SP2 CPU

Programming Model

The SP2 CPU has a 16-bit datapath, with all address and data registers 16 bits wide. Its uniform address space uses *word addressing*, so that individual memory bytes are not directly accessible. Its *dual stack architecture* consists of just the seven programmer-visible entities shown here.

entity	Description
DS	This part of the **data stack** is a LIFO structure which is internally addressed modulo its size, resulting in no overflow or underflow. That means pushing an extra item overwrites the oldest item and popping an extra item rereads the oldest item (circular access). The **data stack** is accessible only by pushing or popping **D**.
D	This register is the second element of the **data stack**. It is used in most alu operations, is the source for stores, and is the destination for loads.
A	This register is the top element of the **data stack**. It is used in all alu operations, is an address for loads and stores, and is the link between the data and return stacks.
R	This register is the top element of the **return stack**. It is used to save the last subroutine return address, is an address for loads and stores, holds the decrementing index for counted loops, and may be pushed and popped for temporary storage.
RS	This part of the **return stack** is a LIFO structure which is internally addressed modulo its size, resulting in no overflow or underflow. As with the data stack, that means pushing an extra item overwrites the oldest item and popping an extra item rereads the oldest item (circular access). The **return stack** is accessible only by pushing or popping **R**.
P	This register holds the **program counter**, holding the next sequential program address.
I	This register holds the current **instruction word** or **execution token**.

External interface

Enough control lines are provided so that an external MMU can provide access to application-specific memory, devices, and interrupts. Upon reset the cpu begins in Code State, fetching the instruction word at address zero.

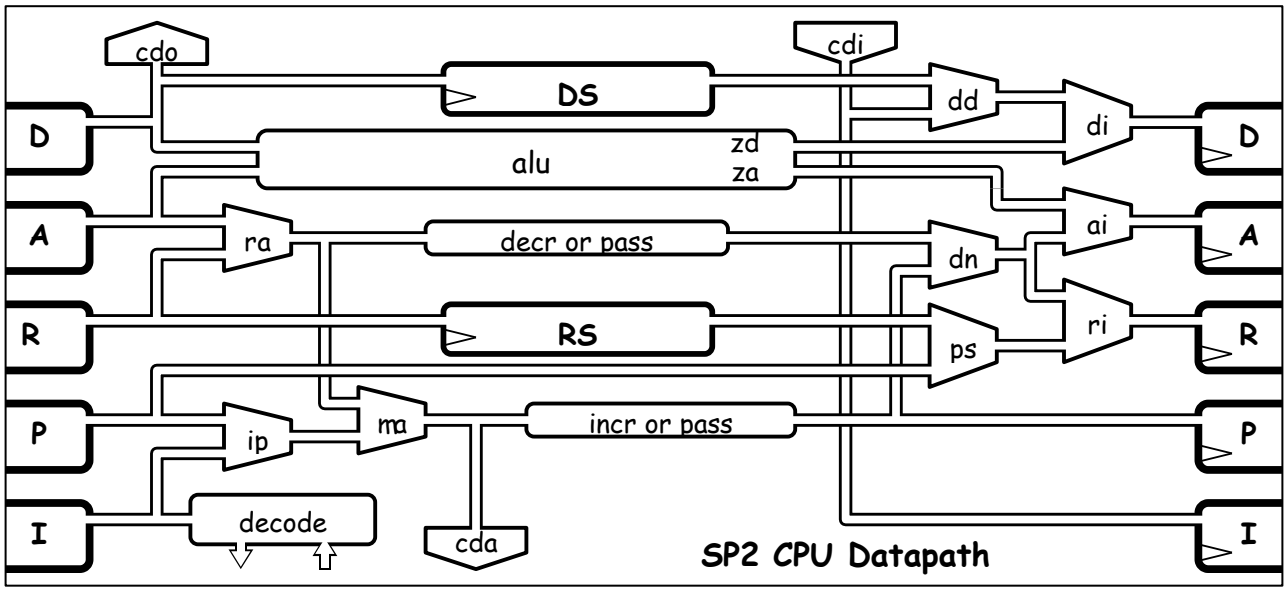
The design is described in Verilog, suitable for compilation into a variety of FPGAs. The depth of the two stacks is not described in the cpu, but is 256 words in the current implementation because that is the smallest memory block available.

Signal	i/o	width	description
clock	i	1	periodic input triggering all the registers and stacks on its rising edge
reset	i	1	bring high to reset the cpu, fetching a Code State instruction from address 0.
hold	i	1	bring high to prevent the address from advancing
data	o	1	high specifies a data access, low an instruction/token fetch (read only)
read	o	1	high specifies a read access, low a write (both read & data low is no access)
next	o	1	high indicates List State for next cycle, low indicates Code State
cda	o	16	data address from the cpu
cdo	o	16	data output from the cpu
cdi	i	16	data input to the cpu
done	i	1	high signals the completion of a read/write/fetch access, low otherwise

CPU Details

The limits to the SP2 instruction set are the result of the single-cycle datapath design shown below. All registers, including the stacks, change on the rising edge of the clock. The remaining modules (decode, increment, decrement, alu, and multiplexers) are combinatorial.

The diagram depicts one clock cycle running from left to right, with the register values available on the left and latched on the right. The multiplexer and alu labels are the names of their local output buses.



I. Code State

Instruction words and opcodes

SP2's dual stack architecture requires few operands to be explicitly named, it has only 42 *opcodes*, most only 5 bits wide. This allows up to three opcodes and an optional subroutine return to be packed into each instruction word and fetched as a unit.

The opcodes within an instruction word are executed sequentially, one per cpu or memory access cycle, starting with the one in slot 0 (bits 14:10), followed by the ones in slot 1 (bits 9:5) and slot 2 (bits 4:0). If there isn't a control-group opcode in slots 0-2, an additional slot 3 opcode is used to fetch a new instruction word.

bit 15	bits 14:10	bits 9:5	bits 4:0
slot 3	slot 0	slot 1	slot 2

Control group

The jump and jump-subroutine opcodes conditionally change the sequence of execution, using the low-order address bits used to fetch the next instruction word.

bit 15	bits 14:10	bits 9:5	bits 4:0
jsr/jmp	slot 0/1 control-group opcode	10-bit page address	
jsr/jmp	other opcode	slot 0/1 control-group opcode	5-bit page address
ret/nxt	other opcode	other opcode	slot 2 control-group opcode
ret/nxt	other opcode	other opcode	other opcode

The low-order 5 or 10 bits of **I** are used to **replace** the corresponding bits of **P**. While this isn't as versatile as being added or subtracted, it's considerably faster. In the small embedded systems for which this processor is targeted, the code is expected to consist of many small subroutines and loops, so fast execution of jumps and subroutine calls is required. To reach locations beyond the 10 bit range of a slot 0 jump, a longjump opcode may be constructed by pushing a value into **R** and executing the return opcode (and see **List State** below).

The behavior of the jumps in slot 0 and slot 1 depends on bit 15:

opcode	value	I[15]	Slot 0 & Slot 1 Behavior description
jmp	0x00	0	Fetch the next instruction word from the address {**P**[15:10], **I**[9:0]} or {**P**[15:5], **I**[4:0]}, depending on slot. Replace **P** with the incremented address.
jmpd	0x01	0	If **R** is non-zero, decrement it and fetch the next instruction word from the address {**P**[15:10], **I**[9:0]} or {**P**[15:5], **I**[4:0]}, depending on slot. Otherwise fetch the next instruction word from the address in **P** and pop **RS** to **R**. In both cases replace **P** with the incremented address.
jmpp	0x02	0	If **A**[15] is zero (positive), fetch the next instruction word from the address {**P**[15:10], **I**[9:0]} or {**P**[15:5], **I**[4:0]}, depending on slot. Otherwise fetch the next instruction word from the address in **P**. In both cases replace **P** with the incremented address.
jmpz	0x03	0	If **A** is zero, fetch the next instruction word from the address {**P**[15:10], **I**[9:0]} or {**P**[15:5], **I**[4:0]}, depending on slot. Otherwise fetch the next instruction word from the address in **P**. In both cases replace **P** with the incremented address.

opcode	value	I[15]	Slot 0 & Slot 1 Behavior description
jsr	0x00	1	Fetch the next instruction word from the address {**P**[15:10], **I**[9:0]} or {**P**[15:5], **I**[4:0]}, depending on slot, push **R** to **RS**, and move **P** to **R**. Otherwise fetch the next instruction word from the address in **P**. In both cases replace **P** with the incremented address.
jsrn	0x01	1	If **R** is non-zero, fetch the next instruction word from the address {**P**[15:10], **I**[9:0]} or {**P**[15:5], **I**[4:0]}, depending on slot, push **R** to **RS**, and move **P** to **R**. Otherwise fetch the next instruction word from the address in **P**. In both cases replace **P** with the incremented address.
jsrp	0x02	1	If **A**[15] is zero (positive), fetch the next instruction word from the address {**P**[15:10], **I**[9:0]} or {**P**[15:5], **I**[4:0]}, depending on slot, push **R** to **RS**, and move **P** to **R**. Otherwise fetch the next instruction word from the address in **P**. In both cases replace **P** with the incremented address.
jsrz	0x03	1	If **A** is zero, fetch the next instruction word from the address {**P**[15:10], **I**[9:0]} or {**P**[15:5], **I**[4:0]}, depending on slot, push **R** to **RS**, and move **P** to **R**. Otherwise fetch the next instruction word from the address in **P**. In both cases replace **P** with the incremented address.

In slot 2 there are no address bits available, so the control opcodes conditionally repeat execution of the current instruction word beginning at slot 0. When any condition is not met, slot 3 controls where to fetch the next instruction word. The opcode value decoded as an unconditional jump or call in slots 0 and 1 is used in slot 2 to enter **List State** (see below).

opcode	value	I[15]	Slot 2 Behavior description
into	0x00	0	Enter **List State** to fetch the next execution token from the address in **P**. Replace **P** with the incremented address.
exit	0x00	1	Return to **List State** to fetch the next execution token from the address in **R**. Replace **P** with the incremented address and pop **RS** to **R**.
repd	0x01	-	If **R** is non-zero, decrement it and repeat the current instruction word from slot 0. Otherwise pop **RS** to **R** and continue with slot 3.
repp	0x02	-	If **A**[15] is zero (positive), repeat the current instruction word from slot 0. Otherwise continue with slot 3.
repz	0x03	-	If **A** is zero, repeat the current instruction word from slot 0. Otherwise continue with slot 3.

When slot 0 and slot 1 contain no control group opcodes, bit 15 determines where the next instruction word is found.

opcode	value	I[15]	Slot 3 Behavior description
(nxt)	-	0	Fetch the next instruction word from the address in **P**. Replace **P** with the incremented address.
ret	-	1	Fetch the next instruction word from the address in **R** and pop **RS** to **R**. Replace **P** with the incremented address.

Move group

The opcodes in this group mostly move data between **R** and **A**.

opcode	value	behavior description
get	0x04	Push **D** to **DS**, move **A** into **D**, and copy **R** into **A**.
pop	0x05	Push **D** to **DS**, move **A** into **D**, move **R** into **A**, and pop **RS** to **R**.
drop	0x06	Delete *A*: move **D** into **A**, and pop **DS** to **D**.
push	0x07	Push **R** to **RS**, move **A** into **R**, move **D** into **A**, and pop **DS** to **D**.

Memory group

The opcodes in this group move data between the address space and **D**, pushing or popping **DS**, using an address from **A**, **R**, or **P**. The address register used may be incremented after the memory access.

opcode	value	behavior description
lit	0x08	Literal: push **D** to **DS** and read data into **D** using the address in **P**. Increment **P**.
ldri	0x09	Push **D** to **DS** and read data into **D** using the address in **R**. Increment **R**.
lda	0x0a	Push **D** to **DS** and read data into **D** using the address in **A**.
ldai	0x0b	Push **D** to **DS** and read data into **D** using the address in **A**. Increment **A**.
str	0x0c	Write data from **D** using the address in **R** and pop **DS** to **D**.
stri	0x0d	Write data from **D** using the address in **R** and pop **DS** to **D**. Increment **R**.
sta	0x0e	Write data from **D** using the address in **A** and pop **DS** to **D**.
stai	0x0f	Write data from **D** using the address in **A** and pop **DS** to **D**. Increment **A**.

ALU group

These instructions do various arithmetic, logical, and movement operations between **A** and **D**, pushing or popping **DS** as specified.

opcode	value	behavior description
swap	0x10	Exchange **A** and **D**.
nop	0x11	Do nothing.
eqz	0x12	Replace **A** with 0xffff if it is zero, with zero otherwise.
inv	0x13	Bitwise invert **A**.
add	0x14	Replace the pair **D.A** with the **sum** of **A** and **D** (unsigned addition).
lsl	0x15	Shift the pair **D.A** one bit to the left, inserting zero into bit 0 of **A**.
asr	0x16	Shift the pair **D.A** one bit to the right, replicating bit 15 of **D**.
lsr	0x17	Shift the pair **D.A** one bit to the right, inserting zero into bit 15 of **D**.
over	0x18	Push **D** to **DS** and exchange **A** and **D**.
dup	0x19	Push **D** to **DS** and copy **A** to **D**.
zero	0x1a	Push **D** to **DS**, move **A** to **D**, and set **A** to zero.
one	0x1b	Push **D** to **DS**, move **A** to **D**, and set **A** to one.
nip	0x1c	Delete **D**: Pop **DS** to **D**.
or	0x1d	Replace **A** with the **bitwise inclusive-or** of **A** and **D** and pop **DS** to **D**.
xor	0x1e	Replace **A** with the **bitwise exclusive-or** of **A** and **D** and pop **DS** to **D**.
and	0x1f	Replace **A** with the **bitwise and** of **A** and **D** and pop **DS** to **D**.

II. List State

Description

In addition to the usual **Code State** in which all microprocessors execute their native machine instructions, the SP2 has an additional **List State** in which nested lists of subroutine calls, often referred to as **threaded code**, may be executed with minimal overhead. This is particularly useful on the SP2 because of the limited range of the **jmp** and **jsr** opcodes and the relatively high cost of jumping or calling outside that range.

The inclusion of **List State** on the SP2 allows an efficient implementation of interrupts. It also permits an especially compact implementation of a threaded-code interpreter for the Forth language.

Implementation

There are just three **List State** instructions, called **execution tokens** to distinguish them from **Code State**'s instruction words. Non-zero execution tokens are simply 16-bit addresses of code and list routines. The low bit of the address is used to differentiate the two: code routines start at even addresses, list routines start at odd addresses. The address zero, the hardware reset address, is also used as the **exit** token, which returns to the previous list routine. Very little additional decode and no additional data paths are needed to implement **List State**.

type	token in I	behavior
exit	exit token (zero)	List return: Remain in **List State** to fetch the next execution token from the address in **R** and pop **RS** to **R**. Replace **P** with the incremented address.
call	code token (even address)	Code call: Enter **Code State** to fetch an instruction word from the address in **I**. Push **R** to **RS** and move **P** to **R**. Replace **P** with the incremented address.
nest	list token (odd address)	List call: Remain in **List State** to fetch an execution token from the address in **I**. Push **R** to **RS** and move **P** to **R**. Replace **P** with the incremented address.

The **Code State into** and **exit** opcodes, described in the control group above, are how machine code routines enter **List State**.

Note: Only the initial address of code and list routines (found as execution tokens in **I**) must be at even and odd addresses, respectively--subsequent list or code routine addresses (as found in **P** and **R**) have no such restrictions.

Interrupts

The SP2 CPU will normally have an associated MMU which handles the allocation and timing of the data accesses outside the CPU. If that MMU can provide an execution token (code or list routine address) to the CPU in response to an external signal, a very simple interrupt mechanism results.

One of the SP2 CPU's inputs is **hold**, which controls whether the **cda** address is incremented before being saved in **P**. If the MMU brings this line high while the SP2 is in **List State** and provides an execution token on the **cdi** bus, the CPU will execute that routine rather than the one normally fetched from the address. By not incrementing the address the specified routine will be executed when the interrupt routine returns.

The result is that execution in **Code State** is uninterruptible. If an appropriate interrupt acknowledge signal is provided, execution of at least one **List State** token between interrupts can be guaranteed, which will prevent a stuck interrupt line from locking the cpu.