HOKSTER Reference Manual

Update 6

05/26/2020

Description of architecture

HOKSTER is a reduced instruction set computer (RISC) with only 38 pre-defined commands, but with room for future expansion. It is a load-store architecture, where most arithmetic operations can take place only between registers.

Memory

This is a Harvard architecture, where program bus (***p-bus***) and data bus (***d-bus***) are separated. The program bus is fixed at 8 bits, and fetches the next 8-bit program instruction into `progword`. However, the data bus can be defined generically as 8, 16 or 32 bit at synthesis time. The 8-bit data bus is the HOKSTER baseline configuration; 16- and 32-bit capability is reserved for future expansion.

The program memory (PRAM) consists of up to 2^{12} (4096) bytes. The data memory (DRAM) consists of up to 2^{16} bytes (65536). Available memory is defined at synthesis time. All memory addressing is performed using (up to) 16 bit addressing. Memory addresses (both to internal DRAM and external peripherals) are communicated on the address bus (***a-bus***).

Registers

There are 16 general purpose 8-bit registers, **a7** .. **a0** and **r7** .. **r0**. For most operations with source <src> and destination <dst>, registers are indexed as 15 ..0, where **a7** = 15, **a6** = 14, … **a0** = 8, **r7** = 7, … **r0** =0. However, to allow 16-bit memory addressing, commands operating with a source or destination as a memory address use pairs of **a** and **r** registers, e.g., **a7** & **r7**, **a6** & **r6**, … **a0** & **r0**, where $\mathbf{a}_i$ are the upper 8 bits of an address, and $\mathbf{r}_i$ are the lower 8 bits of an address. This 16-bit convention is likewise used to read and write to the special purpose interrupt enable (**ie**) register, which has up to 16 bits corresponding to up to 16 interrupts.

There are several special purpose registers:

Program counter (**pc**) is up to 12 bits, allowing for up to 2^{12} bytes of program memory.

Stack pointer (**sp**) is up to 16 bits. The stack pointer is initialized to a valid memory address by the user, which is usually the top of defined memory (though not to top of available address space), and grows downward during stack usage. As there is no pre-segmentation of stack versus global memory, care must be exercised not to overwrite memory during stack-intensive operations.

Status register (**sr**) is 8 bits, and contains pre-defined flags X, C, N, Z, as well as 4 user-defined flags. The status register can be saved to the stack using the str command, and restored from the stack using the lsr command. Status register is as follows: - - - - X C N Z. Z = zero flag (set on arithmetic result of zero), N = negative flag (set when most significant bit is set to 1 in arithmetic operations), C = carry flag (set when carry is generated during addition, or borrow during subtraction). X is reserved for user definition.

Interrupt vectors (**iv**). There are up to 16 **iv** consisting of 8 bits each. If an interrupt occurs on an enabled interrupt, program control will be transferred to the 12-bit address, consisting of the 8 bits stored in the corresponding **iv**, shifted left 4 bits, i.e., **pc** = <**iv**[i]> << 4. Interrupt vectors are initialized with the mvv instruction. The **pc** will be stored in a shadow register.

Interrupt enable (**ie**). There is 1 16-bit **ie** register, where each bit pertains to an interrupt i15..i0. The rie command is used to read from the **ie** register, and sie command is used to write to the **ie** register. To mask individual interrupts, one can read **ie** to a general purpose register pair $\mathbf{a}_i$ & $\mathbf{r}_i$, perform required logical operations, and write result to ie using sie. Lower-numbered interrupts are considered to have higher priority. There is no general interrupt enable; all desired interrupts must be explicitly enabled by the user.

Functional Units

There are two arithmetic logic units: alu and aluc. The pre-defined unit is alu, and consists of standard arithmetic functions, such as add, sub, and logical operations. adi and sbi can perform increment or decrement operations on single operands from 1 to 16. All alu instructions execute in 2 clock cycles.

There is an alu for custom instructions, called aluc. This allows custom operations on two operands, as defined by the user. All aluc instructions execute in a variable number of clock cycles, from 2 to an indefinite amount. At an aluc instruction, execution will halt until the `wait_req` signal is deasserted by the corresponding hardware instruction set extension. See the section on Instruction Set Extensions for more information.

Peripherals

HOKSTER supports memory-mapped peripherals which use the same 2^{16} address space available to DRAM. Peripherals are addressed using the `auxaddr` signal, and transmit data from processor to peripherals using `auxdin` signal. Data returns to the processor from peripherals using the `auxdout` bus, which enables external peripherals to write to HOKSTER data bus by asserting the `auxdoutsel` signal. Up to 16 peripherals can independently assert interrupts to HOKSTER via the ***i-bus***.

Interrupts

HOKSTER supports interrupts from external peripherals. Up to 16 prioritized interrupts can be supported, i0 .. i15, where i0 has the highest priority. Interrupts are individually enabled through the 16-bit **ie** register, where a 1 indicates that an interrupt is enabled, and 0 indicates that an interrupt is disabled. Upon receipt of an external interrupt through the ***i-bus***, the corresponding bit of the ***i-bus*** is ANDed with the bit of **ie**. The highest priority enabled interrupt is then priority-encoded, and signals the core to process an interrupt. Upon processing an interrupt, a set of shadow registers will be used to save the current state of registers. The **pc** and **sr** will be stored in shadow registers **pcs** and **srs** respectively. The number of general purpose register pairs (a_i, r_i) saved, N_S, can be defined generically between 1 and 8. Registers a_i, r_i are stored in shadow registers as_i, rs_i for i = 0 to (N_s – 1). At least 1 pair of $\mathbf{a}_i$, $\mathbf{r}_i$ will always be saved, while N_s = 8 means all 16 general purpose registers are saved. The program then branches to the address identified in the corresponding interrupt vector (**iv**) register.

In the interrupt service routine (ISR), it is recommended that the user query the peripheral as necessary, and acknowledge the interrupt using the sys command, by encoding the interrupt to be acknowledged in the sys instruction. The user must then use the rti command to return from the interrupt to normal program flow.

Interrupts are individually enabled and disabled through the rie and sie commands, and interrupt vectors are initialized using the mvv commands. There is no need to disable interrupts when entering the ISR, as nested interrupts are not supported.

Instruction Set Architecture

Instructions are issued 8 bits at a time on progword[7..0]. Instructions execute in a variable number of clock cycles. The number of clock cycles depends on the form of the instructions. Abbreviations are <opc1> = Opcode 1, <opc2> = Opcode 2, where opcodes are each 4-bits in length. Im[b..a] denotes an immediate operand with bits b down to a (where b is the most significant bit), and indicates a value encoded directly into the program. <src>, <src1>, and <src2> refer to source operands, and <dst> refers to destination operand. $\mathbf{a}_i$ and $\mathbf{r}_i$ refer to the ith general purpose register a and r, respectively. $\mathbf{a}_i$ & $\mathbf{r}_i$ indicates a concatenation of $\mathbf{a}_i$ and $\mathbf{r}_i$ (valid for 8-bit datapaths only).

A description of the ISA follows:

<opc1>

0000 mvs **sp** = Im[11..0]: Move (immediate) to stack pointer. Used to initialize stack pointer. The value written into **sp** is (Im[11..0] << 4) - 1, i.e., a 16-bit value is written into **sp**. The value of **sp** is the user's choice, however, is normally set to the top of physically defined DRAM. Executes in 2 clock cycles; 1st cycle recognizes mvs and Im[11.8]; 2nd cycle completes the operation.

<opc1>

0001 mvv <opc2=dst> = Im[7..0]: Move (address) to interrupt vector register **iv15** … **iv0** specified in `progword`[3..0] = <dst>. The address in next `progword`, Im[7..0] is the upper 8 bits of 12 bit interrupt vector. Executes in 2 clock cycles; 1st cycle: Recognize mvv <dst>. 2nd cycle: Complete operation.

<opc1>

0010 jmp **pc** = Im[11..0]: Jump (unconditionally) to address specified in Im[11..0], where upper 4 bits of Im[11.0] are the lower 4 bits of `progword`[3..0], and lower 8 bits are contained in next `progword`. Executes in 2 clock cycles; 1st cycle: Recognize jmp and store Im[11..8]. 2nd cycle: **pc** = Im[11..8] & Im[7..0].

<opc1>

0011 jsr **pc** = Im[11..0]: Jump to subroutine specified in Im[11..0], where upper 4 bits of Im[11.0] are the lower 4 bits of `progword`[3..0], and lower 8 bits are contained in next `progword`. Return address (**pc** + 2) is stored on stack. Executes in 2 clock cycles; 1st cycle: Recognize jsr, store Im[11..8], [**sp**] = (**pc**+2)[11..8], **sp** = **sp** – 1; 2nd cycle: **pc** = Im[11..8] & Im[7..0], [**sp**] = (**pc** + 1)[7..0], **sp** = **sp** – 1.

<opc1>

0100 bzi **pc** = Im[11..0]: Branch (conditionally) if Z flag set, to address specified in Im[11..0], where upper 4 bits of Im[11.0] are the lower 4 bits of `progword`[3..0], and lower 8 bits are contained in next `progword`. Executes in 2 clock cycles; 1st cycle: Recognize bzi and store Im[11..8]. 2nd cycle: **pc** = Im[11..8] & Im[7..0]. Does not affect **sr**.

<opc1>

0101 bni **pc** = Im[11..0]: Branch (conditionally) if N flag set, to address specified in Im[11..0], where upper 4 bits of Im[11.0] are the lower 4 bits of `progword`[3..0], and lower 8 bits are contained in next `progword`. Executes in 2 clock cycles; 1st cycle: Recognize bni and store Im[11..8]. 2nd cycle: **pc** = Im[11..8] & Im[7..0]. Does not affect **sr**.

<opc1>

0110 bci **pc** = Im[11..0]: Branch (conditionally) if C flag set, to address specified in Im[11..0], where upper 4 bits of Im[11.0] are the lower 4 bits of `progword`[3..0], and lower 8 bits are contained in next `progword`. Executes in 2 clock cycles; 1st cycle: Recognize bci and store Im[11..8]. 2nd cycle: **pc** = Im[11..8] & Im[7..0]. Does not affect **sr**.

<opc1>

0111 bxi **pc** = Im[11..0]: Branch (conditionally) if X flag set, to address specified in Im[11..0], where upper 4 bits of Im[11.0] are the lower 4 bits of `progword`[3..0], and lower 8 bits are contained in next `progword`. Executes in 2 clock cycles; 1st cycle: Recognize bxi and store Im[11..8]. 2nd cycle: **pc** = Im[11..8] & Im[7..0]. Does not affect **sr**.

<opc1>

1000 mvi <dst> = Im[7..0]: Move (immediate) 8-bit value Im[7..0] to register specified in <dst>. Lower 4 bits of `progword`[3..0] are <dst>, and Im[7..0] is contained in next `progword`. Executes in 2 clock cycles; 1st cycle: Recognize mvi and store <dst>. 2nd cycle: <dst> = Im[7..0].

<opc1>

1001 alu

Alu operations execute in 2 clock cycles. The 1st 8-bit word consists of <opc1=alu><opc2>. The second 8-bit word is formatted as <src1><src2=dst> or Im[3..0]<src=dst>. Some alu operations affect **sr**. All changes to **sr** are described. In the absence of a specific change, **sr** is not affected.

<opc2>

0000 add <src1> + <src2=dst> = <dst>: Adds <src1> to <src2> and stores result in <src2=dst>. Updates Z, N, C, X flags. Executes in 2 clock cycles; 1st cycle: Recognize alu + add. 2nd cycle: Complete operation.

0001 sub <src1> - <src2=dst> = <dst>: Subtracts <src2> from <src1> and stores result in <src2=dst>. Updates Z, N, C, X flags. Executes in 2 clock cycles; 1st cycle: Recognize alu + sub. 2nd cycle: Complete operation.

0010 and <src1> and <src2=dst> = <dst>: ANDs <src1> to <src2> and stores result in <src2=dst>. Executes in 2 clock cycles; 1st cycle: Recognize alu + and. 2nd cycle: Complete operation.

0011 lor <src1> (logical) or <src2=dst> = <dst>: Logically ORs <src1> to <src2> and stores result in <src2=dst>. Executes in 2 clock cycles; 1st cycle: Recognize alu + lor. 2nd cycle: Complete operation.

0100 sll <src2=dst> << <src1>= <dst>: Logically shifts <src2> left the number of bits indicated in <src1> and stores result in <src2=dst>. Executes in 2 clock cycles; 1st cycle: Recognize alu + sll. 2nd cycle: Complete operation.

0101 rol <src2=dst> <<< <src1>= <dst>: Logically rotates <src2> left the number of bits indicated in <src1> and stores result in <src2=dst>. Executes in 2 clock cycles; 1st cycle: Recognize alu + rol. 2nd cycle: Complete operation.

0110 srl <src2=dst> >> <src1>= <dst>: Logically shifts <src2> right the number of bits indicated in <src1> and stores result in <src2=dst>. Executes in 2 clock cycles; 1st cycle: Recognize alu + srl. 2nd cycle: Complete operation.

0111 ror <src2=dst> >>> <src1>= <dst>: Logically rotates <src2> right the number of bits indicated in <src1> and stores result in <src2=dst>. Executes in 2 clock cycles; 1st cycle: Recognize alu + ror. 2nd cycle: Complete operation.

1000 Reserved

1001 Reserved

1010 not !<src> -> <dst>: Takes bitwise complement of <src> stores result in <dst>. Executes in 2 clock cycles; 1st cycle: Recognize alu + not. 2nd cycle: Complete operation.

1011 xor <src1> (exclusive) or <src2=dst> = <dst>: XORs <src1> to <src2> and stores result in <src2=dst>. Executes in 2 clock cycles; 1st cycle: Recognize alu + xor. 2nd cycle: Complete operation.

1100 adc **sr**[c] + <src1> + <src2=dst> = <dst>: Adds **sr**[c] + <src1> to <src2> and stores result in <src2=dst>. Updates Z, N, C, X flags. Executes in 2 clock cycles; 1st cycle: Recognize alu + adc. 2nd cycle: Complete operation.

1101 sbc <src1> - <src2=dst> - **sr**[c] = <dst>: Subtracts <src2> and **sr**[c] from <src1> and stores result in <src2=dst>. Updates Z, N, C, X flags. Executes in 2 clock cycles; 1st cycle: Recognize alu + sbc. 2nd cycle: Complete operation.

1110 adi Im[3..0] + 1 + <src=dst> = <dst>: Adds 4-bit immediate value Im[3..0] + 1 to <src> and stores result in <src=dst>. Updates Z, N, C, X flags. Executes in 2 clock cycles; 1st cycle: Recognize alu + adi. 2nd cycle: Complete operation.

1111 sbi <src=dst> - (Im[3..0] + 1) = <dst>: Subtracts 4-bit immediate value Im[3..0] + 1 from <src> and stores result in <src=dst>. Updates Z, N, C, X flags. Executes in 2 clock cycles; 1st cycle: Recognize alu + sbi. 2nd cycle: Complete operation.

<opc1>

1010 aluc

Aluc operations execute in 2 + x clock cycles, where x is any integer greater than or equal to 0. The 1st 8-bit word consists of <opc1=aluc><opc2>. The second 8-bit word is formatted as <src1><src2=dst> or Im[3..0]<src=dst>. These operations can affect **sr**. See the section on Instruction Set Extensions for more information about the hardware implementing aluc operations.

<opc2>

TBD

<opc1>

1011 reserved

<opc1>

1100 General instruction 1 (gen1)

<opc2>

0000 mov <src><dst>: Moves value in register <src> to register <dst>. Requires 2 clock cycles. 1st clock cycle recognizes gen1 mov; 2nd clock cycle recognizes <src> and <dst>, and completes operation.

0001 ldb (ldh, ldw) [src]<dst>: Load value in memory to register, with no auto post-increment. Memory address is specified in $\mathbf{a}_i$ & $\mathbf{r}_i$, where 3-bit i (7..0) is specified in 3-bit <src>, which is in `progword`[6..4]. Register <dst> is specified in `progword`[3..0]. `progword`[7] = 0 (no auto-post increment). Executes in 2 clock cycles, where 1st cycle gen1 ldb (ldh, ldw) are recognized, and operation completes on 2nd clock cycle.

0001 lpb (lph, lpw) [src]<dst>: Load value in memory to register, with auto post-increment. Memory address is specified in $\mathbf{a}_i$ & $\mathbf{r}_i$, where 3-bit i (7..0) is specified in 3-bit <src>, which is in `progword`[6..4]. Register <dst> is specified in `progword`[3..0]. `progword`[7] = 1 (auto-post increment). Executes in 2 clock cycles, where 1st cycle gen1 ldb (ldh, ldw) are recognized, and operation completes on 2nd clock cycle. On 2nd clock cycle, $\mathbf{r}_i = \mathbf{r}_i + 1$ (note: there is no carry into $\mathbf{a}_i$).

0010 stb (sth, stw) <src>[dst]: Store value in register to memory, with no auto post-increment. Register <dst> is specified in `progword`[7..4]. Memory address is specified in $\mathbf{a}_i$ & $\mathbf{r}_i$, where 3-bit i (7..0) is specified in 3-bit [dst], which is in `progword`[2..0]. `progword`[3] = 0 (no auto-post increment). Executes in 2 clock cycles, where 1st cycle gen1 stb (sth, stw) are recognized, and operation completes on 2nd clock cycle.

0010 spb (sph, spw) <src>[dst]: Store value in register to memory, with auto post-increment. Register <dst> is specified in `progword`[7..4]. Memory address is specified in $\mathbf{a}_i$ & $\mathbf{r}_i$, where 3-bit i (7..0) is specified in 3-bit [dst], which is in `progword`[2..0]. `progword`[3] = 1 (auto-post increment). Executes in 2 clock cycles, where 1st cycle gen1 stb (sth, stw) are recognized, and operation completes on 2nd clock cycle. On 2nd clock cycle, $\mathbf{r}_i = \mathbf{r}_i + 1$ (note: there is no carry into $\mathbf{a}_i$).

0011 ret: Return from subroutine to **pc** stored on stack, and pops return address off stack. Executes in 2 clock cycles. On 1st clock cycle, **pc**[7..0] = [**sp**+1], and **sp** = **sp** + 1. On 2nd clock cycle **pc**[11..8] = [**sp**+1], and **sp** = **sp** + 1.

0100 str: Store status register on stack. Executes in 1 clock cycle: [**sp**] = **sr**, **sp** = **sp** – 1.

0101 lsr: Load status register from stack. Executes in 1 clock cycle: **sr** = [**sp**+1], **sp** = **sp** + 1.

0110 rie: Read interrupt enable register to <dst>, where <dst> is 3-bit i in `progword`[6..4], and represents ai & ri. `progword`[7] and `progword`[3..0] are unused. Executes in 2 clock cycles. On 1st cycle, gen1 and rie are recognized. On 2nd cycle, operation completes.

0111 sie: Set interrupt enable register using value in <src>, where <src> is 3-bit i in `progword`[6..4], and represents $\mathbf{a}_i$ & $\mathbf{r}_i$. `progword`[7] and `progword`[3..0] are unused. Executes in 2 clock cycles. On 1st cycle, gen1 and sie are recognized. On 2nd cycle, operation completes.

1000 hlt: Halt. Program halts, pending receipt of a valid interrupt. Executes in 1 clock cycle.

1001 rti: Return from interrupt service routine. Restores saved register contents from shadow registers. Executes in 1 clock cycle. For i = 0 to (N_S – 1): $\mathbf{a}_i = \mathbf{as}_i$ and $\mathbf{r}_i = \mathbf{rs}_i$; **sr** = **srs**, **pc** = **pcs**.

1010, 1011, 1100, 1101, 1110 Reserved

1111 sys: System call Im[7..0]. Asserts Im[7..0] onto external system bus (***s-bus***), where Im[7..0] is specified in next `progword`[7..0]. Executes in 2 clock cycles. On 1st cycle, gen1 sys is recognized. Operation completes on 2nd clock cycle. Note: A 4-bit region of the ***s-bus*** can be reserved for acknowledgment of interrupts, i.e., 2^4 = 16 interrupts can be acknowledged through a decoder.

Note: Currently sys 0xFF is reserved to send an "end" signal, i.e., the last command that the core processes. sys 0x10 through 0x1F are expected to be reserved for interrupt acknowledgment.

<opc1>

1101 psh <src>: Push value in register <src> onto stack. Executes in 1 clock cycle. [**sp**] = <src>, **sp** = **sp** – 1.

1110 pop <dst>: Pop value from stack into register <dst>. Executes in 1 clock cycle. <dst> = [**sp**+1], **sp** = **sp** + 1.

1111 yyy (reserved)

The above instructions are summarized below in Table 1.

Table 1 – Summary of HOKSTER Instruction Set Architecture

Instr	Cycle 1		Cycle 2	
	<opc1>	<opc2>		
mvs	0000	Im[11:8]	Im[7:4]	Im[3:0]
mvv	0001	<dst>	Im[7:4]	Im[3:0]
jmp	0010	Im[11:8]	Im[7:4]	Im[3:0]
jsr	0011	Im[11:8]	Im[7:4]	Im[3:0]
bzi	0100	Im[11:8]	Im[7:4]	Im[3:0]
bni	0101	Im[11:8]	Im[7:4]	Im[3:0]
bci	0110	Im[11:8]	Im[7:4]	Im[3:0]
bxi	0111	Im[11:8]	Im[7:4]	Im[3:0]
mvi	1000	<dst>	Im[7:4]	Im[3:0]
alu				
add	1001	0000	<src1>	<src2=dst>
sub	1001	0001	<src1>	<src2=dst>
and	1001	0010	<src1>	<src2=dst>
lor	1001	0011	<src1>	<src2=dst>
sll	1001	0100	<src1>	<src2=dst>
rol	1001	0101	<src1>	<src2=dst>
slr	1001	0110	<src1>	<src2=dst>
ror	1001	0111	<src1>	<src2=dst>
reserved	1001	1000		
reserved	1001	1001		
not	1001	1010	<src>	<dst>
xor	1001	1011	<src1>	<src2=dst>
adc	1001	1100	<src1>	<src2=dst>
sbc	1001	1101	<src1>	<src2=dst>
adi	1001	1110	Im[3:0]	<src=dst>
sbi	1001	1111	Im[3:0]	<src=dst>
aluc				
	1010	See ISE section for details		
reserved	1011			
gen1				
mov	1100	0000	<src>	<dst>
ldb	1100	0001	0<src>	<dst>
lpb	1100	0001	1<src>	<dst>
stb	1100	0010	<dst>	0<src>
spb	1100	0010	<dst>	1<src>
ret	1100	0011		

str	1100	0100		
lsr	1100	0101		
rie	1100	0110	<dst>	
sie	1100	0111	<src>	
hlt	1100	1000		
rti	1100	1001		
reserved	1100	1010		
reserved	1100	1011		
reserved	1100	1100		
reserved	1100	1101		
reserved	1100	1110		
sys	1100	1111	Im[7:4]	Im[3:0]
psh	1101	<src>		
pop	1110	<dst>		
reserved	1111			

In addition to the hardware-implemented instructions, the following pseudo-instructions are supported by HoksterAsm:

nop: No operation. The processor completes "no operation" in 2 clock cycles. This command is implemented as mov r0, r0.

An example of each instruction in the ISA can be found in the "Instruction Set Examples" section.

<u>External interface</u>

The external interface to the HOKSTER core is shown in Fig. 1. Inputs include clock (`clk`), synchronous reset (`rst`), external program in (`extprogin`), external program address (`extpaddr`), external program load (`extprogload`), start, external data in (`extdin`), external data address (`extdaddr`), external data load (`extdataload`), `extdaddrsel`, auxiliary data out (`auxdout`), auxiliary data selector (`auxdatasel`), and the interrupt bus (`ibus`). Outputs include external data out (`extdout`), system bus (`sbus`), auxiliary data in (`auxdin`), and auxiliary data address (`auxaddr`).

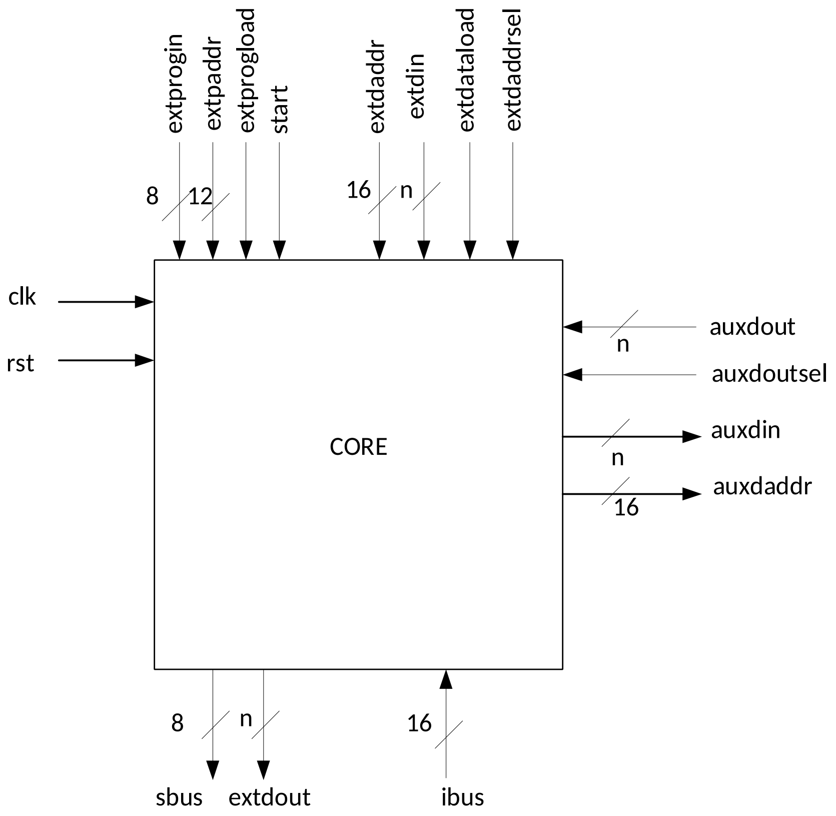

Figure 1 – HOKSTER internal interface, with default single-bit bus width

<u>General processor operations</u>

1. The processor has a synchronous reset (`rst`). Assertion of `rst` does not start processor operations, but ensures that the processor initializes in a defined state. It is recommended to assert `rst` upon power-up or hot restart.

2. A program must be externally loaded by an external module, called *loader*. Program addresses are asserted on `extpaddr`, which individual commands are asserted on `extprogin`, while `extprogload` is asserted. Note: The loader currently performs the role of an operating system as well, however, this architecture can be extended to have a more robust operating system.

3. Data may be preloaded to data memory (DRAM) by asserting `extdataload` and `extdaddrsel`, while asserting addresses on `extdaddr`, and data words on `extdin`. Signal `extdaddrsel` is required to drive the address bus by and external user, such as *loader*.

4. The processor commences operations at address 0x000 upon assertion of start for 1 clock cycle.

5. Data output may be continuously observed by an external user on `extdout`, however, the data address bus can only be driven by the external user by asserting `extdaddrsel`. Assertion of `extdaddrsel` is not recommended during core operation, as this can cause contention on

the data address bus. If a program should communicate data to external users during operation, the recommended procedure is to signal an event using the ***s-bus***, halt, and await acknowledgement using the ***i-bus*** before resuming processing.

6. Operation of memory-mapped peripherals is interleaved with processor operations. Writes to memory on the data bus (***d-bus***) are echoed externally via the `auxdin` signal, and are available to memory-mapped peripherals. Addresses asserted on the address bus (***a-bus***) are echoed externally via the `auxaddr` signal. Peripherals are responsible for recognizing their own addresses. If directed, peripherals should respond with data via the `auxdout` signal, while asserting `auxdoutsel`. It is the responsibility of peripherals to not assert `auxdoutsel` unless directed, otherwise, there could be contention on the ***d-bus***. Refer to section on “interfacing with memory mapped peripherals” for recommended interface protocols for memory-mapped peripherals.

7. Programs can halt indefinitely while waiting for an external stimulus, including those generated by peripherals, or the loader.

8. Programs announce termination by asserting an END signal via the ***s-bus*** (as agreed to by core and loader, e.g., `sbus` = 0xFF). At this point, a program can only be restarted by asserting start. With the exception of **pc**, registers and memory are not cleared nor initialized by asserting start; it is the responsibility of the user to initialize memory and registers as desired.

A detailed diagram of the core is shown in Fig. 8.

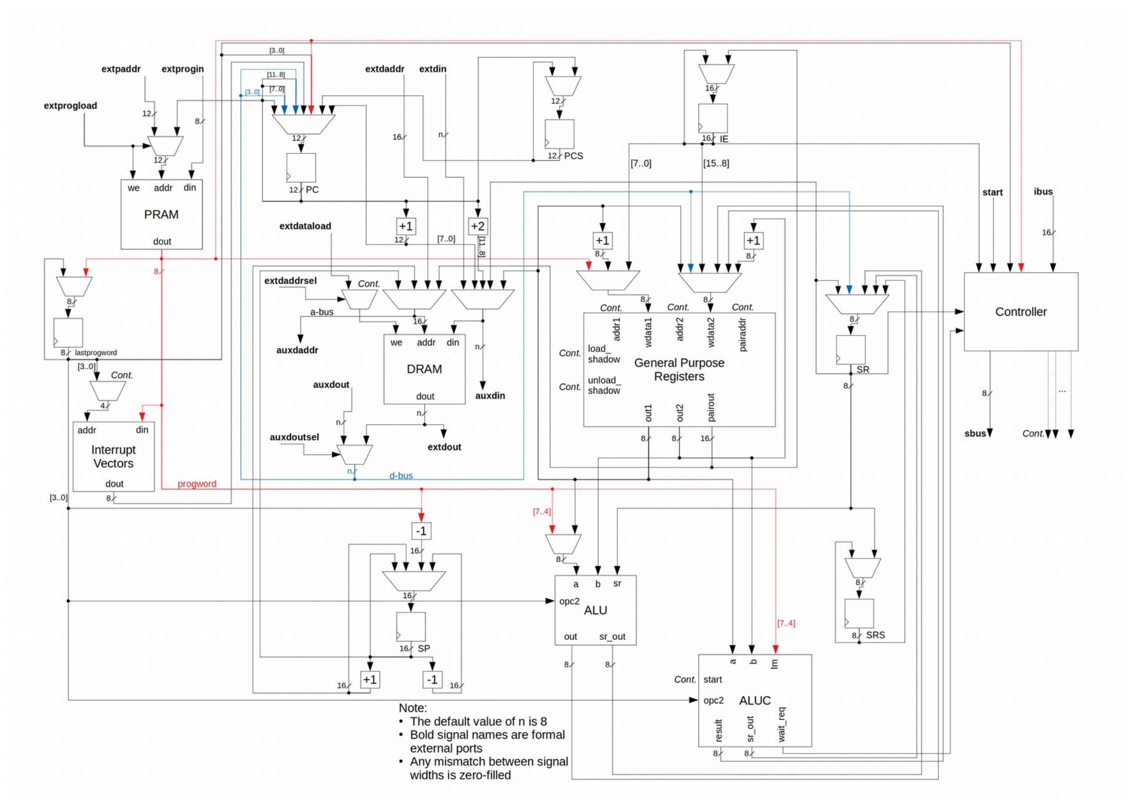

Figure 8. Block Diagram of HOKSTER Core

Instruction Set Extensions

HOKSTER supports hardware instruction set extensions (ISEs). These extensions implement user-defined aluc operations on register values and immediates. Either the contents of two registers or one immediate value and the contents of one register can be passed to an ISE. Unlike regular alu operations, these operations can take a variable number of clock cycles to complete. Specifically, the number of clock cycles for these operations is 2 + *x* where *x* is an integer greater than or equal to 0. They also receive the current value of **sr** and can set all 8 **sr** flags.

ISEs must conform to the interface shown in Fig. 2. The 8-bit input signal `a` is the contents of the register specified in <src1> of the aluc operation. Instead of specifying a register, those four bits can be an immediate value: `Im`. The next signal, `b` is data from another register specified in <src2>. The `sr` signal is the current value of **sr**. The final input to the ISE is the `start` signal, which gets asserted in the second clock cycle of an aluc operation. The purpose of this signal is to notify the ISE to begin operation. In cases where the aluc operation takes more than 2 clock cycles to complete, this signal is needed so that the ISE does not begin operation when not needed.

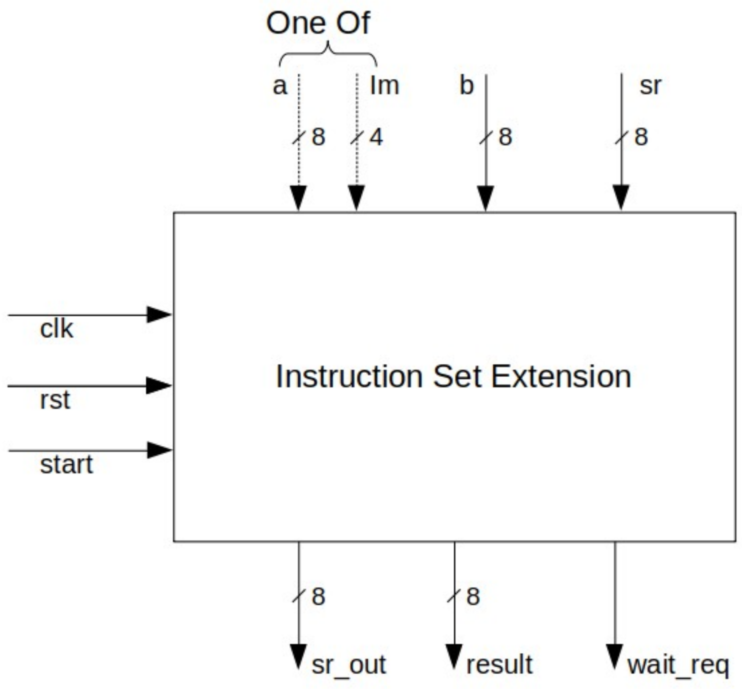

Figure 2 – External Interface for Instruction Set Extensions

The interface requires the output signals `sr_out`, `result`, and `wait_req`. In response to `start` being asserted, if the ISE needs more than two clock cycles to complete its operation, the `wait_req` signal must be asserted for as long as the ISE needs. The processor will halt all other operations while `wait_req` is asserted. This includes not handling interrupts as all operations are atomic. If the ISE can complete operation in two clock cycles, then `wait_req` does not need to be asserted. Either way, once `wait_req` is deasserted, the processor will continue normal operations. The signals `sr_out` and `result` are loaded into **sr** and the register specified in <dst> in the aluc operation, respectively. ALUC operations are not required to update **sr** flags, but if the operation does affect a flag, then all eight flags must be affected (i.e., either set or reset). If no flags are affected, then the ISE must forward the input `sr` signal to the output `sr_out`. All pending interrupt requests are then handled. This handshake is shown in Fig. 3 for an ISE with *x* = 3.

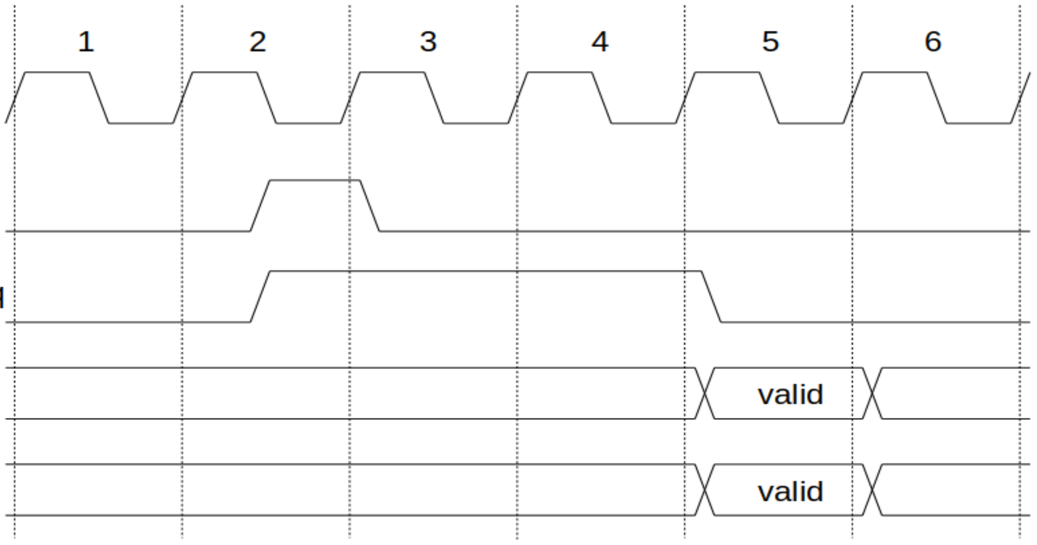

Figure 3 – Operation of an ISE with *x* = 3. In the 2nd clock cycle, `start` is asserted and the ISE immediately asserts `wait_req` in response. The ISE asserts `wait_req` for a total of 3 clock cycles and, when `wait_req` is deasserted, provides `sr_out` and `result` to the processor.

Currently implemented ISE are designed for use with the AES, CHAM, and GIFT block ciphers and are defined as follows. A summary of their usage and details is shown in Tables 2 and 3.

All call formats follow a stack-like sequence (i.e., load in order A,B,C means load out order C,B,A).

AES ISE:

0000 asb AES_SBOX(<src1>) -> <src2=dst>: Applies 8-bit AES SBOX to <src1> and places the result in <src2=dst>. Executes in 2 clock cycles. 1st cycle: Recognize aluc + ASB. 2nd cycle: completion

0001 sib AES_SBOX^{-1}(<src1>) -> <src2=dst>: Applies 8-bit AES inverse SBOX to <src1> and places the result in <src2=dst>. Executes in 2 clock cycles. 1st cycle: Recognize aluc + AIB. 2nd cycle: completion

0010 amc (See call sequence): Performs mix column operation on 32-bit values inputted through multiple calls (5 total). Mix columns calculation will require multiple cycles (exact performance will be determined once implemented).

Call sequence:

1. S_0 = <src1>, S_1 = <src2>
2. S_2 = <src1>, S_3 = <src2>, Calculate MC(S), <src2=dst>=S'_3
3. <src2=dst>=S'_2
4. <src2=dst>=S'_1

5. <src2=dst>=S’$_0$

Example usage:

If the AES state is:

a[0,0]	a[0,1]	a[0,2]	a[0,3]
a[1,0]	a[1,1]	a[1,2]	a[1,3]
a[2,0]	a[2,1]	a[2,2]	a[2,3]
a[3,0]	a[3,1]	a[3,2]	a[3,3]

to perform the first mix columns operation one could load the values a[0,0], a[1,0], a[2,0], and a[3,0] into register a0, a1, a2, and a3 respectively. Then the calls would look as follows:

amc a0, a1
amc a2, a3
amc a2, a2
amc a1, a1
amc a0, a0

The registers a0, a1, a2, and a3 will hold a’[0,0], a’[1,0], a’[2,0], and a’[3,0] respectively.

0011 aic (See call sequence): Performs inverse mix column operation of 32-bit values inputted through multiple calls (5 total). Inverse mix columns calculation will require multiple cycles (exact performance will be determined once implemented).

Call sequence:

1. S$_0$ = <src1>, S$_1$ = <src2>
2. S$_2$ = <src1>, S$_3$ = <src2>, Calculate MC(S), <src2=dst>=S’$_3$
3. <src2=dst>=S’$_2$
4. <src2=dst>=S’$_1$
5. <src2=dst>=S’$_0$

Example usage: See AMC example.

CHAM ISE:

0100 swd (See call sequence): 32-bit barrel circular shifter. Per the requirements of the CHAM cipher, this rotates left by the number of bits specified by the register in the 3rd call. Executes in 6 calls each taking 2 cycles. 1st cycle: Recognize aluc + SWD. 2nd cycle: completion

Call sequence:

1. Input[7:0]=<src1>,Input[15:8]=<src2>
2. Input[23:16]=<src1>,Input[31:24]=<src2>
3. Circular shift amount = <src1>, output=(input <<< <src1>). <src2=dst>=output[31:24]
4. <src2=dst>=output[23:16]
5. <src2=dst>=output[15:8]
6. <src2=dst>=output[7:0]

Example usage:

Take a 32-bit value A that needs to be circularly shifted left by B bits. This could be implemented as follows: a0 = A[7:0], a1 = A[15:8], a2 = A[23:16], a3 = A[31:24], a4 = B.

```
swd a0, a1
swd a2, a3
swd a4, a3
swd a2, a2
swd a1, a1
swd a0, a0
```

At the end of this call sequence we would have concat(a3, a2, a1, a0) == (A <<< B).

GIFT-128 ISE:

0101 gsp (GIFT SBOX PERM - See call sequence): Applies 4-bit GIFT SBOX to each 4-bit chunk of input (one 8-bit register holds 2 4-bit inputs) as well as the 128-bit permutation. There are 8 calls to load (16-bits of input per call) and 15 additional calls to unload (8-bits of output per call). A total of 23 calls each taking 2 cycles will be required. 1st cycle: Recognize aluc + GSP. 2nd cycle: completion

Call sequence:

9. Input[7:0]=SBOX(<src1>),Input[15:8]=SBOX(<src2>)
10. …
11. Input[119:112]=SBOX(<src1>),Input[127:120]=SBOX(<src2>), output = PERM(input[127:120]), <src2=dst>=output[127:120]
12. <src2=dst>=output[119:112]
13. …
14. <src2=dst>=output[7:0]

Example usage:

Due to the large input size, there will have to be memory accesses during the load and unload calls to gsp. These will be left out of the example since they are specific to the assembly implementation. However, assuming you have a 128-bit “state” stored somewhere in memory that needs to have the SBOX and 128-bit permutation applied. The instructions to do so could look like the following:

```
<Load input[63:0] into a7-a0 (lowest byte in a0, highest byte in a7)>
gsp a0, a1
gsp a2, a3
gsp a4, a5
```

gsp a6, a7
<Load input[127:64] into a7-a0 (lowest byte in a0, highest byte in a7)>
gsp a0, a1
gsp a2, a3
gsp a4, a5
gsp a6, a7
gsp a6, a6
gsp a5, a5
gsp a4, a4
gsp a3, a3
gsp a2, a2
gsp a1, a1
gsp a0, a0
<storea7-a0 into output[127:64] (lowest byte from a0, highest byte froma7)>
gsp a7, a7
gsp a6, a6
gsp a5, a5
gsp a4, a4
gsp a3, a3
gsp a2, a2
gsp a1, a1
gsp a0, a0
<storea7-a0 into output[63:0] (lowest byte from a0, highest byte froma7)>

This will result in output containing the value of the 128-bit input after the SBOX and permutation.

0110 gip (GIFT INVERSE SBOX PERM - See call sequence): Applies 4-bit GIFT inverse SBOX to each 4-bit chunk of output (one 8-bit register holds 2 4-bit inputs) after applying the inverse 128-bit permutation. So, there are 8 calls to load (16-bits of input per call) and 15 additional calls to unload (8-bits of output per call). A total of 23 calls each taking 2 cycles will be required. 1st cycle: Recognize aluc + GSP. 2nd cycle: completion

Call sequence:

1. Input[7:0]=<src1>,Input[15:8]=<src2>
2. …
3. Input[119:112]=<src1>,Input[127:120]=<src2>, output = PERM(input[127:0]), <src2=dst>= SBOX^{-1}(output[127:120])
4. <src2=dst>=SBOX^{-1}(output[119:112])
5. …
6. <src2=dst>= SBOX^{-1}(output[7:0])

Example usage: See gsp.

Table 2 – Summary of HOKSTER Instruction Set Extensions

Instr	Cycle 1		Cycle 2	
	<opc1>	<opc2>		
		aluc		
asb	1010	0000	<src1>	<src2=dst>
aib	1010	0001	<src1>	<src2=dst>
amc	1010	0010	<src1>	<src2=dst>
aic	1010	0011	<src1>	<src2=dst>
swd	1010	0100	<src1>	<src2=dst>
gsp	1010	0101	<src1>	<src2=dst>
gip	1010	0110	<src1>	<src2=dst

Table 3 – ISE Details

ISE	# of Calls	Wait Cycles	Total Cycles	LUTs*	FFs*
asb	1	0	2	5	0
aib	1	0	2	5	0
amc	5	3	13	43	52
aic	5	23	33	59	101
swd	6	1	13	181	49
gsp	23	1	47	97	160
gip	23	1	47	99	160

(*) synthesized in an Artix-7 FPGA

Memory-mapped Peripherals

HOKSTER supports memory-mapped peripherals. Memory-mapped peripherals are intended for future expansion. Interface with peripherals must adhere to the load-store architecture of the core. Therefore, a recommended procedure is to write a desired operand to an address mapped to a peripheral, possibly with a unique purpose of signaling a start. This can be accomplished using the stb (sth, stw) command. To retrieve a value from the peripheral, a user can load a result from an address mapped to the peripheral, using the ldb (ldh, ldw) command. If multi-cycle operation of the peripheral is required, the processor can either halt (hlt) and wait for completion, or can continue in normal operations, and process an interrupt upon receipt.

The external interface of peripherals should generally follow the interface shown in Fig. 4, although fewer or additional signals are possible. Refer to HOKSTER system diagrams for definition of the signals `auxdaddr`, `auxdin`, `auxdoutsel`, `auxdout`, `extdaddr`, `extdaddrsel`, `extdout`, and `extwe`. (The `extwe` signal corresponds to `extdataload` in the core's external interface.) The `irq` (interrupt) signal will be tied to an incoming interrupt line i0 … i15, per user preference, and a corresponding interrupt vector will be assigned by the user. The `ack` (interrupt acknowledge) signal will either come directly from the ***s-bus***, or from an interrupt acknowledge decoder sourced by the ***s-bus***.

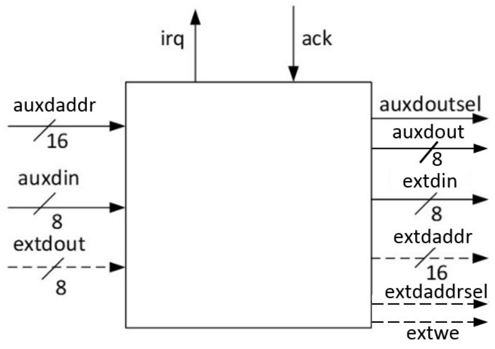

Figure 4 – External Interface for Memory-mapped Peripherals

The protocol for using a memory-mapped peripheral is as follows:

1. The user communicates data and control information to the peripheral using the `auxdaddr` and `auxdin` signals on ***a-bus*** and ***d-bus***, respectively.
2. The peripheral uses the `irq` signal to communicate to the processor that an action is complete.
3. The peripheral can directly communicate an n-bit result to the processor ***d-bus*** using `auxdout`, and simultaneously asserting `auxdoutsel`.
4. The peripheral can optionally communicate with DRAM by additionally using `extdaddr`, `extdin`, `extdout`, and `extwe`. The `extdaddrsel` signal should be asserted when using `extdaddr`.
5. The processor should acknowledge the interrupt by asserting `ack`.

An example of a memory-mapped application is shown in Fig. 5, and described below. The peripheral, a direct memory access (DMA) controller, transfers an arbitrary block of memory from a source in DRAM, to a destination in DRAM. The source and transfer blocks must not overlap, and must contain the same number of bytes. The full range of source and transfer blocks must physically exist in DRAM. A sample use of the DMA would be "transfer 16 bytes of memory from addresses 0x0020 to 0x002F to addresses 0x0030 to 0x003F."

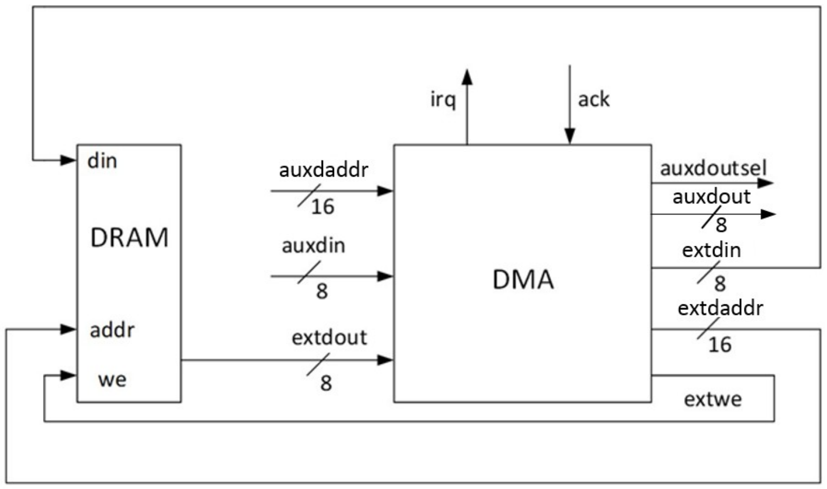

Figure 5 – Direct Memory Access (DMA) Peripheral

A description of the protocol necessary to complete the above operation is described as follows:

1. Assume a DRAM contains at least 512 bytes of memory.

2. Let the following memory-mapped addresses be defined as follows:
0x0100 Control address
0x0101 least-significant 8 bits of source start address
0x0102 most-significant 8 bits of source start address
0x0103 least-significant 8 bits of destination start address
0x0104 most-significant 8 bits of destination start address
0x0105 number of bytes to transfer = n (encoded as 0 to 255). The number of bytes to be transferred will be (n+1) << G, meaning that the minimum number of bytes to be transferred will be 2^G (e.g., n = 0; G = 2; (0+1) << G = 4), and the maximum to be transferred (assuming G = 2) would be 1024 (e.g., n = 255; G = 2; (255 + 1) << G = 1024). G is defined by the user at synthesis time in a generic between 0 and 4.

3. The programmer uses the stb (sth/stw) command to write appropriate values into addresses 0x0101 – 0x0105.

4. When ready to start the DMA transfer, the programmer writes 0xFF to 0x0100 using the stb (sth/stw) command.

5. The programmer uses the hlt instruction to halt the processor until it receives an interrupt. Optionally, the program can continue normal execution, however, user operations on the ***a-bus*** and ***d-bus*** are not permitted (which severely limits the usefulness of the processor).

6. The DMA takes over operation of DRAM, and transfers all bytes from the source region to all bytes in the destination region.

7. When the DMA is complete, it signals completion of the operation by asserting `irq`. Optionally, the DMA can assert a status message on `auxdout`.

8. The programmer acknowledges the interrupt by asserting `ack`, and the DMA resets to idle.

HOKSTER Development, Verification, and Benchmarking Environment

There are various paths to develop, verify, and benchmark HOKSTER applications. The programmer starts by developing an application using the HOKSTER assembly language and accompanying preprocessor directives. The file is stored in a text (<file>.txt) file. The program is then assembled using HoksterAsm. The output file appends "_prog" to the initial filename <file>; the result is <file>_prog.hex. Data used to initialize data memory is output by the assembler into <file>_data.hex.

In future expansion, compilation directly from a subset of C into HOKSTER object code will be possible using HoksterCompiler (hcc). Linking of multiple object codes and libraries will also be possible using HoksterLink.

The assembled code can be verified and benchmarked using a tailored simulator, the hard loader, or soft loader. The use of the simulator is shown in Fig. 6, and described subsequently in a separate section. To use the hard loader, the user instantiates the core, two fileloader.vhd instances (one for <file>_prog.hex and the other for <file>_data.hex), and any and peripherals, inside loader.vhd, sets applicable parameters, and runs the loader_tb in a simulator like Vivado Simulator. Benchmarking statistics, such as latency, throughput, and area, can be determined by implementing core and desired memories and peripherals. The hard loader verification and benchmarking process is shown in Fig. 6.

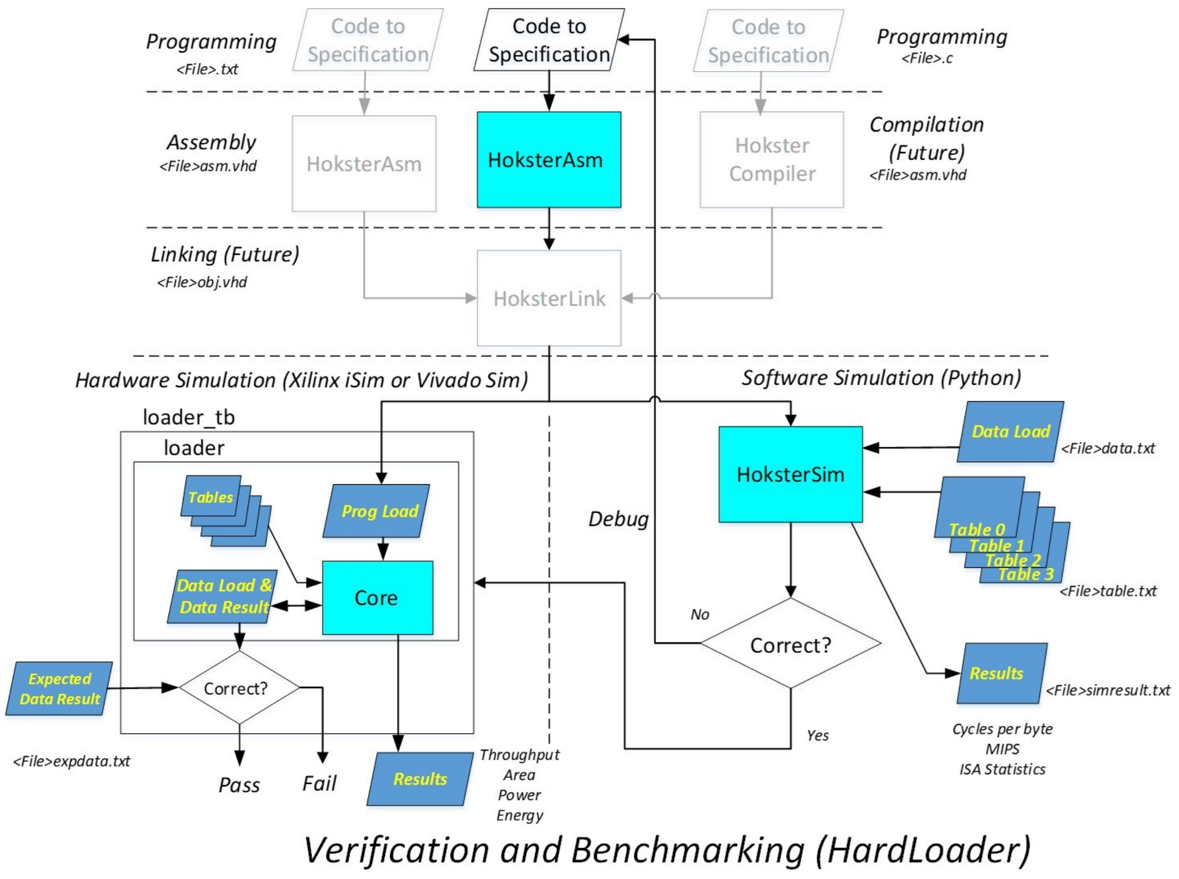

Figure 6. Hard Loader Verification and Benchmarking Process, including Simulator

The soft loader contains a runnable instance of the core, together with memories and accelerators, in a Xilinx Block Design together with a microblaze soft core. An accompanying C program called loader.c is used to load and launch the HOKSTER application. Using Xilinx SDK, the user updates loader.c by inserting the program, data, and tables as data arrays. The Core API, written in C, is then used to load and run the program, and collect results per user desires. The core output can be observed through examination of an output text file, or on a console supplied by a UART connection. The soft loader will also permit benchmarking including latency, throughput and area, keeping in mind that area statistics will include a microblaze and required AXI peripherals. The soft loader verification and benchmarking process is shown in Fig. 7.

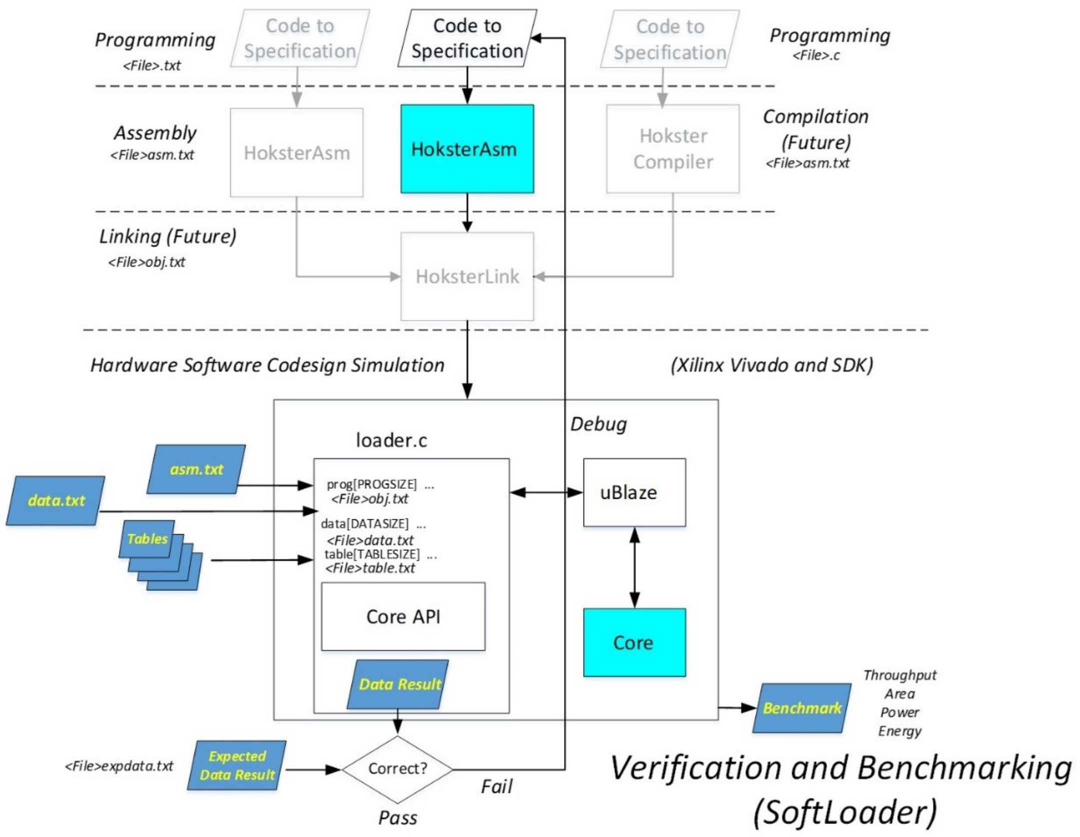

Figure 7. Soft Loader Verification and Benchmarking Process

Code preparation and use of HoksterAssembler

HOKSTER applications can be efficiently created using the HoksterAsm (HoksterAsm.py) assembler. The assembler catches most syntax errors and illegal references, though programmer caution and diligence is advised. Steps for creating a program are as follows:

1. The source file should be a .txt file located in the same directory as the HoksterAsm.py.

2. This is a multi-pass assembler which resolves two types of references: one-byte (8-bit) data references, and one-and-a-half byte (12-bit) program references.

3. All references and constants should be entered in hexadecimal (eg. 0x0A).

4. 1 byte data references should be placed in the beginning of the code using the .equ directive. Example:
.equ tmpvar 0x00

This will permit tmpvar to be used as a constant reference in the program, for example:
mvi tmpvar, r0 (moves tmpvar = 0x00) to register r0

5. 1 ½ byte program references are placed as applicable in the code as “labels” using the .lbl directive. Example:

```
stb r0, r2
mvi tmpvar, r1
.lbl nxtloop
mvi op1ptr, r0
…
bzi nxtloop
```

The assembler will resolve .lbl nxtloop into an address such as 0x0E3. The object code will insert the 0x0E3 reference into the bzi instruction.

6. The user can supply data used to initialize memory using the .dat directive. A .dat directive must always contain a 16-bit address, at least one byte of data, and an arbitrary number of subsequent bytes of data. .dat directives should be placed at the beginning of the code before programming statements. Example:

```
.dat 0x0000 0x01
.dat 0x0010 0x12, 0x34, 0x56, 0x78
…
```

In the above code, the value 0x01 is intended for memory address 0x0000, and values 0x12, 0x34, 0x56, and 0x78 are intended for memory addresses 0x0010, 0x0011, 0x0012, and 0x0013, respectively. Data assignment used for initialization of memory will be assembled into a separate text file called <file>_data.hex, where <file>.txt was the name of the source file.

7. The user should specify the end address (i.e., stop point) using the .end directive. The last runnable command in the code should be “sys 0xFF”, where 0xFF is currently reserved as the code to end program execution and return control to the loader or operating system.
Example:

```
sys 0xFF
.end
```

8. Comments should start with a # character. Anything after the # character on the same line will be considered a comment.

9. All commands and directives should be lower case (in this version).

10. The name of the source file should be specified as an argument when running HoksterAsm.py, for example: “python HoksterAsm.py testcode.txt”. The object code is a .hex file with name <source>_prog.hex, for example: “testcode_prog.hex”. The data file is a .hex file with name <source>_data.hex, e.g., “testcode_data.hex”.

11. To facilitate writing of Interrupt Service Routines (ISRs), the “.align” directive aligns the next command to a 16-byte boundary. The memory between the last written command and the command following the .align directive is filled with 0x00s.
Example:

```
…
(addr=0x032)    sys 0xFF
```

```
.end
.align
.lbl isr2
(addr=0x040)     sys 0x12
…
```

Additional features and interfaces will be described in subsequent updates.

<u>Hokster Program simulation</u>

Object code assembled by HoksterAsm can be simulated in HoksterSim. HoksterSim.py is a simulator which runs in a Python environment. The instructions for using the HoksterSim.py utility are outlined below:

1. Start with an object file compiled by HoksterAsm

2. Run HoksterSim using python HoksterSim.py [object filename].

3. Use the following commands:

"quit" - terminates the event and returns to command line interface (CLI)

"rmem <start> <end>" or "r <start> <end>" – displays data memory locations from [start] to [end] in groups of 16 bytes. <start> and <end> should always be four-digit hex numbers:
Ex: rmem 0000 0100

"lmem [datafile] <start>" – loads the data file to consecutive data memory locations starting at <start>. <start> should always be a four-digit hex number.
Ex: lmem aesdata.txt 0000

"wmem <addr> <value>" – writes the two-digit hex value at <value> to the four-digit hex data memory address <addr>.
Ex: wmem 00ce a0

"jump <newPC>" – adjusts the program counter to the three-digit hex value in <newPC>.
Ex: jump 03a

"step" or "s" – Executes the instruction at the current PC.

"run" – Starts the program from the origin specified in the [object file]. The program will run until the end location specified in the [object file], a user-defined breakpoint, or an error occurs.

"cont" – Continues program execution from current PC location. The program will run until the end location specified in the [object file], a user-defined breakpoint, or an error occurs.

"setbreak <breakpoint>" – Sets a user-defined breakpoint at the three-digit hex address specified in <breakpoint>
Ex: setbreak 1ae

"clrbreak" – Clears the user-defined breakpoint

"wreg <reg> <value>" – writes a two-digit hex <value> into a register. Register should be r0, r1, r2, or r3.
Ex: wreg r1 f6

"ltab [table filename] <table>" – Loads the contents of table from filename [table file] into a user-defined <table>. <table> should be t0, t1, t2, or t3.
Ex: ltab aestable.txt t0

"rtab <table>" – Displays the contents of a user-defined table, in groups of 16 bytes. <table> should be t0, t1, t2, or t3.
Ex: rtab t2

"reset" – resets the program counter, zeroes all registers, clears all flags, clears breakpoints, and clears statistics (with exception of program sequence list). Note: for a clean program sequence list the user should restart the application prior to executing a program.

"status" or "st" – displays current PC, instruction code and mnemonic, register contents, breakpoint status, and condition flag status

"statistics" or "stats" – displays total cycles, total instructions, instructions per cycle (IPC), and cumulative total usage of each instruction in the ISA.

"dumpstats" – writes cumulative total usage of each instruction to "HoksterSimStats.txt" and writes the program sequence list to "HoksterSimProgSeq.txt" in comma separated value (csv) for importation to Microsoft Excel or MATLAB.

"logon" – Turns on recording of all HoksterSim instructions to "HoksterSimLog.txt"

"logoff" – Turns off recording of HoksterSim instructions to "HoksterSimLog.txt"

"traceon" – Causes a status to be displayed on every clock cycle during program execution using the "run" or "cont" commands. If "logon" is selected, additionally dumps this information to HoksterSimLog.txt

"traceoff" – Turns off the display of each instruction to the screen and to HoksterSimLog.txt.

Additional available features and interfaces will be described in subsequent updates.

<u>Instruction Set Examples</u>

Table 4 – Examples of HOKSTER Instructions

Instruction	**Syntax**	**Example**	**Description**
mvs	mvs Im[15:0]	mvs 0x0040	Loads 0x003F (0x0040 – 1) to the stack pointer register

mvv	mvv Im[11:0], <dst>	mvv 0xFF0, iv4	Moves 0xFF0 to interrupt vector register iv4
jmp	jmp Im[11:0]	jmp 0x044	PC immediately changes to 0x044 (labels also accepted)
jsr	jsr Im[11:0]	jsr 0x080	Jumps to the subroutine located at address 0x080 (labels also accepted)
bzi	bzi Im[11:0]	bzi ForLoop	Branches to the label ForLoop if the Z flag is set
bni	bni Im[11:0]	bni Whileloop	Branches to the label WhileLoop if the N flag is set
bci	bci Im[11:0]	bci 0x022	Branches to the address 0x022 if the C flag is set
bxi	bxi Im[11:0]	bxi ZERO	Branches to the address at label ZERO if the X flag is set
mvi	mvi Im[7:0], <dst>	mvi 50, r1	Move 50 to register r1
alu			
add*	add <src1>, <src2=dst>	add a1, r2	Adds a1 and r2, stores the result in r2
sub*	sub <src1>, <src2=dst>	sub r4, r3	Computes r4 - r3 and stores it in r3
and	and <src1>, <src2=dst>	and r0, a0	Computes r0 AND a0, stores the result in a0
lor	lor <src1>, <src2=dst>	lor a0, a1	Computes the logical OR of a0 and a1, stores the result in a1
sll	sll <src1>, <src2=dst>	sll r0, a0	Shifts a0 to the left by the number of bits indicated in r0
rol	rol <src1>, <src2=dst>	rol r0, a0	Rotates a0 to the left by the number of bits indicated in r0, stores the result in a0
srl	srl <src1>, <src2=dst>	srl r2, a1	Shifts a1 to the right by the number of bits indicated in r2
ror	ror <src1>, <src2=dst>	ror r0, a0	Rotates a0 to the right by the number of bits indicated in r0, stores the result in a0
not	not <src>, <dst>	not r1, r2	Inverts all the bits in r1 and stores the result in r2
xor	xor <src1>, <src2=dst>	xor r0, a0	Computes r0 XOR a0 and stores the result in a0
adc*	adc <src1>, <src2=dst>	adc r1, r2	Computes r1 + r2 + **sr**[C] and stores the result to r2
sbc*	sbc <src1>, <src2=dst>	sbc r3, r2	Computes r3 - (r2 + **sr**[C]) and stores the

			result in r2
adi*	adi Im[4:0], <src=dst>	adi 5, a0	Computes a0 + 5 and stores the result in a0
sbi*	sbi Im[4:0], <src=dst>	sbi 1, a1	Computes a1 - 1 and stores the result in a1
		aluc	
asb	asb <src1>, <src2=dst>	See section on ISEs for examples	
aib	aib <src1>, <src2=dst>		
amc	amc <src1>, <src2=dst>		
aic	aic <src1>, <src2=dst>		
swd	swd <src1>, <src2=dst>		
gsp	gsp <src1>, <src2=dst>		
gip	gip <src1>, <src2=dst>		
		gen1	
mov	mov <src>, <dst>	mov a4, r0	Moves the value in a4 to r0
ldb	ldb Im[2:0], <dst>	ldb 3, a1	Loads the value at address a3 & r3 into register a1
lpb	lpd Im[2:0], <dst>	lpb 0, r1	Loads the value at address a0 & r0 into register r1 and increments r0
stb	stb <dst>, Im[2:0]	stb r3, 1	Stores the value in register r3 to memory address a1 & r1
spb	spb <dst>, Im[2:0]	spb a0, 7	Stores the value in register a0 to memory address a7 & r7 and increments r7
ret	ret	ret	Returns from subroutine to PC value stored on the top of stack
str	str	str	Stores the status register on the stack
lsr	lsr	lsr	Loads the status register from the stack
rie	rie <dst>	rie 1	Reads the interrupt enable register and stores it in registers a1 & r1
sie	sie <src>	sie 4	Sets the interrupt enable register to the value stored in registers a4 & r4
hlt	hlt	hlt	Program halts until interrupt is received
rti	rti	rti	Returns from the interrupt service routine
sys	sys Im[7:0]	sys 0xFF	Asserts the value 0xFF on the system bus
psh	psh <src>	psh a0	Pushes a0 onto the top of the stack

pop	pop <dst>	pop r7	Pops top value off the stack and stores it in r7
		pseudo	
nop	nop	nop	Performs no operation for 2 cycles

(*) These instructions update the status register