

---

# *CodeScape*

## *Pipeline Simulator User Guide*

*for Hitachi SH4 microprocessors*

---

---

# Legal Notice

## IMPORTANT

The information contained in this publication is subject to change without notice. This publication is supplied "as is" without warranty of any kind, either express or implied, including but not limited to the implied warranties or conditions of merchantability or fitness for a particular purpose. In no event shall Cross Products be liable for errors contained herein or for incidental or consequential damages, including lost profits, in connection with the performance or use of this material whether based on warranty, contract, or other legal theory.

This publication contains proprietary information which is protected by copyright. No part of this publication may be reproduced in any form, or stored in a database or retrieval system, or transmitted or distributed in any form by any means, electronic, mechanical photocopying, recording, or otherwise, without the prior permission of Cross Products Limited.

## Revision History

Release Version 3.0 January 2000

Release Candidate 1, 7 October 1996; beta 2, 26 August 1996; beta 1, 26 July 1996 - 97, 2.0.0; March 1998, Version 2.0.5a, May 1998; Version 2.1.0. alpha build 86, July 1998; Version 2.1.2. beta build 94, October 1998.

Release Version 2.2.0: March 1999; July 1999; October 1999

© 1996 - 2000 Cross Products Limited. All rights reserved.

Microsoft, MS-DOS, and Windows are registered trademarks, and Windows NT and JScript are trademarks of Microsoft Corporation in the United States and other countries. CodeScape and SNASM are registered trademarks of Cross Products Limited in the United Kingdom and other countries. Brief is a registered trademark of Borland International. CodeWright is a registered trademark of Premia Corporation. Multi-Edit is a trademark of American Cybernetics, Inc. All other trademarks or registered trademarks are the property of their respective owners.

**Cross Products Ltd**  
23 The Calls, Leeds, West Yorkshire, LS2 7EH  
telephone: +44 113 242 9814  
facsimile: +44 113 242 6163  
www.crossproducts.co.uk  
email sales: enquiry@crossproducts.co.uk  
email support: support@crossproducts.co.uk

---

# Contents

<b>About This Guide .....</b>	<b>1</b>
<b>Simulating Processor Pipelines .....</b>	<b>3</b>
SH4 pipelines and their stages .....	4
Initial stages of pipeline execution .....	6
Decode rate .....	6
Resource dependency .....	8
Locked stage .....	10
<b>Running the Simulator and Interpreting the Results .....</b>	<b>11</b>
Information Displayed .....	12
Using the Simulator .....	15
Function optimization example .....	16
Simulator output .....	18
SH4 memory model .....	22
<b>Appendix A: Status Bar Messages .....</b>	<b>25</b>
<b>Appendix B: Simulator's Shortcut Menu .....</b>	<b>27</b>
<b>Appendix C: CodeScape's Debug Menu .....</b>	<b>29</b>



# About This Guide

---

This guide is divided into three sections:

**Simulating processor pipelines** explains the purpose of the Simulator, the processors it supports, the processor pipelines, and common causes of pipeline stalls.

**Running the Simulator and interpreting the results** tells you how to run the Simulator and explains what the target and your computer do during simulation. It describes the Simulator's display and a typical simulation including an example routine. The example shows how a function can be optimized and most pipeline stalls can be avoided by re-ordering the sequence of assembly instructions.

**Appendixes** that: list the Simulator's Status Bar messages, explain how to use the Simulator's shortcut menu, and describe CodeScape's key debugging options.

---

**NOTE:**      *The example Simulator files and the original C++ code referred to in this guide are included on the release CD in the `Tutorials\SH4Simulator` directory.*

---

---

# *Simulating Processor Pipelines*

---

This CodeScape pipeline Simulator is for optimizing code that runs on Hitachi SH4 microprocessors. The Simulator helps you optimize timing critical sections of code by highlighting pipeline operations that stall the flow of instruction execution.

SH4's are superscalar pipelining microprocessors that can execute two instructions in parallel. The execution cycles depend on the processor version, for more information refer to your *Hitachi SuperH™ (SH) 32-Bit RISC MCU/MPU Series Hardware Manual*.

This guide assumes that you fully understand SH4 internal architecture. Refer to your *Hitachi SuperH™ (SH) 32-Bit RISC MCU/MPU Series Hardware Manual* for more information.

This section describes SH4 pipeline operation including the pipeline stages and how they execute. It also describes the three common causes of pipeline stalls.

## SH4 pipelines and their stages

SH4's have seven pipelines: General, General Load/Store, Special, Special Load/Store, Floating Point, Floating Point Extended, and FDIV/FSQTR. Each pipeline stage has a specific task that can run in parallel with other stages of the same pipeline and with some stages of other pipelines. Overlapping instructions by running pipeline stages in parallel increases processor throughput. The stages of each pipeline follow.

---

**NOTE:**      *Processing many instructions in parallel can cause complex pipeline behaviour.*

---

### **General Pipeline**

I	D	EX	NA	S
Fetch	Instruction decode Issue Register read Destination address calculation for PC relative branch.	Operation	Non-memory data access	Write-back

### **General Load/Store Pipeline**

I	D	EX	MA	S
Fetch	Instruction decode Issue Register read	Address calculation	Memory data access	Write-back

### **Special Pipeline**

I	D	SX	NA	S
Fetch	Instruction decode Issue Register read	Operation	Non-memory data access	Write-back

### **Special Load/Store Pipeline**

I	D	SX	MA	S
Fetch	Instruction decode Issue Register read	Address calculation	Memory data access	Write-back

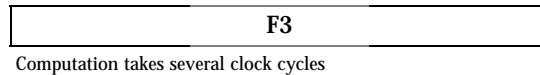
### **Floating Point Pipeline**

I	D	F1	F2	FS
Fetch	Instruction decode Issue Register read	Computation 1	Computation 2	Computation 3 Write-back



**Floating Point Extended Pipeline**

I	D	F0	F1	F2	FS
Fetch	Instruction decode Issue Register read	Computation 0	Computation 1	Computation 2	Operation 3 Write-back

**FDIV/FSQTR Pipeline**

**NOTE:** Although the FDIV/FSQTR Pipeline is described as a separate pipeline it is only used as a sub-pipeline of the Floating Point Pipeline.

**Example pipeline operations**

Assume that the op-codes (see Figure 1) are processed in the General Pipeline and that each operation takes one clock cycle. On the first clock cycle the instruction at address 0x0000 is fetched from memory and on the fifth clock cycle five instructions are processed concurrently.

The instruction at address:

- 0x0000 writes the result back to register.
- 0x0002 performs 'non-memory operations address'.
- 0x0004 performs the addition operation.
- 0x0006 is decoded and issued, and R3 is read.
- 0x0008 is fetched.

Figure 1: Instruction execution during five clock cycles

			Time										
	Address	Instruction	1	2	3	4	5	6	7	8			
	0x0000	ADD #1, r0	I	D	Ex	Na	S						
	0x0002	ADD #1, r1		I	D	Ex	Na	S					
	0x0004	ADD #1, r2			I	D	Ex	Na	S				
	0x0006	ADD #1, r3				I	D	Ex	Na	S			
	0x0008	ADD #1, r4					I	D	Ex	Na			
	0x000a	ADD #1, r5						I	D	Ex			
	0x000c	ADD #1, r6							I	D			

## Initial stages of pipeline execution

The two initial stages of pipeline execution “Fetch and Decode” are common to all instructions. In a single clock cycle, the SH4 fetches two instructions, and can decode up to two previously fetched instructions. In some conditions it cannot fetch two instructions, these include branch destinations not set on 32-bit boundaries.

## Decode rate

The decode rate is the number of instructions that are decoded and successfully issued. It is limited by the type of instructions in the pipeline and any resource requirements. An instruction type is dependent on the pipeline used. See “SH4 pipelines and their stages” on page 4.

A dual issue occurs when two instructions decode and issue simultaneously. Some instructions cannot dual issue and cause pipeline stalls. An instruction cannot dual issue if:

- It has a decode rate that does not allow dual issues.
- It has a resource dependency problem.
- Another instruction locks a required stage in the pipeline.

SH4 instructions are grouped in six groups: MT, EX, BR, LS, FE, and CO. Table 1 shows the instructions that can dual issue.

Table 1: SH4 instructions and dual issue restrictions

		Second instruction					
First instruction		MT	EX	BR	LS	FE	CO
	MT	✓	✓	✓	✓	✓	X
	EX	✓	X	✓	✓	✓	X
	BR	✓	✓	X	✓	✓	X
	LS	✓	✓	✓	X	✓	X
	FE	✓	✓	✓	✓	X	X
	CO	X	X	X	X	X	X

---

**NOTE:** A ✓ indicates that the instruction pair can dual issue providing there are no resource dependencies.

---

### Example decode rate stall

The following code section shows a series of LS and EX instructions with the issue tests highlighted. Time Slot 4 shows a pipeline stall because two LS instructions were fetched in Time Slot 3 but only one may be dispatched.

Figure 2: Decode rate stall

			Time										
	Address	Instruction	Type	1	2	3	4	5	6	7	8		
	0x0000	MOV @r3, r8	LS	I	D	Ex	Ma	S					
	0x0002	ADD #1, r1	EX	I	D	Ex	Na	S					
	0x0004	MOV @r4, r9	LS		I	D	Ex	Ma	S				
	0x0006	ADD #1, r3	EX		I	D	Ex	Na	S				
	0x0008	MOV @r3, r10	LS			I	D	Ex	Ma	S			
	0x000a	MOV @r4, r11	LS			I	i	D	Ex	Ma	S		
	0x000c	ADD #1, r5	EX				I	D	Ex	Na	S		
	0x000e	MOV @r0, r12	LS					I	D	Ex	Ma	S	
	0x0010	ADD #1, r6	EX					I	D	Ex	Na	S	

**NOTE:** Time Slots represent the number of clock cycles that have occurred at that point. For example, at Time Slot 4, four clock cycles have occurred. For more information on Time Slots see “Information Displayed” on page 12.

## Resource dependency

Resource dependency is a common cause of pipeline stalls. Resource dependency stalls occur when:

- An instruction needs a resource to complete an operation and cannot release the resource until the operation is complete.
- The resource is also required by another instruction that must wait for the resource to be released before it can execute.

### Example resource dependency stall

EX and LS instructions normally dual issue but do not in this example due to a resource dependency stall. The stall occurs because R1 contains the result of the ADD but the value is not available for the MOV to issue until Time Slot 3.

*Figure 3: Resource dependency stall*

			Time										
	Address	Instruction		1	2	3	4	5	6	7	8		
	0x0000	ADD r3, r1	LS	I	D	Ex	Na	S					
	0x0002	MOV.L r1+, @ r0	EX	I	i	D	Ex	Ma	S				
					->								
				1 latency stall									

The example in *Figure 3* only shows a short stall, resource dependencies can also cause very long stalls. *Figure 4* shows that the instruction FADD DRm, DRn could stall the next instruction's decode by up to 9 Time Slots.

### ***Examine the instruction latency***

When trying to resolve a resource dependency stall, examine the instruction latency to discover when an operation completes generating the required data. The latency is the number of Time Slots after an instruction is issued before the data is available.

Refer to your *Hitachi SuperH™ (SH) 32-Bit RISC MCU/MPU Series Hardware Manual* for more information. *Figure 4* is from the manual and includes the definition for ADD. (ADD has a latency of 1.)

Figure 4: Instruction latency of ADD

Instruction	Group	Issue	Latency	Pattern	Lock		
					Stage	Start	Cycle
ADD Rm, Rn	EX	1	1	#1	-	-	-
MOV.L @Rm, Rn	LS	1	2	#2	-	-	-
MOV.L @Rm+, Rn	LS	1	1/2	#2	-	-	-
MOV Rm, Rn	MT	1	0	#1	-	-	-
FADD DRm, DRn	FE	1	(7/8) /9	#39	F1	2	6
BT	BR	1	2 or 1	#1	-	-	-

These instructions show special cases:

- MOV.L @Rm+, Rn has two latency values, one for each of the two registers it changes (Rm + 4 takes 1, the loaded value takes 2).
- MOV Rm, Rn has a latency of 0, this implies that its result is available immediately.
- FADD DRm, DRn has two values in brackets. The values define the delay on the two halves of the 64-bit value.
- Conditional branches have two possible values. The value used is dependent on whether the branch is taken or not.

## Locked stage

Some instructions require extensive use of certain stages of the pipeline. This can cause instructions later in the pipeline to stall.

### Example locked stage stall

The CLRMAC instruction locks the F1 stage of the floating-point pipelines for two cycles. Stalls of this type are unusual but removing them substantially increases performance.

*Figure 5: Locked stage stall*

					Lock		
Instruction	Group	Issue	Latency	Pattern	Stage	Start	Cycle
CLRMAC	C0	1	3	#28	F1	3	2

# *Running the Simulator and Interpreting the Results*

---

When you run the Simulator, CodeScape first reads the target processor's cache to your computer where it is maintained until you close the Simulator. The Simulator uses the target's memory and registers as the source and destination of each assembly instruction and switches CodeScape from debugging mode to simulation mode. Switching to simulation mode halts the target and processor, and all execution operations such as run, stop, and trace are simulated on your computer.

To run the Simulator:

- Click Tools, Simulate Processor.
- OR-
- Configure a breakpoint's *Trigger Actions*. In the *Configure Breakpoint(s)* dialog select *Cause processor simulation to start* to tell the Simulator to run when the breakpoint has triggered.

See “*Configure a breakpoint*” on page 31.

On close, the Simulator executes any remaining instructions in the simulated pipeline then returns CodeScape to normal debugging on the target. However, the cache is not restored.

---

**NOTE:**      *Use the Simulator's shortcut menu commands to configure the Simulator and access the debugging functions.*

---

## Information Displayed

When you simulate a function, the simulated pipeline's operation is described in the Simulator output regions. The Simulator output shows the flow of instruction execution in Slots down the screen over Time. Each Time Slot describes pipeline operations that are completed concurrently.

The display has two cursor bars. The vertical bar highlights the instructions executing in active Time Slot. The horizontal bar is a marker that highlights the assembly instruction that is considered to be executing. The information on the Status Bar relates to the active Time Slot:

**Diagnosis** describes the stall type encountered and what caused it.

**Cache** describes memory operations that affect the cache when reading and writing data.

**System clock** shows the total time taken for processor operations up to the current cursor position.

The information generated by the Simulator can be printed, or saved in a file with the `sim` extension. For information on reading the printed or saved information see *“Understanding the printed or saved results”* on page 16. For information on how to print and save files see *“Appendix B: Simulator’s Shortcut Menu”* on page 27.

### SH4 pipeline operation display

The Simulator evaluates each instruction's functionality at the appropriate stage of the pipeline. For example, the instruction `mov.l@r0,r3` tells the processor to read 32 bits from the address stored in R0, then put the results in R3.

---

**NOTE:**     *Debugging in another region when viewing Simulator can be confusing. This is because the Simulator only reports complete Time Slot operations and does not show information about an instruction until the whole Time Slot is complete.*

---

Different colors describe the state of the processor, and mnemonics describe specific operations at each stage of the pipeline. An operation shown in:

**Black** means the processor was OK.

**Red** indicates the processor stalled.

**Blue** indicates the processor missed the cache.

**Pink** indicates the processor stalled and missed the cache.



Table 2: Instruction execution and pipeline operations

The instruction mnemonic:	Indicates this operation:
IF	Instruction fetch.
if	Dummy instruction fetch where external memory is not accessed.
ID	Instruction decoded/issued/register read.
D	Decode stage locked.
d	Decode stage. (Register read only.)
EX	Instruction execution.
SX	Execution phase, the SX stage used.
SX*	SX stage locked not used.
NA	Memory not accessed/no operation address.
MA	Memory accessed/operation address.
WB	Register write back (data stored to registers after operation).
F0	Floating point 0 stage accessed. (Special Stage inner product/transforms.)
F1	Floating point 1 stage accessed.
F1*	Floating point 1 stage locked and not accessed.
f1	Floating point 1 stage partial usage (can overlap with other f1's but not F1).
F2	Floating point 2 stage accessed.
F3	Floating point 3 stage accessed. (Special Stage divide/square root).
FS	Floating point store/writeback.
>FPSCR<	Floating point status register updated.

**NOTE:** *In the `sim` file all of the instructions are represented by the mnemonics listed above except, >FPSCR< which is represented by FC.*

## Example Simulator display

The screen shot shows 148 clock cycles occurred before the Register Usage Stall in the active Time Slot where:

- Five instructions are processed concurrently.
- It takes 55 clock cycles to complete the Time Slot (see header of active column).  
A Time Slot's completion time is determined by the parallel operation that takes the most time to complete. In the example it is the memory access.
- R15 causes a stall in the decode phase.
- A data read missed the cache so a new cache line had to be fetched from memory.  
The new cache line fetch was delayed by a copy back already in progress.




Figure 6: Register Usage Stall

Simulator SH4(7091)-MASTER: T1-P1															
Address	Op-code	CPU time	Dissassembly	1	1	1	1	1	55	28	1	28			
0C010624	C708	002	movs #_stack_ref,r0												
0C010626	6F02	---	mov.l @r0,r15												
0C010628	C706	001	movs #_main_ref,r0												
0C01062A	6002	029	mov.l @r0,r0												
0C01062C	400B	030	jsr @r0												
0C01062E	0009	---	nop	NA/SX*	WB/SX*										
0C010114	4F22	002	sts.l pr,0-r15	ID	EX	NA	WB								
0C010116	903D	056	mov.w #ff7b4,r0	IF	ID	SX/D	NA/SX	WB/NA	WB						
0C010118	3F0C	028	add r0,r15	if	if	if	ID	EX	NA	WB					
0C01011A	64F3	---	mov r15,r4				if	if	ID	EX	NA				
0C01011C	743C	001	add #63c,r4					if	ID	EX	NA				
0C01011E	BFC8	029	bss cbaseclass1						if	ID	EX				
0C010120	0009	---	nop							IF	ID	EX			
0C0100B2	7FFC	001	add #-f04,r15												
0C0100B4	2F42	---	mov.l r4,@r15												
0C0100B6	E200	001	mov #600,r2												
0C0100B8	D304	056	mov.l #1,r3												
0C0100BA	2322	028	mov.l r2,@r3												
0C0100BC	60F2	---	mov.l @r15,r0												
0C0100BE	7F04	001	add #f04,r15												
0C0100C0	000B	003	rts												
Diagnosis: Register Usage Stall				Cache: Copy-Back in Progress, Data Read Miss, Line Fetched								System Clock: 148			

## Using the Simulator

Most pipeline stalls can be avoided by re-ordering the sequence of assembly instructions. The Simulator highlights pipeline stalls helping you to decide where instructions can be re-ordered. It is recommended that you only run the Simulator on small sections of code.

A typical method for using the Simulator is:

1. Start CodeScape.  
On the Windows desktop, click the Start button, choose Programs and select CodeScape.
2. Load a Program file.  
Click File, Load Program file and select the file you want to simulate.
3. Create a Source region.  
Click Window, New Window then right-click in the region and select Source.
4. Add a breakpoint at the point you want to start the simulation.  
Place the cursor where you want to start the simulation then click  on the Breakpoint toolbar to add a start breakpoint.
5. Add a breakpoint at the point you want to end the simulation.  
Place the cursor where you want to end the simulation then click  on the Breakpoint toolbar to add an end breakpoint.
6. Run your program to the start breakpoint.  
Right-click in the Source region, select Execution... and click Run. Run until the start breakpoint occurs.
7. Start the Simulator.  
Click Tools, Simulate Processor.
8. Run to the end breakpoint.  
Right-click in the Simulator, select Execution... and click Run. Run until the end breakpoint occurs.
9. Analyze Results. You can print the pipeline results or save them to a file.  
Right-click in the Simulator and click Print... or Save to file...
10. Close the Simulator.  
On the Simulator's title bar click .
11. Based on the simulation results, edit or re-order your source code to optimize your program and improve pipeline operation.
12. Simulate the optimized code.

## Understanding the printed or saved results

When you print or save a simulation any stalls are represented by different letters. Where a stall occurs the appropriate letter is shown at the bottom of the Time Slot that generated it. Control instruction stalls are only avoided by using an alternative instruction in place of a control instruction. Resource conflict stalls and same group stalls are avoided by re-ordering the Time Slot instructions.

Table 3: How stalls appear in the Simulator's printed or saved output

Letter:	Shows:
w>	A write back to the register when a memory access is incomplete.
i>	An instruction generated stall.
R>	Resource conflict. For example, one instruction trying to write to a register when another instruction is trying to read the same register.
s>	The SX stage of the instruction is locked.
f>	A floating point pipeline stall caused by multiple use of F0, F1, or F3 stages.
c>	One or more control group instructions being dual issued.
g>	Instructions of the same type occurring together and causing a dispatch failure. For example, EX + EX, LS + LS, BR + BR, FE + FE.
?>	An unknown stall type. (Error in the Simulator.)

## Function optimization example

### C/C++ function

The C/C++ function calculates the cross product of two vectors then returns the magnitude.

```
struct Vector {
    float x, y, z;
} ;

float CrossProduct(Vector *v1, Vector *v2, Vector *res)
{
    res->x = (v1->y)*(v2->z)-(v2->y)*(v1->z);
    res->y = (v2->x)*(v1->z)-(v1->x)*(v2->z);
    res->z = (v1->x)*(v2->y)-(v2->x)*(v1->y);

    return fsqrt(res->x * res->x + res->y * res->y + res->z * res->z);
}
```

## First hand-coded assembler version

The first hand-coded assembler version of the C/C++ function code follows the algorithm. The code is written in assembler to reduce the number of memory reads/writes but does not address any of the pipeline problems. The simulation of the code is shown in *figure 7 on page 19*. The hand-coded assembler is shown in the right-hand column under Disassembly.

```
.EXPORT _CrossProduct
_CrossProduct:
; Load Vectors
fmov.s    @r4,fr4      ; v1->x
fmov.s    @r5,fr7      ; v2->x
mov       #4, r0
fmov.s    @(r0,r4), fr5; v1->y
fmov.s    @(r0,r5, fr8; v2->y
mov       #8, r0
fmov.s    @(r0,r4), fr6; v1->z
fmov.s    @(r0,r5, fr9; v2->z

; calculate res->x
fmov      fr5, fr1
fmul      fr9, fr1; v1->y * v2->z
fmov      fr8, fr0
fmul      fr6, fr0; v2->y * v1->z
fsub      fr0, fr1; (v1->y * v2->z) - (v2->y * v1->z)
fmov.s    fr1, @r6

; calculate res->y
fmov      fr7, fr2
fmul      fr6, fr2; v2->x * v1->z
fmov      fr4, fr0
fmul      fr9, fr0; v1->x * v2->z
fsub      fr0, fr2; (v2->x * v1->z) - (v1->x * v2->z)
mov       #4, r0
fmov.s    fr2, @(r0, r6)

; calculate res->z
fmov      fr4, fr3
fmul      fr8, fr3 ; v1->x * v2->y
fmov      fr7, fr0
fmul      fr5, fr0; v2->x * v1->y
fsub      fr0, fr3; (v1->x * v2->y) - (v2->x * v1->y)
mov       #8, r0
fmov.s    fr3, @(r0, r6)

; calculate the magnitude
fldi0     fr0
fipr      fv0, fv0; (res->x*res->x)+(res->y*res->y)+(res->z*res->z)
fsqrt     fr3      ; sqrt
rts
fmov      fr3, fr0; return result in fr0 with Time Slot operation.
```

## Simulator output

The three example simulations show how the function is examined, then optimized through two iterations using information generated by the Simulator to re-order the code. It is assumed that all memory accesses hit the cache and take 1 clock cycle, and no registers are saved or restored as part of the function call. No instructions have been removed or changed. Performance improvements are due to changing the instruction execution order.

---

**NOTE:**      *The assembler for each simulation is in the right-hand column under Disassembly.*

---

### **First simulation** See Figure 7 on page 19

In the first simulation, the function has Control Group Instruction stalls, Same Group stalls, and Resource Usage stalls. The:

- Control Group Instruction stalls are due to the JSR and RTS and cannot be avoided without removing the function call.
- Same Group stalls are due to the code reading all input data at the top of the program.
- Resource Usage stalls are due to the linear nature of the calculation.

### **Second simulation** See Figure 8 on page 20

In the second simulation, time is saved by:

- Delaying the output vector save until the magnitude calculation starts. The two large resource dependency stalls waiting for the result of the inner product and the square root free the stages needed to save the vector to memory.
- Interleaving the input vector loads with the calculation so more operations dual issue.

### **Third simulation** See Figure 9 on page 21

In the third simulation, time is saved by reducing the resource dependency stalls between the FMUL and FSUB, and the FMOV and FMUL.

The third version of the function shows a time improvement of about 20%. This can make a substantial difference to your program if the function is called many thousands of times.

---

**NOTE:**      *The code can be optimized even further if the code is changed to re-use FR4-FR9 instead of FR0 and if FNEG and FMAC are used instead of FMUL and FSUB.*

---

Figure 7: First simulation

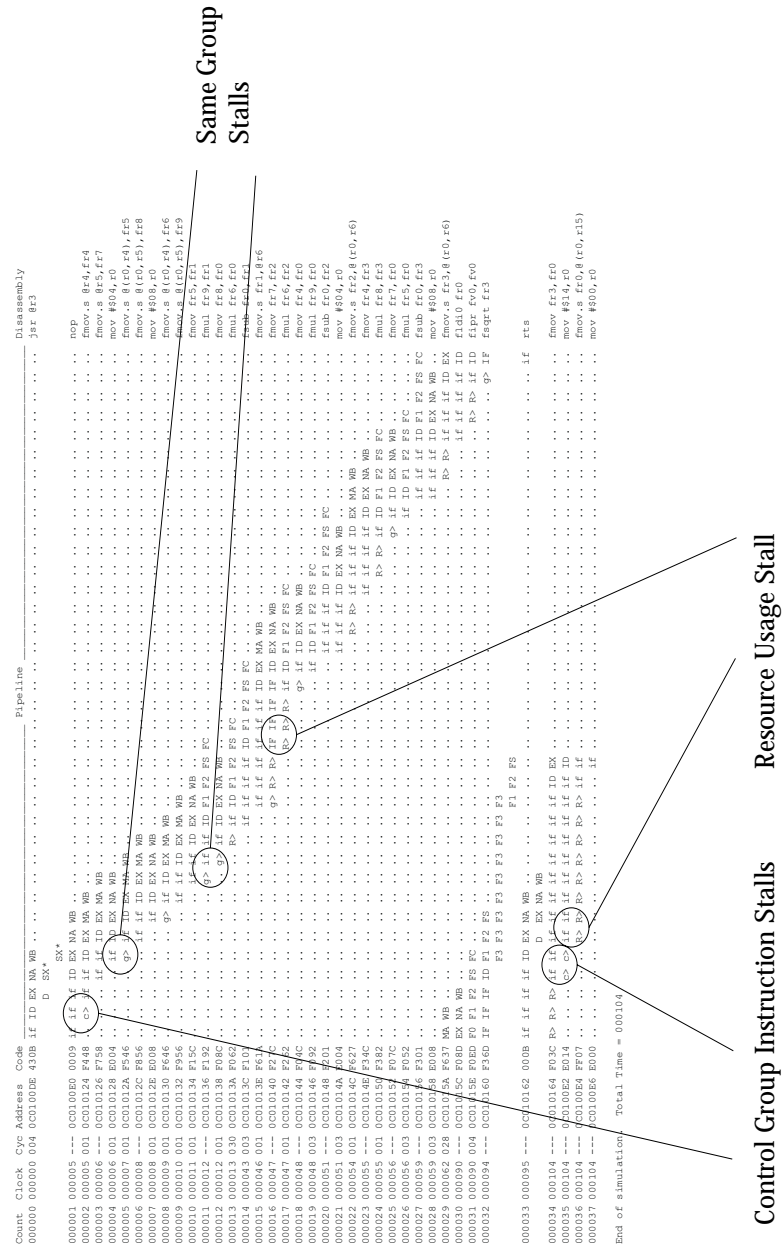


Figure 8: Second simulation

Count	Clock	Cyc	Address	Code	Pipeline	Disassembly
000000	000000	004	0C0100DE	430B	If ID EX NA WB .. .. .	3xR Rr3
000001	000005	---	0C0100E0	0009	if if ID EX NA WB .. .. .	nop
000002	000005	---	0C010124	E004	.. c> if if ID EX NA WB .. .. .	mov \$804, r0
000003	000005	001	0C010126	F448	.. .. . if ID EX NA WB .. .. .	fmov.s 8(r0,r5), r4
000004	000006	001	0C010128	F856	.. .. . if ID EX NA WB .. .. .	fmov.f4, r0
000005	000007	001	0C01012A	F34C	.. .. . if ID EX NA WB .. .. .	fmov.f4, r3
000006	000008	---	0C01012C	F382	.. .. . if ID EX NA WB .. .. .	fmov.f4, r3
000007	000008	001	0C01012E	F546	.. .. . if ID EX NA WB .. .. .	fmov.s 8(r0,r4), r5
000008	000009	---	0C010130	F584	.. .. . if ID EX NA WB .. .. .	mov \$800, r0
000009	000009	002	0C010132	F758	.. .. . if ID EX NA WB .. .. .	fmov.f7, r0
000010	000011	---	0C010134	F07C	.. .. . if ID EX NA WB .. .. .	fmov.f7, r0
000011	000011	001	0C010136	F052	.. .. . if ID EX NA WB .. .. .	fmov.f7, r0
000012	000012	029	0C010138	F956	.. .. . if ID EX NA WB .. .. .	fmov.s 8(r0,r5), r5
000013	000041	---	0C01013A	F301	.. .. . if ID EX NA WB .. .. .	fmov.f0, r3
000014	000041	001	0C01013C	F15C	.. .. . if ID EX NA WB .. .. .	fmov.f5, r1
000015	000042	---	0C01013E	F192	.. .. . if ID EX NA WB .. .. .	fmov.f5, r1
000016	000042	001	0C010140	F646	.. .. . if ID EX NA WB .. .. .	fmov.f5, r1
000017	000043	001	0C010142	F08C	.. .. . if ID EX NA WB .. .. .	fmov.f5, r1
000018	000044	003	0C010144	F062	.. .. . if ID EX NA WB .. .. .	fmov.f5, r1
000019	000044	---	0C010146	F101	.. .. . if ID EX NA WB .. .. .	fmov.f5, r1
000020	000047	001	0C010148	F201	.. .. . if ID EX NA WB .. .. .	fmov.f5, r1
000021	000048	---	0C01014A	F242	.. .. . if ID EX NA WB .. .. .	fmov.f5, r1
000022	000048	001	0C01014C	F04C	.. .. . if ID EX NA WB .. .. .	fmov.f5, r1
000023	000049	003	0C01014E	F092	.. .. . if ID EX NA WB .. .. .	fmov.f5, r1
000024	000052	---	0C010150	F201	.. .. . if ID EX NA WB .. .. .	fmov.f5, r1
000025	000052	001	0C010152	F08D	.. .. . if ID EX NA WB .. .. .	fmov.f5, r1
000026	000053	001	0C010154	E008	.. .. . if ID EX NA WB .. .. .	fmov.f5, r1
000027	000054	001	0C010156	F637	.. .. . if ID EX NA WB .. .. .	fmov.f5, r1
000028	000055	031	0C010158	F08D	.. .. . if ID EX NA WB .. .. .	fmov.f5, r1
000029	000086	---	0C01015A	F36D	.. .. . if ID EX NA WB .. .. .	fmov.f5, r1
000030	000086	---	0C01015C	F61A	.. .. . if ID EX NA WB .. .. .	fmov.f5, r1
000031	000087	---	0C01015E	E004	.. .. . if ID EX NA WB .. .. .	fmov.f5, r1
000032	000088	---	0C010160	F627	.. .. . if ID EX NA WB .. .. .	fmov.f5, r1
000033	000089	---	0C010162	000B	.. .. . if ID EX NA WB .. .. .	fmov.f5, r1
000034	000096	---	0C010164	F03C	.. .. . if ID EX NA WB .. .. .	fmov.f5, r1
000035	000096	---	0C0100E2	E014	.. .. . if ID EX NA WB .. .. .	fmov.f5, r1
000036	000096	---	0C0100E4	F070	.. .. . if ID EX NA WB .. .. .	fmov.f5, r1
000037	000096	---	0C0100E6	E000	.. .. . if ID EX NA WB .. .. .	fmov.f5, r1

End of simulation. Total Time = 000096



Figure 9: Third simulation

Count	Clock	Cyc	Address	Code	Pipeline	Disassembly
000000	000000	004	0C0100DE 430B	if ID EX NA WB D SX*	...	jsr @r3
000001	000005	---	0C0100E0 0009	if if ID EX NA WB SX*	...	nop
000002	000005	001	0C010124 5004	.. > if ID EX NA WB	...	mov #504,r0
000003	000006	001	0C010124 5004	.. > if ID EX NA WB	...	mov.s @r4,r4
000004	000006	001	0C010126 F836	.. > if ID EX NA WB	...	mov.s @r4,r5
000005	000007	---	0C01012A F546	.. > if ID EX NA WB	...	mov.s @r0,r4,r5
000006	000007	---	0C01012C F508	.. > if ID EX NA WB	...	mov #508,r0
000007	000008	001	0C01012C F508	.. > if ID EX NA WB	...	fmov.s @r5,r7
000008	000008	---	0C010130 F34C	.. > if ID EX NA WB	...	fmov f4,r3
000009	000009	001	0C010132 F382	.. > if ID EX NA WB	...	fmul f8,r3
000010	000010	---	0C010134 F07C	.. > if ID EX NA WB	...	fmov f7,r0
000011	000010	001	0C010136 F052	.. > if ID EX NA WB	...	fmul f5,r0
000012	000011	001	0C010138 F956	.. > if ID EX NA WB	...	fmov.s @r0,r5
000013	000012	028	0C01013A F646	.. > if ID EX NA WB	...	fmov.s @r0,r4
000014	000040	---	0C01013C F931	.. > if ID EX NA WB	...	fsub fr0,fr3
000015	000040	001	0C01013E F15C	.. > if ID EX NA WB	...	fmov f5,fr1
000016	000041	---	0C010140 F192	.. > if ID EX NA WB	...	fmul fr9,fr1
000017	000041	001	0C010142 F08C	.. > if ID EX NA WB	...	fmov fr8,fr0
000018	000042	001	0C010144 F062	.. > if ID EX NA WB	...	fmul fr6,fr0
000019	000043	001	0C010146 F262	.. > if ID EX NA WB	...	fmov f6,fr2
000020	000043	002	0C010148 F262	.. > if ID EX NA WB	...	fsub fr0,fr1
000021	000045	---	0C01014A F101	.. > if ID EX NA WB	...	fmov f4,fr0
000022	000045	001	0C01014C F04C	.. > if ID EX NA WB	...	fmul fr9,fr0
000023	000046	---	0C01014E F032	.. > if ID EX NA WB	...	mov #508,r0
000024	000046	003	0C010150 E008	.. > if ID EX NA WB	...	fsub fr0,fr2
000025	000049	---	0C010152 F201	.. > if ID EX NA WB	...	fldl0 fr0
000026	000049	001	0C010154 F08D	.. > if ID EX NA WB	...	fmov.s fr3,@r0,r6
000027	000050	002	0C010156 F637	.. > if ID EX NA WB	...	fmov.s fr0,@r0,r15
000028	000052	031	0C010158 F08D	.. > if ID EX NA WB	...	fldr fr0,fr0
000029	000083	---	0C01015A F36D	.. > if ID EX NA WB	...	fsqrt fr3
000030	000083	---	0C01015C F61A	.. > if ID EX NA WB	...	fmov.s fr1,@r6
000031	000084	---	0C01015E 5004	.. > if ID EX NA WB	...	mov #504,r0
000032	000085	---	0C010160 F627	.. > if ID EX NA WB	...	fmov.s fr2,@r0,r6
000033	000086	---	0C010162 000B	.. > if ID EX NA WB	...	rts
000034	000093	---	0C010164 F03C	.. > if ID EX NA WB	...	fmov f3,r0
000035	000093	---	0C0100E2 E014	.. > if ID EX NA WB	...	fmov #514,r0
000036	000093	---	0C0100EA F077	.. > if ID EX NA WB	...	fmov.s fr0,@r0,r15
000037	000093	---	0C0100E6 5000	.. > if ID EX NA WB	...	mov #500,r0

End of simulation. Total Time = 000093

## SH4 memory model

The previous examples assumed that all operations took 1 clock cycle. For memory accesses this is not true. Access (read/write) to external memory takes several cycles to complete. The Simulator mimics many of the features present on the SH4, it models:

- Cached and uncached memory including cache update methods. Examples are: write-back, copy-back, RAM mode.
- The store queues.
- Data bus access size (byte, word, long, double and cache line).
- Instructions that operate on cache memory such as PREF, MOVCA, OCBI.

To configure this model the Simulator reads information from two sources. It reads:

- On startup, the cache control register and the processor cache memory (instruction and data caches). These are used to initialize its internal representation of the cache.
- The user specified timing information and valid memory areas from a configuration file (see Figure 10).

Figure 10: Example DASH4 memory configuration file

```
[DASH MasterSH4EVA_ValidMemory]
; General Address Space - optimal transfer size is LONG
;
; Access      Start      End      Size  Expr      Restrictions      WRITE      READ
; Method      Start      End      Size  Expr      Restrictions      Byte Word Long Quad Cache  Byte Word Long Quad Cache
ReadWrite = 0x0C000000, 0x0CFFFFFF, LONG, , NoRestrictions , 4, 4, 4, 4, 16, 7, 7, 7, 7, 28
ReadWrite = 0x0D000000, 0x0DFFFFFF, LONG, , SimulatorHardwarePort , 1, 1, 1, 1, -1, 1, 1, 1, 1, -1
ReadWrite = 0x8C000000, 0x8CFFFFFF, LONG, , NoRestrictions , 4, 4, 4, 4, 16, 7, 7, 7, 7, 28
ReadWrite = 0xAC000000, 0xADFFFFFF, LONG, , NoRestrictions , 4, 4, 4, 4, 16, 7, 7, 7, 7, 28
ReadWrite = 0xF4000000, 0xF4FFFFFF, LONG, , NoRestrictions , 1, 1, 1, 1, -1, 1, 1, 1, 1, -1
ReadWrite = 0xFC000000, 0xFC0003ff, LONG, , NoRestrictions , 1, 1, 1, 1, -1, 1, 1, 1, 1, -1

; Areas for Java Script
ReadWrite = 0xFF000010, 0xFF00001F, LONG, , NoRestrictions , 1, 1, 1, 1, -1, 1, 1, 1, 1, -1
ReadWrite = 0xFF800000, 0xFF800003, LONG, , NoRestrictions , 1, 1, 1, 1, -1, 1, 1, 1, 1, -1
ReadWrite = 0xFF800004, 0xFF800005, WORD, , NoRestrictions , 1, 1, 1, 1, -1, 1, 1, 1, 1, -1
ReadWrite = 0xFF800008, 0xFF800017, LONG, , NoRestrictions , 1, 1, 1, 1, -1, 1, 1, 1, 1, -1
ReadWrite = 0xFF80001c, 0xFF80001d, WORD, , NoRestrictions , 1, 1, 1, 1, -1, 1, 1, 1, 1, -1
ReadWrite = 0xFF800020, 0xFF800029, WORD, , NoRestrictions , 1, 1, 1, 1, -1, 1, 1, 1, 1, -1
Write = 0xFF940190, 0xFF940190, BYTE, , NoRestrictions , 1, 1, 1, 1, -1, 1, 1, 1, 1, -1

[DASH MasterSH4EVA_SharedMemory]
SharedMemory = 0x0C000000, 0x0CFFFFFF, Tag:B
SharedMemory = 0x8C000000, 0x8CFFFFFF, Tag:B
SharedMemory = 0xAC000000, 0xACFFFFFF, Tag:B

[DASH MasterSH4EVA_Settings]
AutoSoftBreakEnabled = 0
```

## Memory configuration file: valid memory section

The valid memory section of the configuration file describes specific areas of memory to the Simulator.

The available fields are:

- Supported access method.  
For example, Read only (Read), Write only (Write), or Read and Write (ReadWrite).
- Start and end address of the memory section being defined.
- Access size (BYTE, WORD, LONG).  
This tells CodeScape how to request memory from the target for this memory area. Access size is not used by the Simulator.
- Access Restrictions (NoRestrictions or SimulatorHardwarePort).  
Defines the part of CodeScape, or the part of the Simulator, that can see or access this memory area.
- Timing information.  
Defines the access time for the Simulator when accessing this memory area with the set access size. A value of -1 means that the access type is not possible and generates an error if attempted. For example, cache accesses to the P2 and P4 memory spaces are marked "-1".

---

**CAUTION:**    *Do not change the default values in the memory configuration file.*

---

---

**NOTE:**        *Optimizing cache using the PREF instruction can substantially improve performance.*

---



# *Appendix A: Status Bar Messages*

---

The messages on the Simulator's status bar provide stall and cache operation information about the current instruction.

**Stall messages are:**

- Register Usage Stall.
- SX Stage Stall.
- Floating Point Pipeline Stall.
- CO Group Dispatch Failure Stall.
- Same Group Dispatch Failure Stall.
- Memory Access/Instruction Fetch Stall.
- Write Back/Register Usage Stall.
- Multiplier Usage Stall.
- Instruction Generated Stall.
- Memory Access/Memory Access Stall.

**Cache operation messages are:**

- Write Miss.
- Read Miss, Line Fetched.
- Data Write Miss.
- Data Read Miss.
- Data Write Hit.
- Data Read Hit.
- Instruction Read Miss.
- Store Queue Write.
- Block.
- Copy-Back.
- Line Fetched.
- Write-Back.
- Write-Through.
- Allocation.
- Purge.
- Invalidate.
- Write-Back.
- Prefetch in Progress.
- Copy-Back in Progress.
- Write-Back in Progress.

# Appendix B: Simulator's Shortcut Menu

---

Table 4: The shortcut menu in the Simulator

Select:	To:
Highlight Cache Misses	See in which Time Slot a pipeline operation missed the cache and what caused it.
Highlight Pipeline Stalls	See in which Time Slot a pipeline operation stalled and what caused it.
Show Stall Type	Show the type of stall generated: "R>", "R", "c>".
Show Uppercase	Show instructions in upper case.
Show Symbols	Show operand values as symbols.
Show EAs & Lits.	Show the effective address and literals.
Source / Disassembly tracking	Track the Simulator cursor with the source and disassembly regions.
Print	Print the results of program simulation.
Save to file...	Save the results of program simulation to a file.
Execution	Run, stop, and restart a program. Run a program to the cursor position, or until it executes a specified address. Run all of your programs simultaneously.
Breakpoints	Enable, disable, configure, reset, and remove breakpoints.
Overlay Management	View and manage overlay properties.
Properties...	Configure fonts and colors. Set the update rate for a single region, a region type, and each processor. Change the tab settings.





# *Appendix C: CodeScape's Debug Menu*

---

CodeScape's regular debugging functions are available when the Simulator is running. The debugging functions include commands for: controlling program execution, stepping code, using breakpoints, and setting the cursor to the PC and vice versa.

When you single step in a Simulator region the cursor is shown at the instruction currently executing in the pipeline. During simulation the PC fetches instructions ahead of the current instruction (when you single step in any other CodeScape region, the cursor is shown at the PC). Some instructions are not executed because of changes in the program flow. For example, instructions fetched after a branch.

---

*NOTE:*      *You cannot run the Profiler and the Simulator at the same time.*

---

## Execute a program

In a Source or Disassembly region, run your program in one of the following ways:

- On the Debug menu, click Run to run your program (F9).
- On the Debug menu, point to Execution then click Run All to run all of your programs simultaneously (CTRL+F9).
- On the Debug menu, point to Execution then click Run to Cursor to run your program to the cursor position (ALT+F9).
- On the Debug menu, point to Execution then click Run to Address... Type an address in the Expression field of the Run to Address/Instructions dialog to run your program until it executes a specified address (SHIFT+F9).


## Step into (trace) code

In a Source or Disassembly region, step into your code in one of the following ways:

- Click Debug, click Step to single step a line of code (F7).
- On the Debug menu, click Forced Step to step a line of source code at the disassembly level (SHIFT+F7).
- On the Debug menu, click Step Over to step over a line of source code (F8).

## Add a breakpoint

In a Source or Disassembly region:

1. Click Debug, click Goto Address. Enter an expression address to go to, click OK.
2. On the Breakpoint toolbar, click  to set a breakpoint.
3. On the Debug menu, click Execution, click Run to Address... Your program will run until the breakpoint is reached.

## Configure a breakpoint

CodeScape enables breakpoint configuration including data accesses within memory ranges and breakpoints on external peripheral devices.

To configure a breakpoint:

- Click Debug, point to Breakpoints then click Configure Breakpoint(s)... (CTRL+F5).

The *Configure Breakpoint(s)* dialog appears. Specify any options you require. For more information on using the dialog, refer to the online help supplied with your version of CodeScape.

