SoftCore

Overview of Soft Core Microprocessor
William Diehl
Version 1.3
1/19/2017

Introduction.

This softcore microprocessor (SoftCore) is designed using register-transfer level (RTL) design and coded in VHDL. It is an 8-bit microprocessor designed for embedded applications under extreme hardware or power budgets.

The SoftCore processor contains three types of memory: Program (RAM), Data (RAM), and Table (ROM) (Future upgrades could easily permit Table RAM). The program and data memories are separate instantiations. The program memory is byte addressable and contains up to 2^{12} (4096) bytes, where 2^8 (256) bytes is the default instantiation. The program memory can only be written by an external interface; the microprocessor cannot alter the program. The data memory is byte addressable and contains up to 2^{16} (65536) bytes, where 2^8 bytes is the default instantiation. The data memory can be written by either the microprocessor or an external interface, and can always be read on both internal and external ports. The microprocessor contains up to four tables of 256 bytes each for a total of 1024 bytes, and are optionally instantiated depending on the application. The Table ROMs can be read by the processor but cannot be written by internal or external interfaces. A top-level interface for SoftCore is shown in Figure 1.

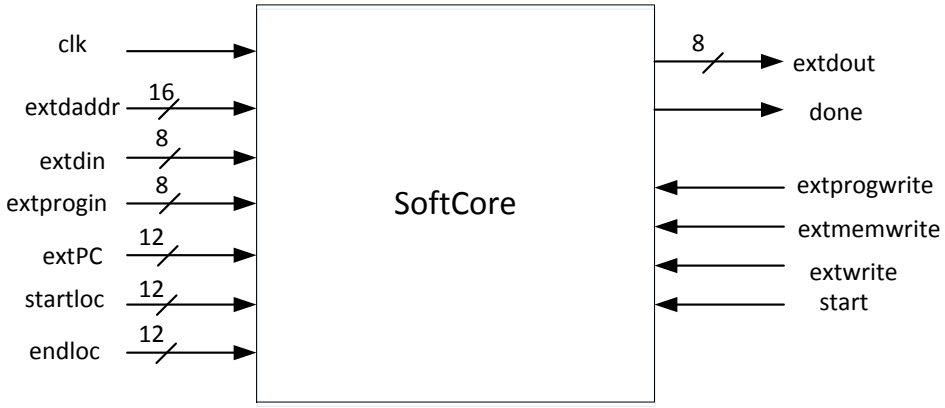


Figure 1 – External Interface for SoftCore Microprocessor

Port	Bus width	Explanation
extdaddr	16	External memory address to write and read data RAM
extdin	8	External input to write data RAM
extdout	8	External output to read data RAM
extprogin	8	External input to write program RAM
extPC	12	External program address to write program RAM
extwrite	1	Asserted when writing to program or data RAM
extprogwrite	1	Asserted when writing to program RAM
extmemwrite	1	Asserted when writing to data RAM
startloc	12	Program execution begins from this address
endloc	12	Program execution terminates at this address
start	1	Asserted for one clock cycle to start execution
done	1	Asserted by SoftCore when when endloc is reached

Table 1 – Explanation of SoftCore External Interface Ports

Program RAM.

The program RAM can be written by an external interface, such as the loader, or is addressed by the PC (program counter) register. The PC is a 12-bit register.

Data RAM.

The data RAM can be written by an external interface, such as a loader, or is addressed by the microprocessor through a 16-bit data address bus. The instruction set allows the user options for either 8-bit (short) or 16-bit (long) memory addresses. All data loads and stores are on 8-bit bytes, however. The instruction set also allows for storing and retrieving program addresses and status register to support subroutine calls and returns.

Instruction Set Architecture (ISA)

This 8-bit CPU is a RISC architecture that supports 31 standard commands (plus two application-specific commands in this version). All commands execute in one or two clock cycles. This is a "load and store" architecture." Only values in registers are permitted to be stored in memory, and loads from memory must be placed in registers. The ISA is introduced below and is described in detail in Appendix A.

Registers.

There are four (4) general purpose 8-bit registers, r0, r1, r2, and r3. Although registers are general purpose, the combination of registers r1:r0 and r3:r2 are used for 16-bit (long) stores and loads.

Additional specialized registers PC, SP, and SR cannot be directly written or read by the programmer. The 12-bit PC contains the address in Program RAM of the instruction currently being executed. The Stack Pointer (SP) is a 16-bit address which contains the location in data RAM into which the next stack push

should occur. The stack is automatically initialized upon program start to point to the uppermost address in data RAM. For example, if the user specifies a data RAM size of 2^8 or 256 bytes, the SP will be initialized to point to address 0xFF (255). The stack grows downward in RAM. Each JSR command pushes two bytes onto the stack, and each RET command pops two bytes from the stack. STR pushes one byte to the stack, and LSR pops one byte from the stack.

Caution: It is the programmer's responsibility to ensure stack and data separation in the data RAM. If the stack is corrupted by a memory write, the program will not likely recover.

The status register (SR) is an 8-bit register with most significant bits SR[7:3] not used, SR[2] is the Carry flag, SR[1] is the negative flag, and SR[0] is the zero flag.

The Carry flag (C) is set only on ADD and INC operations. The negative flag (N) is set whenever the most significant bit is set after an ALU operation. The zero flag (Z) is set whenever the result in the destination register (dst) is equal to 0. In general N and Z flags are set on all ALU operations – even in cases where an interpretation of a negative or a zero result is undefined (such as logical operations). The programmer is advised to exercise care with the status register.

It is straightforward to implement one additional status on the SR. An example would be "parity" (P).

The SR cannot be directly read or written, but can be stored on the stack and retrieved using the STR and LSR commands, respectively. Likewise, various conditional branch instructions act on SR contents.

Arithmetic instructions:

There are 11 basic arithmetic instructions and two application-specific instructions. The instruction set is easily extendable to eight additional two-cycle ALU instructions; the AES-specific instructions in this version are examples of extensions.

The INC, DEC, and NOT instructions execute in one clock cycle and use only one operand (i.e., the source is the destination). The ADD, SUB, AND, LOR (i.e., "logical OR"), ROR, SLR, ROL, and SLL instructions have source and destination operands, and execute in two clock cycles. The XOR instruction executes in one clock cycle.

Table transformations.

SoftCore is designed to conduct cryptographic operations, and can execute transformations (such as an 8-bit S-Box) in one clock cycle by referencing one of the defined tables t0, t1, t2, or t3.

Unconditional and Conditional Branches

SoftCore can complete unconditional 12-bit (i.e., full program) jumps using the JMP command from any address to any address. These jumps take two clock cycles.

The SoftCore supports two types of conditional branches. The first type is the conditional immediate branch. If the C, N, or Z flag is set (as applicable), the program transfers control to the address consisting of the two most significant bits of the PC PC[11:10] and the 8-bit immediate address. The conditional immediate jump can only occur within a 1024-byte program RAM block. Attempts to jump outside of this

range will generate assembly-time errors. Note that this can easily be overcome by a conditional branch to a JMP and thence to anywhere in the Program.

SoftCore also supports conditional (register) indirect branches. These jumps execute in one clock cycle. If the C, N, or Z flag is set (as applicable), the program transfers control to the address consisting of the four most significant bits of the PC PC[11:8] and the 8-bits of the destination register. The conditional indirect branch can only occur within a 256-byte block of the program. This restriction cannot be enforced at assembler-time, so the programmer must exercise caution to ensure that jumps occur within the appropriate block, or erratic or unrecoverable behavior could result.

SoftCore also supports subroutines. A user can branch to a subroutine using the JSR command, which allows a branch to anywhere in the program. The RET command is used to return from a subroutine. Branches to subroutines take two clock cycles. Note that the status register SR is not automatically saved during a JSR call; it is up to the user to save the SR using the STR command and recover using the LSR command if desired.

Soft Core Development, Verification, and Benchmarking Environment

Soft core applications can be developed, assembled, debugged, simulated, verified and benchmarked using a suite of applications. The suite includes the assembler SoftAsm.py, the simulator SoftSim.py, and the loader loader.vhd. Figure 2 shows the integrated development, verification, and benchmarking flow.

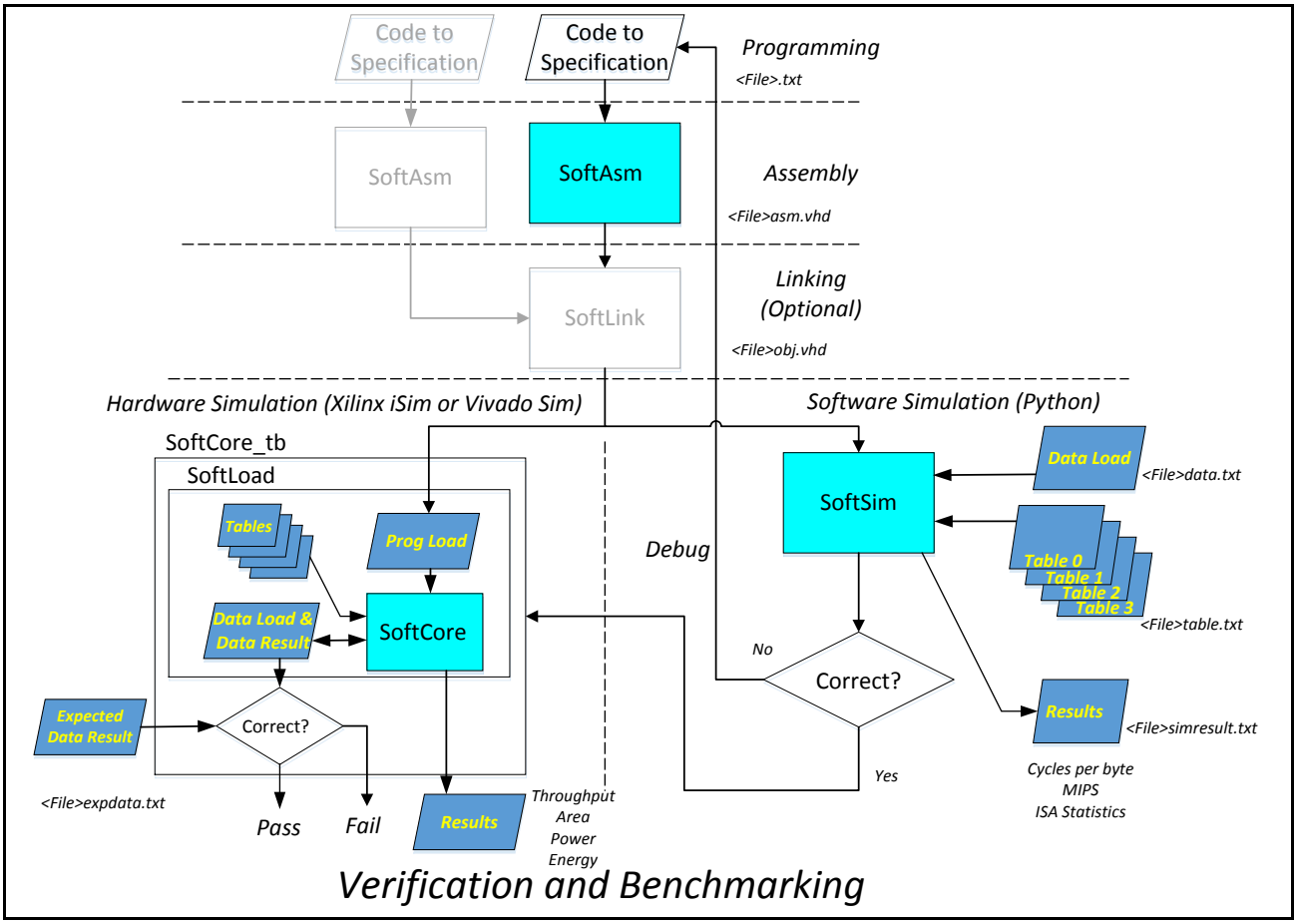


Figure 2 – Soft Core Development, Verification, and Benchmarking Flow

Code Preparation

SoftCore applications can be efficiently created using the SoftAsm (SoftAsm.py) assembler. The assembler catches most syntax errors and illegal references. It will occasionally throw an exception and quit due to a programmer error, however, the line number in which the error occurred will be the last line compiled and is easily rectified. However, it currently cannot detect the case where a user enters a non-defined command, so use caution. Steps for creating a program are as follows:

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

2. This is a two-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:

```
sts 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 may optionally specify the start address of the program using the .start directive, for example: ".start 0x01A". This enables separate assembly of multiple code segments and eventual combination. However, there is no "linker" application at this time (e.g., SoftLink), i.e., there is no application to resolve external references. Additionally, a start location (other than 0x000) is supported by the assembler and the SoftCore, but is not currently supported by the loader.

7. The user should specify the end address (i.e., stop point) using the .end directive. It is recommended to follow the .end directive with a nop command to ensure intended operation. For example:

```
.end
nop
```

6. Comments should start with a # character. Comments should be on their own line

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

8. The name of the source file should be specified as an argument when running SoftAsm.py, for example: "python SoftAsm.py testcode.txt". The object code is also a .txt file with name [source]obj.txt, for example: "testcodeobj.txt".

<u>SoftCore Program simulation</u>

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

1. Start with an object file compiled by SoftAsm

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

3. Use the following commands:

“quit” - terminates the event and returns to 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 “SoftSimStats.txt” and writes the program sequence list to “SoftSimProgSeq.txt” in comma separated value (csv) for importation to Microsoft Excel or MATLAB.

“logon” – Turns on recording of all SoftSim instructions to “SoftSimLog.txt”

“logoff” – Turns off recording of SoftSim instructions to “SoftSimLog.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 SoftSimLog.txt

“traceoff” – Turns off the display of each instruction to the screen and to SoftSimLog.txt.

SoftCore Program Execution

The SoftCore has a versatile interface that allows for loading, execution, and extraction of results by a higher-level entity. However, the SoftCore is currently supported by a higher-level wrapper called “loader.” The SoftAsm.py is currently configured to generate an object file in VHDL format, designed to be easily inserted into the loader utility called “progload.” Once the code is inserted into progload, it is important to note two values: endprogloadloc, and endloc. endprogloadloc is the value of PC annotated in the object code at the last location of the code; this tells the loader how many bytes to load. endloc is chosen by the programmer as the stop point for their code; the execution will halt and the done signal will assert when this location is reached. It is advisable to enter a NOP command at this location. The start location “startloc,” end location “endloc,” and highest program address “endprogloadloc” are specified in the header in the assembler-generated object file.

The SoftAsm assembler currently does not support generation of memory files to load into data RAM. There is a loader entity called “dataload,” where the user enters the desired starting RAM configuration into VHDL table format.

The loader works as follows

1. Loads program from progload into program RAM, beginning at address specified at startloc (default = 0x000) and terminating at address endprogloadloc.

2. Loads data from dataload into data RAM, beginning at address 0x0000 and terminating at address enddataloc.

3. Asserts “start” signal and executes program until “done” is asserted.

4. Dumps memory from data RAM, beginning at address 0x0000 and terminating at address enddataloc.

The default sizes for both program RAM and data RAM are G_PMEM_SIZE:=8 and G_DMEM_SIZE:=8, respectively. It is the user's responsibility to set these generics to the desired RAM sizes.

Status of testing.

The architecture has been successfully tested on all commands and addressing modes, although it has not been verified for every possible sequence of instructions.

Verification:

In order to verify the assembler and architecture, and to demonstrate the utility of this soft core processor, an AES application has been successfully executed in SoftCore. This AES consists of AES encryption only, 10-cycle with 128 bit key, and includes round key generation. The Softcore processor required 14375 clock cycles for a complete AES encryption, including 5458 for round key generation and 8917 for AES encryption. The round keys remain in RAM after execution, so additional encryptions would not require additional round key generation. This program occupied 422 bytes of program RAM, approximately 200 bytes of data RAM (174 bytes for round keys, 16 bytes for plaintext/ciphertext, and some additional space for variables), and 256 bytes of table ROM (i.e., S-Box).

The execution is verified in Xilinx iSim.

Implementation.

SoftCore has been implemented in the Virtex-6 FPGA using Xilinx 14.7 ISE, and on the Kintex-7 using Vivado 2014.4 Design Suite. The implementation statistics are as follows:

Platform	Virtex-6 (Xilinx 14.7 ISE)		Kintex-7 (Vivado 2014.4)	
Application	SoftCore	SoftCore (AES)	SoftCore	SoftCore (AES)
IO Pins	71	71	71	71
LUT	475	391	453	382
Slices	169	134	140	127
Clock Period (ns)	7.143	5.647	5.5	4.5
Frequency (MHz)	140	177.1	181.8	222.2

Table 2 – Implementation of SoftCore applications on Xilinx FPGAs

These implementation statistics show a SoftCore application that was configured as necessary to run the AES Encryption (including key scheduling), including 512 bytes program RAM, 256 bytes data RAM, 256 bytes table ROM. The SoftCore (AES) retains only features required to run this AES application, indicating that AES uses about 82% of total SoftCore features. The Vivado frequencies are estimated using iterative reductions of target frequency and the PerformanceExplore_PostRoutePhysOpt Implementation strategy. Instantiation of memories in both the Virtex-6 and Kintex-7 cases were exclusively through distributed RAM (DRAM). Each DRAM block of 256 bytes requires approximately 32 LUTs

Appendix A
Instruction Set Architecture (ISA)

LDS [src], dst Load (Short): Loads the byte value located at memory address with lower 8 bits contained in register src and upper 8 bits equal to x"00" to register dst.

OPCODE
0000 src=[3:2],dst=[1:0] (Cycle 1)

Example: lds r0, r1 (Load the byte at address 00[r0] to register r1)

LDL [src], dst Load (Long): Loads the byte value located at memory address with lower 8 bits contained in register src1 and upper 8 bits contained in register src2 to register dst. Register src combinations as follows:

src	src(binary)	address (src2 || src1)
r0	00	r1||r0
r2	10	r3||r2
r1	01	undefined
r3	11	undefined

The [src] operand must be specified as r0 or r2. Attempts to specify [src] as r1 or r3 will generate an assembler error.

OPCODE
0001 src=[3:2],dst=[1:0] (Cycle 1)

Example: ldl r0, r2 (Load the byte at address [r1][r0] to register r2)

STS src, [dst] Store (Short): Stores the byte value located in register src to memory address with lower 8 bits contained in register dst and upper 8 bits equal to x"00".

OPCODE
0010 src=[3:2],dst=[1:0] (Cycle 1)

Example: sts r3, r2 (Store the byte in register r3 to address 00[r2])

STL src, [dst] Store (Long): Stores the byte value located in register src to memory address with lower 8 bits contained in register dst1 and upper 8 bits contained in register dst2. Register dst combinations are as follows:

dst	dst(binary)	address (dst2 || dst1)
r0	00	r1||r0
r2	10	r3||r2
r1	01	undefined

r3 11 undefined

The [dst] operand must be specified as r0 or r2. Attempts to specify [dst] as r1 or r3 will generate an assembler error.

OPCODE
0011 src=[3:2],dst=[1:0] (Cycle 1)

Example: stl r0, r2 (Store the byte in register r0 to the address [r3][r2])

MOV src, dst Move: transfers byte in register src to register dst.

OPCODE
0100 src=[3:2],dst=[1:0] (Cycle 1)

Example: mov r0, r1 (Transfer byte in register r0 to register r1)

MVI Im, dst Move (immediate): Transfers the immediate byte Im to register dst.

OPCODE
0101 - - [3:2],dst=[1:0] (Cycle 1)
Im = [7:0] (Cycle 2)

Example: mvi 0x0A, r2 (Load the value 0x0A into register r2)

INC dst Increment: Register dst:= dst + 1. Sets C, N, Z flags.

OPCODE
0110 00 dst=[1:0] (Cycle 1)

Example: inc r0 (Increments register r0)

DEC dst Decrement: Register dst:= dst - 1. Sets N, Z flags.

OPCODE
0110 01 dst=[1:0] (Cycle 1)

Example: dec r3 (Decrements register r3)

NOT dst Negation: Register dst := ~dst.

OPCODE
0110 10 dst=[1:0] (Cycle 1)

Example: not r2 (Bitwise negation of register r2)

ADD src, dst Addition: Register dst:= src + dst. Sets C, N, Z flags.

OPCODE
0110 11 – (Cycle 1)
0000 src=[3:2], dst=[1:0] (Cycle 2)
Example: add r0, r2 (Adds contents of register r0 to contents of register r2 and places result in r2)

SUB src, dst Subtraction: Register dst:= src - dst. Sets N, Z flags.

OPCODE
0110 11 – (Cycle 1)
0001 src=[3:2], dst=[1:0] (Cycle 2)

Example: sub r3, r2 (Subtracts contents of register r2 from register r3, places results in r2)

AND src, dst Logical AND: Register dst:= src & dst.

OPCODE
0110 11 – (Cycle 1)
0010 src=[3:2], dst=[1:0] (Cycle 2)

Example: and r0, r1 (Logically ANDs registers r0 and r1, places results in r1)

LOR src, dst Logical OR: Register dst:= src | dst.

OPCODE
0110 11 – (Cycle 1)
0011 src=[3:2], dst=[1:0] (Cycle 2)

Example: lor r2, r0 (Logically ORs registers r2 and r0, places results in r0)

ROR src, dst Rotate Right: Register dst:= dst >>> src[2:0]. The byte in register dst is rotated right by src[2:0] bits. Sets C, N, Z flags.

OPCODE
0110 11 – (Cycle 1)
0100 src=[3:2], dst=[1:0] (Cycle 2)

Example: ror r2, r1 (Rotates the contents of register r1 right by the number of bits corresponding to the three least significant bits of register r2; result is placed in r1)

SRL src, dst Shift Right Logical: Register dst:= dst >> src[2:0]. The byte in register dst is shifted right by src[2:0] bits and filled by zeros in the most significant src[2:0] bits. Sets C, N, Z flags.

OPCODE
0110 11 – (Cycle 1)
0101 src=[3:2], dst=[1:0] (Cycle 2)

Example: srl r2, r1 (Shifts the contents of register r1 right by the number of bits corresponding to the three least significant bits of register r2; result is placed in r1)

ROL src, dst Rotate Left: Register dst:= dst <<< src[2:0]. The byte in register dst is rotated left by src[2:0] bits. Sets C, N, Z flags.

OPCODE
0110 11 – (Cycle 1)
0110 src=[3:2], dst=[1:0] (Cycle 2)

Example: rol r2, r1 (Rotates the contents of register r1 left by the number of bits corresponding to the three least significant bits of register r2; result is placed in r1)

SLL src, dst Shift Left Logical: Register dst:= dst << src[2:0]. The byte in register dst is shifted left by src[2:0] bits and filled by zeros in the least significant src[2:0] bits. Sets C, N, Z flags.

OPCODE
0110 11 – (Cycle 1)
0111 src=[3:2], dst=[1:0] (Cycle 2)

Example: sll r2, r1 (Shifts the contents of register r1 left by the number of bits corresponding to the three least significant bits of register r2; result is placed in r1)

STR Store Status Register: Stores the status register at the memory location referenced by [SP]. The stack pointer is post-decremented (SP := SP – 1).

OPCODE
0111 ---- (Cycle 1)

Example: str (Stores the contents of the status register on the stack)

LSR Load Status Register: Loads the status register with the contents of the byte at memory location [SP+1]. The stack pointer is post-incremented (SP := SP + 1)

OPCODE

1000 ---- (Cycle 1)

Example: lsr (Loads the status register using the contents of the status byte stored on the stack)

JSR addr[11:0] Jump to Subroutine: Stores the return address in two consecutive bytes in memory at [SP] and [SP-1]. Updates PC = addr. The stack pointer is post-decremented once during each clock cycle for a total decrement of 2 (SP := SP – 2). Note: The status register is not saved. In order to save the SR, you must manually execute the STR command prior to JSR. Note: This command cannot be used if PC modulo 256 > 253 (i.e., the least significant 8 bits of PC are 254 or 255). This is a boundary violation enforced by the assembler. This can be resolved by placing one or two NOPs in the code until the error goes away).

OPCODE
1001 addr[11:8] (Cycle 1)
Addr[7:0] (Cycle 2)

jsr nxtloop (Jumps to the subroutine at address referenced by nxtloop. Upon completion from the subroutine, control is returned to the subsequent instruction.)

RET Return from subroutine: Transfers the return address in two consecutive bytes from memory at [SP+1] and [SP+2] to the program counter PC. The stack pointer is post-incremented once during each clock cycle for a total increment of 2 (SP := SP + 2). Note: The status register is not restored. In order to restore the SR, you must manually execute the LSR command after RET.

OPCODE
1010 ---- (Cycle 1) (Note: this command occupies only one byte in program memory but takes 2 cycles)
-------- (Cycle 2)

Example: ret (Returns to the two-byte PC stored on the stack)

XOR src, dst Exclusive OR: Register dst := src ^ dst.

OPCODE
1011 src=[3:2], dst=[1:0] (Cycle 1)

Example: xor r0, r1 (XORs the contents of register r0 with register r1 and places result in r1)

TRF op, dst Transformation: Computes transformation op[3:2] on the byte in register dst and stores the result in dst such that dst := TRF{op}(dst). There are 4 possible transformations t0, t1, t2, and t3.

OPCODE
1100 op=[3:2],dst=[1:0] (Cycle 1)

Example: trf t0, r0 (Performs transformation t0 on the contents of register r0, places results in r0)

JMP addr[11:0] Jump (immediate): Jumps to the immediate 12-bit target specified in the operand. PC := addr.

OPCODE
1101 addr[11:8] (Cycle 1)
Addr[7:0] (Cycle 2)

Example: jmp lpend

BCX [dst] Branch on Carry (Indirect): If C flag is set branches to a new address, where addr[11:8] = PC[11:8] (i.e., most-significant 4 bits), and addr[7:0] = [dst], where the least significant 8 bits are contained in register dst. The C, N, and Z flags are reset. Note: the branch target is an absolute target which must be located in the same 256-byte block referenced by addr[11:8].

Warning: This restriction is not enforced by the assembler – the user must monitor for intended effect.

OPCODE
1110 00 dst=[1:0] (Cycle 1)

Example: bcx r0, r1 (If carry flag is set, branch to the address PC[11:8] & [r1])

BNX [dst] Branch on Negative (Indirect): If N flag is set branches to a new address, where addr[11:8] = PC[11:8] (i.e., most-significant 4 bits), and addr[7:0] = [dst], where the least significant 8 bits are contained in register dst. The C, N, and Z flags are reset. Note: the branch target is an absolute target which must be located in the same 256-byte block referenced by addr[11:8].

Warning: This restriction is not enforced by the assembler – the user must monitor for intended effect.

OPCODE
1110 01 dst=[1:0] (Cycle 1)

Example: bnx r0, r1 (If negative flag is set, branch to the address PC[11:8] & [r1])

BZX [dst] Branch on Zero (Indirect): If Z flag is set branches to a new address, where addr[11:8] = PC[11:8] (i.e., most-significant 4 bits), and addr[7:0] = [dst], where the least significant 8 bits are contained in register dst. The C, N, and Z flags are reset. Note: the branch target is an absolute target which must be located in the same 256-byte block referenced by addr[11:8].
Warning: This restriction is not enforced by the assembler – the user must monitor for intended effect.

OPCODE

1110 10 dst=[1:0] (Cycle 1)

Example: bzx r0, r1 (If zero flag is set, branch to the address PC[11:8] & [r1])

BCI addr[9:0] Branch on Carry (Immediate): If C flag is set branches to a new address, where addr[11:10] = PC[11:10] (i.e., most-significant 2 bits), and addr[9:0] = Im[9:0], where the least significant 10 bits are contained in the immediate operand. The C, N, and Z flags are reset. Note: the branch target is an absolute target which must be located in the same 1024-byte block referenced by addr[11:10]. This restriction is enforced by the assembler.

OPCODE
1111 00 addr[9:8]=[1:0] (Cycle 1)
Addr[7:0] (Cycle 2)

Example: bci lpjmp (If carry flag is set, branch to the address PC[11:10] & lpjmp[9:0])

BNI addr[9:0] Branch on Negative (Immediate): If N flag is set branches to a new address, where addr[11:10] = PC[11:10] (i.e., most-significant 2 bits), and addr[9:0] = Im[9:0], where the least significant 10 bits are contained in the immediate operand. The C, N, and Z flags are reset. Note: the branch target is an absolute target which must be located in the same 1024-byte block referenced by addr[11:10]. This restriction is enforced by the assembler.

OPCODE
1111 01 addr[9:8]=[1:0] (Cycle 1)
Addr[7:0] (Cycle 2)

Example: bni lpjmp (If negative flag is set, branch to the address PC[11:10] & lpjmp[9:0])

BZI addr[9:0] Branch on Zero (Immediate): If Z flag is set branches to a new address, where addr[11:10] = PC[11:10] (i.e., most-significant 2 bits), and addr[9:0] = Im[9:0], where the least significant 10 bits are contained in the immediate operand. The C, N, and Z flags are reset. Note: the branch target is an absolute target which must be located in the same 1024-byte block referenced by addr[11:10]. This restriction is enforced by the assembler.

OPCODE
1111 10 addr[9:8]=[1:0] (Cycle 1)
Addr[7:0] (Cycle 2)

Example: bci lpjmp (If zero flag is set, branch to the address PC[11:10] & lpjmp[9:0])

NOP No operation. This instruction is implemented by MOV r0, r0. The status register is unaffected, and the program counter is incremented.

OPCODE
0100 00 00 (Cycle 1)

Example: nop (Performs no operation; registers and status registers unchanged; PC = PC + 1)

Appendix B
User-defined Instruction Set Extensions (ISE)

Note: The following instructions are ALU instruction set extensions for use in specified applications. They are not included in the baseline version of SoftCore, and must be specifically instantiated in the VHDL source code for SoftCore.

Warning: As there are only 8 spaces available for user-defined ALU instructions, it is possible that the OPCODES of the below instructions could be contradictory. Care should be taken to ensure that the OPCODES in the ALU1 multiplexer match the encoding performed by the assembler, and evaluated by the simulator.

GF4 src, dst Galois Field multiplication of two 4-bit operands where $P(x) = x^4 + x + 1$: Register dst:= src * dst modulo P(x) polynomial. Note: This is a specialized instruction for LED.

OPCODE
0110 11 – (Cycle 1)
1000 src=[3:2], dst=[1:0] (Cycle 2)

Example: gf4 r2, r0 (Multiplies the contents of register r2[3:0] by r0[3:0] and places results in register r0. The upper nibble of r0, i.e., r0[7:4], is cleared).

GF2 src, dst Galois Field multiplication by a constant of 2 where $P(x) = x^8 + x^4 + x^3 + x + 1$: Register dst:= src * 2 modulo AES polynomial. Note: This is a specialized instruction for AES.

OPCODE
0110 11 – (Cycle 1)
1001 src=[3:2], dst=[1:0] (Cycle 2)

Example: gf2 r2, r0 (Multiplies the contents of register r2 by 2 and places results in register r0)

GF3 src, dst Galois Field multiplication by a constant of 3 where $P(x) = x^8 + x^4 + x^3 + x + 1$: Register dst:= src * 3 modulo AES polynomial. Note: This is a specialized instruction for AES.

OPCODE
0110 11 – (Cycle 1)
1010 src=[3:2], dst=[1:0] (Cycle 2)

Example: gf3 r2, r0 (Multiplies the contents of register r2 by 3 and places results in register r0)

PRW dst, src Writes the value of register <src> to the permutation register specified in <dst>. The allowable values of <dst> are 0 through 9, which correspond to registers p0 to p9. The least significant nibble of <dst> is used for the permutation register, i.e., dst[3:0]. The programmer is cautioned that values of <dst> greater than 9 legally encode, but are undefined in hardware. Note: This is a specialized instruction for PRESENT.

OPCODE
0110 11 – (Cycle 1)
1011 dst=[3:2], src=[1:0] (Cycle 2)

Example: prw r2, r0 (Writes the contents of r0 into the permutation register specified by r2)

PRR src, dst Reads the unpermutated and unrotated value of the register p0 to p9 specified in register <src> and places the result in register r0 to r3 specified in <dst>. The allowable values of <src> are 0 through 9, which correspond to registers p0 to p9. The least significant nibble of <src> is used for the permutation register, i.e., src[3:0]. The programmer is cautioned that values of <src> greater than 9 legally encode, but are undefined in hardware. Note: This is a specialized instruction for PRESENT.

OPCODE
0110 11 – (Cycle 1)
1100 src=[3:2], dst=[1:0] (Cycle 2)

Example: prr r2, r0 (Reads the non-permutated and non-rotated contents of permutation register specified in r2 and writes the results into register specified in r0)

PRP src, dst Reads the permutated value of the register p0 to p7 specified in register <src> and places the result in register r0 to r3 specified in <dst>. The allowable values of <src> are 0 through 7, which correspond to registers p0 to p7. The least significant 3 bits of <src> are used for the permutation register, i.e., src[2:0]. The programmer is cautioned that values of <src> greater than 7 legally encode, but are undefined in hardware. Note: This is a specialized instruction for PRESENT. The permutation is the 64-bit permutation defined in the PRESENT cipher.

Caution: The correct permutations in registers p0 to p7 are not defined until all 8 registers p0 to p7 have been written. Attempts to read the permutation registers before writing all p0 to p7 will produce undefined results.

OPCODE
0110 11 – (Cycle 1)
1101 src=[3:2], dst=[1:0] (Cycle 2)

Example: prp r2, r0 (Reads the permutated contents of permutation register specified in r2 and writes the results into register specified in r0)

PRS src, dst Reads the rotated value of the register p0 to p9 specified in register <src> and places the result in register r0 to r3 specified in <dst>. The allowable values of <src> are 0 through 9, which correspond to registers p0 to p9. The least significant nibble of <src> is used for the permutation register, i.e., src[3:0]. The programmer is cautioned that values of <src> greater than 9 legally encode, but are undefined in hardware. Note: This is a specialized instruction for PRESENT. The rotation is a 61-bit left circular shift performed on an 80-bit word defined by p9 (MSB) to p0 (LSB). This 61-bit circular shift permutation is defined in the in the PRESENT cipher.

Caution: The correct rotations in registers p0 to p9 are not defined until all 10 registers p0 to p9 have been written. Attempts to read the permutation registers before writing all p0 to p9 will produce undefined results.

OPCODE
0110 11 – (Cycle 1)
1110 src=[3:2], dst=[1:0] (Cycle 2)

Example: prs r3, r2 (Reads the rotated contents of permutation register specified in r3 and writes the results into register specified in r2)

MTW dst, src Writes the least significant nibble of <src> to the 4x1 column vector position y, where y is position 0 to 3, specified in <dst>. The allowable values of <dst> are 0 through 3, which correspond to column positions 0 to 3. The least significant 3 bits of <dst> are used for the multiplier register, i.e., dst[1:0]. The programmer is cautioned that values of <dst> greater than 3 legally encode, but are undefined in hardware. A write to column position 0 initializes registers m0 to m3 with the product of <src> * constant $\{a_0, b_0, c_0, d_0\}$ mod $P(x) = x^4 + x + 1$. A write to column position y (where y = 1 to 3) accumulates the sum of m0 to m3 with <src> * constant $\{a_y, b_y, c_y, d_y\}$ mode P(x). The scalar product of each 1x4 constant row vector by 4x1 variable column vector is only defined after MTW has been called on all positions 0 to 3 in succession.

Note: This is a specialized instruction for LED. The 4x4 matrix of constants is shown below. The values for the constants are available in the LED cipher specification.

a_0 a_1 a_2 a_3
b_0 b_1 b_2 b_3
c_0 c_1 c_2 c_3
d_0 d_1 d_2 d_3

OPCODE
0110 11 – (Cycle 1)
1100 dst=[3:2], src=[1:0] (Cycle 2)

Example: mtw r3, r2 (writes the product of the value in r2 and the constants [a<r3>, b<r3>, c<r3>, d<r3>] to the multiplication registers m0 to m3)

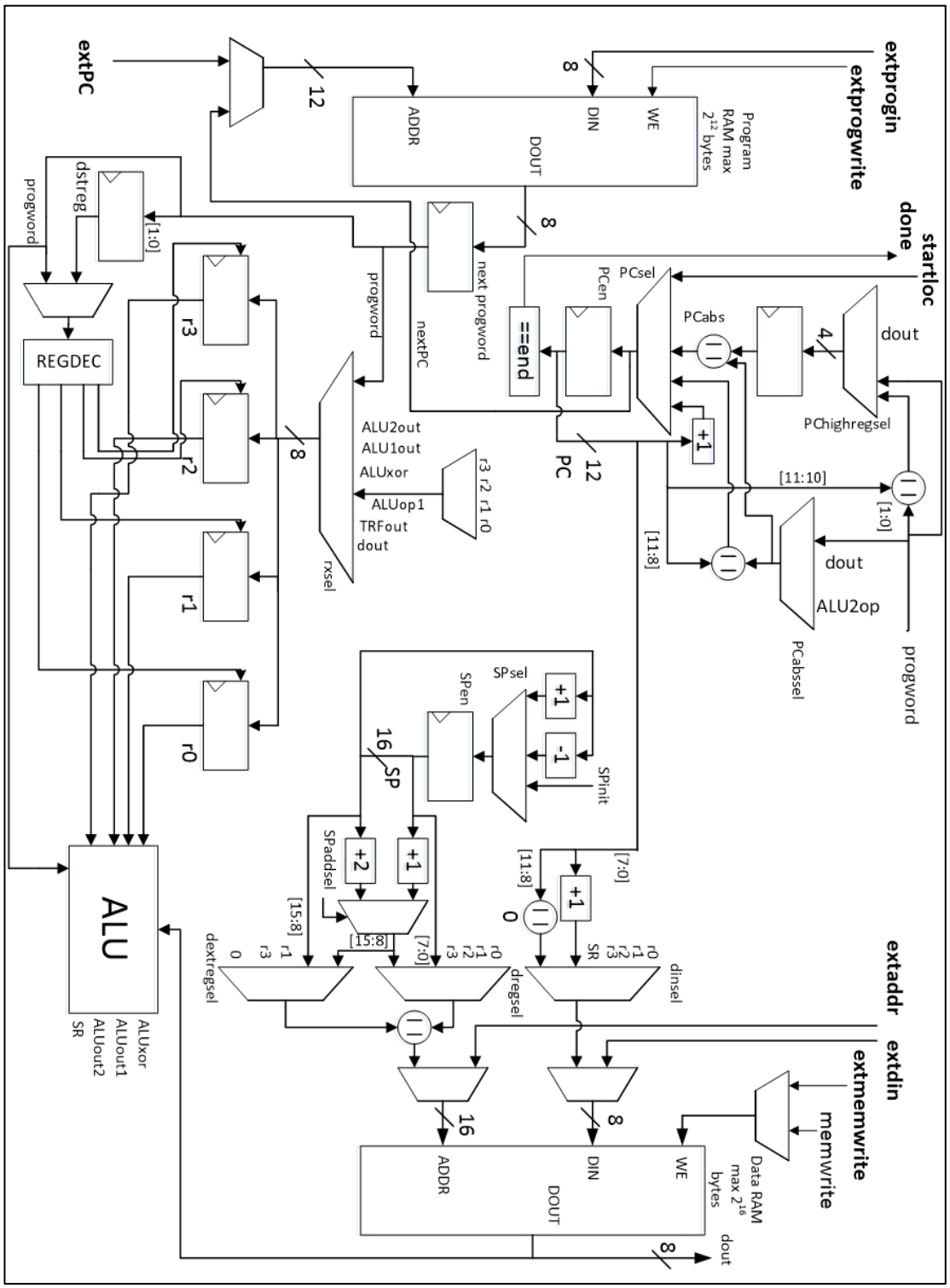
MTR src, dst Reads the scalar column matrix result at column position y specified in <src>, where y is 0 to 3, and places result in register <dst>. Caution: MTR scalar results can be read in any order, but should only be read after all four MTW instructions have been performed in the correct order on column vector positions 0 to 3, or results will be undefined.

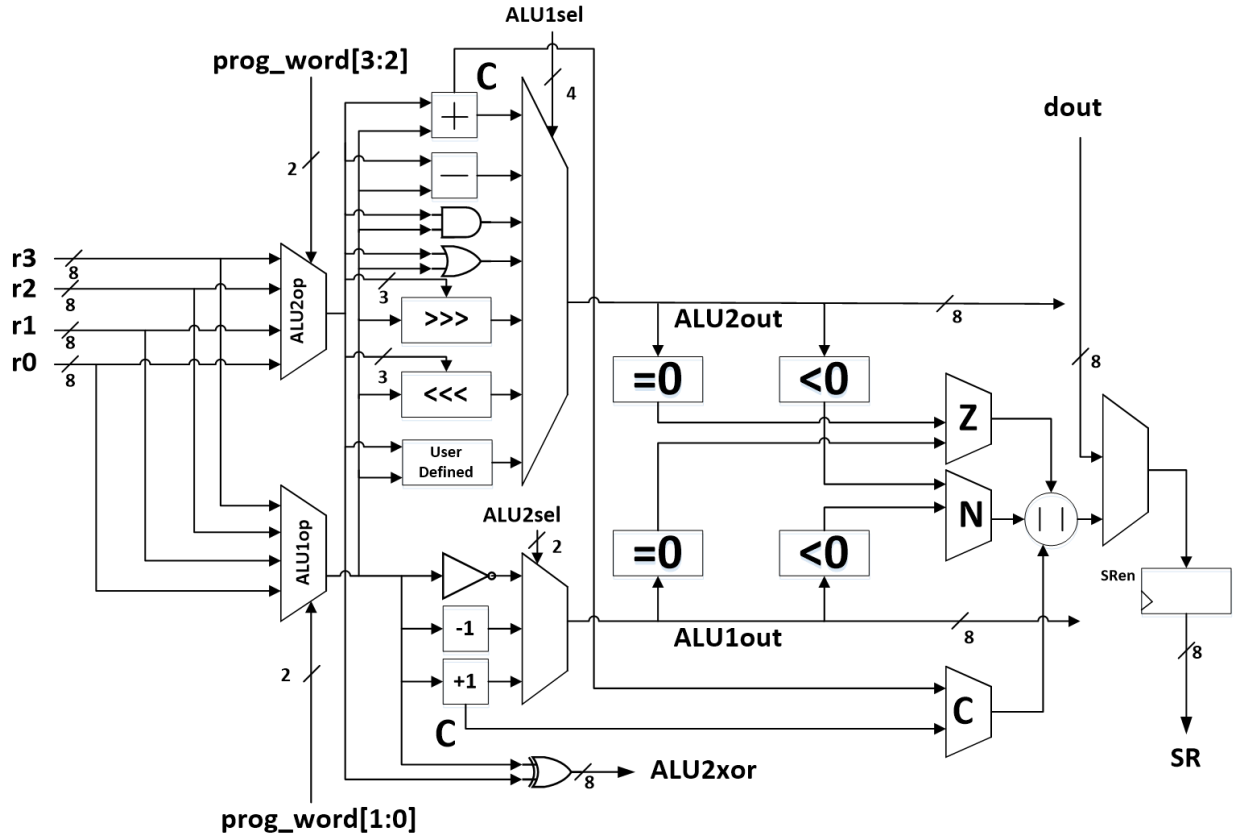
Note: This is a specialized instruction for LED.

OPCODE
0110 11 – (Cycle 1)
1111 src=[3:2], dst=[1:0] (Cycle 2)

Example: mtr r2, r1 (writes the scalar result of column vector position y in <r2> and places result in r1. The upper nibble r1[7:4] is cleared.)

Appendix C
Soft Core Block Diagrams




ALU1sel
prog_word[3:2]
C
4
dout
2
r3
r2
r1
r0
8
ALU2op
3
>>>
<<<
User Defined
ALU2out
8
=0
<0
Z
N
8
ALU2sel
2
=0
<0
SRen
ALU1op
-1
+1
ALU1out
8
2
C
C
8
ALU2xor
SR
prog_word[1:0]