YAM
Yet Another Micro-controller
user manual

Table of Contents

1 Introduction

1.1 Motivation

YAM is primarily a micro-controller aiming at simple yet fast IO sequencing with integer processing capabilities.

The *YAM* specification has been driven by a few basic ideas.

- Scalable data width from 8 bits to any (like 32 and larger) size.
- Possibly a large number of registers (up to 128 of them).
- Deterministic execution times (no pipeline).
- No privileged mode (definitely a micro-controller style).
- Highly configurable to meet different requirements.
- Keep compact in area and preserve fast clocking, avoiding superfluous sophistications.

Therefore the base instruction list is restricted to a minimal set of instructions.
Even the possible extensions have been carefully analyzed against new functionality versus area/speed cost.

YAM architecture is simple:

- the arithmetic and logic unit only operates on registers;
- the data memory access is limited to register load or store.

There is room left in the instruction decoding, so it would be easy to add more instructions to *YAM*. This should only be motivated to address a new crystal clear requirement.

1.2 Presentation

YAM main features are:

- “Harward” architecture with dedicated instruction memory.
- Arithmetic and logic unit with scalable to any data width.
- Rich (up to 128) regular register array, two operand simultaneous access.
- Even more registers and fast context switch by register bank swapping (optional).
- Selectable 2- or 3-operand style ALU instruction set.
- Efficient immediate data management.
- Full word centric ALU (sparse byte management support).
- Program counter and stack for subroutines branch and return.
- Data memory (optional), externally dual ported (optional).
- External interrupt manager (optional).
- External input/output port interface (optional).
- External co-processor interface (optional).
- Deterministic execution time, 2 clock cycles per instruction in most cases.

YAM is extremely scalable since most of its resources are specified by generic parameters (see section 3).

A co-processor interface allows for even more extensions. Available co-processor are

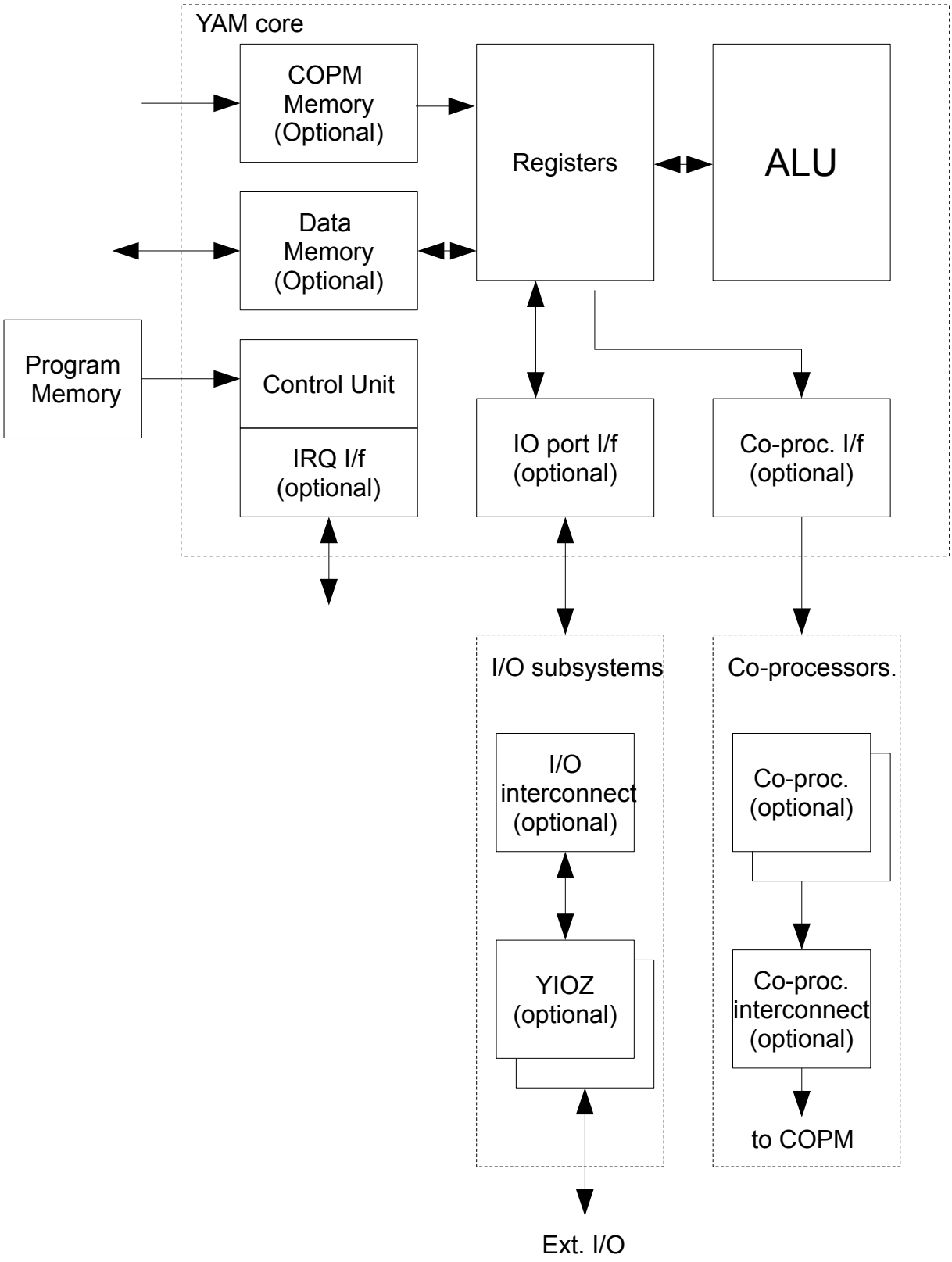
- 32-bit floating point unit.
- Fixed point unit.

External input/output connections are ready for several interfaces

- YAM native;
- Wishbone;
- AXI4 lite;
- E-bone (http://www.ohwr.org/projects/e-bone);
- Basic system-on-chip (timer, breakpoint, UART), namely YIOZ.

YAM has been written in VHDL. It is target technology independent.

YAM architecture



2 YAM component interfacing

2.1 Source files

YAM source files are found in the /hdl/yam directory.

2.1.1 List

yam.vhd	YAM top level
yam_pkg.vhd	YAM package (for components declaration)
yam_imem.vhd	Instruction memory, R/W dual ported
yam_irom.vhd	Instruction memory, read only
yam_dram0.vhd	RAM single port
yam_dram1.vhd	RAM dual symmetrical ports
yam_dram2.vhd	RAM dual asymmetrical ports
yam_dmadrs.vhd	Data memory address processing
yam_dmem.vhd	Data memory
yam_copr.vhd	Co-processor result memory
yam_regs.vhd	Register array
yam_sregs.vhd	Special Registers
yam_pc.vhd	Program Counter and associated stack
yam_sr.vhd	Status Register
yam_wp.vhd	Workspace pointer management
yam_alu.vhd	ALU top level
yam_alu_arith.vhd	Arithmetic unit
yam_alu_logic.vhd	Logic unit
yam_alu_shift.vhd	Shift unit with barrel shifter
yam_alu_shift1.vhd	Shift unit without barrel shifter
yam_alu_lvl1.vhd	ABS, CLZ, Rotate and SHL16
yam_alu_lvl2.vhd	Byte move...
yam_alu_mult.vhd	Multiplier
yam_dcode.vhd	Instruction decoding
yam_irq.vhd	Interrupts management
yam_fsm.vhd	Core sequencer
yam_disasm.vhd	YAM disassembler for simulation purpose

2.1.2 Synthesis warnings!

The memory descriptions *yam_dramX.vhd* have been written in a way they should be actually synthesized by most tools. However *yam_dram2.vhd* (dual asymmetrical ports) may be sub-optimal, yielding two RAMs where a single one would be possible. This might be optimized, depending on the synthesis tool used.

The multiplier description *yam_alu_mult.vhd* assumes the inference of some DSP primitive for the target technology. If this is not possible or not supported by the synthesis tool the multiplier may dropped out by setting the ALU_MULT generic value to “false” (see next).

2.1.3 Disassembler for simulation purpose

YAM includes a disassembler for simulation purpose.
To visualize its text output and the register contents add to the simulator command (*vsim* example, *uut* pointing to YAM) :

```
add wave uut/regs/regs
add wave uut/disasm/YAM
```

For easier visualization the disassembler text is generated as short as possible. All numbers are expressed in hexadecimal, without any prefix.

2.2 Generics

<table>
<tr><th>Name</th><th>Type / Default</th><th>Description</th></tr>
<tr><td>IM_WIDTH</td><td>natural
32</td><td>Instruction memory width
Recommended value: 32 (min. 24 in 2-operand style)</td></tr>
<tr><td>IM_SIZE</td><td>natural
2048</td><td>Instruction memory size.</td></tr>
<tr><td>REG_WIDTH</td><td>natural
32</td><td>Register (and all data paths) width;
min. 8, max. 128 (a practical limitation).</td></tr>
<tr><td>REG_SIZE</td><td>natural
32</td><td>Regular register number;
max. 128</td></tr>
<tr><td>RM_SIZE</td><td>natural
32</td><td>The size of the memory for hosting the regular registers.
<table>
<tr><td>RM_SIZE
= REG_SIZE</td><td>Basic implementation, single register bank.
Context switch instructions are not implemented.</td></tr>
<tr><td>RM_SIZE
> REG_SIZE</td><td>Multiple register banks with workspace pointer.
Context switch instructions are implemented.
Minimum value is REG_SIZE*2.
Also required: REG_WIDTH >= 32.</td></tr>
</table></td></tr>
<tr><td>PC_STACK</td><td>natural
16</td><td>Program counter stack depth
min. 1, max. 64 (a practical limitation).
Actual subroutine nesting depth is N-1, including possible Interrupt service routine</td></tr>
<tr><td>COPM_SIZE</td><td>natural
0</td><td>Co-processor result memory size.
0 (none) to typically 16 words; maximum 256.
Adds support for co-processor control port when non zero.</td></tr>
<tr><td>DM_SIZE</td><td>natural
32</td><td>Data memory size
0 (none) to 64K words.
When a co-processor is used, locations at addresses $F000 to $FFFF are overloaded by the co-processor result memory.</td></tr>
<tr><td>DM_PIPE</td><td>natural
0</td><td>Insert pipeline stage at DM input or DM and COPM output</td></tr>
</table>

<table>
<tr><td></td><td></td><td><table>
<tr><td>0</td><td>No pipeline;
whenever possible if this fits with clock speed.</td></tr>
<tr><td>1</td><td>DM input address pipeline when reading;
helps timing closure for small memories at fast clock</td></tr>
<tr><td>2</td><td>DM and COPM output data pipeline when reading;
helps timing closure for large memories.</td></tr>
</table></td></tr>
<tr><td>DM2_WIDTH</td><td>natural
1</td><td>DM external port width.
If DM2_WIDTH < REG_WIDTH none is actually generated;
else DM2_WIDTH must be REG_WIDTH*(2**N)</td></tr>
<tr><td>ALU_LEVEL</td><td>natural
0</td><td>Determines the ALU configuration.<table>
<tr><td>0</td><td>Base implementation.
Single position shift replaces the barrel shifter.</td></tr>
<tr><td>1</td><td>Add barrel shifter, rotate and SHL16 instructions.
Add ABS and CLZ instructions.
Not possible when REG_WIDTH<16.</td></tr>
<tr><td>2</td><td>Add byte related instructions.
Add LDYX and MOVX instructions.
Not possible when REG_WIDTH<16.</td></tr>
</table></td></tr>
<tr><td>ALU_3OP</td><td>boolean
false</td><td>Selects the ALU instructions 2-operand (false) or 3-operand (true) style.</td></tr>
<tr><td>ALU_MULT</td><td>boolean
false</td><td>Generates the integer multiplier when true.
Some “DSP/multiplier” primitive must be available for the target technology.</td></tr>
<tr><td>ALU_SPR</td><td>boolean
false</td><td>Generates the Special Registers when true.</td></tr>
<tr><td>IRQ_IF</td><td>boolean
false</td><td>Adds support for external interrupts interface when true.</td></tr>
<tr><td>IOP_IF</td><td>boolean
false</td><td>Adds support for external input/output port when true.</td></tr>
<tr><td>IOP_RMW</td><td>boolean
false</td><td>Adds support for IORMW instruction when true.</td></tr>
<tr><td>YAM_DBG</td><td>boolean
false</td><td>Adds support for debugging/stepping mode when true.
Also requires IOP_IF being true.</td></tr>
<tr><td>YAM_CLK</td><td>natural
0</td><td>YAM system clock frequency in MHz.
Only for reporting in special register.</td></tr>
<tr><td>YAM_SREG3</td><td>std16
0</td><td>User's open usage; value is reported into special register #3.</td></tr>
</table>

2.3 Ports

2.3.1 Short list

Name	Dir.	Type	Description
clk_i	in	stdl	System clock
rst_i	in	stdl	System reset
sleep_i	in	stdl	Sleep request
dbg_rq_i	in	stdl	Switch to debugging/stepping mode
dbg_stat_o	out	stdlv8	Debugging mode status
im_adrs_o	out	stdlv	Program counter
im_datr_i	in	stdlv	Program instruction data
io_wr_o	out	stdl	IO port write strobe
io_rd_o	out	stdl	IO port read validation
io_wait_i	in	stdl	IO port read handshake
io_busy_i	in	stdl	IO port write pending
io_adrs_o	out	stdlv	IO port address
io_datw_o	out	stdlv	IO port data write
io_datr_i	in	stdlv	IO port data read
irq_i	in	stdlv4	Interrupt request
iack_o	out	stdlv4	Interrupt acknowledge
irqp_o	out	stdl	Interrupt request pending
cop_valid_o	out	stdl	co-processor port write strobe
cop_chan_o	out	stdl	co-processor logical channel assignment
cop_id_o	out	stdlv	co-processor identifier
cop_data_o	out	stdlv	co-processor data input
cop_inst_o	out	stdlv	co-processor instruction
cop_busy0_i	in	stdl	co-processor channel#0 busy
cop_busy1_i	in	stdl	co-processor channel#1 busy
clkr_i	in	stdl	co-processor system clock
copr_wr_i	in	stdl	co-processor result write enable
copr_adrs_i	in	stdlv	co-processor result address
copr_data_i	in	stdlv	co-processor result data

clk2_i	in	stdl	DM external port asynchronous clock
dm2_wr_i	in	stdl	DM external port write strobe
dm2_adrs_i	in	stdlv	DM external port address
dm2_dati_i	in	stdlv	DM external port data write
dm2_dato_o	out	stdlv	DM external port data read

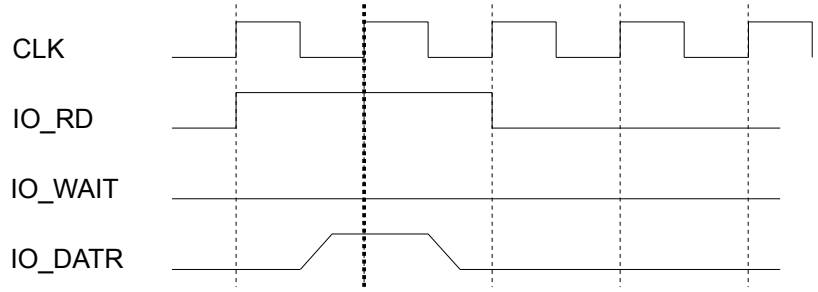
Note: the instruction memory is external to YAM core (see section 2.4)

2.3.2 Input/Output port interfacing

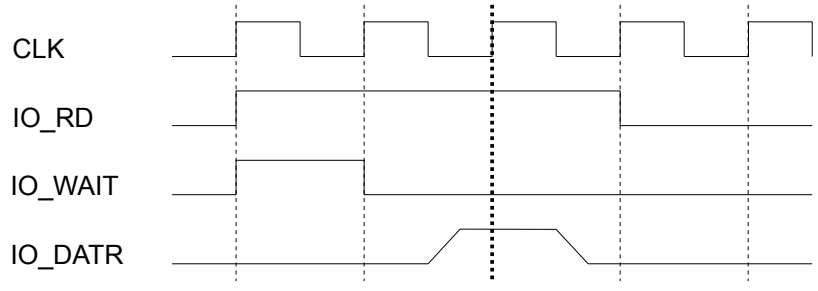
Input/Output port is optional.

Important rule: when reading from the Input port the data must become available in the same clock cycle where the address has been issued for the instruction execution *not* being delayed. In other words the data *cannot* be clocked (this is done internally to YAM). The data from the input port must just follow the input address, unless handshaking is used.
Read handshaking: However a slow input port may pause the YAM core by asserting the *io_wait_i* signal, for as long as necessary. Then re-timing clocks may be added as required.

Sampling time, no wait state

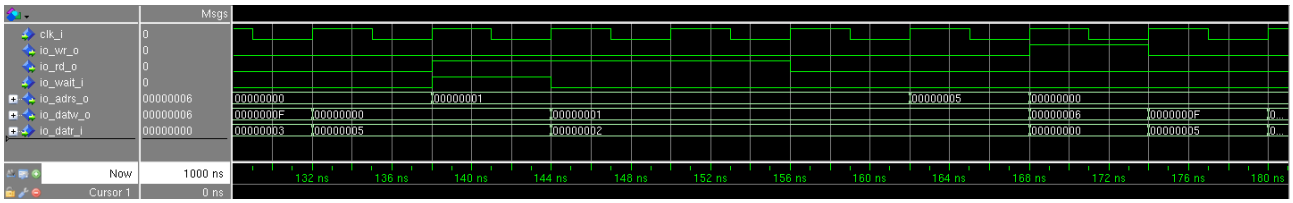


Sampling time, one wait state



It is noteworthy that the *io_rd* signal extends one clock past the actual sampling time.
When writing to the output YAM inserts a clocked (pipelined) register. An example of Input/Output port interfacing follows.

Signalling waveforms example



The input *io_busy_i* , not shown, is addressed in section 8.

Basic input/output VHDL coding example

```
-- IO port (16 registers)

type io_typ is array(15 downto 0) of std_logic_vector(REG_WIDTH-1 downto 0);
signal ioreg   : io_typ;
signal io_wr   : std_logic; -- write enable
signal io_rd   : std_logic; -- read enable
signal io_rd2  : std_logic; -- read enable, delayed for generating io_wait
signal io_wait : std_logic; -- extend read cycle

process(clk)
begin
   if rising_edge(clk) then
      if io_wr = '1' then
         ioreg(to_integer(unsigned(io_adrs(3 downto 0)))) <= io_dati;
      end if;
      io_rd2    <= io_rd;
      io_datr_o <= ioreg(to_integer(unsigned(io_adrs(3 downto 0))));
   end if;
end process;

io_wait <= io_rd AND NOT io_rd2; -- read delayed by one clock
```

Read-modify-write

Atomic IO read-modify-write cycle may be generated, with the aim of implementing exclusion semaphores between several *YAMs*. Assuming no wait states, the sequence is as follows.

- In the first clock cycle *io_rd_o* is asserted and the read data is captured.
- In the next clock cycle *io_wr_o* is asserted and the read data with its MSbit set is written back. As for any read cycle *io_rd_o* remains asserted.

2.3.3 Interrupt interfacing

Interrupt logic is optional.
There are 4 interrupt sources. The *irq_i(N)* input triggers the *interrupt(N)* internal signal and must remain asserted until acknowledged; N in range 1 to 4. YAM core branches to the address N, where a jump to some interrupt service routine (ISR) must be implemented. All interrupts are automatically masked off, so the ISR cannot be interrupted. When returning from the ISR the output *iack_o(N)* is asserted. It must clear the *irq_i(N)* input (otherwise the IRQ would be re-entered). It is possible to individually mask off each interrupt. An example of interrupt interfacing follows.

```
process(clk)
begin
   if rising_edge(clk) then
      for ii in 1 to 4 loop
       if iack(ii) = '1' OR rst = '1' then -- Clear ASAP
         irq(ii) <= '0';
       elsif irq_trig = '1' then           -- Pulsed request
         irq(ii) <= '1';
       end if;
      end loop;
   end if;
end process;
```

The *irqp_o* output is asserted when any interrupt is presented to the YAM core (after crossing the interrupt mask).

2.3.4 Sleep mode

When the *sleep_i* input is asserted, YAM finishes off with the current instruction execution and stops fetching new instruction until *sleep_i* is de-asserted. The *irqp_o* output may be used to clear the *sleep_i* input thus waking up YAM on purpose.

2.3.5 Debugging mode

When the *dbg_rq_i* input is asserted, YAM finishes off with the current instruction execution and switches to the debugging mode. Then YAM enters the loop described thereafter.

- YAM dumps the register contents to *io_datw_o* (value), with *io_adrs_o* pointing to the register address.
- YAM concurrently asserts the *dbg_stat_o(7)* output thus validating the above data burst.
- YAM sleeps until the *sleep_i* input is pulsed high.
- YAM fetches the next instruction.

The *dbg_rq_i* signal must remain asserted at all times. If *dbg_rq_i* is de-asserted YAM continues in normal mode of operation.

The *dbg_stat_o* format is described in next table.

03-00	YAM interrupt mask.
05-04	YAM status register (N flag, Z flag).
06	YAM_DBG generic status
07	YAM data register dump valid.

2.4 Instruction memory

The instruction memory (IM) is a separated component.
It may be implemented with two variants.

- *yam_imem.vhd*; dual ported, with a read/write port for program initialization and read back.
- *yam_irom.vhd*; single port read only.

The assembler generates two files for initializing the IM, depending on its implementation.

- File of integers (*.dat* extension) that may be used for dynamically downloading the program in a dual ported IM.
- File of *bit_vector* values (*.bvt* extension) that must be used for initializing a ROM type IM. It is expected that the synthesis tool will correctly translate this format to initialize the ROM.

In the latter case the generic parameter *IM_FILE* (of string type) is the path to the *.bvt* file. Other generics are *IM_WIDTH* and *IM_SIZE*, common to YAM core.

3 YAM configuration help

3.1 Instruction word width

The instruction word width may be enlarged or shortened, depending on the IM_WIDTH generic parameter value.

The instruction word is decoded left justified from the most significant bit regarding the 16 upper bits (Operation code, addressing modes,...), then right justified from bit zero for the rest (including branch address and immediate data). Even for the smallest configurations the instruction word width cannot be less than 24, so as to preserve a minimal space for branch addresses (limited to 255 when IM_WIDTH is 24). Instruction larger than 32 bit is of little interest.

Almost ever the instruction is 32 bits in width.

3.2 Instruction memory size

Although the theoretical maximum instruction memory size is 8 Mega words, in practice it will be much less; typically a few Kilo works. The IM_SIZE parameter may significantly impact the maximum clock frequency. This parameter should be tuned as low as possible. This is the price to pay for the simple, non pipelined *YAM* architecture.

3.3 Data memory vs regular registers

	Data Memory	Regular registers
ALU aware	no	yes
Immediate write	no	yes
Clock penalty	none for small size +1 when reading and pipelined	none
Direct address range	64K	128
External port	yes (optional)	no
Timing penalty (as size increases)	May become high	Small

Summary
If external connection is required then DM with external second port is the only choice.
If the aim is to extend the number of registers (for storage, as the ALU cannot operate on DM) the DM is better than regular registers.

3.4 Data memory vs banked registers

See section 4.1 thereafter for the banked register organization.
The question arises when a total of 128 registers is not enough. The register number may be increased further multiplying the banks (ex. 4 banks of 128 is offering 512 registers).

	Data Memory	Banked registers
ALU aware	no	yes, within a single bank at a time.
Storage	yes, direct addressing	Limited to a single bank at a time.

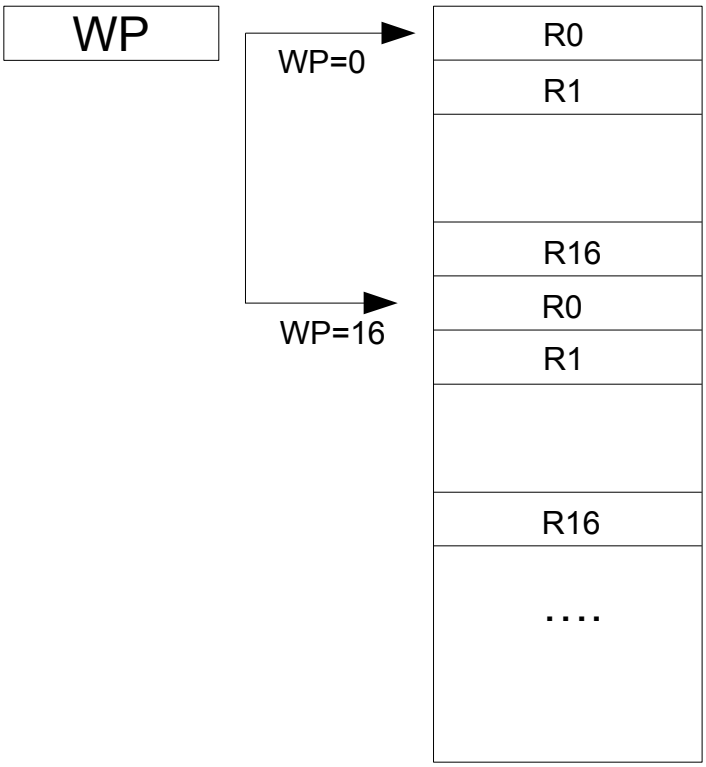
4 Instruction set architecture

4.1 Bit order notation and addressing

Bit zero (right) is the least significant bit; bit REG_WIDTH-1 is the most significant bit.
Data memory addressing is only possible by full word; direct byte addressing is not possible. So the address just increments by one to point to the next word.

4.2 Register banks and Workspace Pointer

There may be several register banks, implemented in the so called register memory.
A dedicated register, the Workspace Pointer (WP) points to register offset zero of the active bank.



**Register banks
and Workspace Pointer
(example with 16 registers)**

There are instructions to branch and switch to a new register bank whilst saving the active context (in the R0 register) and then switch backward, restoring the old context (from the R0 register).

The workspace pointer is managed modulo 16.

Nothing prevents the workspace pointer to get any value (modulo 16). Then some new registers may partially overlap with the old ones (when using 32 or more of them).

When a single bank is enough (RM_SIZE = REG_SIZE) the WP is forced to zero and the corresponding logic won't be generated.

4.3 Index pointers to register memory

At high-end configuration (ALU_LEVEL of 2) there are two pointers to the register memory, namely RX and RY. The RX and RY pointers are for making easier moving blocks of data between the register memory and the DM, in both directions.

The RX pointer allows indirect addressing of the register memory. It is self-incrementing.
The RY pointer is used to check the upper bound of RX. The Z flag is set when RX equals RY.

4.4 Addressing modes

4.4.1 Main addressing modes

There are 3 main addressing modes (written using the assembler syntax).

```
%xx    Regular register
@%xx   Indirect register
&aaa   Direct addressing
```

These are applying to both the source and the destination operands.

In addition an immediate constant may be sourced, using a number of variants.

```
#DDD   Immediate data sign extended
>DDD   Immediate data unsigned
<DDD   Immediate data unsigned shifted left
```

4.4.2 Immediate data variants

The symbol used thereafter (#D16) is referring to a frequent instruction width of 32 bits (IM_WIDTH generic). This leaves a 16 bit field for defining immediate values. However this may be different. For example if IM_WIDTH is 24, #D16 would be 8 bits in width.

Immediate data may be pre-processed in 4 different ways, from the instruction word to destination storage. In next table the $ sign prefixes an hexadecimal number.

REG_WIDTH > IM_WIDTH-16 (usually 16)

Type	Assembler Notation	Processing	Example IM_WITH=32, REG_WIDTH=32
Short signed (3-op. only)	#D8	MSbit (sign) extended	From IM_WITH-24 to REG_WIDTH $87 => $FFFFFF87
Signed	#D16	MSbit (sign) extended	From IM_WITH-16 to REG_WIDTH $8765 => $FFFF8765
Unsigned	>D16	Up filled with zeros	From IM_WITH-16 to REG_WIDTH $8765 => $00008765
Unsigned shifted	<D16	Left shifted by IM_WIDTH-16	From IM_WITH-16 to REG_WIDTH $8765 => $87650000

REG_WIDTH < IM_WIDTH-16 (usually 16)

Type	Notation *yamasm*	Processing	Example IM_WITH=32, REG_WIDTH=8
Unsigned	#D16 >D16 <D16	Truncated Truncated Truncated (not shifted)	From IM_WITH-16 to REG_WIDTH $8765 => $65 Shifting left would not make sense!

For simplicity we shall use #D16 for representing any of the 3 cases (but short signed) in this manual.

4.5 ALU flags

There are two ALU flags:

- Zero (Z); result is zero.
- Negative (N) arithmetic only; result is negative.

There is no dedicated Carry flag.
However the instruction ADDC/SUBC (Add/Sub with Carry) allows to accumulate past the register dynamical range. In that specific case the Carry is defined as:
Carry := Negative
from the previous (hopefully ADD/SUB) instruction which operates on unsigned integers.

For every ALU related instruction updating (or not) the flags is selectable ('S' bit in the instruction).

4.6 Core internal registers

These are summarized in next table.

Name	Width (max)	Reset value
Program Counter (PC)	23	0
Index pointers (RX, RY) to register memory, optional	7	X
Workspace Pointer (WP), optional	10	0
Interrupt mask (IMask), optional	4	“1111” = all masked off
Status Register (SR) 5-2 | 1 | 0 Reserved | N flag | Z flag	6	0

4.7 Special registers

A set of read only Special Registers (SPR) is used for reporting with the following format.

SPR0	Upper	13	12	11-10	09-00
Value	Zero	ALU_3OP	ALU_MULT	ALU_LEVEL	YAM_CLK

SPR1	Upper	15-08	07-04	03-00
Value	Zero	REG_WIDTH	Log2(RM_SIZE)	Log2(REG_SIZE)

SPR2	Upper		07-04	03-00
Value	Zero		Log2(COPM_SIZE)	Log2(DM_SIZE)

SPR3	Upper	15-00
Value	Zero	YAM_SPR3 generic value.

SPR7	Upper	15-00
Value	Zero	Multiplier result most significant word: bits REG_WIDTH+15 down to REG_WIDTH. Only significant when REG_WIDTH is 16 or less.

The special registers are either filled up with zeros, or left truncated to fit the actual register width.

4.8 Exception vectors

The reserved branch addresses to the instruction memory are summarized in next table.

0	Reset
1	IRQ1
2	IRQ2
3	IRQ3
4	IRQ4
7-5	Reserved

4.9 ALU instructions main options

These are summarized in next table.

ALU_LEVEL := 0	3-operand style ALU_3OP := true 2-operand style ALU_3OP := false ALU base
ALU_LEVEL := 1	ALU extension level 1
ALU_LEVEL := 2	ALU extension level 2
ALU_MULT := true	Multiplier extension
ALU_SPR := true	Special registers support

5 Instruction list

Not every addressing mode is possible in any operation. The valid combinations are detailed thereafter. The general syntax is (see *YAM assembler* documentation for all details)

```
<Op. Code> <Destination>, <Source> [, S]
```

The third argument ('S') is only applicable to ALU instructions for controlling the ALU flags update. Alternatively the assembler *S_ON* / *S_OFF* directives allows setting / clearing the S bit within a whole sequence of instructions.

When *YAM* has been configured for 3-operand ALU operations the syntax may extend to

```
<Op. Code> <Destination>, <source2>, <Source> [, S]
```

5.1 ALU base (level 0)

5.1.1 ALU base Arithmetic

ADD	Add unsigned
ADDC	Add unsigned with Carry
SUB	Subtract unsigned
SUBC	Subtract unsigned with Carry
SUBS	Subtract signed
CMP	Compare (SUB without assignment) unsigned
CMPS	Compare (SUBS without assignment) signed

2-operand style

```
OPC %dd, %ss  [, S]
OPC %dd, #D16 [, S]
```

3-operand style

```
OPC %dd, %s2, %ss  [, S]
OPC %dd, %dd, #D16 [, S]  ; mind second source == destination
OPC %dd, %s2, #D8  [, S]  ; mind short 8-bit immediate limitation
```

Z flag: set when result is zero.
N flag: carry (ADD case) or sign of result (SUB case).
Note: Subtraction is ordered as *%s2* minus *%ss* (*%dd* minus *%ss* in the 2-operand style).
Note: Subtraction unsigned versus signed impacts the N flag (not the result).
Note: CMP, CMPS does not make sense without the S bit set.
Note: ADDC does not make sense without a previous (usually ADD) instruction and the S bit set.

5.1.2 ALU base Logic

AND, OR, XOR	Bitwise logic
TST	Test (AND without assignment)

Same addressing modes as arithmetic, above.

Z flag: set when result is zero.
N flag: not touched.
Note: TST does not make sense without the S bit set.

5.1.3 ALU base shift

SHR	Shift Right logical
SHRA	Shift Right Arithmetic (sign bit propagation)
SHL	Shift Left logical

2-operand style

```
SHX %dd, #1 [, S]
```

3-operand style

```
SHX %dd, %s2, #1 [, S]
```

Z flag: inverted MSbit (shift left) or LSbit (shift right) of the source register (before shifting).
N flag: MSbit of result.

5.1.4 ALU base immediate source commented

There are three variants (signed extended, unsigned, left shifted unsigned) depending on the prefix #, > or < (see 4.4.2). Some examples follow, assuming a 32-bit data width.

2-operand style

```
ADD %dd, #$FFF2 ; adds $FFFFFFF2
ADD %dd, >$FFF2 ; adds $0000FFF2
ADD %dd, <$FFF2 ; adds $FFF20000
```

3-operand style

Similar syntax may be used, with the second source identical to the destination.

```
ADD %dd, %dd, #$FFF2 ; adds $FFFFFFF2
ADD %dd, %dd, >$FFF2 ; adds $0000FFF2
ADD %dd, %dd, <$FFF2 ; adds $FFF20000
```

However when the second source differs from the destination, only the short immediate *signed extended* remains possible.

```
ADD %dd, %s2, #$F2   ; %dd = %s2 + $FFFFFFF2
ADD %dd, %s2, >$F2   ; Error!
ADD %dd, %s2, <$F2   ; Error!
ADD %dd, %s2, #$FFF2 ; Error! Out of range.
```

5.2 ALU extension level 1

5.2.1 ALU1 shift, dynamical

SHR	Shift Right logical
SHRA	Shift Right Arithmetic (sign bit propagation)
SHL	Shift Left logical

2-operand style

```
SHX %dd, %ss [, S]
SHX %dd, #N  [, S]
```

3-operand style

```
SHX %dd, %s2, %ss [, S]
SHX %dd, %s2, #N  [, S]
```

Shift count is dynamically sourced from a regular register or is an immediate value. Shift count is limited to the 0-15 range.
Z flag: inverted MSbit (shift left) or LSbit (shift right) of the source register (before shifting).
N flag: MSbit of result.

5.2.2 ALU1 shift left fixed by 16

```
SHL16  %dd, %ss [, S]
```

Z flag: inverted MSbit of the source register (before shifting).
N flag: MSbit of result.
Note: Similar shift right is not available.
Note: Useful for large (more than 32 bits) constant or half word manipulation.

5.2.3 ALU1 rotate fixed by 1

```
ROTR %dd, %ss [, S]
ROTL %dd, %ss [, S]
```

Z flag: inverted MSbit (shifting left) or LSbit (shifting right) of the source register (before shifting).
N flag: MSbit of result.

5.2.4 ALU1 absolute value

```
ABS  %dd, %ss [, S]
```

Z flag: set when result is zero.
N flag: MSbit of the source.

5.2.5 ALU1 Count Leading Zeros

```
CLZ %dd, %ss [, S]
```

Operates on the 16 most significant bits of the regular source register. Returns 0 to 16.
Z flag: set when the 16 most significant bits of the source are zero; %dd is 16.
N flag: MSbit of the source; %dd is zero.

5.3 ALU extension level 2

5.3.1 ALU2 Byte Store

```
BSTN %dd, %ss [, S]
```

N in the range 0 to 3.
%dd<8*N+7 : 8*N> ← %ss<7:0>

Z flag: set when the source byte is zero.
N flag: MSbit of source byte.

5.3.2 ALU2 Byte Load

```
BNLD %dd, %ss [, S]
```

N in the range 0 to 3.
%dd<7:0> ← %ss<8*N+7 : 8*N>
%dd<REG_WIDTH-1:8> ← Zero

Z flag: set when the source byte is zero.
N flag: MSbit of source byte.

5.3.3 ALU2 Byte Load with sign propagation

```
BNLDS %dd, %ss [, S]
```

N in the range 0 to 3.
%dd<7:0> ← %ss<8*N+7 : 8*N>
%dd<REG_WIDTH-1:8> ← %dd<7>

Z flag: set when the source byte is zero.
N flag: MSbit of source byte.

5.3.4 ALU2 (block move) index pointers management

```
LDYX  %03, %01 [, S] ; RY:=3, RX:=1
```

Load the RY and RX indexes.

Z flag: set when RX=RY.
N flag: zero.
Note: the syntax may be misleading as the registers (%03, %01 in this example) are not directly involved. Only RX and RY are loaded to point to the actual registers.

```
MOVX @%dd, @%X+ [, S] ; to DM
MOVX @%X+, @%ss [, S] : from DM
```

The MOVX instruction moves a register contents pointed to by the RX index.
Then RX self increments. Z flag reports on RX and RY being equal.

Z flag: set when RX=RY.
N flag: zero.

Albeit looking like a MOV instruction, the MOVX is an ALU family instruction. So it possibly updates the flags. The addressing modes are not derived from the (MOV like) fields, but forced from the main operation decoding. An example of usage follows.

```
;; Moving R1-R5 to DM at address 7 onwards
     MOV  %0f, #7        ; pointer to DM
     LDYX %05, %01       ; start from R1, stops at R5
loop MOVX @%0f, @%X+, S  ; move reg. to DM, RX++, RX=RY?
     ADD  %0f, #1        ; next in DM, preserving flags form MOVX!
     JMP  loop, NZ       ; done?
```

WARNING! RX and RY are not saved during an interrupt context switch. Therefore they should not be used within an interrupt service routine.

5.4 ALU Multiply

```
MUL %dd, [%s2], %ss
MUL %dd, [%s2], #D16
```

Both operands are resized (or truncated) to 16 bit unsigned. The 32 bit unsigned result is resized to REG_WIDTH.
Flags: not touched.
ALU_MULT false: Not supported.

5.5 Move

To regular register

```
MOV %dd, %ss
MOV %dd, #D16
```

Data Memory (DM)

DM addressing may be direct or indirect (from a regular register). It is possible to move from/to DM to/from a register.

```
MOV  @%dd, %ss
MOV  %dd, @%ss
MOV  &aaa, %ss
MOV  %dd, &aaa
```

5.6 Branch

`JMP <label> [,<condition>]`	Jump at direct address
`JMP @%dd [,<condition>]`	Jump at indirect address from register %dd

The condition checks

- the ALU flags;
- the IRQ flag;
- the co-processor flags (see section 7);
- the IO system flag (see section 8).

5.7 Subroutine

`JSR <label>`	Jump to subroutine at direct address
`JSR @%dd`	Jump to subroutine at indirect address from register %dd
`RET`	Return from subroutine
`RLD %dd, #D16`	Return from subroutine and load immediate data to register %dd #D16 is restricted to the sign extended immediate variant.

JSR/RET may be nested to a level depending on the stack depth, as defined by the PC_STACK generic. In the case PC_STACK as the one value, JSR cannot be nested, but still work with a single return address.

Note:

A sequence of RLD instructions may be used to create a table of constants in the Instruction Memory. Getting a value from the table is done by JSR at the appropriate offset from the table origin. An example follows.

```
i2c_map
   RLD  %02, #$98  ;  #0, chan. 0 , dev. 0, ADC LT2990 power
   RLD  %02, #$90  ;  #1, chan. 0 , dev. 1, TMP108 carrier
   …
   MOV   %0a, #i2c_map
   ADD   %0a, %07     ; %0a points at some input in table
   JSR   @%0a         ; %02 is assigned a value from table
```

5.8 Context switch

`JSRWP <label>, <WPoffset>`	Jump to subroutine at direct address, with context switch WP ← WPoffset; %00 ← Old context WPoffset value must be a multiple of 16. Branch address is limited to 16 bits in width.
`RETWP`	Return from subroutine, with context restoration Context ← %00;

The context (saved in register zero) format is as follows.

31-26	25-20	19-16	15-00
WP/16	SR	IMask	PC (for returning)

Of course, after switching, the register zero should be left untouched, or exceptionally modified with extreme care.
For the context switch to be usable, REG_WIDTH must be at least 32.

REG_SIZE = RM_SIZE: Not supported.

5.9 Interrupt management

When the *irq_i(N)* is detected:

- IMask is saved;
- SR is saved;
- all interrupts are masked (IMask ← ”1111”);
- program execution jumps to the ISR at address N (related to the *irq_i(N)* input).

`RETI`	Return from ISR, restoring IMask and SR.
`IRQ_ON <mask>`	Unmask dedicated interrupt (actually bitwise NAND to IMask).
`IRQ_OFF <mask>`	Mask dedicated interrupt (actually bitwise OR to IMask).
`IRQM_SS`	IMask save then set (mask all).
`IRQM_RLD`	IMask reload from saved.

IRQ management instructions should not be executed inside an ISR.
IRQM_SS/RLD instructions cannot be nested as saving is done to a dedicated register, not to some stack.

A dedicated IRQ flag may be tested in a regular branch instruction as follows:

```
halt  JMP halt, NIRQ  ; loop back here
      ; continue thereafter when some ISR has been executed
```

The IRQ flag is set by RETI last instruction of the ISR and cleared at next instruction (usually a conditional JMP to make sense).

IRQ_IF false: Not supported.

5.10 IO port management

IO port indirect addressing is the only way.
The data is moved to/from a regular register, or may originate from an immediate value.

```
MOVIO %dd, @%ss   Input from port
MOVIO @%dd, %ss   Output to port
MOVIO @%dd, #D16
```

IO_IF false: Not supported.

```
IORMW %dd, @%ss   read-modify-write (MSbit set is written back)
```

IO_RMW false: Not supported.
Note: the IORMW does not set any flag. It must be followed by some ALU instruction to actually check the destination register (usually its MSbit for implementing some exclusion mechanism).

IMPORTANT WARNING!
After an IO write instruction, at least one other instruction must be inserted before issuing any IO read. For example the following sequence is wrong.

```
MOVIO @%00, #D16  ; out
MOVIO %01, @%02   ; in cannot follow out!
```

This is because the write is posted and actually takes place in the beginning of the next instruction. However in the case the write and read addresses are identical, and the IO system is YAM core synchronous, it will work as expected.

5.11 Co-processor control

Section 7 is dealing in depth with co-processors.

```
COPN %ss, #D16, C
```

N co-processor identifier (0 to 7)
%ss Source register pair (must be even) %ss - %ss+1
#D16 Instruction to co-processor
C Logical channel (0 or 1)

COPM_SIZE = 0: Not supported.

5.12 Special registers

The Special Registers may only be read out by switching the ALU to a dedicated state on that purpose. Then the ALU normal state must be restored. A typical sequence of instructions is as follows.

```
SPR_ON        ; switch to SPR readout
MOV %04, %00  ; SPR0 stored to %04
MOV %05, %01  ; SPR1 stored to %05
MOV %06, %02  ; SPR2 stored to %06
SPR_OFF       ; switch back to normal state
```

ALU_SPR false: Not supported.

Note: the source register in the instruction is only used for addressing one out of the Special Registers.
Note: SPR_ON also does IRQM_SS; SPR_OFF also does IRQM_RLD; so interrupts are automatically masked off during the whole sequence.

5.13 Loading immediate data

Assuming a 32 bit instruction width, whatever REG_WIDTH, immediate value in range [-$8000, +$7FFF] can be loaded in a single instruction using the sign extended format.

```
MOV    %dd, #$4567  ; $00004567
MOV    %dd, #$8999  ; $FFFF8999
```

Larger values may be managed considering that immediate data may also *not* be sign extended and shifted left by 16. Examples follow (with *yamasm* syntax).

REG_WIDTH := 32

```
MOV    %dd, <$4567  ; $45670000
OR     %dd, >$8999  ; $45678999
```

REG_WIDTH := 48

```
MOV    %dd, <$4567  ; $000045670000
OR     %dd, >$8999  ; $000045678999
SHL16  %dd          ; $456789990000
OR     %dd, >$F234  ; $45678999F234
```

5.14 Warning on Z and N flags setting

Flags setting might look unusual. Mind the following reminder and check every specific case in the full instruction description!

Shifts:
Z is the first bit (of the source) shifted out; destination is *not* checked against zero.
N is the most significant bit of the result.
Byte move:
Z is set when the *source* byte is zero; destination is *not* checked against zero.
N is the most significant bit of the *source* byte.
Multiplication: does *not* touch any flag.

5.15 Instruction execution time

As a general rule, an instruction executes in 2 clock cycles.

However some more clock cycles may be inserted as follows.

+1	Data memory read when DM_PIPE is not zero.
+1+W	IO port read (plus external wait state(s)).

5.16 Interrupt latency

The time from the interrupt asserted to the service routine being executed is minimum 3 and typically 4 clock cycles.

5.17 Instruction word format

Note (*): 3-operand style only.

5.17.1 Main frames

31-23	22-16	14-08	06-00
Op. code / addrs. mode	Reg. destination	Reg. source 2 (*)	Reg. source 1

31-23	22-16	14-08	07-00
Op. code / addrs. mode	Reg. destination	Reg. source 2 (*)	Immediate short (*)

31-23	22-16	15-00
Op. code / addrs. mode	Reg. destination	Immediate data

31-23	22-00
Op. code	Branch address

5.17.2 Source addressing mode encoding

Code	Description	Notation	Applicable to
000	Register	%ss	MOV, MOVIO, ALU
010	Indirect register	@%ss	MOV, MOVIO
011	Direct address	&aaa	MOV
100	Imm. short signed	#DD	ALU (*)
101	Immediate signed	#DDDD	MOV, MOVIO, ALU
110	Immediate unsigned	>DDDD	MOV, MOVIO, ALU
111	Immediate shifted left	<DDDD	MOV, MOVIO, ALU

5.17.3 Destination addressing mode encoding

Code	Description	Notation	Applicable to
00	Register	%ss	MOV, MOVIO
10	Indirect register	@%ss	MOV, MOVIO
11	Direct address	&aaa	MOV

5.17.4 ALU

<table>
<tr><th>31-27</th><th>26</th><th>25-23</th><th>22-16</th><th>15-08</th><th>07-00</th></tr>
<tr>
<td>Op. Code
<table>
<tr><td>00000</td><td>TST</td></tr>
<tr><td>00001</td><td>AND</td></tr>
<tr><td>00010</td><td>OR</td></tr>
<tr><td>00011</td><td>XOR</td></tr>
<tr><td>00100</td><td>CMP</td></tr>
<tr><td>00101</td><td>CMPS</td></tr>
<tr><td>00110</td><td>SUB</td></tr>
<tr><td>00111</td><td>SUBS</td></tr>
<tr><td>01000</td><td>ADD</td></tr>
<tr><td>01001</td><td>ADDC</td></tr>
<tr><td>01010</td><td>SUBC</td></tr>
<tr><td>01110</td><td>MUL</td></tr>
</table></td>
<td>'S' bit
Set flags
0 = do
1 = don't update</td>
<td>Source mode</td>
<td>Dest. Reg.</td>
<td>Source Reg. 2 (*)
or
Immediate MSByte</td>
<td>Source Reg. 1
or
Immediate LSByte
or
Imm. short (*)</td>
</tr>
<tr>
<td><table>
<tr><td>10000</td><td>SHIFT 15-0</td></tr>
</table></td>
<td>Set flags</td>
<td><table>
<tr><td>000</td><td>SHR %,%</td></tr>
<tr><td>001</td><td>SHRA%,%</td></tr>
<tr><td>010</td><td>SHR %,#</td></tr>
<tr><td>011</td><td>SHRA %,#</td></tr>
<tr><td>100</td><td>SHL %,%</td></tr>
<tr><td>110</td><td>SHL %, #</td></tr>
</table></td>
<td>Dest. Reg.</td>
<td>Source Reg. 2 (*)</td>
<td>Source Reg.
or
Immediate data</td>
</tr>
<tr>
<td><table>
<tr><td>10010</td><td>Level 1 Ext.</td></tr>
</table></td>
<td>Set flags</td>
<td><table>
<tr><td>000</td><td>ROTR</td></tr>
<tr><td>100</td><td>ROTL</td></tr>
<tr><td>110</td><td>SHL16</td></tr>
<tr><td>001</td><td>ABS</td></tr>
<tr><td>011</td><td>CLZ</td></tr>
</table></td>
<td>Dest. Reg.</td>
<td></td>
<td>Source Reg.</td>
</tr>
<tr>
<td><table>
<tr><td>10011</td><td>Level 2 Ext.</td></tr>
</table></td>
<td>Set flags</td>
<td><table>
<tr><td>0NN</td><td>Byte offset</td></tr>
</table></td>
<td>Dest. Reg.</td>
<td><table>
<tr><td>00000001</td><td>BST</td></tr>
<tr><td>00000010</td><td>BLD</td></tr>
<tr><td>00000110</td><td>BLDS</td></tr>
</table></td>
<td>Source Reg.</td>
</tr>
</table>

		000 Not used		00010100 LDYX 00011000 MOVX to DM 00011100 MOVX fr. DM	

5.17.5 MOV

31-28	27-26	25-23	22-16	15-00
1010	Destination mode (but Direct address.)	Source mode	Dest. Reg.	Source Reg. or Immediate data or Direct address
	Destination mode (11 = Direct address)	Source mode (000 = Register)	Source Reg.	Direct address

5.17.6 MOVIO, IORMW

31-28	27-26	25-23	22-16	15-00
1011	Destination mode (but Direct address)	Source mode (but Direct address.)	Dest. Reg.	Source Reg. or Immediate data

IORMW is coded like a MOVIO read instruction, but bit 15 is set to one.

5.17.7 JMP

<table>
<tr><td>31-28</td><td>27</td><td colspan="2">26-23</td><td>22-16</td><td>15-00</td></tr>
<tr><td>1100</td><td>1</td><td colspan="2">Condition</td><td>Indirect Reg.</td><td></td></tr>
<tr><td rowspan="14"></td><td rowspan="14">0</td><td colspan="2">Condition</td><td rowspan="14">Direct address MSbits</td><td rowspan="14">Direct address LSbits</td></tr>
<tr><td>0000</td><td>Always</td></tr>
<tr><td>0001</td><td>Z</td></tr>
<tr><td>0010</td><td>NZ</td></tr>
<tr><td>0011</td><td>GT</td></tr>
<tr><td>0100</td><td>LT</td></tr>
<tr><td>0101</td><td>GE</td></tr>
<tr><td>0110</td><td>LE</td></tr>
<tr><td>0111</td><td>NIRQ</td></tr>
<tr><td>1000</td><td>B0</td></tr>
<tr><td>1001</td><td>B1</td></tr>
<tr><td>1010</td><td>NB0</td></tr>
<tr><td>1011</td><td>NB1</td></tr>
<tr><td>1100</td><td>BIO</td></tr>
</table>

5.17.8 COP

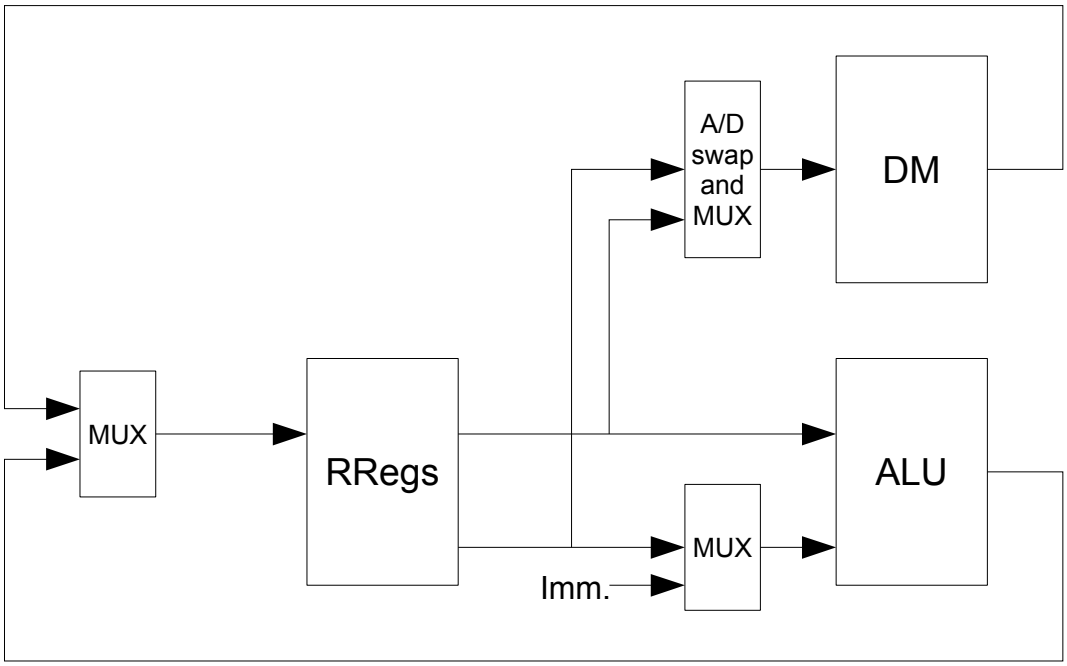
31-28	27-25	24	23	22-16	15-00
1110	ID	Reserved	Channel	Source Reg. pair	Immediate data

5.17.9 Miscellaneous

<table>
<tr><th>31-28</th><th>27-23</th><th>22-16</th><th>15-00</th></tr>
<tr><td>1111</td><td>
<table>
<tr><td>00001</td><td>JSR Direct</td></tr>
<tr><td>10001</td><td>JSR @%dd</td></tr>
<tr><td>00101</td><td>RLD</td></tr>
<tr><td>00010</td><td>JSRWP</td></tr>
<tr><td>00100</td><td>IRQ_ON/OFF</td></tr>
</table>
</td><td>
<table>
<tr><td colspan="2">Address MSbits</td></tr>
<tr><td colspan="2">Source Reg.</td></tr>
<tr><td colspan="2">Dest. Reg</td></tr>
<tr><td colspan="2">WP/16</td></tr>
<tr><td>22</td><td>19-16</td></tr>
<tr><td>0/1 = on/off</td><td>Mask</td></tr>
</table>
</td><td>
<table>
<tr><td>Address LSbits</td></tr>
<tr><td></td></tr>
<tr><td>Immediate data</td></tr>
<tr><td>Address</td></tr>
<tr><td></td></tr>
</table>
</td></tr>
<tr><td></td><td>00000</td><td>
<table>
<tr><td>0000000</td><td>RET</td></tr>
<tr><td>0000001</td><td>RETWP</td></tr>
<tr><td>0000010</td><td>RETI</td></tr>
<tr><td>0000100</td><td>SPR_OFF</td></tr>
<tr><td>0000101</td><td>SPR_ON</td></tr>
<tr><td>0000110</td><td>IRQM_RLD</td></tr>
<tr><td>0000111</td><td>IRQM_SS</td></tr>
</table>
</td><td></td></tr>
</table>

6 Implementation

6.1 YAM (simplified) data paths



6.2 System clock frequency

Rough estimate follows for REG_WIDTH := 32.
Target technology: XILINX KINTEX7 speed grade 2 (medium).
Depending on the configuration the maximum clock frequency is in the 125 - 200 MHz range.

6.3 Area resources

YAM being largely scalable, this severely impacts the area resources. Rough estimates based on a few examples follow. Target technology: XILINX KINTEX7 or SPARTAN6.
This is for 2-operand style. Add 10-15% for the 3-operand style, depending on the configuratiion.

The smallest, 16 8-bit registers
SP6: ~60 slices (10% of the smallest part SP6SLX4)

Small configuration, 32 16-bit registers
SP6: ~125 slices (20% of the smallest part SP6SLX4)
KX7: ~90 slices (0.2% of medium range XC7K325)

Medium configuration, 128 32-bit registers, 256 word DM,
ALU_LEVEL := 0;
KX7: ~200 slices (0.4% of medium range XC7K325)
ALU_LEVEL := 1;
KX7: ~260 slices (0.5% of medium range XC7K325)
ALU_LEVEL := 2;
KX7: ~300 slices (0.6% of medium range XC7K325)

7 co-processor interfacing

7.1 Overview

There can be up to 8 co-processors, numbered 0 to 7.
A co-processor is activated by the YAM core using the specific COPN instructions.
YAM sends data to co-processor using a dedicated output port.
Co-processor stores back its results through the YAM co-processor result input port.
Co-processor subsystem clock may differ from the YAM system clock.

7.2 Logical channel and flags

A maximum of two co-processors may be active at the same time. At activation time a co-processor is dynamically assigned a logical channel (0 or 1), under software control.
An active co-processor drives one of the *cop_busy(0 or 1)* signals (depending on its logical channel) until the operation has been completed.
The *cop_busy* signals can be tested as dumb flags by a YAM branch instruction.

7.3 Data communication

7.3.1 Input to co-processor

In a single COPN instruction YAM sends out:

- a register pair contents (restricted to first register being at even offset);
- a 16-bit “instruction” (that may be openly defined and used);
- the logical channel assigned to.

In case more data need be transmitted, repeated COP instructions should be used. The logical channel is memorized in the first access.

7.3.2 Output from co-processor

Co-processor writes back its result in the YAM dedicated co-processor memory (COPM), through a co-processor interconnect core (see the next section).
The COPM memory is managed like the DM memory with the following constraints.

- Its word width is defined by REG_WIDTH, the register width.
- It is located at address $FF00 onwards.
- It can only be read out by YAM.

By default the recommended storage offset in COPM (which is usually very limited in size) are:

- from 0 onwards for logical channel 0;
- from 4 onwards for logical channel 1.

Different schemes may be agreed upon, for example using the “instruction” parameter.
It is noteworthy that logical channels are dynamically assigned under software control. In general the channel is *not* automatically related to the co-processor identifier.

Regarding the data width a typical 32-bit implementation the co-processor

- gets 64-bit data plus the 16-bit instruction in one shot;
- stores a 32-bit result in one clock cycle (plus the processing time).

7.4 co-processor interconnect core

7.4.1 Overview

The interconnect is based on the following few rules.

- A co-processor not ready to transmit drives its outputs to zero.
- The interconnect generates a *"sync0"* signal that toggles at every clock cycle.
- A co-processor assigned to channel 0 should validates its output when *sync0* is high.
- A co-processor assigned to channel 1 should validates its output when *sync0* is low.

7.4.2 Source files

Co-processor related source files are found in the /hdl/ycop32 directory.
The co-processor deliverables includes

- Core interconnection between co-processors (32-bit).
- YAM 32-bit floating point co-processor.
- YAM fixed point co-processor.
- A general purpose co-processor template.

ycop_pkg.vhd	Package for component declarations
ycop_core32.vhd	co-processor interconnection, 32-bit data width
ycop_float32.vhd	Floating point 32-bit co-processor
fx_add.vhd	Fixed point; elementary terms adder
fx_bshift.vhd	Fixed point; barrel shifter
fx_mult.vhd	Fixed point; elementary term multiplier
fx_muls.vhd	Fixed point; full multiplier
ycop_fixed.vhd	Fixed point co-processor
ycop_template.vho	co-processor template

7.4.3 Generics

Name	Type	Description
COP_NB	natural	Number of co-processors
COP_PIPE	boolean	Add pipeline registers within interconnect; P&R help.

7.4.4 Ports

Name	Dir.	Type	Description
			YAM clock domain (combinatorial OR)
cc_bsy0_i	in	stdlv	Channel 0 busy from co-processors
cc_bsy1_i	in	stdlv	Channel 1 busy from co-processors
cop_busy0_o	out	stdl	Co-processor channel 0 busy to YAM
cop_busy1_o	out	stdl	Co-processor channel 1 busy to YAM
			DM/co-processor clock domain

clkr_i	in	stdl	DM/co-processor clock
cc_wr_i	in	stdlv	Write enable from co-processors
cc_adrs_i	in	stdlv array	Destination address from co-processors
cc_datw_i	in	stdlv array	Data from co-processors
cc_sync0_o	out	stdl	Channel zero synchronization
copr_wr_o	out	stdl	COPM external port write strobe
copr_adrs_o	out	stdlv	COPM external port address
copr_data_o	out	stdlv	COPM external port data write

7.5 Signalling waveforms (co-processor)

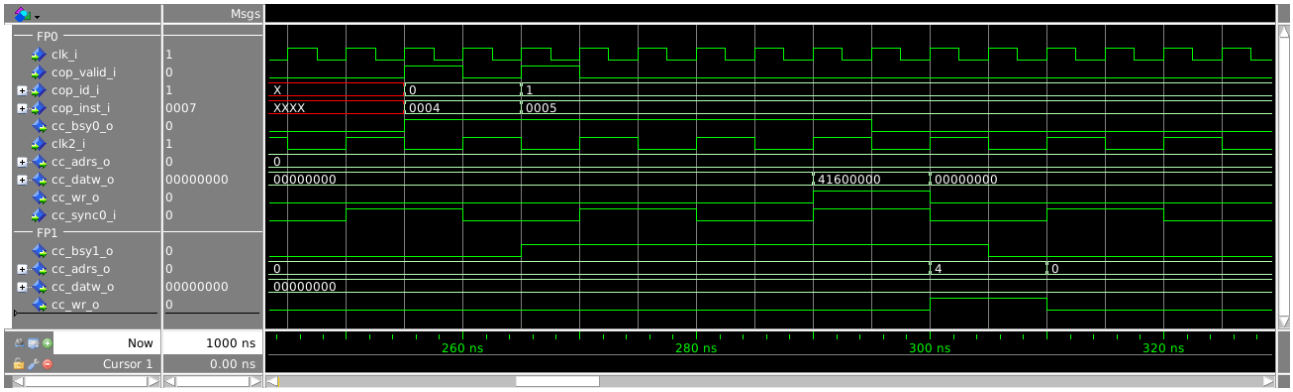


Figure above shows a sequence of two instructions COP channel#0 ;COP channel #1.
The co-processor clock frequency is half YAM clock frequency.
Mind the co-processor assigned to channel #0 writing when *sync0* is asserted, and channel #1 when *sync0* is de-asserted.

7.6 Procedure

The YAM co-processor logic is only responsible for transmitting the data.
The whole co-processor channel management is under software control.
Co-processor must not be used within interrupt service routine. The *cop_busy* flags are neither saved nor restored as part of the SR.

A basic software sequence may be as follows.

```
     COP2  %04, $1234, 0  ; activate cop#2 on channel#0 (data = %04-%05)
wait JMP wait, B0         ; check busy#0 (since assigned to channel#0)
     MOV   %02, &8000     ; get result from COPM
```

7.7 Floating point (32-bit) co-processor

7.7.1 Overview

A co-processor is available that implements basic 32-bit floating point instructions as defined in the IEEE-754 standard. A multiplier-accumulator is also included as an option.

7.7.2 Generics

Name	Type	Description
COP_ID	std3	co-processor identifier
IEEE754	boolean	True for a full IEEE754 implementation. False turns off processing of NAN (invalid or overflow) and de-normal numbers.
FP32_LEVEL	natural	0 = Basic 1 = Add the multiplier-accumulator

7.7.3 Ports

Name	Dir.	Type	Description
			YAM clock domain
clk_i	in	stdl	System clock
rst_i	in	stdl	System reset
cop_valid_i	in	stdl	co-processor write strobe
cop_chan_i	in	stdl	co-processor logical channel assignment
cop_id_i	in	stdlv	co-processor identifier
cop_data_i	in	stdlv	co-processor data input
cop_inst_i	in	stdlv	co-processor instruction
cop_busy0_i	in	stdl	Channel 0 busy from interconnect
cop_busy1_i	in	stdl	Channel 1 busy from interconnect
cc_bsy0_o	out	stdl	Channel 0 busy driver
cc_bsy1_o	out	stdl	Channel 1 busy driver
			COPM/co-processor clock domain
clkr_i	in	stdl	COPM/co-processor clock
cc_wr_o	out	stdl	Write enable
cc_adrs_o	out	stdlv	Destination address
cc_datw_o	out	stdlv	Data result
cc_sync0_i	in	stdl	Channel zero synchronization

7.7.4 Instructions

DM destination address output
The destination offset in COPM is generated depending on the logical channel assigned to:

- 0 for channel #0;
- 4 for channel #1.

Operands (data word input) format

bits 63-32	bits 31-00
Operand B (float32 type)	Operand A (float32 or signed integer type)

The Operand A type is determined by the instruction bit 5.

Instruction format
In next table Z represents the (co-processor internal) multiplier-accumulator register. The 32-bit result is computed as indicated thereafter.

5	4-0	
`0 = A is float` `1 = A is signed integer`	`00000`	`A+B`
	`00001`	`A-B`
	`00010`	`A*B`
	`00011`	`A/B`
	`00100`	`float(A); A is signed integer;` `bit 5 not used.`
	`00101`	`signed integer(A); A is float;` `bit 5 not used.`
	`00111`	`Z`
	`10000`	`Z := A; COPM not touched.`
	`10001`	`Z := Z+(A*B); COPM not touched.`

7.7.5 Synthesis warning!

The floating point co-processor description assumes that the *IEEE float_pkg* package is known from the synthesis tool. At XILINX for example, this is true for VIVADO but *not* for ISE; thus XILINX 6-family and before *cannot* infer the floating point co-processor. Also some tools are searching for the *ieee* library whereas others are using *ieee_proposed*. The source may need be edited depending on the synthesis tool.

7.7.6 Area estimates

Rough estimates follow. Target technology: XILINX KINTEX7.
FP32_LEVEL = 0: ~300 slices (0.6 % of medium range XC7K325), 2 DSP cells.
FP32_LEVEL = 1: ~365 slices (0.7 % of medium range XC7K325), 2 DSP cells.

7.8 Fixed point co-processor

7.8.1 Overview

A co-processor is available that implements basic fixed point instructions. An accumulator is also included as an option.

The "fixed point" arithmetic is essentially integer arithmetic, but the data is split into an integer and a fraction part. The point position delimits the integer from the fraction part. It needs additional care for "fixed".

The following boundary conditions should be met.

- Both operands are scaled the same, that is with the point at the same position.
- The result is also scaled the same.

Then it results the following.

- Addition or subtraction result do not need any scaling.
- Multiplication result do need shifting right (by the fraction size) to recover the correct point position.

As implemented the "fixed" co-processor actually operates on integers. The integer range is determined by the generic FIXED_FSIZE. It is up to the application to know the point position at all times.

7.8.2 Generics and ports

Name	Type	Description
COP_ID	std3	Co-processor identifier
FIXED_FSIZE	natural	Operand full size (integer + fraction) min. 32, max. 48 (recommended but not constraint).
FIXED_LEVEL	natural	0 = Basic 1 = Add the internal accumulator

Ports are those already described in the floating point co-processor section.

7.8.3 Instructions

DM destination address output
The destination offset in COPM is generated depending on the logical channel assigned to:

- 0,1 for channel #0;
- 4,5 for channel #1.

Two 32-bit words are written back.
In the case of the multiplication the result (which may overflow the 64-bit width) is saturated to 64 bits (however after shifting, see thereafter).

Operands input

In general there are two operands, each wider than 32 bits.
So a sequence of two (YAM 32-bit) COP instructions should be used.

- First instruction loads the A operand.
- Next instruction loads the B operand and (starts) the operation.

Instruction format

In next table Z represents the (co-processor internal) internal accumulator register. The 32-bit result is computed as indicated thereafter.

7-4	3-0	
Fraction shift size (N=0-15)	0000	Load B; (A+B)>>N; result to COPM
	0001	Load B; (A-B)>>N; result to COPM
	0010	Load B; (A*B)>>N; result to COPM
	0011	Load B; B>>N; result to COPM
	0100	Load A
	0111	Z result to COPM (also loads A)
	1000	Z := Z+(A+B)>>N; COPM not touched.
	1001	Z := Z+(A-B)>>N; COPM not touched.
	1010	Z := Z+(A*B)>>N; COPM not touched.
	1100	Z := A; COPM not touched.

Note:
The specific operation B>>N operates on the full 64-bit word; it is not limited to the FIXED_FSIZE range.

Note:
N is zero (don't shift) for dumb integer processing.
Shift is arithmetic to the right direction.

8 System On Chip (SOC)

8.1 Overview

The IO system connects to the YAM IO port interface.

Any user's custom logic may be directly attached to the IO port. This could be the case when simple interfacing is enough, like for a few external registers.

However an intermediate IO interconnect core is available to help when dealing with larger IO subsystems. It will be used to hook the IO system onto YAM.
YAM is supported by a number external components. The purpose of this section is twofold.

- Describes the existing SOC components included in the YAM delivery.
- Promotes standardization for SOC user's developments.

The IO system in a whole may be of either of two types.

- Synchronous: it is clocked by YAM clock.
- Asynchronous: it uses its own clock. In the latter case the IO clock must be slower than the YAM clock. Clock domains crossing it managed by the interconnection.

Some simple rules are strongly recommended for using it.

Rule 1
The address most significant bits (typically 2-3 of them) must be decoded to further split the IO system into several (typically 4-8) subsystems (numbered #0, #1, …).

Rule 2
Subsystem#0 is reserved and named *YIOZ* (YAM IO subsystem Zero). It is used by YAM for implementing basic ancillary components.

Rule 3
Every subsystem may have a 16-bit status register that is connected to subsystem#0. This enforces software standardization in the management of commonly used subsystems (Whishbone, E-bone, AXI4-lite, …).

8.2 IO core interconnect

8.2.1 Overview

Except for the case YIOZ (or some other) is a single component hooked to the IO bus, an IO core interconnect component should be used. It is responsible for the following functions.

- Decode the address most significant bits to generate the dedicated subsystem selector.
- Multiplex the data read from the different subsystems to select the active one.
- Generate the *io_wait* signal (multiplex from the subsystems).
- Generate the *io_busy* signal.
- In the asynchronous case, manage clock domains crossing.

8.2.2 Source files

IO interconnect source files are located in the /hdl/yioz directory.
Four variants (32- and 16-bit data width combined with synchronous and asynchronous) are delivered.
The asynchronous IO subsystem interconnection variants must be used when YAM clock differs from IO clock.

o_core32_s.vhd	IO subsystem interconnection, 32-bit data width, synchronous
io_core16_s.vhd	IO subsystem interconnection, 16-bit data width, synchronous
io_core32_a.vhd	IO subsystem interconnection, 32-bit data width, asynchronous
io_core16_a.vhd	IO subsystem interconnection, 16-bit data width, asynchronous

The required types and component declarations are in the *yam_pkg.vhd* main package.

8.2.3 Generics

Name	Type	Description
SUBS_NB	natural	Number of subsystems
REG_WIDTH	natural	YAM register word width The IO system data width may differ. Data will be re-sized to YAM register width at the interface.

The *reversed* ordered MSbits of the address are decoded to select a given subsystem, as follows.

MSbits	Subsystem #
000...	0
100...	1
010...	2
110...	3
001...	4

So the addressing of a given subsystem is independent from the total number of subsystems.

8.2.4 Ports

Asynchronous case

Name	Dir.	Type	Description
clk_i	in	stdl	YAM system clock
io_XXX			to/from YAM IO ports
io_busy_o	out	stdl	IO busy flag, actually write pending status.
subs_clk_i	in	stdl	IO system clock
subs_sel_o	out	stdlv	Decoded subsystem selectors
subs_wr_o	out	stdl	IO write
subs_wr_o	out	stdl	IO read
subs_adrs_o	out	stdlv	IO address
subs_datw_o	out	stdlv	IO data write
subs_datr_i	in	stdlv array	Data read from the different subsystems
subs_wait_i	in	stdlv	*io_wait* signals from the different subsystems

Synchronous case

Name	Dir.	Type	Description
io_XXX			to/from YAM IO ports
io_busy_o	out	stdl	IO busy flag, forced to zero.
subs_sel_o	out	stdlv	Decoded subsystem selectors
subs_wr_o	out	stdl	*io_wr_i* pass through
subs_wr_o	out	stdl	*io_rd_i* pass through
subs_adrs_o	out	stdlv	*io_adrs_i* pass through
subs_datw_o	out	stdlv	*io_datw_i* pass through
subs_datr_i	in	stdlv array	Data read from the different subsystems
subs_wait_i	in	stdlv	Wait signals from the different subsystems

8.2.5 Low level IO assembler programming

IO read
In that case YAM is stalled (by the *io_wait* signal) until the data input becomes available. No problem regarding the next instruction.

IO write, asynchronous IO system
YAM core posts the IO write and immediately fetches the next instruction. The problem arises when the next instruction is an IO read, as the previous write is still pending. Inserting enough extra instructions between the write and the read instruction would solve the problem. But how many is “enough” might be difficult to guess by the programmer, and not portable. The clean way is to test the BIO (Busy IO) flag, actually the status of the *io_busy* signal generated by the interconnection. An example follows.

```
        MOVIO @%00, #D16  ; writing out
pausio  JMP pausio, BIO   ; loop until IO is done
        MOVIO %01, @%02   ; reading in is now safe
```

IO write, synchronous IO system
The *io_busy* signal is forced to zero by the interconnection. Keeping the same example:

```
        MOVIO @%00, #D16  ; writing out
pausio  JMP pausio, BIO   ; execute only once when synchronous
        MOVIO %01, @%02   ; reading in is now safe
```

The JMP quits immediately. It meets the general requirement of inserting some instruction in a write-read sequence, without loosing extra time. So the same program coding style may be used in both synchronous and asynchronous cases.

IO write – IO read at same address, synchronous IO system
Under these conditions the next example works:

```
        MOVIO @%00, #D16  ; writing out
        MOVIO %01, @%00   ; reading in from same address is OK
```

8.3 YIOZ

8.3.1 Overview

The YIOZ subsystem complements the YAM core with some ancillary components. These are:

- Status registers ready for feeding from others subsystems.
- Core timer logic.
- Breakpoint logic.
- Simple UART peripheral; fixed S-8-S format; no parity; fixed baud rate.
- Reserved for extensions.

8.3.2 Source files

YIOZ source files are found in the /hdl/yioz directory.

yioz.vhd	YIOZ top level
yioz_ioif.vhd	YIOZ registers interfacing logic
yioz_bkpt.vhd	YIOZ breakpoint logic
yioz_ctim.vhd	YIOZ core timer logic
yioz_irq.vhd	YIOZ interrupt logic
yioz_uart.vhd	YIOZ simple UART (asynchronous serial peripheral)

The YIOZ component declaration is in the *yam_pkg.vhd* main package.

8.3.3 Generics and ports

All generics must be identical to those for the YAM core instantiation.
New generics specific to YIOZ, are as follows.

Name	Type	Description
YIOZ_CLK	natural	IO subsystem clock frequency in MHz. This value is used for configuring a 1 MHz pre-scaler to feed the timer and UART. If YAM_CLK equals YIOZ_CLK it is assumed that YAM clock and IO clock are physically the same. That is a fully synchronous design is implemented. Else (asynchronous case) some clock domain crossing logic will be inserted.
UART_RATE	natural	UART peripheral baud rate. Max. 38400. To be used together with YAM_CLK for configuring the baud rate generator.
BKPT_IF	boolean	Adds support for breakpoint logic when true.
UART_IF	boolean	Adds support for UART peripheral when true.

Most ports connect to YAM (synchronous case) or to the IO subsystem core interconnect (synchronous case). Only specific ports are addressed thereafter.

Name	Dir.	Type	Description
yam_clk_i	in	stdl	YAM system clock
clk_i	in	stdl	IO subsystem clock
yioz_sel_i	in	stdl	Subsystem#0 select Connect to *io_core_x* interconnect *subs_sel_o(0)*.
subs_stat_i	in	stdlv array	Collects the status from the different subsystems
yioz_irq0_i	in	stdl	Free interrupt request
yioz_irq1_i	in	stdl	Free interrupt request
uart_rx_i	in	stdl	UART receiver input
uart_tx_o	out	stdl	UART transmitter output

8.3.4 Programming model

YIOZ internal registers are 16 bit in width.

<table>
<tr><th>Offset</th><th>Type</th><th>Name</th><th>Description</th></tr>
<tr><td>0</td><td>R/W</td><td>YIOZ_CTRL</td><td>
<table>
<tr><td>bit</td><td></td></tr>
<tr><td>0</td><td>CTimer mode
0 = free run, no IRQ
1 = On threshold IRQ and self-clear.</td></tr>
<tr><td>1</td><td>CTimer enable</td></tr>
<tr><td>3-2</td><td>Reserved</td></tr>
<tr><td>4</td><td>Breakpoint IRQ enable</td></tr>
<tr><td>5</td><td>CTimer IRQ enable</td></tr>
<tr><td>6</td><td>IRQ0 enable</td></tr>
<tr><td>7</td><td>IRQ1 enable</td></tr>
<tr><td>8</td><td>UART IRQ enable</td></tr>
</table>
</td></tr>
<tr><td>1</td><td>R/W</td><td>YIOZ_BKPTA</td><td>Breakpoint address.
When entering the breakpoint interrupt service routine the instruction at this address has already been executed.</td></tr>
<tr><td>2</td><td>R/W</td><td>YIOZ_CTIMUP</td><td>CTimer threshold (say N), in us.
Because the 1 MHz pre-scaler is free running, the first crossing value is anywhere in the interval [N-1, N].
Writing in also clears the timer counter.</td></tr>
<tr><td>3</td><td>R/W</td><td>YIOZ_TX</td><td>UART TX data (8 bits)
Writing in starts the transmitter.</td></tr>
<tr><td>4-8</td><td></td><td>Reserved</td><td></td></tr>
<tr><td>9</td><td>R</td><td>YIOZ_RX</td><td>UART RX data (8 bits)
Reading out clears the RX ready flag.</td></tr>
<tr><td>10</td><td>R</td><td>YIOZ_BKPTI</td><td>Breakpoint instruction reached, 16 MSbits.</td></tr>
<tr><td>11</td><td>R</td><td>YIOZ_CTIMRD</td><td>CTimer value, in us.</td></tr>
<tr><td>12</td><td>R</td><td>YIOZ_STAT</td><td>
<table>
<tr><td>bit</td><td></td></tr>
<tr><td>3-0</td><td>IRQ status: irq1 - irq0 - CTimer - Breakpoint.</td></tr>
<tr><td>4</td><td>UART RX ready; UART IRQ on rising edge.</td></tr>
<tr><td>5</td><td>UART TX busy; UART IRQ on falling edge.</td></tr>
</table>
</td></tr>
<tr><td>15-13</td><td>R</td><td></td><td>Reserved</td></tr>
<tr><td>31-16</td><td>R</td><td></td><td>Subsystems status registers</td></tr>
</table>

8.3.5 Data width resizing

YIOZ is designed to fit any YAM data bus width (REG_WIDTH generic). Internal YIOZ data width is resized to YAM data width when interfacing to the IO bus. However depending on the actual REG_WITH value some restrictions may apply. Next table comments on the limitations for popular data width values.

REG_WIDTH	
8	YIOZ registers are truncated to bits 7-0. So some information is lost. YIOZ might still be used however as most critical data are located in bits 7-0. Mind breakpoint address is limited to 255!
16 and more	No limitations.

8.3.6 Asynchronous case

YIOZ determines if it is used in an asynchronous environment when the generics YAM_CLK and IO_CLK differ. A few specific logic is then inserted to manage the interrupt interfacing to YAM.

8.4 E-bone simple master subsystem

8.4.1 Description

This subsystem provides a minimal E-bone master. Multiple data burst, broad-call and broad-cast are not supported. A single data is transferred by executing a MOVIO instruction. Addressing the segment 1 is assumed. YAM data width is resized to 32 to fit the E-bone data width.
The generic EBS_AD_RNGE limits the actual E-bone addressing range.

E-bone interfacing source files are found in the /hdl/ebone directory.
Source files are:

io_ebm.vhd	E-bone simple master
io_ebm_pkg.vhd	E-bone simple master package for component declaration

8.4.2 Write cycle

Data is write posted and YAM continues with next instruction. Finishing off the write cycle takes some more clocks periods. So two E-bone write instructions cannot be to close, or the status register must be checked.

8.4.3 Read cycle

The YAM sequencer waits until the incoming data is ready (*io_wait_i* signal handshake). Then execution continues.

8.4.4 Status register

bit	Meaning
0	E-bone cycle in progress
1	E-bone error memorized. Cleared a every cycle start.

8.5 Wishbone simple master subsystem

8.5.1 Description

This subsystem provides a minimal Wishbone master. Multiple data burst are not supported. A single data is transferred by executing a MOVIO instruction. YAM data width is resized to 32 to fit the Wishbone data width.
The generic WBS_AD_RNGE limits the actual Wishbone addressing range.

Wishbone interfacing source files are found in the /hdl/wbone directory.
Source files are:

wbs_core.vhd	Whishbone simple core interconnect
io_wbm.vhd	Wiishbone simple master
io_wbm_pkg.vhd	Package for components and dedicated types declaration

8.5.2 Write cycle

Data is write posted and YAM continues with next instruction. Finishing off the write cycle takes some more clocks periods. So two Wishbone write instructions cannot be to close, or the status register must be checked.

8.5.3 Read cycle

The YAM sequencer waits until the incoming data is ready (*io_wait_i* signal handshake). Then execution continues.

8.5.4 Status register

bit	Meaning
0	Wishbone cycle in progress
1	Wishbone error memorized. Cleared a every cycle start.
2	Time out on endless cycle.

8.6 AXI4 lite simple master subsystem

8.6.1 Description

This subsystem provides a minimal AXI4 lite master. A data is transferred by executing a MOVIO instruction. YAM data width is resized to 32 to fit the AXI4-lite data width.
The generic AXI_AD_RNGE limits the actual AXI4 addressing range.

AXI4 lite interfacing source files are found in the /hdl/axil directory.
Source files are:

io_axil.vhd	AXI4 lite master
io_axil_pkg.vhd	Package for components and dedicated types declaration

8.6.2 Write cycle

Data is write posted and YAM continues with next instruction. Finishing off the write cycle takes some more clocks periods. So two AXI4 lite write instructions cannot be to close, or the status register must be checked.

8.6.3 Read cycle

The YAM sequencer waits until the incoming data is ready (*io_wait_i* signal handshake). Then execution continues.

8.6.4 Status register

bit	Meaning
0	AXI4 cycle in progress
1	Zero (AXI4 lite does not actually support the response channel)
2	Time out on endless cycle.

9 Annex: YAM instructions summary

“Flags” means ALU flags updated under S bit control

“AL” means ALU implementation level

Mnemonic	Flags	AL (Note)	Description	Destination	Source
AND	Z	0	Logical AND	%dd	%ss; #D16
OR	Z	0	Logical OR	%dd	%ss; #D16
XOR	Z	0	Logical XOR	%dd	%ss; #D16
TST	Z	0	Logical AND without assignment	%dd	%ss; #D16
ADD	Z, N	0	Unsigned ADD	%dd	%ss; #D16
ADDC	Z, N	0	Unsigned ADD with carry in	%dd	%ss; #D16
SUB	Z, N	0	Unsigned SUB	%dd	%ss; #D16
SUBC	Z, N	0	Unsigned SUB with carry in	%dd	%ss; #D16
SUBS	Z, N	0	Signed SUB	%dd	%ss; #D16
CMP	Z, N	0	Unsigned SUB without assignment	%dd	%ss; #D16
CMPS	Z, N	0	Signed SUB without assignment	%dd	%ss; #D16
MUL		(**)	Unsigned Multiply	%dd	%ss; #D16
SHL	Z, N	0/1(*)	Shift left logical by 0-15	%dd	%ss; #N
SHR	Z, N	0/1(*)	Shift right logical by 0-15	%dd	%ss; #N
SHRA	Z, N	0/1(*)	Shift right arithmetical by 0-15	%dd	%ss; #N
SHL16	Z, N	1	Shift left logical by 16	%dd	%ss
ROTL	Z, N	1	Rotate left by one	%dd	%ss
ROTR	Z, N	1	Rotate right by one	%dd	%ss
ABS	Z, N	1	Absolute value	%dd	%ss
CLZ	Z, N	1	Count leading zeros	%dd	%ss
BSTN	Z, N	2	Byte store	%dd	%ss
BLDN	Z, N	2	Byte load	%dd	%ss
BLDSN	Z, N	2	Byte load, signed	%dd	%ss
LDYX	Z, N	2	Load RY, RX		
MOVX	Z, N	2	Move with indexing	@%dd, @%X+	@%ss, @%X+

JMP			Jump	@%dd; label	
JSR			Jump to subroutine	@%dd; label	
JSRWP		(**)	Jump to sub. with context switch	label, offset	
RET			Return form subroutine		
RLD			Return and load register	%dd	#D16
RETWP		(**)	Return with context switch		
MOV			Move	%dd; @%dd; &AAA	%ss; @%ss; #D16; &AAA
MOVIO		(**)	Move to/from IO bus (input/output)	%dd; @%dd	%ss; @%ss; #D16
IORMW		(**)	IO bus read-modify-write	%dd	@%ss
SPR_ON		(**)	Switch to SPR readout		
SPR_OFF		(**)	Switch to ALU normal state		
COPx		(**)	co-processor activate		%ss - #D16
IRQ_ON		(**)	IRQ mask bitwise clear	mask	
IRQ_OFF		(**)	IRQ mask bitwise set	mask	
IRQM_SS		(**)	IRQ mask save and set		
IRQM_RLD		(**)	IRQ mask reload from saved		
RETI		(**)	Return from Interrupt		

(*)
When AL=0 shift is restricted to one position.

(**)
Available depending on YAM generics configuration.

10 Annex: YAM distribution tree

YAM distribution includes both the core HDL and the assembler. It is organized as follows.

documentation/ YAM hardware related documentations
firmware/
    hdl/ YAM core
    src/libyam Referenced by the assembler
tools/yamasm YAM assembler

11 Revision history

Rev.	Date	Who	Comment
0.1	09/11/15	herve	Preliminary, YAM 0.1 release
0.2a	01/12/15	herve	Preliminary, YAM 0.2 release
0.2b	06/01/16	herve	Updated, YAM 0.2 release. Added co-processor interface. Added IO subsystem and YIOZ.
0.3a	19/01/16	herve	Updated, YAM 0.3 release. Removed External register. Revisited Immediate data management. Added selectable update of ALU flags.
0.3b	12/07/16	herve	Added asynchronous IO system and BIO flag. Added 32-bit floating point co-processor. Added Multiplier. Data Memory synthesis issue fixed.
1.0	24/08/16	herve	Added special registers. First release; Full suite test passed.
1.1	20/09/16	herve	Bug N flag fixed. Bug dis-assembler fixed. Remove SHRA16 instruction. Add SUBC instruction. Add ABS instruction to ALU level 1. Add Byte move instructions to ALU level 2. Add co-processor dedicated result memory. Add co-processor result input port. Add COPM_SIZE generic; remove COP_NB generic.
1.2	20/10/16	herve	Add self-incrementing index pointers to register memory and related LDYX, MOVX instructions. Add SPR7 multiplier MSW. Add debugging/stepping mode. Add YAM_DBG generic and *dbg_rq_i, dbg_wr_o* ports.
1.3	20/11/16	herve	Add fixed point co-processor. Floating point co-processor revisited. Bug COP Memory indirect addressing fixed.
1.4	16/03/17	herve	Instruction set upgrade to 3 operands, selectable ALU level 1 instructions are all becoming 2-operand type. Added IORMW instruction and associated generic IO_RMW. External address ports are now constraint in size. Bug ABS instruction flag Z is incorrect has been fixed. Fixed point co-processor synthesis failure fixed.