sf16bu

16-bit microprocessor

IMA (Implementation Architecture) Reference Manual

Revision 0.9
24 December 2013

Author: Martin Raubuch

Revision History

Revision	Date	
0.9	24Dec2013	First version

Table of contents

1 Overview

1.1 Introduction

The sf16 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 fixed length 16-bit instruction coding the architectural focus is on high clock rates and small core implementations.

The sf16 family 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 sf16bu, the (u) ultra light implementation of the sf16 (b) base ISA.

1.2 Feature Summary

The following list summarizes the sf16bu's main features

- Focused on small core size
- 16-bit wide instruction and data interfaces
- Register file with 1/1 read/write ports and one cycle read-latency, can be implemented as RAM
- Single cycle effective execution of computation instructions with one register source operand
- Two cycles effective execution of computation instructions with two register source operands
- Two cycles effective execution of most load and store instructions
- Iterative shift execution with one bit per cycle
- Average IPC (Instructions Per Cycle) of 0.5 for typical code sequences
- 2 x 16-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 130MHz on low end FPGAs
- Clock rates >300MHz with deep sub-micron std-cell generic and low-power technologies

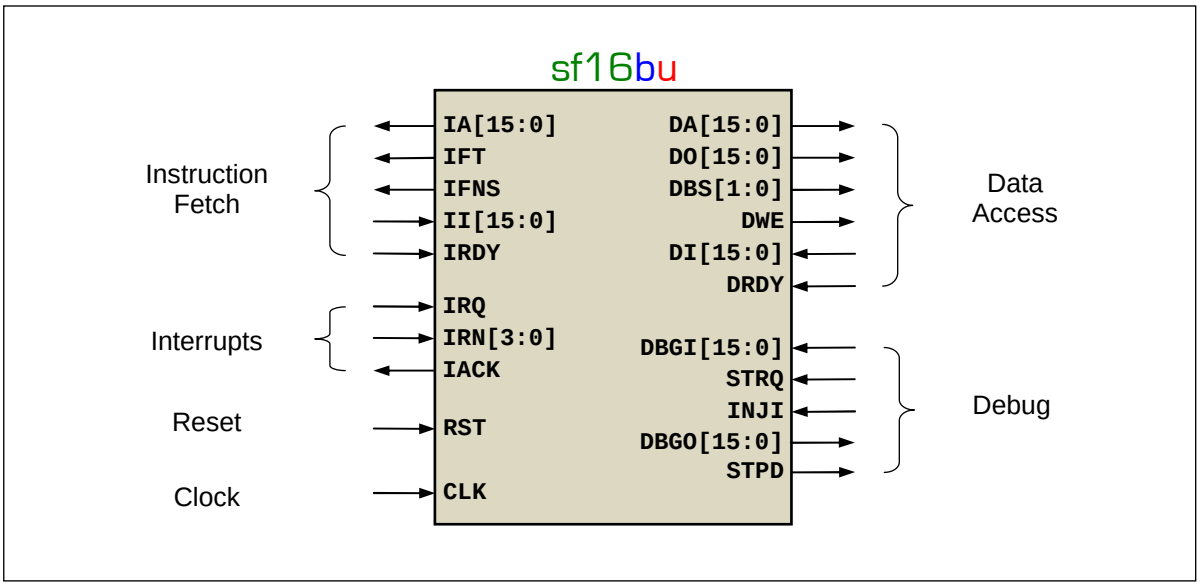
1.3 Scope of this manual

This sf16bu 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[15:0]	Input	16	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[15:0]	Input	16	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 sf16bu 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**. To enable very small core sizes especially on FPGAs the register file can be implemented as synchronous RAM with one cycle read latency, driven by **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[15:0]** 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 16-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[15:0]	Instruction In (input); When **IRDY** is asserted **II[15:0]** must be a valid instruction word. When **IRDY** is de-asserted **II[15:0]** 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 sf16bu instruction fetch timing is designed for direct connection of synchronous memories. The following rules apply:

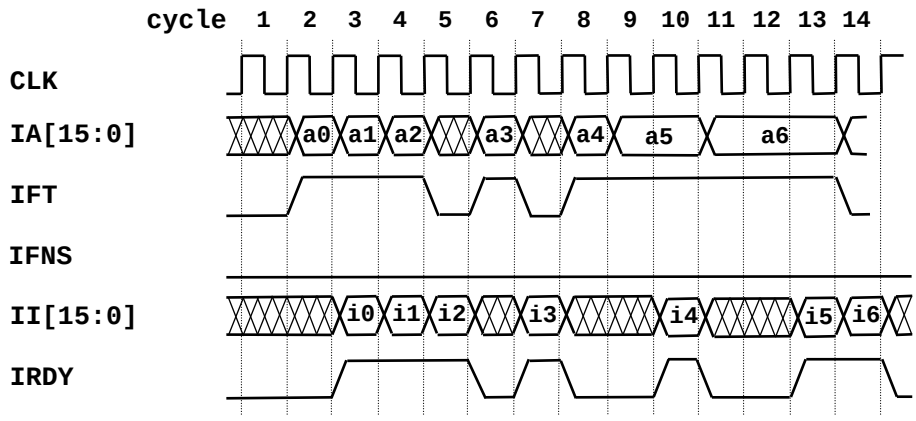
- Based on a handshake with **IFT** as request and **IRDY** as acknowledge
- For zero wait states fetches **II[15:0]** must be provided and **IRDY** must be asserted in the next cycle following an **IFT** request.
- If **II[15:0]** is not ready in the next cycle following **IFT** an arbitrary number of wait cycles can be inserted by delaying the assertion of **IRDY** until **II[15:0]** 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[15:0]** are interpreted as response to the non-sequential fetch.

Sequential Fetches

The figure below shows sequential instruction fetches with **IFNS** de-asserted. There are gaps with no instruction fetches in cycles 1, 5 and 7. This is because typical sf16bu execution rates are < 1 per cycle and 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.

The fetches with wait states show an important behavior of the sf16bu'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. 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** must be latched in a register because it is not available anymore at the interface in cycle 9.

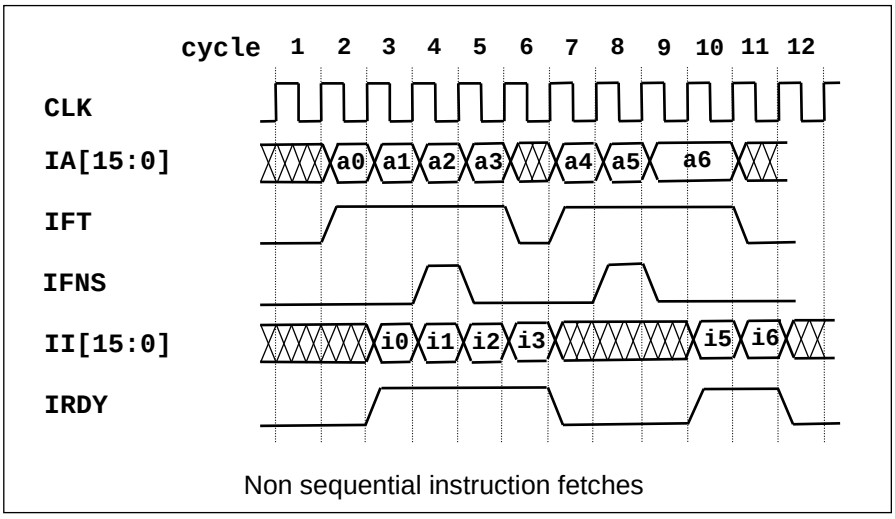


Sequential instruction fetches with and without wait states

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 preceding fetch. If **IRDY** is asserted in the same cycle the corresponding instruction word **II[15:0]** 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 preceding 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.



Non sequential instruction fetches

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.

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

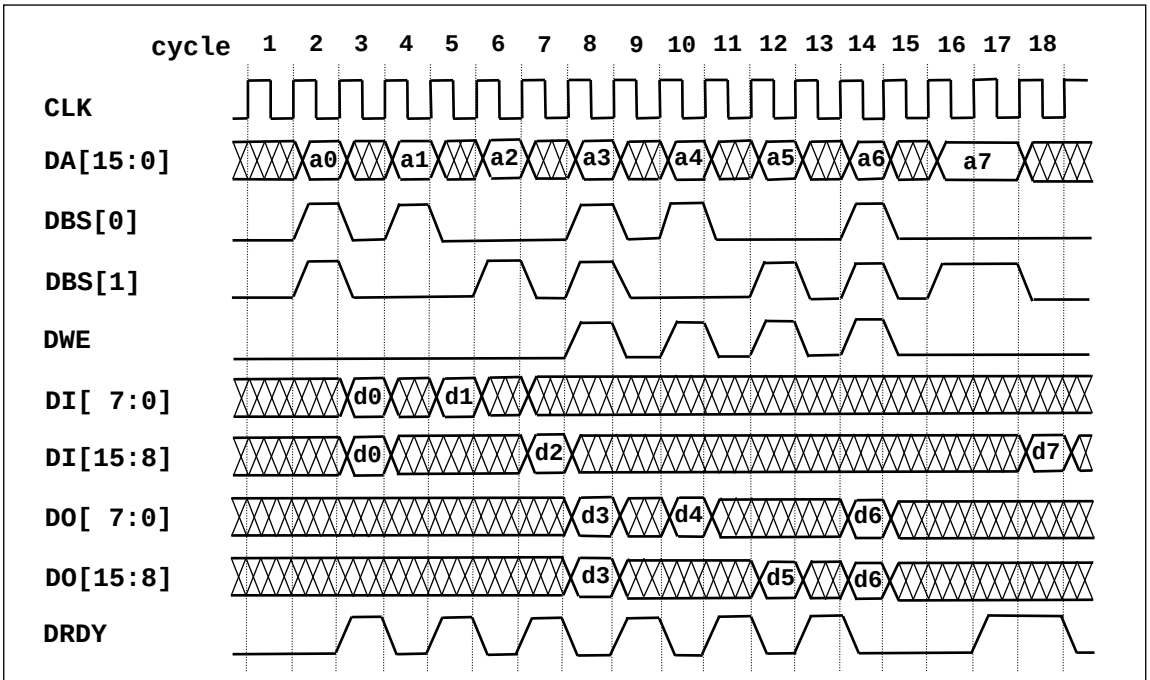
As with the instruction interface the sf16bu data interface is designed for direct connection of synchronous memories. A specialty of the (u) ultra light implementation is that ALU and load/store hardware resources are shared. As a result the sf16bu can perform data accesses only in every second cycle. 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**.
- The maximum access rate is one access every two cycles. But the interface is still pipelined. In case of wait states two accesses can be completed in two consecutive cycles.

Timing

The following diagram shows the sf16bu data access timing. Cycles 2 to 13 are zero wait state accesses of all possible types regarding read/write and data size (low-byte, high-byte or 16-bit). They show that the active part of the data in/out interfaces depends on **DBS[1:0]**. For write accesses the active part of **DO[15:0]** depends on the state of **DBS[1:0]** in the same cycle. For read accesses the active part of **DI[15:0]** in the cycle where **DRDY** is asserted depends on the state of **DBS[1:0]** in the corresponding request cycle.

Cycles 14 to 18 are data accesses with wait states and show two effects that are important to keep in mind when designing data bus control logic for the sf16bu. As with the instruction fetch interface with no pending transaction output signals are driven for only one cycle. Example is the 16-bit write of data **d6** to address **a6** in cycle 14. The processor then starts a high byte read from **a7** in cycle 16. Because the write from cycle 14 is not completed yet the output signals (in this case **DA[15:0]** and **DBS[1]**) are extended until the preceding access is completed in cycle 17. In systems with data access wait states the bus control logic must latch the output signals to be able to perform the access later when these signals are no longer valid.



Data access timing

The second effect is the bus pipelining. Although the maximum access rate is one access every two cycles the bus is still pipelined as shown in cycles 17 and 18 where two accesses are completed in two consecutive cycles.

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.

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

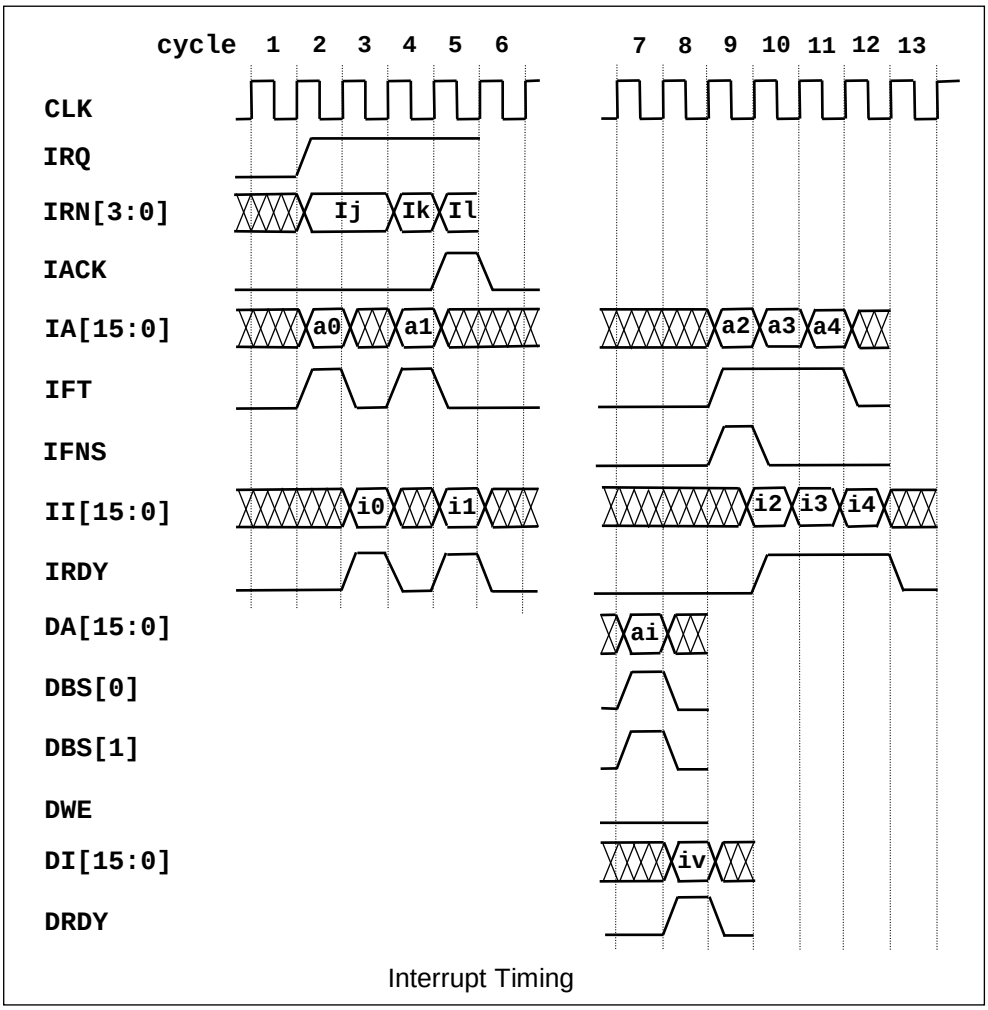
The following diagram is an example interrupt timing of a sf16bu system. Interrupt processing affects also signals of the instruction fetch and data access interfaces. To keep the diagram simple and clear only sections that are relevant for interrupt processing are shown for each signal and all instruction 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 means 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 instructions fetches before an interrupt is started are sequential or non-sequential.

Starting with the cycle where **IACK** is asserted instruction fetching stops and the processor waits until all pre-fetched instructions have been executed and the pipeline is completely empty. The number of cycles required to complete this phase is not deterministic and depends on the following:

- Number of instructions in the pre-fetch buffer
- A possible pending instruction fetch (one more instruction to execute)
- Wait states of a pending instruction fetch
- Data dependencies and associated pipeline stalls caused by the instructions to execute
- Data access wait states in case there are load/store instructions to execute

With zero wait state instruction fetches and data accesses typical times to flush the pipeline are 6-10 cycles.



Interrupt Timing

When the pipeline has been flushed the start address of the interrupt service routine is read from the interrupt vector table in data memory. **DBS[0]** and **DBS[1]** both asserted in cycle 7 (16-bit data access) with **DWE** = 0 (read) indicate the reading of the interrupt vector from address **ai**. With no wait states the data access is completed in cycle 8 with **DRDY** asserted and the instruction vector **iv** available at **DI[15:0]**.

In cycle 9 fetching of instructions of the interrupt service routine starts. The first fetch from **a2** (**a2** = **iv**) is non-sequential and **IFNS** is asserted. Because the pre-fetch buffers and execution pipeline are completely empty there are at least three consecutive instruction fetches as shown in the diagram.

3.4 Debug

Signals

DBGI[15:0] 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[15:0]** is interpreted as 16-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[15:0]**; 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 sf16bu 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[15:0]**, **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.

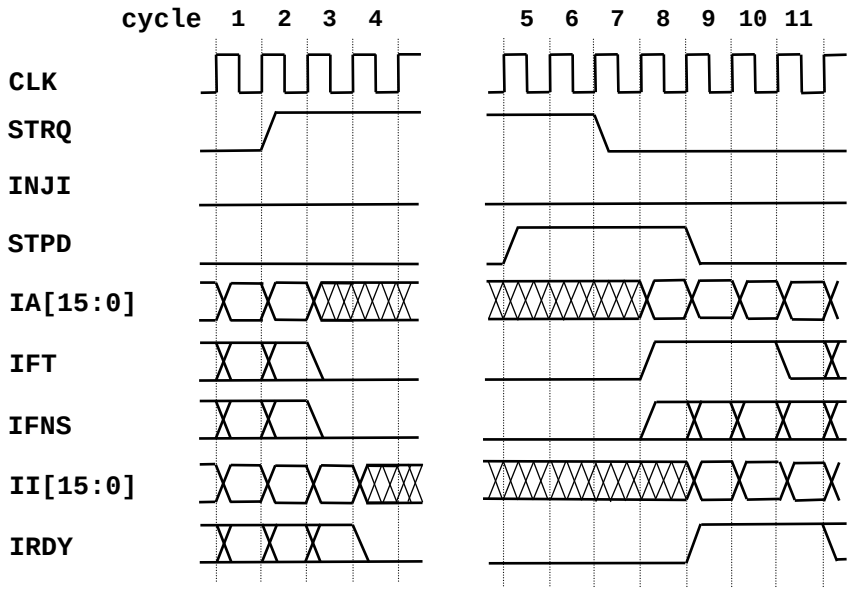
Timing

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 same as after an interrupt acknowledge. 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.



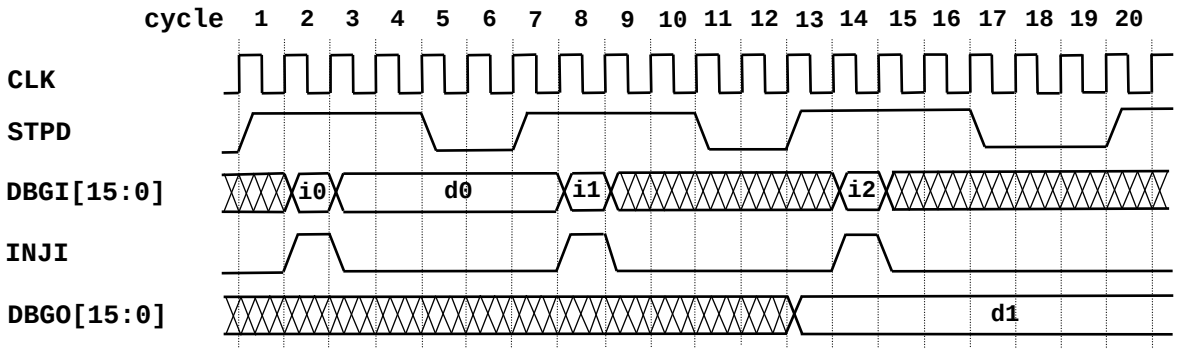
Entering and leaving the stopped state

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[15:0]**. 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

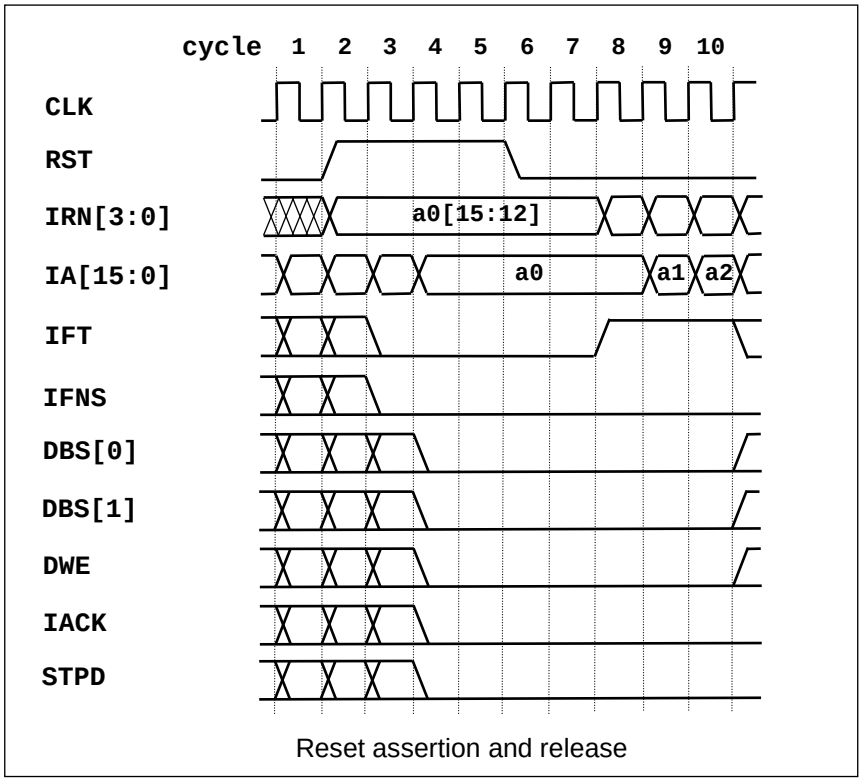
Signals

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 sf16bu 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	**C9$_S$,Rd**	1	D2	-
mtsr	**C7,SRLd**	1	D2	-
adsp	**C7$_S$,Ad**	1	D2, D4	-
addt subf addh andb iorb	**C8$_U$,Rb**	1	D1, D2	-
comp	**C8$_A$,Rs1**	1	D1	-
move negt absl inv clzr sxbt sxsh adcf sbcf	**Rs,Rd**	1	D1, D2	-
cpcf	**Rs**	1	D1	-
mtsr	**Rs,SRd**	1	D1, D2	-
mfsr	**SRs,Rd**	1	D1, D2	-
btst btcl bttg	**SHC4,Rb**	1	D1, D2	-
btts	**SHC4,Rs**	1	D1	-
addt subf addc subc andb iorb xorb mult mlhu mlhs	**Rs0,Rs1,Rd**	2	D1, D2	2 register read cycles
btst btcl bttg			D1, D2, D3	
shlz shlf shru shrs		3 + n	D1, D2, D3	n = shift-count (see note after table)
	SHC4,Rb	1 + n	D1, D2	
comp cmpc	**Rs0,Rs1**	2	D1	2 register read cycles
stbt stsh	**Rs,DA8$_U$**	2	D1	data access wait states add to the effective execution time n = #of register in RGS
	Rs,(DO5$_S$,An)			
	Rs,(An)+			
	Rs,-(An)			
	Rs,(An)*	3		
	Rs,(Rx,An)		D1, D3	
	RGS,-(An)	2*n	D1	
ldbt ldsh	**DA8$_U$,Rd**	2	-	Data access wait states add to the effective execution time n = #of registers in RGS
	(DO5$_S$,An),Rd		D1	
	(An)+,Rd			
	-(An),Rd			
	(An)*,Rd	3		
	(Rx,An),Rd		D1, D3	

	(An),RGS	2*n	D1	
siah	**IAH4**	1	-	-
stie clie scie rsie stas clas	**implied**	1	-	-
jump jpsr	**IA12**	2	-	-
jump jpsr	**implied (TA)**	3	D5	TA dependency
rtsr	**implied**	3	D6	SP dependency
rtir			-	-
bral	**IO8$_S$**	2	-	-
brlc	**IO8$_S$**	2	D7	branch taken
		1		branch not taken
brxx (conditional)	**IO8$_S$**	2	D8	branch taken
		1		branch not taken

Shift Instructions

Two special cases exist regarding effective execution times of shift instructions:

1. With shift count n=1 the execution time is 1 with the **SHC4,Rb** addressing mode (instead of 1+n = 2) and is 3 with the **Rs0,Rb** addressing mode (instead of 3+n = 4)
2. When during iterative shifting the operand becomes zero the instruction finishes immediately with zero as destination operand regardless of the remaining shift count.

4.2 Stall Conditions

Extra cycles add to best case execution times if stall conditions occur during instruction execution. Two types of stall conditions exist for the sf16bu:

1. Resource constraints; a pipeline architecture performs multiple actions on multiple instructions in the various pipeline stages simultaneously and it can happen that more than one stage tries to use the same hardware resource. In such cases one stage takes priority and the other has to wait. There is only one condition of this type named **R1**.
2. Operand dependencies; instructions have to wait if one or more of their source or destination operands are scheduled to be updated by a preceding instructions that has not finished execution yet. These conditions are instruction and addressing mode specific. Eight conditions exist named **D1** to **D8**. 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.

R1 (Register write)

The sf16bu register file has a single write port accessed by the ALU output and by loads from data memory. Because load instructions have two more pipeline stages compared to computation instructions it can happen that a computation instruction following a load instruction tries to write its destination operand in the same cycle as the load. There are two cases with different solutions if this conflict occurs:

1. The destination registers are different; the load instruction takes priority because it was first in sequence and the computation instruction has to wait (is stalled).
2. The destination register is the same; the computation instruction takes priority because it is second in sequence and overwrites the destination of the load. This case is connected to the **D2** condition.

This condition is caused by the RAM implementation of the register file and is difficult to avoid. The following can be done to reduce the statistical probability:

- Avoid computation instructions as the second instruction following a load instruction
- Group loads from memory together as much as possible; means avoid frequent switching between load and computation instructions.

Computation Latency

With the sf16bu 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 **D3** 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 cycle 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 instruction between the load from memory and the instruction that uses the load destination as source.

D2 (Destination operand pending update)

This affects all instructions with register destination operands. In the cycle where computation instructions update register destination operands the register destination operand of a directly preceding load from memory instruction has not been updated yet. If both instructions have the same destination register the computation instruction has to be stalled until the load from memory has updated its destination. In fact the sf16bu detects this case and stalls the computation instruction until the load tries to write the destination. But then in this cycle the computation takes priority and the operand from memory is never written (see also **R1** condition).

In closed software functions this case should never occur because it doesn't make sense to load a value from memory into a register and then overwrite the register with the next instruction without using it as source operand. It's necessary anyway to detect and implement this stall condition. It can occur e.g. at the end of interrupts if the interrupt service routine restores registers from the stack (loads from memory) and the first instruction of the interrupted code sequence e.g. is a move to one of the restored registers.

D3 (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 **D3** stalls instructions must be re-ordered such that the instruction that generates the register destination operand is not the directly preceding instruction.

D4 (**adsp** with SP pending update)

In principle this is the same as **D1** but for register **SP** (Stack Pointer) only and only the **adsp** instruction is affected. The **adsp** instruction uses **SP** as source operand. A directly preceding instruction that updates **SP** can cause a stall if there is no forwarding or if the update latency is more than one cycle. This is the case for **ldsh (An)+,RGS** with **SP** contained in the register list (2 cycles latency) and **jpsr** (one cycle latency but no forwarding).

D5 (TA pending update)

This is similar to **D4** 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 **ldsh (An)+,RGS** instructions with **TA** in the register list and a cycle distance < 2 and for directly preceding **mtsr Rn,TA** instructions.

D6 (**rtsr** with SP pending update)

This is similar to **D5** but for register **SP** (Stack Pointer) and only for the **rtsr** instruction that uses **SP** as source operand. There is no forwarding when **SP** is used as return address. A preceding instruction that updates **SP** with a latency > the cycle distance to the **rtsr** causes stall cycles. This is the case for **ldsh (An)+,RGS** instructions with **SP** in the register list and with a cycle distance < 2 and for directly preceding **mtsr Rn,SP** instructions.

D7 (**brlc** with LC pending update)

Again this is similar to **D4**-**D6** 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 directly preceding **mtsr Rn,LC** instruction causes a one cycle stall.

D8 (**brxx** with CC pending update)

This stall condition affects conditional branches which use the condition codes register **CC** as source operand. There is no forwarding of the **CC** source operand. If the last preceding instruction that updates **CC** has a cycle distance < 2 the conditional branch will stall until **CC** is updated. If a **CC** updating instruction is directly preceding a conditional branch there will be two stall cycles. To avoid such stalls instructions should be re-ordered if possible.

5 Compatibility

5.1 Software

The sf16bu is fully compatible with the sf16 (b) (base) ISA.

5.2 Hardware

The sf16bu interface signals and timing are the same as those of the sf16bl and sf16dl. These three processors can replace each other without changing any surrounding hardware.

5.3 Replacement Options

sf16bu and sf16bl are software compatible and can replace each other regarding both hardware and software.

The sf16dl can replace sf16bu and sf16bl regarding both hardware and software because it supports the sf16 (b) (base) ISA.