SAM Architecture Reference (Revised)

Mika Nyström

Department of Computer Science
California Institute of Technology
Pasadena, California, U.S.A.

2000

April 19, 2000

Revised April 2, 2002

1. Introduction

This report describes the Simple Asynchronous Microprocessor (SAM) architecture. SAM is a simple 32-bit RISC architecture intended for hardware demonstration projects. Its design reflects a desire of making a high-performance implementation as easy as possible. This is not without merit on the software level; for instance, as a result of the desire of keeping the hardware as simple as possible, the instruction set of the SPAM processor is completely orthogonal; i.e., all instructions use the same addressing mode and instruction format.

2. SPAM overview

The SPAM architecture defines eight general-purpose registers, `gpr[0]` through `gpr[7]`, of which `gpr[0]` is always read as zero, although it may be written by any instruction. The remaining state of the processor is the program counter `pc`. The instructions provided are arithmetic instructions, load/store instructions, and `pc`-changing instructions. Changes to `pc` take effect immediately—there is no "branch delay slot." The architecture does not define floating-point operations, interrupts, or exceptions.

3. SPAM instruction format

All SPAM instructions have the same format. The instruction format is a four-operand RISC format with three register operands and a single immediate operand. The opcode format has two fields, which are also the same across all instructions. These fields are the operation unit and the operation function. The operation "Y-mode," which determines the addressing mode used to conjure operand `opy`, is further defined in a fixed position in the instruction.

SPAM instructions are 32 bits wide. Considering a SPAM instruction i as a 32-bit array of bits, we identify the fields of the instruction:

1. The `opcode` $= i[31 \ldots 27]$, further divided into:
   a. The unit number `unit` $= i[31 \ldots 30]$.
   b. The function `fxn` $= i[29 \ldots 27]$.
2. The Y-mode `ymode` $= i[26 \ldots 25]$.
3. The result register number `rz` $= i[24 \ldots 22]$.
4. The X-operand register number `rx` $= i[21 \ldots 19]$.
5. The Y-operand register number `ry` $= i[18 \ldots 16]$.
6. The immediate field `imm` $= i[15 \ldots 0]$.

4. SPAM instruction semantics

Because the SPAM instruction set is orthogonal, we may define the semantics of instructions in a modular way. An instruction execution consists of the following steps:

1. Generation of the operands:

   `opx` := `gpr[`i`.rx]` and `opy` := `YMODE(`i`.ymode)(gpr[`i`.ry],`i`.imm)`

2. Computation of the result:

   `opz` := `OP(`i`.opcode)(opx,opy)`

   2a. Computation of the next `pc`:

   `pc` := `PCOP(`i`.opcode)(pc,opx,opy)`

3. Write-back of `opz`:

   `gpr[`i`.rz]` := `opz`

4.1. Operand generation

The first operand, `opx`, is always the contents of `gpr[`i`.rx]`. The second operand, `opy`, is computed from the contents of `gpr[`i`.ry]` and the immediate field, depending on i`.ymode`.

Allowable values for i`.ymode` are as follows, where *sext* signifies sign extension:

i`.ymode` Mnemonic	Decimal value	Operand generated
`YMODE_REG`	0	`opy := gpr[`i`.ry]`
`YMODE_IMM`	1	`opy :=` *sext*`(`i`.imm)`
`YMODE_IMMSHIFT`	2	`opy :=` i`.imm << 16`
`YMODE_REGIMM`	3	`opy := gpr[`i`.ry] +` *sext*`(`i`.imm)`

4.2. Operation definitions

Operations are defined on two's-complement numbers. There are no flags or condition codes. We divide the operations by unit:

4.2.1. ALU operations i`.unit` = `UNIT_ALU` = 0

All ALU operations take two operands and produce one result. The *bitwise_NOR* is included in the instruction set for the express purpose of computing the bitwise inverse of `opx` using a zero operand for `opy`.

Mnemonic	Name	i`.fxn`	Operation
`add`	Add	0	`opz := (opx + opy)`$_{31\ldots0}$
`sub`	Subtract	1	`opz := (opx - opy)`$_{31\ldots0}$
`nor`	NOR	4	`opz :=` *bitwise_NOR*`(opx,opy)`
`and`	AND	5	`opz :=` *bitwise_AND*`(opx,opy)`
`or`	OR	6	`opz :=` *bitwise_OR*`(opx,opy)`
`xor`	Exclusive OR	7	`opz :=` *bitwise_XOR*`(opx,opy)`

4.2.2. Branch operations i.`unit` = `UNIT_BRCH` = 1

All branch operations unconditionally produce the same result as `opz`, namely the value of `pc`, right-shifted by two. Likewise, a branch taken will branch to the address denoted by `opy` left-shifted by two and incremented by one. The shifting avoids alignment errors.

Note that the mechanism described for branch addresses allows a simple compilation of function call/return linkage. The function call jump saves the current PC, and then the function return jump calls back through the saved address. Coroutine linkage is compiled similarly. (The SPAM architecture leaves unspecified function parameter linkage conventions and register save masks, etc.)

The `hlt` instruction halts the processor. An external action, not defined within the architecture, is required to restart the machine.

Conditional branches branch on the value of `opx`.

Mnemonic	Name	i.`fxn`	Branch if	Target
`hlt`	Halt	0	**true**	$\perp$
`beq`	Branch on Equal	1	`opx` $= 0$	`opy`$_{29\ldots0}\|00$
`bne`	Branch on Not Equal	2	`opx` $\neq 0$	`opy`$_{29\ldots0}\|00$
`bgt`	Branch on Greater Than	3	`opx` > 0	`opy`$_{29\ldots0}\|00$
`blt`	Branch on Less Than	4	`opx` < 0	`opy`$_{29\ldots0}\|00$
`ble`	Branch on Less or Equal	5	`opx` ≤ 0	`opy`$_{29\ldots0}\|00$
`bge`	Branch on Greater or Equal	6	`opx` ≥ 0	`opy`$_{29\ldots0}\|00$
`jmp`	Jump	7	**true**	`opy`$_{29\ldots0}\|00$

4.2.3. Memory operations i.`unit` = `UNIT_DMEM` = 2

Only two memory operations are defined, load word, `lw`, and store word, `sw`. The address of the memory access is determined by `opy`. On a memory load, `opx` is ignored, whereas on a store, it becomes the value stored. A store returns `opy` (the computed address) as `opz`.

Mnemonic	Name	i.`fxn`	Operation
`lw`	Load Word	0	`opz := dmem[opy]`
`sw`	Store Word	4	`dmem[opy] := opx, opz := opy`

4.2.4. Shifter operations i.`unit` = `UNIT_SHFT` = 3

The SPAM architecture defines a restricted shifter that is capable only of logical shifts. Arithmetic shifts must be simulated using `blt`. The SPAM shifter can shift by one or eight. Shifts-by-eight are provided so that byte memory operations can proceed at a reasonable speed.

Mnemonic	Name	i.fxn	Operation
sr1	Shift Right by One	0	opz := $0\|$opy$_{31\ldots1}$
sr8	Shift Right by Eight	1	opz := $00000000\|$opy$_{31\ldots8}$
sl1	Shift Left by One	2	opz := opy$_{30\ldots0}\|0$
sl8	Shift Left by Eight	3	opz := opy$_{23\ldots0}\|00000000$

4.2.5. Undefined operations

Operations not defined in this reference manual are reserved for future expansion and must not be used. The behavior of the undefined operations is UNDEFINED.

4.2.6. System reset

The mechanism to cause a system reset is implementation-dependent. On system reset, the processor starts execution with pc $= 8$ and arbitrary data in all general purpose registers except gpr[0].

5. Assembly language conventions

The SPAM architecture uses a simple, understandable assembly language syntax that is free from the traditional confusion about which register identifier names the operand and which names the result.

5.1. The SPAM assembly format

The SPAM assembly format is best illustrated with an example:

```
;;; Compute sum of 100 first integers
;;; Do some other things to test the processor
.=0x8
                  jmp Start          ; comment
.=0x100
Start:
                  li r1=100
                  li r2=0U           ; upper immediate
                  jmp r3=Detour      ; comment
Label:                               ; comment
                  add r2=r1,r2
                  sw r2,(100)
                  lw r2=(r1+0x3ff)
                  lw r2=(100)
                  sub r1=r1,1
                  bne r1,Label
                  hlt
                  jmp zero           ; shouldnt get executed
                  nop
.=0x200                              ; test comment
Detour:           jmp r3
```

5.1.1. Assembly instruction syntax

In the example, we see the use of some standard assembler conventions, such as `.` for setting the desired memory location of the current instruction. We also see that the syntax of the instructions is $<$ *mnemonic* $><$ *result register* $>=<$ *operands* $>$. Parentheses are used for memory instructions to denote the indirection that comes from using a register to make a memory reference. Labels can be used directly by the branches. Any field not specified will be assembled as zero; this has several benefits—e.g., not specifying the target register of an operation makes the target `gpr[0]`, which simply means that the result will be discarded.

5.1.2. Specification of immediates

Immediates are specified either in decimal or in hexadecimal. Hexadecimal numbers must be preceded with the string `0x` to flag their base. Following an immediate with the roman capital `U` flags it as being an "upper" immediate; i.e., it will be shifted 16 bits left before it is used.

5.1.3. Pseudo-instructions

There are also several *pseudo-instructions* in the example program that are understood by the assembler and mapped to the machine language instructions presented earlier. The pseudo-instructions understood by the assembler are:

Pseudo-instruction	Name	Operation
`li rz=opy`	Load immediate	`or rz=rx,opy`
`nop`	No operation	`add r0=r0,r0`
`not rz=opy`	NOT	`nor rz=0,opy`

Notice that the `nop` pseudo-instruction assembles to an all-zeros instruction word.