)

−−**with-builtin-includes** Use the pre-defined built-in include paths when calling the C pre-processor (these are set in the iopasm makefile)

## 2.2 Architecure specific command line options

The tables 1 and 2 list all the machine dependent options that can be passed to the SPU and MPU architecture. All of these options are invoked using the −m switch, and are switch off by default. Th The SPU does not have any machine-

| **Command line switch** | **Explanation** |
|---|---|
| auto-align-64bit-inst | Force the MPU code generator to align 64-bit instructions automatically |

Table 1: MPU specific command line flags

| **Command line switch** | **Explanation** |
|---|---|
| - | *No command line switches has been defined for the SPU* |

Table 2: SPU specific command line flags

specific options at present time. The FSM optimizer is partly built in to the back end, but certain parts of it is architecture independent.

## 2.3 Assembly-language syntax

The assembly-language syntax used by IOPASM is very simple. Two different modes are supported: FSM and sequential mode. The two modes can be mixed in the same source-file seamlessly, provided that the selected architecture supports the FSM mode.

### 2.3.1 Assembler directives

The assembler supports the following directives:

### 2.3.2 Sequential mode syntax

The format of the sequential will not be described in great detail here as it closely resembles that of the GNU Assembler for the CRIS architecture *(gas)*.

| Operand | Description |
|---|---|
| `.align` *boundrary* | Align next instruction to the boundrary specified by *boundrary* (measured in bits) |
| `.dword` *expression* | Convert *expression* to a 32-bit integer and store it at current location |
| `.end` | Mark end of file ***(Required)*** |
| `.fsm` | Switch to FSM mode |
| `.org` *address* | Set PC for next instruction |
| `.seq` | Switch to sequential mode |

Table 3: Assembler directives (pseudo operands

As the reader of this document is assumed to be familiar with assemblers and *gas*, only a brief introduction will be given. For a list of supported instructions and their operands, please refer to [1]. Below is a simple example of sequential code:

```
          .seq                              ; 1 - not necessary
a_label:  moveq 0x10,r0                     ; 2 - Hexadecimal number
          moveq $1A,R2                      ; 3 - Same as above
          moveq 10,r5                       ; 4 - Decimal immediate
          moveq \%100_1001_11,r6            ; 5 - Binary immediate
          addq (((7 + 4 ) << 3) / 2),r1,r2 ; 6 - Complex operand
          addq r1,(((7 + 4) << 3) / 2),r2  ; 7 - operand transforms
1:   nop                                    ; 8 - Local labels are supported
  addq r7,(a_label << 2) + 4, r8   ; 9 - Labels can be used
          moveq a_label,r4                  ; 10 - Labels are immediates
          ba a_label                        ; 11 - Rule precursor
          nop                               ; 12 - Delay slot rules
          ba 1b                             ; 13 - Use of local labels (backward)
  nop                               ; 14 - standard delay slot
  ba  2f                            ; 15 - Use of local labels (forward)
  nop                               ; 16 - Standard delay slot
2:   nop                               ; 17 - Local label definition
          .end
```

The example above examplifies the following features of the assembler:

- Numbers can be written in either hexadecimal, decimal or binary format. The tokens:

  **0x or $** are the prefix for hexadecimal numbers

  % The % sign is the prefix for binary digits. Binary digits can be grouped through use of the _ i.e. the decimal number 160 can be written as %1010_0000 in binary form

  Decimal numbers do not require a prefix

- Operands can be complex arithmetric expressions as long as the expression evaluates to a constant once label addresses have been calculated *(line 6 and 9)*
- The assembler performs operand-transformations. Lines 6 and 7 will generate exactly the same op-code. This is only supported in instructions where the result of the transformation is unambigous
- Labels are generally treated like immediates, however the value of a label is not calculated before stage 6 *line 9*
- Local labels can be reused. Local labels are referenced by combining the number of the label with the letters *b*ackward or *f*orward
- Certain instructions have different rules associated with them. All branch instructions must have an instruction in the delay-slot otherwise a warning is generated.

### 2.3.3 FSM mode syntax

The FSM syntax looks like the following:

```
state_name : optional_flags              ; Optional
             seq sequential instruction  ; Optional
             timer instruction           ; Optional
             0-8 state transitions
```

The supported flags are shown in table The states transitions are built from

| Instruction field | IOPASM Syntax | Description |
|---|---|---|
| break | `fsm_halt` | Set breakpoint flag. Should generally not be used |
| do_seq | `do_seq` | The sequential instruction will be executed every cycle while this state is active. Do not use this flag if no sequential instruction is present |
| go_seq | `go_seq` | Switch to sequential mode |
| sel_inputs | `inp = value` | Set value of input selector. The *value* is a 4-bit number *See [1]* |
| sel_outputs | `outp = value` | Set value of output selector. The *value* is an 8-bit number *See [1]*ă |
| Event Mask | `emask = value` | Set value of the Event mask register. The *value* is a 4-bit number interpreted as a vector *See [1]* |
| Event Update Mask | `umask = value` | Set value of event update field. The *value* is a 4-bit number interpreted as a vector *See [1]* |

Table 4: Supported flags in iopasm

three different components according to the following:

*input : outputs : next_state*

The *input*-field is coded using the values 0, 1 and ?. The interpretation of these values are 0, 1 and do not care respectively.

The *output*-field has an additional specifier: *p* which is short for pulse (a pulse generally lasts one cycle). Specifying a *p* will result in a pulse being generated on the output. An example is shown below

0_1_1_? : ?_0_1_p : state_4

The statement above will be interpreted according to the following:

1. Selected input bits 3, 2, 1 and 0 should be 0, 1, 1, X (do not care) respectively
2. Selected output bits 2 and 1 will be set to 0 and 1 respectively. Bit 3 will remain unchanged and a pulse will be generated on bit 0
3. Next state to be executed is *state_4*

Two special state transition statements can also be used:

- *Only state_4* - Used in conjunction with sequential instructions to set the *seq_only*[2] flag and thus generate a compact instruction
- *Always state_4* - This is a built-in macro that will be expanded to the following: *?_?_?_? : ?_?_?_? : state_4*

The last element of a state transition is the timer element. A timer statement has the following syntax:

*timer timer_value : output_conditions : next_state*

The only new element here is the *timer_value*. This field can be either a register or an integer. Thus both of the following statements are valid timer statements:

```
timer r0 : 1_0_0_0 : state_4
timer 1000 : 0_0_0_0 : state_5
```

The priority of timers is implied by the relative position within the state-description. If timer statements are preceeded by always or state-transition statements the priority is set to zero. This is shown in the example below:

```
state_3 :
          timer r0 : 1_0_0_1 : state_4 ; Timer priority implicitly set to 1

state_4 :
```

[2] For further details see chapter 13 in the ETRAX FS designers reference

```
1_0_0_1 : 1_0_1_1 : state_3
timer 1000 : 1_1_1_0 : state_3 ; Timer priority implicitly set to 0
```

The assembler always starts in sequential mode. As the parser stage is totally independent from the code generation stages FSM-mode will be correctly parsed. However the code generator will report an error if the selected architecture (selected at invocation using the *-M* or *−−arch* swithces) does not support FSM code. The overall structure of a sourcefile is shown in the example below:

```
; The assembler starts in sequential mode per default

foo:     move r0,r1
         addq 10,r1,r2
         ba no_bar ; (bar is branch register)
         nop
         .fsm   ; Switch to fsm mode

state_1: do_seq
         inp = 4
         outp = 0xff
         seq addq 1,r1,r2
         only state_2

state_2:
         timer r2 : ?_1_1_0 : state_1  ; Timer priority implicitly set to 1

         .seq  ; More sequential code
no_bar:  addq r2,10,r2
         ba foo

         .end ; File ends here
```

## 2.4 C Macro generation

The ability to generate C-macros for all instructions and registers defined in the architecture of the SPU and MPUs is a unique feature of IOPASM. IOPASM generates these macros according to a set of built-in rules as certain instructions have different op-codes depending on the type of operand. Three different types of operands are defined:

**I** Argument is an integer

**R** Argument is a register

**S** Argument is a special register

The operand type is concatenated to the macro in order to differentiate macros from each other, thus indicating what type, and how many arguments the macro expects. Each macro is prepended by a prefix which is architecture specific. The prefix for the MPU is *MPU_* and the prefix for the SPU is *SPU_*.

As an example, let us assume that we want to use a macro to generate the MPU-specific opcode for `moveq 0x10,R0`. This leads us to the following macro:

1. The macro should work generate a valid MPU-opcode, so the prefix is *MPU_*
2. The macro should take two [macro] arguments, as the instruction has two operands.
3. The instruction operands are one immediate and one register, so the argument-suffix of the macro should be *_IR*

The macro `MPU_MOVEQ_IR(...)` is thus the correct macro. To generate the opcode for `moveq 0x10,R0` and assign the result to the variable "op_code", the macro should be invoked like this:

```
op_code = MPU_MOVEQ_IR(0x10,R0);
```

The macros does not support any range-checking for immediates. However integers are rounded down. If the integer is out of range, the used value should be considered to be erroneous.

## 2.5 The FSM memory optimizer

This section covers the FSM memory optimizer. The section only gives an introductory overview to the optimization algorithm, as it is deemed important to gain a rough understanding of the algorithm in order to be able to use it. To summarize, the FSM optimizer serves two important purposes:

- Saves SPU memory through memory "hole" elimination
- Removes the limitations imposed by the SPUs banked memory

The "holes" in the memory map is a result of the memory being fragmented due to the fact that the assembler will assemble code in the same sequence as it finds it in the source file. As sequential and FSM code can be freely mixed the result is that the SPU memory is littered with "holes" required to align the various code fragments properly. The first step taken by the optimizer is thus to repack all sequential code at the low end of the memory map. It should be noted that all *org* and *align* instructions are removed. All labels will be recalculated, so the optimization process is opaque to the programmer.

### 2.5.1 The FSM optimization process

The FSM optimization process is a lot more complicated than the sequential memory optimization stage (which is really just a relocation). In short this process can be divided into 5 different steps:

**State identification** During this phase the various states are identified and a state dependency matrix is built

**State interdependency calculation** From the state dependency matrix a number of sets are built. These sets contain all the states that are dependent on each other. A special set for states that have no dependencies are also created. These sets are referred to as optimizer sets

**Optimizer set minimization** During this phase all sets are matched against each other so that $S_0 \cup S_1 \rightarrow S_0 = \{S_0, S_1\}$.

**Set partitioning** The number of sets left after the set minimization process is then partitioned on the remaining amount of memory. The algorithm continuously tries to fit the largest possible set into the remaining area of the memory banks.

**State layout** During this phase states are laid out in memory so that each row in memory is completely filled. States are selected from the partitions given by the *set partitioning* state

Empirical data shows that the state optimization process will save a programmer around 15% of the SPU memory. As an additional benefit the programmer does not have to think about where in the memory the various states are placed making it possible to write code that is totally relocation independent.

The negative side effect induced by the optimizer is that all code is completely reorganized. This makes the code rather hard to follow as the code is no longer formatted in a human readable manner.

# References

[1] Axis Communications AB, *ETRAX FS Designers reference*, http://developer.axis.com/doc/hw/etraxfs/des_ref/des_ref_060120.pdf, 2006

# A Warnings & Errors

Warnings and errors are divided into groups. Tables 6 and 5 list the various errors and warnings and provides a brief explanation for the error. All warnings are configurable and their behaviour can be controlled by the $-e$, $-w$ and $-n$ switches.

| Reference name | Default state | Description |
|---|---|---|
| `dest-reg-usage` | off | Destination register may be used in an incorrect way (i.e. forwarding would use the old register) |
| `delay-slot-usage` | off | Delay slot may hold incorrect insstruction sequence |
| `unaligned-64bit-inst` | off | The 64-bit insstruction is not aligned to a 64-bit boundrary |
| `state-terminated` | off | The state might have been prematurely terminated by a ".fsm" statement |
| `reti-rw-rr-sequence` | Error | rw instruction found in delay slot of "reti". If the next instruction is an "rw" instruction the MPU will crash |

Table 5: Configurable errors, their reference names and their default state

| Error # | Error | Description |
|---|---|---|
| 101 | Unsupported architecture | Architecture specified as an argument to `-M` or `--arch` is not supported |
| 102 | Fork failed | This is an internal IOPASM error indicating that the system-call `fork()` failed. This should *never* happen |
| 103 | cpp execution failed | This is an internal IOPASM error indicating that cpp could not be executed |
| 104 | Report bug | Simple catch-all error, indicating that IOPASMs internal databases are inconsistent |
| 105 | cpp stage failed | cpp did not exit successfully. Probably due to a parser-error |
| 106 | memory region used | An already used memory region is being reused |
| 107 | Invalid code size | Size of generated code does not match the size calculated |
| 108 | Cannot write file | Cannot write to file |
| 109 | Syntax error | Syntax error in source file |
| 110 | Undefined label | Label was not defined |
| 111 | Input file missing | The assembler did not find an input file |
| 112 | Undefined forward local label | Forward referencing local label not found before end of file |
| 113 | Undefined backward local label | Backward referensing local label not found before after start of file |
| 114 | Input file is empty | Assembler invoked on an empty input file |
| 115 | Illegal warning flag | The input file does not contain any code |
| 116 | Illegal warning mode | It is not possible to set the configurable error as specified (deprecated) |
| 117 | Parser mode error | Missing .fsm or .seq statement |
| 118 | Illegal local address label | Use of local address labe l is incorrect (e.g. in a state ) |
| 119 | Extraneous mode switch | |
| 201 | Argument expected | Argument expected for instruction |
| 202 | Unexpected argument | Argument was not expected |
| 203 | Unknown Arg type | Argument type is unknown by Architecture |
| 204 | Illegal Argument | Argument type is not allowed as operand |
| 205 | Dword pseudo instruction evaluates to a non-constant expression | The .dword expression contains a non-constant value such as a register. |
| 206 | Non constant operand | The operand evaluates to a non-constant expression, but a constant (immediate) was expected. |
| 301 | Range error | Immediate is out of range |
| 302 | Immediate not allowed | Immediate not allowed as operand |
| 401 | Too many seq's | Too many sequential instructions in FSM statement |
| 402 | Too many timers | Too many timers in FSM statement |
| 403 | Too many transitions | Too many state-transitions in FSM statement |
| 404 | Unknown FSM error | General error in FSM |
| 405 | Alignment error | Instruction is illegally aligned |
| 406 | Transition conflict | Transitions is in conflict with *only* statement |
| 407 | Timer conflict | Timer is in conflict with FSM instruction |
| 408 | State address is invalid (crosses bank boundrary) | The "next state" part of a state transition statement crosses the memory bank boundrary. |
| 409 | Timer statement is not allowed | The state contains a 32-bit immediate instuction and a timer statement. |
| 410 | Emask register changed in state containing an only state transition | The emask cannot be changed in a state that contains an only state transition. It results in illegal code. |
| 411 | Emask register found in a state containing | |
| 412 | No state transitions | |
| 501 | Unknown register | Register is not defined by iopasm |
| 502 | Register not allowed | A register cannot be used as argument here |