RACORS

sf20bl

16-bit microprocessor

IMA (Implementation Architecture) Reference Manual

Revision 0.9
21 December 2014

Author: Martin Raubuch

Revision History

Revision	Date	
0.9	21Dec2014	First version

Table of contents

1 Overview

1.1 Introduction

The sf20 family of 16-bit microprocessors is targeted at embedded control applications that have high performance requirements and are satisfied with a direct addressable data space of 64kBytes. With 20-bit instruction coding excellent code efficiency is achieved for a fixed length architecture with 16 general purpose registers. The sf20 family is very well suited for FPGAs where 20-bit wide memories can be built efficiently.

The sf20family defines two ISAs (Instruction Set Architectures), a base (b) ISA for general purpose control & computing and a (d) DSP ISA extension for small 16-bit DSP applications. This manual is the IMA (Implementation Architecture) reference of the sf20bl, the (l) light implementation of the sf20 (b) base ISA.

1.2 Feature Summary

The following list summarizes the sf20bl's main features

- Focus on performance and moderate core size
- 20-bit wide instruction interface and 16-bit wide data interface
- Register file with 3/2 read/write ports
- Decoupled unit for instruction fetch and flow instruction execution
- Branch Speculation
- Loop cache, zero cycles loop branch from 2nd interation
- Separate execution pipelines for computation and load/store instructions
- Single cycle effective execution of all computation instructions
- Single cycle effective execution of all load/store instructions except load/store multiple
- Barrel shifter for single cycle effective execution of all shift instructions
- Average IPC (Instructions Per Cycle) of ~0.8 for typical code sequences
- 2 x 20-bit instruction pre-fetch buffer
- Fully synchronous design, all flip-flops are triggered with the rising edge of the clock input
- Clock rates up to 120MHz on low end FPGAs
- Clock rates >300MHz with deep sub-micron std-cell generic and low-power technologies

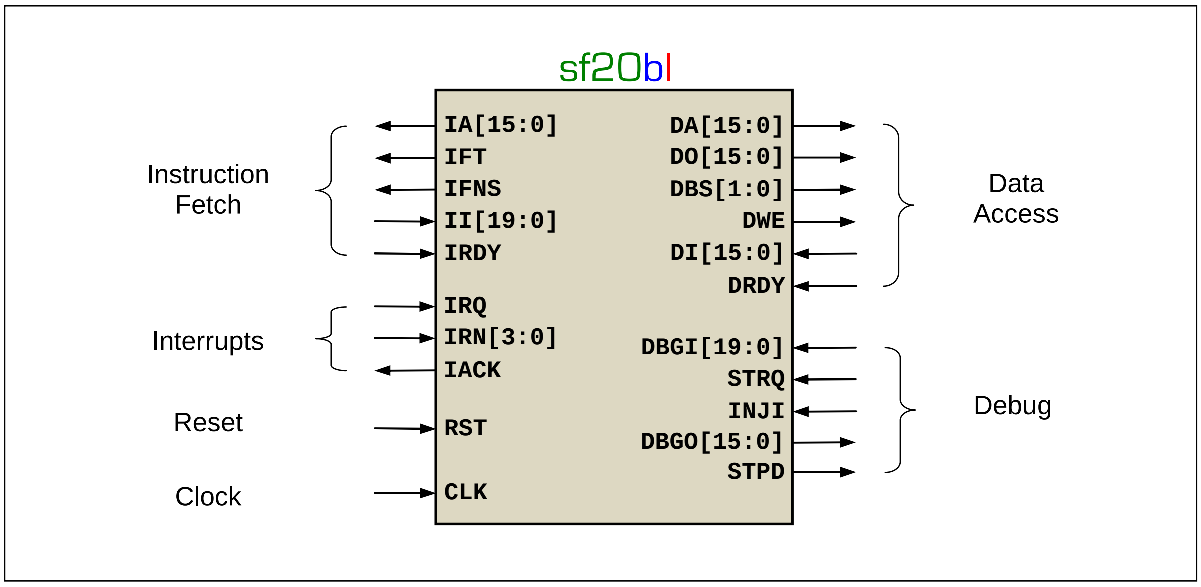
1.3 Scope of this manual

This sf20bl IMA reference manual contains the following detailed descriptions:

- **I/O Overview**, overview of interfaces and I/O signals
- **Interface Details**, detailed I/O signal descriptions and interface timing
- **Instruction Execution Timing**, effective execution time of instructions, data dependencies and stall conditions
- **Compatibility**, hardware and software compatibility, drop in replacement options

ISA specific details such as programming model and instruction set are not part of this IMA reference manual. This information can be found in the base (b) ISA (Instruction Set Architecture) reference manual.

2 I/O Overview



Signal	Direction	Width	Description
`IA[15:0]`	Output	16	Instruction Address
`IFT`	Output	1	Instruction Fetch
`IFNS`	Output	1	Instruction Fetch Non Sequential
`II[19:0]`	Input	20	Instruction In
`IRDY`	Input	1	Instruction Ready
`IRQ`	Input	1	Interrupt Request
`IRN[3:0]`	Input	4	Interrupt Number
`IACK`	Output	1	Interrupt Acknowledge
`RST`	Input	1	Reset
`CLK`	Input	1	Clock
`DA[15:0]`	Output	16	Data Address
`DO[15:0]`	Output	16	Data Out
`DBS[1:0]`	Output	2	Data Byte Strobes
`DWE`	Output	1	Data Write Enable
`DI[15:0]`	Input	16	Data In
`DRDY`	Input	1	Data Ready
`DBGI[19:0]`	Input	20	Debug In
`STRQ`	Input	1	Stop Request
`INJI`	Input	1	Inject Instruction
`DBGO[15:0]`	Output	16	Debug Out
`STPD`	Output	1	Stopped

Clocking

The sf20bl is a fully synchronous design. All flip flops are triggered with the rising edge of the **`CLK`** input. All output changes occur after the rising edge of **`CLK`**. All inputs are sampled with the rising edge of **`CLK`**.

Control signals asserted state

All control signals are active high. The asserted state is ‘1’ and the de-asserted state is ‘0’. The following signals are affected: **`IFT`**, **`IFNS`**, **`IRDY`**, **`IRQ`**, **`IACK`**, **`RST`**, **`DBS[1:0]`**, **`DWE`**, **`DRDY`**, **`STRQ`**, **`INJI`**, **`STPD.`**

Debug Interface

If the debug interface is not used inputs **`STRQ`**, **`INJI`** and **`DBGI[19:0]`** should be connected to ground.

3 Interface Details

3.1 Instruction Fetch

Signals

`IA[15:0]`	Instruction Address (output); When **`IFT`** is asserted **`IA[15:0]`** is the address of the 20-bit instruction word to fetch. When **`IFT`** is de-asserted **`IA[15:0]`** is don't care.
`IFT`	Instruction Fetch (output); **`IFT`** is the main control signal of the instruction fetch interface. When **`IFT`** is asserted outputs **`IA[15:0]`** and **`IFNS`** are valid. When **`IFT`** is de-asserted these outputs are don't care.
`IFNS`	Instruction Fetch Non Sequential (output); When **`IFT`** is asserted **`IFNS`** indicates if the fetch is sequential (**`IA[15:0]`** = address of the preceding fetch + 1) or not (any address due to a change in program flow). When **`IFT`** is de-asserted **`IFNS`** is don't care.
`II[19:0]`	Instruction In (input); When **`IRDY`** is asserted **`II[19:0]`** must be a valid instruction word. When **`IRDY`** is de-asserted **`II[19:0]`** is ignored.
`IRDY`	Instruction Ready (input); **`IRDY`** is the acknowledge handshake signal following **`IFT`** instruction fetch requests. **`IRDY`** must be asserted only as a response to an **`IFT`** request. For zero wait state instruction fetches **`IRDY`** must be asserted in the cycle following an **`IFT`** request. Wait states are inserted by delaying the assertion of **`IRDY`** by the required #of clock cycles.

General Rules

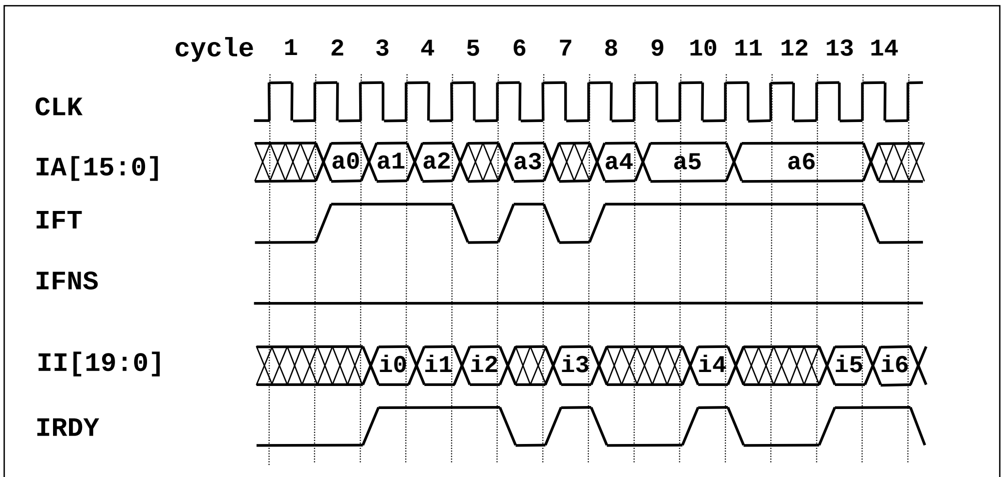
The sf20bl instruction fetch timing is designed for direct connection of synchronous memories. The following rules apply:

- Based on a handshake with **`IFT`** as request and **`IRDY`** as acknowledge
- For zero wait states fetches **`II[19:0]`** must be provided and **`IRDY`** must be asserted in the next cycle following an **`IFT`** request.
- If **`II[19:0]`** is not ready in the next cycle following **`IFT`** an arbitrary number of wait cycles can be inserted by delaying the assertion of **`IRDY`** until **`II[19:0]`** is ready.
- **`IFT`** asserted with **`IFNS`** de-asserted indicates sequential fetches. The address **`IA[15:0]`** is the address of the preceding fetch + 1.
- **`IFT`** and **`IFNS`** both asserted indicate non-sequential fetches. **`IA[15:0]`** can have any value with no relation to the preceding fetch. A preceding fetch not yet completed is aborted. The next **`IRDY`** and related **`II[19:0]`** are interpreted as response to the non-sequential fetch.

Sequential Fetches

The figure below shows sequential instruction fetches in application mode with **`IFNS`** de-asserted. There are gaps with no instruction fetches in cycles 1, 5 and 7. Although the sf20bl pipeline architecture is designed for one instruction per clock throughput instruction fetching gaps can occur as a result of pipeline stalls or execution of multi-cycle instructions, e.g. load/store multiple registers instructions. The processor fetches sequential instructions only when there is space available in its pre-fetch buffers. The pre-fetch buffer concept makes sure that the processor never discards and re-reads sequential instruction words independent of instruction execution times and pipeline stalls.

The fetches in cycles 2, 3, 4, and 6 are done with zero wait states. Instruction words **i0**, **i1**, **i2** and **i3** read from addresses **a0**, **a1**, **a2** and **a3** are provided in the next cycle following the fetch and **`IRDY`** is asserted. Fetching of instruction words **i4** and **i5** from addresses **a4** and **a5** in cycles 8 and 9 is done with 1 and 2 wait states respectively. Fetching of **i6** from **a6** is done with zero wait states again.



Sequential instruction fetches with and without wait states

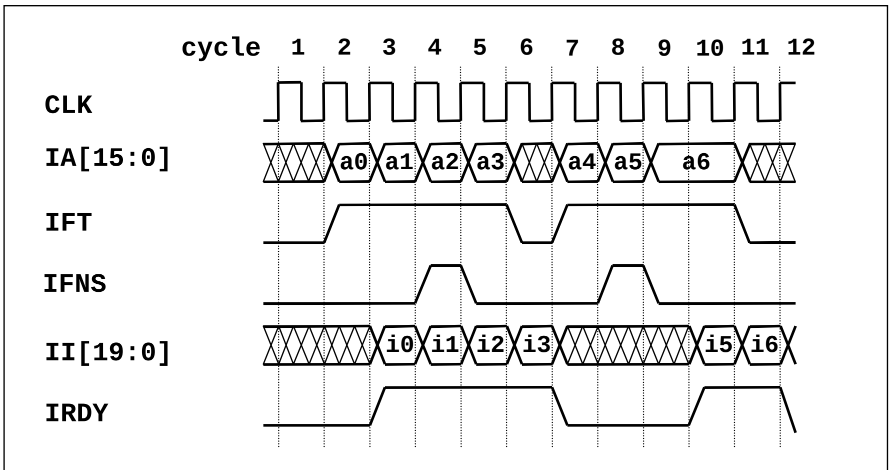
The fetches with wait states show an important behavior of the sf20bl's pipelined instruction interface. With no pending fetch (waiting for **IRDY** of the preceding fetch) **IFT** is asserted with **IA[15:0]** and **IFNS** valid for only one cycle. The fetch from **a4** in cycle 8 of the diagram illustrates the behavior. In cycle 9 the next fetch from **a5** is driven on the interface. Because the fetch from **a4** is not acknowledged yet in cycle 9 the fetch from **a5** remains stable on the interface. This means that bus logic that inserts wait states, e.g. to let another client access the instruction memory must latch the instruction address and control signals. E.g. if in the example shown below the bus controller grants access to the instruction memory to another client in cycle 8 and then reads from **a4** in cycle 9 to have **i4** ready in cycle 10 the address **a4** and corresponding control signals must be latched in registers because they are not available anymore at the interface in cycle 9.

Non Sequential Fetches

Non sequential instruction fetches occur as a result of program flow changes (jump, branch, return or interrupt). The processor flushes the instruction pre-fetch buffer and does not wait for **IRDY** of a pending fetch. If **IRDY** is asserted in the same cycle the corresponding instruction word **II[19:0]** is ignored. A pending instruction fetch that has not been acknowledged yet when a non-sequential fetch occurs is aborted. This means that the first **IRDY** following a cycle with **IFNS** asserted is always interpreted as acknowledge of the non-sequential fetch.

The next figure shows some example non-sequential fetch timings. The first example is in the middle of a fetch sequence with no wait states. In cycle 4 **IFNS** indicates a non-sequential fetch from **a2**. Instruction word **i1** from the fetch in cycle 3 is discarded. The second example shows a case where a pending fetch is aborted. The non-sequential fetch from **a5** is started in cycle 8. Due to wait states the preceding fetch from **a4** is not completed yet. The instruction bus logic aborts this fetch and reads directly from **a5**. Instruction word **i5** is delivered (in the example with one wait state) in cycle 10.

In systems with no instruction fetch wait states, e.g. with a synchronous instruction memory directly connected output **IFNS** can be ignored. In systems with wait states e.g. with an instruction cache or with debug access to the instruction memory **IFNS** must be used to abort pending fetches.



Non sequential instruction fetches

3.2 Data Access

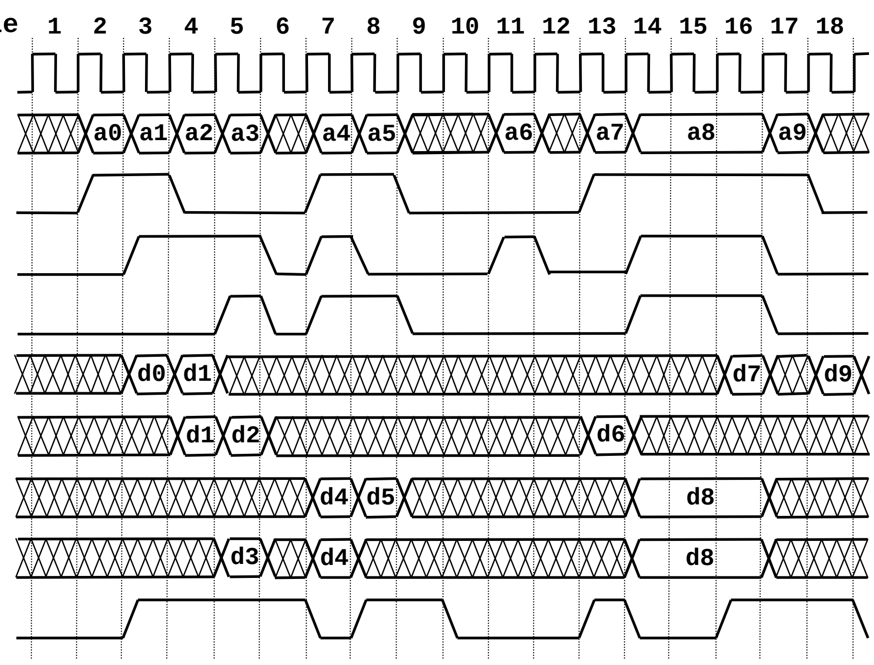
Signals

DA[15:0]	Data Address (output); when **DBS[1:0]** is asserted (!=0) **DA[15:0]** is the byte address of the data access. When **DBS[1:0]** is de-asserted (==0) **DA[15:0]** is don't care.
DO[15:0]	Data Out (output); for write accesses (**DBS[1:0]** != 0 and **DWE** asserted) **DO[7:0]** provides the low byte data if **DBS[0]** is asserted and **DO[15:8]** provides the high byte data if **DBS[1]** is asserted; when **DBS[0]** or **DWE** are de-asserted **DO[7:0]** is don't care; when **DBS[1]** or **DWE** are de-asserted **DO[15:8]** is don't care
DI[15:0]	Data In (input); when **DRDY** is asserted as response to a read access low byte input data is expected at **DI[7:0]** if **DBS[0]** was asserted and high byte data is expected at **DI[15:8]** if **DBS[1]** was asserted; data at **DI[15:0]** is ignored when **DRDY** is de-asserted and when for a read access **DBS[0]** was de-asserted (high byte read, **DO[7:0]** ignored) or **DBS[1]** was de-asserted (low-byte read, **DO[15:8]** ignored)
DBS[1:0]	Byte Strobes (output); **DBS[1:0]** is the main control signal of the data access interface. When **DBS[1:0]** is asserted (!=0) outputs **DA[15:0]** and **DWE** are valid. With **DWE** asserted **DO[15:0]** provides the low and/or high byte data. When **DBS[1:0]** is de-asserted these outputs are don't care. **DBS[1:0]** also indicates if a data access is a low-byte only access (**DBS[0]** asserted, **DBS[1]** de-asserted), a high-byte only access (**DBS[0]** de-asserted, **DBS[1]** asserted) or a 16-bit access (**DBS[0]** and **DBS[1]** both asserted)
DWE	Data Write Enable (output); When **DSB[1:0]** is asserted (!=0) **DWE** indicates if the data access is a read (**DWE**=0) or write (**DWE**=1); When **DSB[1:0]** is de-asserted (==0) **DWE** is don't care
DRDY	Data Ready (input); **DRDY** is the acknowledge handshake signal following **DBS[1:0]** != 0 data access requests. **DRDY** must be asserted only as a response to a **DBS[1:0]** != 0 request. For zero wait state data accesses **DRDY** must be asserted in the cycle following a **DBS[1:0]** request. Wait states are inserted by delaying the assertion of **DRDY** by the required #of clock cycles.

General Rules

As with the instruction interface the sf20bl data interface is designed for direct connection of synchronous memories. The following rules apply:

- Based on a handshake with **DBS[1:0]** as request and **DRDY** as acknowledge
- For zero wait access **DRDY** must be asserted in the next cycle following a **DBS[1:0]** request; for read accesses data must be provided at **DI[15:0]** in that cycle.
- If an access can't be serviced with zero wait states an arbitrary number of wait cycles can be inserted by delaying the assertion of **DRDY**.



Data access timing

Timing

The diagram above shows the sf20bl data access timing. Cycles 2 to 8 are zero wait state accesses of different types (size, read/write). Cycles 11 to 18 are data accesses with and without wait states and show an effect that are important to keep in mind when designing data bus logic for the sf20bl. As with the instruction fetch interface with no pending transaction output signals are driven for only one cycle. E.g. the 8-bit read from **a6** in cycle 11 is completed with one wait state in cycle 13, but the data bus output signals are de-asserted in cycle 12 (before the transaction is completed). Another exampl is the 8-bit read from address **a7** in cycle 13. The processor then starts a 16-bit write to **a8** in cycle 14. Because the read from cycle 13 is not completed yet the output signals (in this case **`DA[15:0]`**, **`DBS[1:0]`**and **`DO[15:0]`**) are extended until the preceding access is completed in cycle 16. In systems with data access wait states the bus logic must latch the output signals to be able to perform the access later when these signals are no longer valid.

3.3 Interrupts

Signals

`IRQ`	Interrupt Request (input); **`IRQ`** asserted signals an interrupt request with number **`IRN[3:0]`** to the processor
`IRN[3:0]`	Interrupt Number (input); when **`IRQ`** is asserted **`IRN[3:0]`** is the number of the requested interrupt; when **`IRQ`** is de-asserted **`IRN[3:0]`** is ignored
`IACK`	Interrupt Acknowledge (output); **`IACK`** is asserted for one cycle when the processor has latched **`IRN[3:0]`** and starts interrupt execution.

The interrupt concept of the sf20bl is designed for maximum performance. When a new interrupt request is acknowledged the processor does not flush the pipeline. Fetching of the interrupt vector from the data address space is done one the fly. As soon as the interrupt vector has been received instruction fetching is diverted to the interrupt start address. In ideal cases with no bus wait states and dependency stalls the only execution time overhead is the execution of the **`rtir`** instruction at the end of the interrupt service routine.

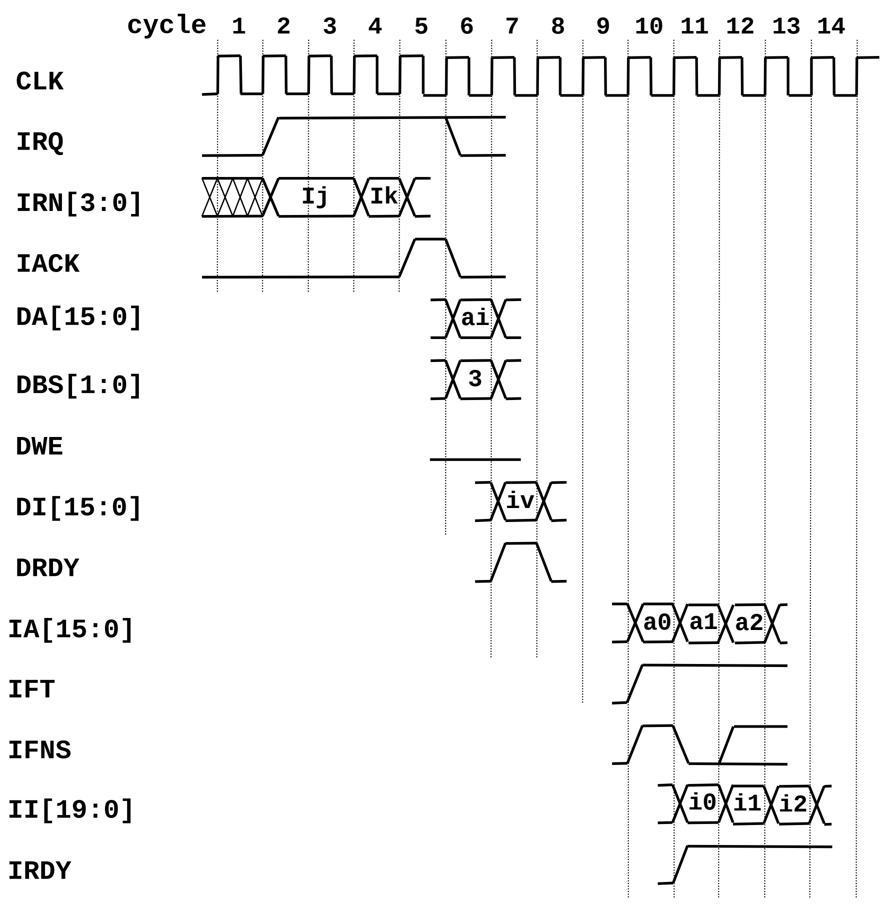
General Rules

- Interrupts are acknowledged and executed only if enabled (see ISA reference manual)
- The interrupt number **`IRN[3:0]`** may be changed from cycle to cycle at any time also while **`IRQ`** is asserted. When **`IACK`** is asserted the **`IRN[3:0]`** of the preceding cycle has been latched and the corresponding service routine will be executed.
- Simple interrupt controllers with no request queuing can ignore the **`IACK`** signal

Timing

The following diagram is an example interrupt timing of a sf2Obl system. Interrupt processing affects also signals of the instruction fetch and data access interfaces. To keep the diagram simple and clear only sections that are relevant for interrupt processing are shown for each signal and all instruction and data accesses are completed with zero wait states.

The sequence starts in cycle 2 with the assertion of **`IRQ`** and interrupt number **Ij** on **`IRN[3:0]`**. In most cases if the processor is not already executing another interrupt **`IACK`** is asserted in the next cycle following **`IRQ`**. In the example **`IACK`** is asserted later in cycle 5 to demonstrate that **`IRN[3:0]`** is allowed to change while **`IRQ`** is asserted. **`IACK`** asserted in cycle 5 indicates that the **`IRN[3:0]`** value **Ik** of cycle 4 has been latched inside the processor and is the interrupt number that will be processed. Output **`IFNS`** is not shown in cycles 1 to 6 because it is not relevant if the last instruction fetches before an interrupt is started are sequential or non-sequential.



Interrupt Timing

In the cycle following the **`IACK`** pulse a 16-bit data read (**`DBS[1:0] = 3`**, **`DWE = 0`**) from address **ai** fetches the interrupt vector (start address).

With no wait states the data access is completed in cycle 7 with **`DRDY`** asserted and the instruction vector **iv** available at **`DI[15:0]`**. Three cycles later in cycle 10 fetching of instructions of the interrupt service routine starts. The first fetch from **a0** (**a0** = **iv**) is non-sequential and **`IFNS`** is asserted.

3.4 Debug

Signals

DBGI[19:0]	Debug In (input); this port is used to inject instructions into the processor and to provide input data for the **mfdp** (move from debug port) and **rspc** (restore PC) instructions; when **INJI** is asserted **DBGI[19:0]** is interpreted as 20-bit opcode of the instruction to be injected; when a **mfdp** or **rspc** instruction is injected source data must be provided at **DBGI[15:0]** from the cycle following the assertion of **INJI**.
STRQ	Stop Request (input); the debug module asserts this signal to bring the processor into the debug state. The processor stops fetching new instructions and flushes its pipeline (executes all pending instructions and instructions in the pre-fetch buffer). As long as **STRQ** remains asserted the processor is held in the debug state; when **STRQ** is released the processor resumes normal operation.
INJI	Inject Instruction (input); when the processor is in the stopped state (**STPD** asserted) the debug module asserts **INJI** for one clock cycle to inject and execute individual instructions; in the cycle where **INJI** is asserted the opcode of the injected instruction must be provided at **DBGI[19:0]**; when the processor is not in the stopped state **INJI** is ignored.
DBGO[15:0]	Debug Out (output); when in the stopped state a **mtdp** (move to debug port) or **svpc** (save PC) instruction is injected and executed destination data is provided at **DBGO[15:0]**.
STPD	Stopped (output); **STPD** asserted indicates that the processor is in the stopped state. The processor enters the stopped state after flushing its pipeline either when **STRQ** is asserted by the debug module or when a **stop** instruction is executed.

General Rules

- To use the sf20bl debug features a separate debug module is required that connects to the processor's debug interface and the debug Host PC. If debug functionality is not required the input signals of the debug port **DBGI[19:0]**, **STRQ** and **INJI** should be tied to GND.
- Injecting and executing instructions via the debug port is possible only when the processor is in the stopped state indicated by the **STPD** output signal.
- The stopped state is entered from normal operation either by asserting the **STRQ** input or by executing a **stop** instruction.
- To resume normal operation when the stopped state has been entered by **STRQ** assertion **STRQ** must be de-asserted.
- To resume normal operation when the stopped state has been entered by executing a **stop** instruction **STRQ** must be asserted and then de-asserted.
- When data is input via **DBGI[19:0]**, the data value must be driven on **DBGI[15:0]**, bits **DBGI[19:16]**are ignored.

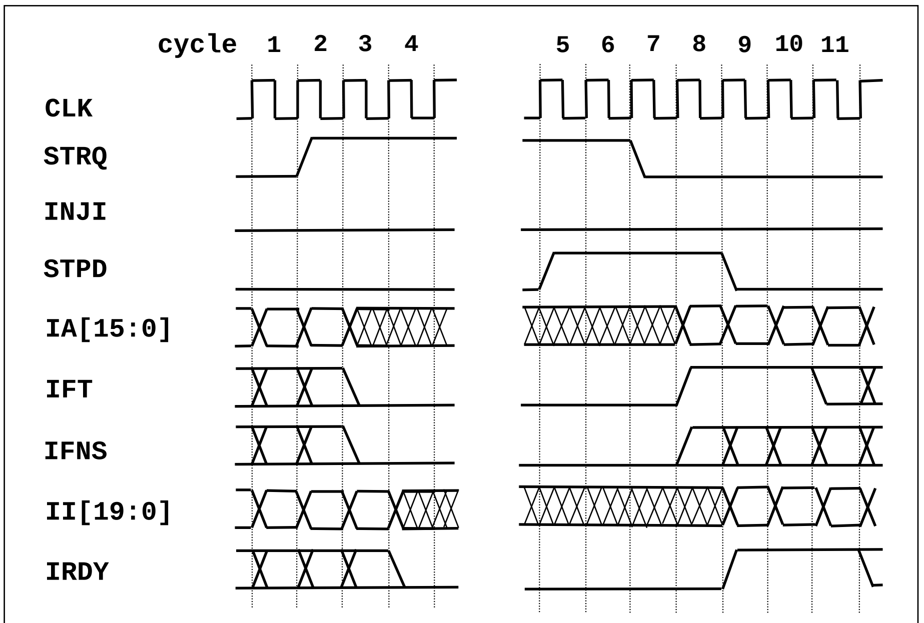
Timing

The first diagram following shows the interface timing at the beginning and end of the stopped state. The signals of the debug and instruction fetch interfaces are shown. In cycle 2 **STRQ** is asserted. Starting with the next cycle the processor stops fetching new instructions. Pending instructions and instructions in the pre-fetch buffer are executed until the pipeline is completely empty. When the pipeline is empty (6-10 cycles with no wait states) the **STPD** output is asserted indicating that the stopped state has been reached.

The stopped state can also be entered by executing a **stop** instruction during normal operation. When a **stop** instruction is executed remaining instructions in the pre-fetch buffer are discarded and the processor asserts the **STPD** output and enters the stopped state when the execution pipeline has been flushed.

In the diagram **STRQ** is de-asserted again in cycle 7 only 2 cycles after the stopped state has been entered. Normally this would not make much sense but the purpose of this diagram is to illustrate the timing only at the beginning and end of the stopped state.

In the next cycle after **STRQ** has been de-asserted the processor starts fetching instructions again. One cycle later in cycle 9 the **STPD** output is de-asserted indicating that the processor has resumed normal operation.



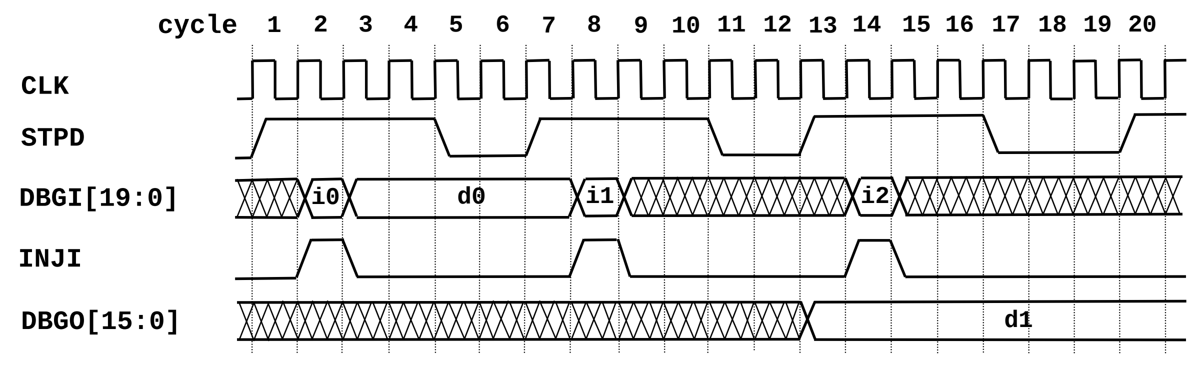
Entering and leaving the stopped state

The second diagram shows the timing of instruction injection and data I/O via the debug port while the processor is in the stopped state. The **STRQ** signal is not shown because it is not relevant if the stopped state has been entered due to **STRQ** assertion or after the execution of a **stop** instruction. Regarding interface timing there are three types of instruction injection:

1. Injection with data input from **DGBI[15:0]**; only the dedicated debug instructions **mfdp** and **rspc** take a source operand from the debug port
2. Injection with data output to **DGBO[15:0]**; only the dedicated debug instructions **mtdp** and **svpc** output a destination operand to the debug port
3. Injection with no data I/O; all other instructions are of this type

The injection and execution behavior is common to all three types. The debug module asserts **INJI** for one cycle and drives the opcode of the instruction at **DBGI[19:0]**. Three cycles later the processor de-asserts **STPD** which indicates the execution of the injected instruction. **STPD** is asserted again when execution has finished and the pipeline is completely empty. The number of cycles **STPD** is de-asserted depends on the instruction. For some flow instructions like **SVPC** or **RSPC** **STPD** is de-asserted for only one cycle. For most computation instructions it is 2 or 3 cycles. For load/store instructions with zero wait states data access it is at least 4 cycles. Data access wait states add to the **STPD** de-asserted time. The debug module must wait for **STPD** being de-asserted and asserted again before it can inject the next instruction.

A type 1 example starts in cycle 2 where opcode **i0** is injected. In the following cycle when **INJI** is de-asserted the source operand **d0** is driven at **DBGI[15:0]**. It must be kept stable until **STPD** is re-asserted.



Instruction injection and execution

A type 2 example starts in cycle 8 where opcode **i1** is injected. When **STPD** is re-asserted in cycle 13 the destination operand **d1** is available at **DBGO[15:0]**. The example shows the behavior of an **mtdp** instruction. The second type 2 instruction **svpc** has different timing. **STPD** is de-asserted for only one cycle and the destination operand appears at **DBGO[15:0]** already in that cycle. The debug module should read the destination operands of type 2 instructions when **STPD** has been re-asserted. **DBGO[15:0]** is always valid then and remains stable until the next type 2 instruction is injected.

A type 3 example starts in cycle 14 where opcode **i2** is injected. **STPD** is de-asserted for three cycles which is a typical value for computation instructions.

3.5 Reset

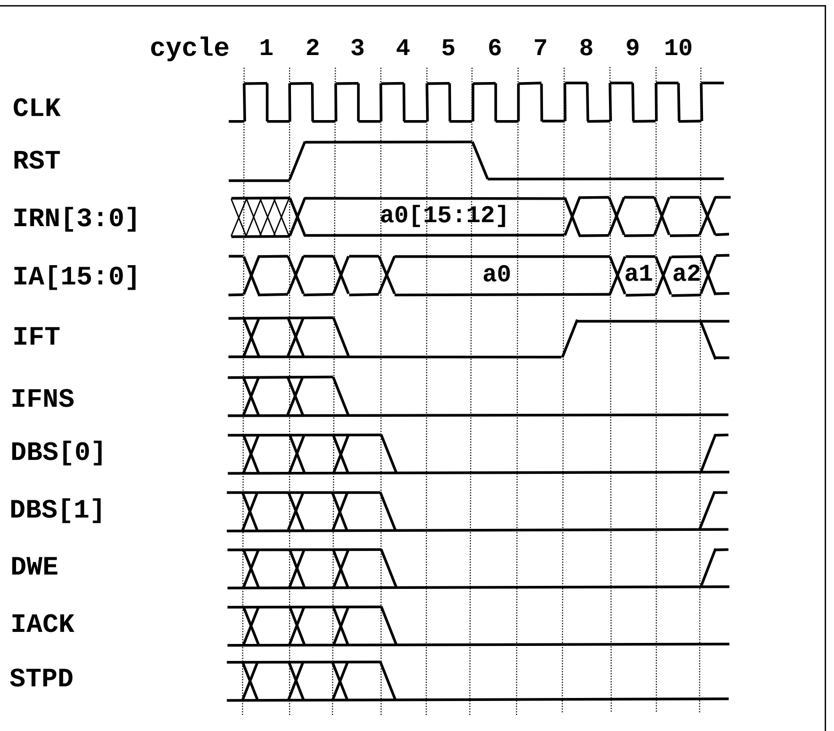
Signals

RST	Reset (input); this is a synchronous reset; when **RST** is asserted the processor is reset with the next rising edge of **CLK**.

General Rules

- **RST** can be asserted at any time. The processor does not wait for any pending interface transactions or instructions.
- **RST** needs to be asserted for only one active edge of **CLK** to fully reset the processor.

- Timing



Reset assertion and release

The diagram shows the **RST** assert and release timing. Only signals that are directly affected are shown. Output signals not shown are either undefined or keep their state. Beside **RST** the only input signal relevant for reset is **IRN[3:0]**.

In cycle 2 **RST** is asserted. In the following cycle (3) instruction fetching stops, **IFT** and **IFNS** are de-asserted. One cycle later (4) all control outputs of the processor are de-asserted. **IA[15:0]** takes the value of the reset start address **a0**: **IA[15:12]** = **IRN[3:0]** and **IA[11:0]** = 0. This state remains unchanged as long as **RST** remains asserted.

In cycle 6 **RST** is de-asserted. **IRN[3:0]** must continue to provide the upper 4 bits of the reset start address for the following 2 cycles. In cycle 8 instruction fetching starts from **a0**. **IRN[3:0]** can take any value from this point.

4 Instruction Execution Timing

4.1 Effective Execution Times

The following table provides effective execution times for all sf2Obl instructions except for the dedicated debug instructions **mtdp**, **mfdp**, **svpc**, **rspc** and **stop** which are not for use in normal program code sequences.

The numbers provided in the **Cycles** column are best case numbers assuming no stalls caused by operand dependencies or data access wait states (load/store instructions). The **Stalls** column contains abbreviations of stall conditions that are further explained in the "**Stall Conditions**" section later in this chapter.

Instructions are grouped by addressing modes and common execution time properties. Instructions with multiple addressing modes may appear in different non-consecutive places.

Instructions	Addressing Mode	Cycles	Stalls	Comment(s)
move	**C10$_S$,Rd**	1	-	-
mvsr	**C10$_S$,Rd**	1	D1	R8 is source register
mtsr	**C10,SRd**	1	-	-
addt subf addh mlcu andb iorb xorb	**C8$_U$,Rb**	1	D1	-
comp	**C10$_S$,Rs1**	1	D1	-
move negt absl inv clzr sxbt sxsh adcf sbcf	**Rs,Rd**	1	D1	-
cpcf	**Rs**	1	D1	-
mtsr	**Rs,SRd**	1	D1	-
mfsr	**SRs,Rd**	1	D1	-
btst btcl bttg	**BTI4,Rs1,Rd**	1	D1	-
btts	**BTI4,Rs1**	1	D1	-
addt subf addc subc mult mlhu mlhs	**Rs0,Rs1,Rd**	1	D1	-
andb iorb xorb				
btst btcl bttg			D1, D2	
shlz shlf shru shrs				
	SHC4,Rs1,Rd		D1	
comp cmpc	**Rs0,Rs1**	1	D1	-
stbt stsh	**Rs,DA11$_S$**	1	D1	data access wait states add to the effective execution time n = #of register in RGS
	Rs,(DO8$_S$,An)			
	Rs,(An)+			
	Rs,-(An)			
	Rs,(An)*			
	Rs,(Rx,An)		D1, D2	
	RGS,(An)+	n	D1	
	RGS,-(An)			
ldbt ldsh	**DA11$_S$,Rd**	1	-	Data access wait states add to the effective execution time n = #of registers in RGS
	(DO8$_S$,An),Rd		D1	
	(An)+,Rd			
	-(An),Rd			
	(An)*,Rd			

	(Rx,An),Rd		D1, D2	
	(An)+,RGS	n	D1	
	-(An),RGS			
stie clie scie rsie	**implied**	1	-	-
jpsr	**IA16**	2	-	-
jump jpsr	**implied (TA)**	2	D3	TA dependency
rtsr	**implied**	2	D4	SA dependency
rtir			-	-
bral	**IO14**$_S$	2	-	-
brlc	**IO10**$_S$**,S**	2	D5	branch taken
		1		branch not taken
		0		loop cache active
brxx (conditional)	**IO10**$_S$	2	D6	branch taken
		1		branch not taken

4.2 Stall Conditions

Extra cycles add to best case execution times if stall conditions caused by operand dependencies occur during instruction execution. Instructions have to wait if one or more of their source or destination operands are scheduled to be updated by a preceding instruction that has not finished execution yet. These conditions are instruction and addressing mode specific. Six conditions exist named **D1** to **D6**. Affected instruction/addressing-mode combinations have the relevant conditions listed in the **Stalls** column of the execution times table.

The following paragraphs are more detailed descriptions of individual stall conditions with hints how they can be avoided.

Computation Latency

With the sf20bl pipeline structure register destination operands of computation instructions are updated only one cycle after a directly following instruction reads its source operands. If the following instruction has the same register as source operand this instruction would have to be stalled by one cycle to wait until the source operand is ready. For most cases a forwarding mechanism is implemented that uses the ALU output as source directly and bypasses the register file to avoid stalls. Exception is the **D2** stall condition.

D1 (Source operand pending update)

The **D1** stall occurs if instructions use the destination register of a directly preceding load from memory instruction as source operand. Load from memory instructions update their destination register two cycles after an immediately following instruction reads its source operands. As with destination operands of computation instruction there is a forwarding mechanism that bypasses the register file and uses the load from memory destination operand as source one cycle before it is written to the register file. But because of two cycles latencies there is still a one-cycles stall if an instruction uses the destination register of a directly preceding load from memory instruction as source operand.

In many cases such stalls can be avoided by instruction re-ordering so that there is at least one other instructions between the load from memory and the instruction that uses the load destination as source.

D2 (Computation destination not forwarded)

As described under **Computation Latency** a forwarding mechanism bypasses the register file to avoid stalls if instructions try to read the destination register of a directly preceding computation instruction as source operand. But there are some exceptions where forwarding is not done because it would create critical timing paths and decrease the processor’s maximum clock rate. The following types of register source operands are affected and cause a one-cycle stall if the register is the destination operand of the directly preceding instruction:

- Indirect shift counts of shift instructions
- Indirect bit index of bit manipulation instructions

- Index of load/store instructions with indirect + index addressing mode

To avoid **D2** stalls instructions must be re-ordered such that the instruction that generates the register destination operand is not the directly preceding instruction.

D3 (TA pending update)

In principle this is the same as **D1** but for register **TA** (Target Address) and only for the **`jump`** and **`jpsr`** instructions with the implied addressing mode that use **TA** as source operand. There is no forwarding when **TA** is used as indirect jump address. A preceding instruction that updates **TA** with a latency > the cycle distance to the **`jump`**/**`jpsr`** with **TA** as source causes stall cycles. This is the case for load instructions with **TA** as destination operand and a cycle distance < 3 and also for directly preceding **`move`** instructions with **TA** as destination operand.

D4 (`rtsr` with SA pending update)

This is similar to **D3** but for register **SA** and only for the **`rtsr`** instruction that uses **SA** as source operand. There is no forwarding when **SA** is used as return address. A preceding instruction that updates **SA** with a latency > the cycle distance to the **`rtsr`** causes stall cycles. This is the case for load instructions with **SA** as destination operand and with a cycle distance < 3 and also for directly preceding **`mtsr`** instructions with **SA** as destination operand.

D5 (`brlc` with LC pending update)

Again this is similar to **D3-D4** but for register **LC** (Loop Counter) and for **`brlc`** instructions which use **LC** as source operand. There is no forwarding when **LC** is used as source operand of loop counter branches. A preceding instruction that updates **LC** with a latency > the cycle distance to the **`brlc`** causes stall cycles. This is the case for directly preceding **`mtsr`** instructions with **LC** as destination operand.

D6 (`brxx` with unresolved speculation)

The sf20bl implements a speculation scheme for conditional branches. Conditional branch instructions do not wait until pending **CC** updates of preceding instructions are completed. However a stall condition occurs if a new conditional branch is decoded and the speculation of a preceding branch is not yet resolved. In this case the new branch has to wait until all **CC** update operations of instructions that have been issued before the preceding branch are completed. **D6** stalls typically occur with instruction sequences where two or more conditional branches are placed close together.

4.3 Loop Cache

The sf20bl implements a single entry loop cache to accelerate the performance of simple loops. When a **`brlc`** (loop counter branch) instruction with backward branch direction is executed and the loop cache is not active yet the processor latches start-address, end-address and the opcode of the first instruction of the loop in internal registers.

From the second iteration on the last instruction (loop branch) and the first instruction of the loop are not fetched from instruction memory to eliminate the 2-cycle latency that the non-sequential instruction fetch of the loop branch would cause. Instead the loop branch is implicitly executed and the opcode of the first instruction is retrieved from the register where it has been stored when the cache became active. As a result the effective execution time of the loop branch is zero cycles.

If beside the implicit loop branch any other non-sequential instruction execution is encountered the loop cache is de-activated immediately. E.g. if a conditional branch inside the loop is taken the cache is de-activated and the next execution of the loop branch is done normally as in the first iteration but with this normal **`brlc`** execution the cache is activated again. If in the next iteration the conditional branch is not taken the cache concept will take effect and the next loop back branch will be done with zero cycles effective.

4.4 Software Controlled Branch Speculation

sf20 opcodes of conditional branch instructions contain a flag (**S** = speculation), that can be used to improve the performance of conditional branch execution. The sf20bl implementation uses the **S** flag to decide whether a conditional branch is taken or not in cases where the branch condition has not been evaluated yet by the time the branch instruction is decoded. If for a conditional branch instruction the preferred case (branch taken or not) is known at compile time or when an assembler routine is written, setting the S flag to the preferred case improves performance.

5 Compatibility

5.1 Software

The sf20bl is fully compatible with the sf20 (b) (base) ISA.

5.2 Hardware

The sf20bl interface signals and timing are the same as those of the sf20bu and sf20dl. These three processors can replace each other without changing any of the surrounding hardware.

5.3 Replacement Options

sf20bl and sf20bu are software compatible and can replace each other regarding both hardware and software.

The sf20dl can replace sf20bu and sf20bl regarding both hardware and software because it supports the sf20 (b) (base) ISA.