Validation of Cycle-Accurate CPU Simulators against Real Hardware

Master's Thesis, 3 October 2000

Sven Montán

Advisor & Examiner:
Jakob Engblom
Uppsala University/IAR Systems

Information Technology
Department of Computer Systems
Uppsala University, Sweden

Abstract

One of the most important issues regarding real-time software performance is the worst case execution time *(WCET). There are several methods for deducing the WCET statically; one such analysis system is currently under development by WCET researchers from Uppsala University in co-operation with C-lab in Panderborn. The system uses a CPU simulator for calculating the execution time of selected parts of the target program, which creates a demand for a very cycle-accurate simulator.*

In this thesis, we give a method for validating a CPU simulator against real hardware CPU. The proposed test method is a black-box test method that relies on hardware analysis for test-case generation. The validation method aims at both determining the accuracy of the simulator and to pinpoint simulator errors for improving the accuracy.

We have tested this validation method on a NEC V850 CPU core simulator, and the results show that the average error-rate drops from 11,2 % to 1,3 % for a set of benchmark programs.

Acknowledgements

The author likes to thank Jakob Engblom, IAR Systems, Uppsala, for irreplaceable support during this work. Acknowledgements are also given the rest of the IAR staff for their help and fruitful comments. Special thanks to Eric Arbin for introduction to and support for the IAR tools and for help with the emulator installation and configuration.

The author also likes to thank IAR Systems for providing tools important to his work, such as the IAR Embedded Workbench (including the IAR C-Spy debugging tool) and the NEC V850E emulator.

Contents

1. Introduction and Background

1.1. WCET Analysis Tools

One of the most crucial aspects of real-time software programming and testing is the determine ration of software performance. One of the most important issues regarding software performance is the estimated *worst case execution time* (WCET). The WCET is required for system modelling and verification, scheduling analysis, hardware dimensioning, etc.

Real-time software developers currently have few options for investigating performance issues such as the WCET. The industrial standard of today is to spend substantial time, perhaps hundreds of hours, running software on real hardware and measuring execution times. This problem becomes even more obvious for embedded system developers, since the applications sometimes must be burned into ROM before performance can be tested.

An estimated WCET must be safe, i.e. it must never underestimate the real value of the WCET. Using measurements for estimating the WCET can never produce a safe WCET since there is no guarantees that the real worst case scenario as been captured by the measurements. A safe estimated WCET can only be produced be formal static analysis.

The need for tools performing static WCET analysis is evident, but still very few such tools have been developed and deployed. However, one research tool for WCET analysis is the Uppsala/Paderborn Generic WCET Analysis Tool [1] currently under development by WCET researchers from Uppsala University in co-operation with C-lab in Panderborn, Germany. This analysis tool performs both advanced high-level and low-level program flow analyses to deduce the worst-case execution time scenario for a given target program. The selected parts of the target program are then run in a cycle-correct CPU simulator and the final execution timing is obtained. This means that the analysis tool is critically dependent on a correct simulator to generate safe times.

1.2. Making a Cycle-accurate CPU Simulator

Designing and implementing a cycle correct CPU simulator is a very difficult task. The main obstacle is probably to obtain correct and detailed CPU specifications. Such specifications are often derived directly from the CPU hardware documentation, written by the CPU manufacturer. There are several sources of errors in such a process:

- Documentation is too superficial. The simulator programmer is then forced to make assumptions about the CPU internal functionality and design.
- Real CPU implementation does not correspond to the documentation on some significant details. Such divergences can arise from inconsistent documentation during CPU development or even simple typing errors.

The simulator specifications then become very unreliable, possibly leading to a bad simulator design. Important internal states and functions of the CPU may be modelled incorrectly or even completely omitted due to a poor abstraction.

A CPU simulator is also a complex piece of software, hence the presence of implementation bugs are probably unavoidable. Such bugs may cause the simulator to render a significantly wrong execution time result for some target programs (while others may be totally unaffected by the bugs).

These issues are only some of the problem simulator vendors must face and they clearly illustrate the need for systematic methods of simulator debugging and validation.

1.3. Why Formal Proofs is not an Option

A CPU simulator can be viewed as just another software program. The target hardware architecture, which the simulator is supposed to imitate, may be specified and documented using a formal hardware description language such as *VHDL* or *Verilog*. This might indicate that formal proofs could be used as validation method framework. However, we consider validation by formal proofs as an infeasible strategy. Some problems of using formal proofs in this case are:

- Insufficient specifications
- High complexity of the software, beyond current technologies

Lack of knowledge of how the architecture is implemented inside the actual hardware will certainly cause CPU simulator vendors to design and implement incorrect performance models without violating any formal specification. Formal proofs may very well be used to validate other parts of a WCET analysis tool, but in the case of simulators the gap between theoretical models and reality (physical target chip) would remain unexplored.

We believe that simulator validation by comparing the simulator to real hardware is probably the only solution to achieve an adequate validation, at least in the foreseeable future.

1.4. Our Method of Validation

In this thesis we present our work on finding a systematic method for CPU simulator validation.
The fundamental idea is to perform *black-box testing* in which performance results from running tests on the CPU simulator are compared to the results of the same tests on real hardware.

Each single test is a simple micro-benchmark program, which includes a short hand-written test code pattern, which is timed. The set of code patterns is derived from characteristics of the simulation target hardware architecture and aims at covering as many execution scenarios as possible.
Every test code is run on real hardware and the execution times are measured. These results act as validation references and are compared to execution times calculated by the CPU simulator.
Divergences indicate simulator errors and the need to correct the simulator.

The basic ideas behind this method of validation are not new. They have been proposed and tested earlier in the work by Black and Shen [2]. Their method aimed at validation of a performance model, using comparison tests between the model and the real hardware. The test suite contained unbiased test cases, as well as random and hand-written test codes. Nevertheless, some key assumptions let us take a slightly different approach, especially on test code pattern generation:

- Our method aims at CPU simulators for microcontroller chips with fairly simple architectures.
- We limit our performance parameters to only include execution time with respect to pipeline execution.
- We do not invoke other performance issues such as cache miss rates, speed of memory accesses etc. This also let us use a simplified memory interface model, letting us assume static and well-known timing for every single memory access.

With these restrictions on hardware complexity we believe that systematic and careful static analysis of hardware architecture can reveal the information needed to generate a complete test suite. This means that no random test cases or other unbiased tests with “ultimate covering” ambitions are necessary to establish an adequate validation test suite.

1.5. Case Study on NEC V850E CPU Core Simulator

We have performed a case study to investigate if such a validation method works in practice.
The NEC V850E 32-bit microcontroller CPU core [6] has been used as hardware target and validation reference. The V850E CPU architecture features simple RISC instructions together with more complex instructions that are specially adapted to embedded system requirements, making it a fairly typical embedded microcontroller.

The V850E pipeline model is rather straightforward but not trivial, since it offers parallel execution of two instructions under certain conditions. The presence of on-board internal ROM and RAM and the absence of caches correspond well with our limitations on memory access.

Our subject to validation test case, the Uppsala/Paderborn V850E CPU Simulator prototype, has originally been based only on written documentation from NEC. Earlier tests [3] of this simulator have shown that the simulator may miscalculate execution time with up to 25 % for some benchmark programs and 11,2 % in average.

Ideally, we would like a perfect correspondence, which is probably not feasible for any non-trivial architecture. For use in static WCET analysis, the CPU simulator must be very close to the hardware but a small divergence can be tolerated if it is bounded. A safety margin may then be added to the WCET to account for such divergence, yielding a safe estimate that is slightly loose.

1.6. Method Evaluation and Results

By comparing results for some benchmark programs before and after debugging the simulator, we are able to estimate how well our method expose simulator incorrectness. The relative improvements of simulator accuracy for these benchmarks act as a measurement on effectiveness of our method.

The results from these benchmark comparison tests show that the average simulator error has dropped from 11,2 to 1,3 % after applying our method to the simulator. These results indicate that our method may be a feasible strategy for debugging and validation of a CPU simulator.

1.7. Paper

Our work is presented as following:

- Related works are presented in chapter 2.
- In chapter 3 we give an overview of our method. Since we invoke testing, we discuss and define test data adequacy criteria. A method cookbook is also presented in this chapter.
- The analysis and test tools used in our case study are introduced in chapter 4.
- Chapter 5 covers the analysis of the V50E CPU pipeline, while the instruction set is analysed in chapter 6
- The structure and design of our test suite is given in chapter 7, and the validation test results are presented in chapter 8. This chapter also includes the results from the method evaluation.

2. Related Work

2.1. Micro-benchmarking

While larger benchmarks are often used with the intention of testing computer system performance under real workloads, micro benchmark aims at measuring specific features of a system. This use of micro benchmark is introduced by Saavedra, Gaines, and Carlton [4], which use micro benchmarks to extensively analyse memory performance of the KSR1 parallel computer.

2.2. Testing WCET Analysis Results

An interesting method of testing the correctness of the result calculated by static WCET analysis is presented by Puschner and Nossal [5]. This method is based on comparison between the result produced by the WCET analysis tool and the result from measuring execution time on real hardware. The test programs are generated automatically using a *generic algorithm (GA).* The generated programs are based on the results of previous test results.

This method shares the property of using comparison tests as main framework with our method. On the other hand, Pushner and Nossal test the entire WCET analyse tool, while our method narrows on testing a CPU simulator. Their main interest is in testing the dependence of input data and associated variances in program flow, and not on low-level CPU issues.

2.3. Performance Model Calibration

Black and Shen propose a method of Performance Model Calibration [2], very closely related to our method. In this method a set of test code patterns are generated, each test code is run on real hardware and the execution times are measured. The performance model also calculates execution times for every test code and the results are compared. After analysing the comparison outcome, the model is debugged and the tests and comparisons are repeated.

The test code set is divided into five test suites; *Alpha, Beta* and *Gamma* test suites contains unbiased tests where each and every assembler instruction is executed individually and in combination with both the same instruction and all other instructions. Several randomised code sequences form the *Random* test suite. Finally, *hand-written* code patterns are used to trigger special hardware functionality and to check model boundary conditions.

By running a similar comparison test with large benchmarks after every debugging phase, the effectiveness at improving model accuracy can be verified.
Since the results presented on effectiveness show a much lower degree of accuracy improvement than expected by the authors, this may indicate that the proposed method is insufficient. However, Black and Shen claim in their final analysis that a larger and more advanced test suite would render an acceptable validation.

One principal point in Black and Shen's method is of course how the test suites are compiled. The Alpha, Beta and Gamma tests together with random test suite are generated automatically, using only basic information on target instruction set. We assume that no prior analysis is needed for this work. However, the special hand-written patterns require an thorough analysis of hardware, instruction set etc, in order to find as many interesting cases as possible.

In our work we have used an approach very similar to the method proposed by Black and Shen. We have adopted the basic idea of a comparison test, invoking a proper test suite, between a simulator and real hardware as the validation method framework. We have also used comparison tests on larger benchmark for evaluating the effectiveness and correctness of our method.

The main difference between Black and Shen’s method and ours is the test suite generation. Our test code patterns are all hand-written based on prior analysis of the hardware architecture. No unbiased tests such as Alpha, Beta, Gamma or random tests are included in our test suite.

3. Method Overview

In this chapter, we discuss CPU simulator validation in terms of *white-box* and *black-box testing*.

We discuss method restrictions on hardware complexity, and a method cookbook is presented. We conclude by defining some problems regarding comparison of measured execution times.

3.1. Software Testing vs. CPU Simulator Validation

CPU simulator validation based on testing against real hardware share many properties with general software testing. Software testing normally involves careful analysis of the software implementation. This includes analysis of the code statements and search for possible execution paths. To reach the best possible test coverage, test criteria such as "execute each statement at least once" or "evaluate each branching structure with all possible values" is stated. This is known as structural or white-box testing, since tests primary derive from knowledge of the software's structure and how the program is implemented.

Another form of software testing is functional or black-box testing. Functional testing means that the test cases are derived only from software specifications and the test results are given by analysing input and corresponding output. Only input and output domains are analysed and classified in order to find the best test data adequacy criteria.

CPU simulator validation by testing against real hardware is very similar to software black-box testing. The simulator program is not analysed in any way and the validation result only depends on simulator input and output. In this case, the input is some assembler instruction sequence and output is the total execution time of this instruction sequence.

Since validation is based on comparison, we consider there to be only one fundamental definition of simulator correctness - the simulator should always agree on execution times (output) with real hardware for any arbitrary instruction sequence (input).

To be able to handle this infinite input set, demands for CPU hardware analysis are more or less unavoidable. Strive for optimal test coverage may also require that every relevant internal state of the simulated CPU be reached during some execution scenario. This implies a profound hardware analysis not unlike the analyses performed in white-box testing. However, if the hardware architecture is too complex, the analysis will certainly fail and other test coverage strategies are required (e.g. unbiased tests may be used).

Our method of simulator validation adopts the idea of black-box testing as the fundamental testing discipline, augmented with a white-box analysis of the hardware. We define *test data adequacy criteria* to quantify our goal of test coverage. These criteria are derived from the simulator-input domain, i.e. sequences of assembler instructions.

The final test suite composition is solely based on results from structured analyses of the real hardware architecture. Hence no unbiased or random test cases are required in our method.

3.2. Method Restrictions on Hardware Complexity

Constraints on hardware complexity are crucial for our method to work in practice. Hence, we have been forced to make some initial restrictions and assumptions before we started our investigation of the V850E hardware.

The memory interface model has been limited to only concern V850E on-chip RAM and ROM. These memories are accessed on independent memory buses and thus no conflicts between data and instruction accesses should ever occur.

The internal ROM holds all instructions and all memory access are completed within a single clock cycle during instruction fetch. Data memory accessing instructions normally access internal RAM only. Such accesses are also completed in a single clock cycle. The V850E instruction set includes two instructions that read data from internal ROM. This exceptional behaviour and its impact on execution timing is explored and analysed as a special case.

The above constraints narrow our analyses and tests to only include the instruction set and the execution environment, i.e. the CPU pipeline.

3.2.1 Case Study Validation Reference Machine

Our method requires the ability to measure execution time on real hardware. Due to limitations in time and available equipment, we have not been able to perform such measurement on a real V850E microcontroller chip. Instead we have used a NEC V850E CPU core emulator as validation reference. The use of an emulator is of course a potential source of error since we depend on the timing accuracy of this emulator. Nevertheless, we do not believe that this has any major effects upon our proposed method. The emulator is developed by real chip manufacture NEC, and they claim to use the same core logic in the microcontroller and the emulator. This also means that even if we have used an emulator in our case study we do not believe that our method is restricted by this fact.

Overall performance monitoring has been limited to a cycle clock counter, due to lack of dedicated hardware counters. This has been quite sufficient since our method disregards performance impacts of the external memory interface, cache etc.

3.3. Execution Parameters

Instead of considering single instructions to be atomic units, we are interested in invoking instructions based on their execution characteristics, e.g. what pipeline resources they use and how they utilise these resources. By analysing the instruction set, we can find these *execution parameters* (EP), and we can also, for each instruction assign *values* to all execution parameters. This means that each instruction can be associated with a *set of tuples (EP, value)*. We will use these sets to find a *reduced instruction set*, which will be the set of instructions we use in our test cases.

The execution environment, i.e. the pipeline and instruction fetch mechanisms, may also have specific execution parameters. An example of scenarios where such execution parameter have impact on overall execution times may be situations where the instruction fetch mechanism fail to fetch an entire instruction due to some memory alignment constraint. These execution parameters must of course be identified in order to reach adequate test coverage.

3.4. Method Cookbook

We have tried to formalise our basic ideas into a validation method cookbook. Each step is described in more details below.

3.4.1 Hardware Analysis

The purpose of hardware architecture analysis is to:

- Identify and classify as many significant execution parameters of the architecture as possible.
- For each instruction, determine the values of the execution parameters of instructions.
- Examine how and under which condition instructions interfere with each other during execution. This analysis includes examining if (and under which conditions) instructions can interfere directly with the third or the forth succeeding instruction without interfering with the intervening instructions.

This requires a deep and structured investigation of the instruction set and pipeline design. It is very hard to give details on what to examine before the actual analysis is commenced, since it is extremely hardware-specific.

The first step will be to analyse the pipeline design, identifying pipeline stages and possible datapaths. Attention must also be paid to how instructions are fetched from instruction memory and under which conditions this may cause pipeline delays.

The next step is to analyse the instruction set, and for each instruction and instruction variant:

- Determine which actions the instruction performs during its execution.
- Match these actions to execution resources, i.e. pipeline stages.
- Analyse any potentially instruction interference scenarios.

Any action that may be suspected to have impact on execution timing is denoted as an execution parameter, and the execution parameter value is determined for each instruction. Note that this is also necessary when designing a cycle-accurate simulator.

3.4.2 Reducing the Instruction Set

To reduce the number of instructions, each instruction or instruction variant is classified regarding its associated set of (EP, value)-tuples. Instructions with the same sets are grouped together into an *equivalence class*.

As a final step, a representative instruction from each equivalence class is selected. These instructions constitute a reduced instruction set, which is used for test code generation.

3.4.3 Test Data Adequacy Criteria

After reduction of the instruction set, each remaining instruction will have a unique set of (EP, value) - tuples. From the instruction set analysis, we know the conditions for instruction interference. From the execution environment (pipeline) analysis, we know the conditions for execution delays caused by the execution environment EP's.

We can now define the test data adequacy criteria for our method as following:

1) Each instruction of the reduced instruction set should be tested when:
   a) The instruction interferes with succeeding instructions (interference conditions satisfied).
   b) The instruction does not interfere with succeeding instructions (interference conditions not satisfied).

2) The test suite should also include test cases where:
   a) Execution environment EP has impact on the execution time of the test case (condition satisfied).
   b) Execution environment EP has no impact on the execution time of the test case (condition not satisfied).

3.4.4 Test Suite Structure

The test suite is divided into several specialised *test case families*, where each family is targeting closely related instructions. Individual test suite families may be derived from:

- "Natural" instruction relations, e.g. a family focusing solely on division instructions.
- Instructions with closely similar sets of (EP, value)-tuples, e.g. "simple" arithmetic instructions.
- Special execution paths such as parallel execution etc.

3.4.5 Test Code Patterns

A test case family consists of a set of short hand-written code patterns. The intention is that each pattern will exercise and reveal the impact of a single or at most a few execution parameters of a target instruction. Thus, each test case should act as a precise error indicator when it comes to debugging the CPU simulator.

The results from the analysis phase will serve as general guidelines for code pattern generation. We assume that each target instruction has some kind of "basic" execution behaviour inside the pipeline, e.g. a simple arithmetic instruction is pipelined perfectly, while a jump instruction cause a static pipeline delay of two clock cycles. Such execution scenarios are easy to find since they are common and also probably well documented. We must, however, write patterns where every unique scenario win which target instructions interfere with adjacent instruction in such a way that additional execution delays may occur. Patterns where no instruction interference is expected, and hence not triggering abnormal execution behaviours, must of course also be included. This improves coverage and helps interpreting test results with respect to simulator debugging.

The code patterns are very short instruction sequences. To be able to measure the execution times distinctly, each pattern must be repeated at least a couple of hundreds of times within each test run. Hence, execution behaviour when a pattern is repeated must be taken into account when designing the code patterns. In most cases, a "neutral" transition between repetitions is desired. In other cases, e.g. patterns with only a single instruction, pattern transition will be most significant to overall execution time.

Another important issue to address, especially with respect to pattern repetition and transition, is that every code pattern must be fully executable on real hardware. Instructions can't be allowed manipulate the contents of registers in such way that succeeding instructions may fail or cause undesired events, e.g. division by zero or overflow.

3.4.6 Test Program Execution

Each test code pattern is repeated and transformed into two equal test programs, one for the CPU simulator and one for the reference machine (real CPU hardware). Some kind of code generation tool should be used to avoid small but devastating divergences between corresponding test programs, e.g. input to simulator is a sequence of one thousand ADD instructions, while real hardware code only incorporates 998 ADD's.

Every test program is executed on the reference machine and the execution time is measured. These values act as validation references; thus these programs need only be executed and measured once. Corresponding test programs are then given to the CPU simulator and the execution times, calculated by the simulator, are compared to the validation reference values. If any divergences are detected, the simulator must be corrected and the tests repeated until all divergences are eliminated.

3.5. Measuring Execution Time

Measuring execution time on real hardware is often very difficult. Interference of unknown or non-deterministic factors may cause large variances on test code execution time. Arriving at accuracy of a few clock cycles require a stable, deterministic and predictable test bed.

In our case study, we have used a V850E emulator as hardware reference machine. This means that we have not been forced to face the more practical difficulties arising when measuring execution time directly on the real CPU chip, and no such problems will be discussed in this thesis.

We have assumed the emulator to be stable and deterministic. All our experience with the emulator supports this assumption. Even though some emulator results have been most surprising, they always have been repeatable. The emulator is presented and discussed in more detail in the next chapter.

3.5.1 Problems when Comparing Measured Execution Times.

There are some important issues regarding measuring and comparing execution time that need to be addressed. Since we want to compare the execution time result given by the simulator to the result given by the emulator, we must ensure that the simulator and the emulator agree on what to measure, in other words:

- When should the measuring start?
- When should it be stopped?

Since we assume that the simulator and emulator are deterministic, measuring must start when the simulator and the emulator are in some equivalent, initial state. Such a “zero” -state could, as an example, be defined as a *cold pipeline* i.e. every resource inside the pipeline is free and empty, including instruction prefetch mechanisms and prefetch instruction queue. This is probably the most natural initial state for the simulator.

Unfortunately, a cold pipeline state is most likely unreachable for the emulator, and definitely unreachable if we should measure on real CPU chip. This requires us to use a *warm pipeline* as initial state i.e. there are already some instructions executing inside the pipeline when first instruction of the test code is fetched and measuring is started. This means those internal states of the emulator and the simulator must be well controlled before the test code initiate execution. In our case, we must be sure that no instructions with “nasty” execution behaviours immediately precede our test code and thereby cause unexpected or uncontrolled execution delays.

Defining the point where measuring should stop is even more difficult. It is very hard to determine at which point in time the last instruction of the test code terminates, especially if this instruction has very long individual execution time e.g. loops in some pipeline stage. The problem is to “synchronise” this point in time, if we decide to stop the emulator measuring clock when the final instruction is being fetched from instruction memory, will the simulator do the same?

Our solution to these problems is presented in chapter 7: Our Test Cases.

4. Analysis and Testing Tools

Our analyses have been performed through studying and interpreting the hardware documentation, and through testing and verifying this information on a hardware emulator. This chapter describes and discusses the documentation used. It also includes a brief description of the emulator and its interface. We conclude with a discussion on emulator versus real hardware chip.

4.1. The V850E User's Manual

All V850E documentation used and referred in the following chapters is taken from the NEC V850/MS1 Preliminary User's Manual (UM)[6]. The essential information has been taken from following sections:

- 5.3 *Instruction Set*. This section includes syntactic and semantic descriptions of each instruction.
- 5.4 *Number of Instruction Execution Clock Cycles*. Includes Table 5-10, which gives some interesting information on instruction interference.
- 8, 8.1, 8.2 *Pipeline*. Include figures of general pipeline configuration and dual issue scenario.
- 8.3 *Pipeline Flow during Execution of Instructions*. Includes Execution Flow Diagrams (EFD) for each type of instruction. Also includes table of memory access timings.

Relying solely on hardware documentation may not be sufficient in order to find all relevant facts about the hardware architecture. Even documentation that can be considered unusually detailed, such as the V850E documentation, may be confusing and easy to misinterpret. During our work we have encountered ambiguous information and even some completely wrong specifications in the User's Manual. All such cases are described in this paper.

4.2. The Emulator

In order to get a better understanding of the CPU, we have used a V850E Core Emulator for testing and clarifying execution behaviours whenever needed.

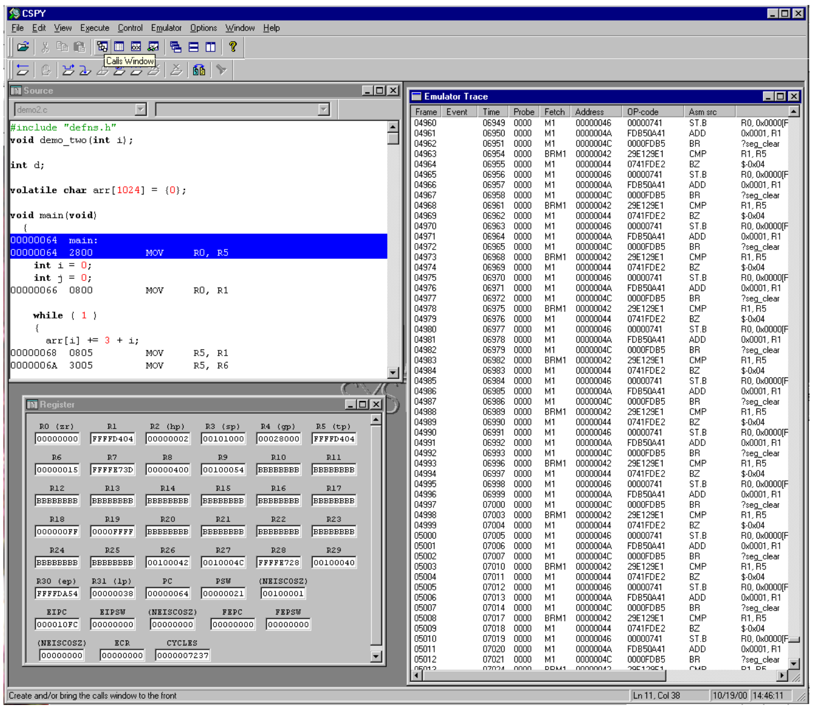
The emulator is programmed and monitored via the IAR V850 C-Spy debugging software [7]. This application includes monitoring of CPU registers, as well as clock cycle counter and execution trace logs. Trace logs show each executed instruction marked with a clock cycle timestamp, indicating execution time in relation to adjacent instructions.

Example: A sequence of instructions with "perfect" pipeline behaviour, e.g. unrelated addition instructions, will render timestamps C, C+1, C+2 and so on. C is the timestamp value for first instruction in the sequence.

It is not always clear how the timestamps should be interpreted. We do not know which execution event sets the value of a timestamp, e.g. execution in some specific pipeline stage, or if timestamps are deduced in some other way. Hence, timestamps only show if a delay has occurred but not why or in which pipeline stage. Whenever the trace log indicates such a delay or other unusual execution timings, documentation has been consulted for clarification, and vice versa.

Figure 4-1 shows the user interface of C-Spy, running an emulator session.

To some extent, the emulator has also been used for verification of rather trivial information, gathered from V850E User's Manual. This has been very enlightening and helped us to reason about the pipeline design and implementation. In general, having an emulator with good control software is a very valuable tool for simulator validation.



```
#include "defns.h"
void demo_two(int i);

int d;

volatile char arr[1024] = {0};

void main(void)
  {
00000064  main:
00000064  2800           MOV      R0, R5
    int i = 0;
    int j = 0;
00000066  0800           MOV      R0, R1

    while ( 1 )
    {
      arr[i] += 3 + i;
00000068  0805           MOV      R5, R1
0000006A  3005           MOV      R5, R6
```

```
 R0 (zr)    R1         R2 (hp)    R3 (sp)    R4 (gp)    R5 (tp)
[00000000] [FFFFD404] [00000002] [00101000] [00028000] [FFFFD404]
 R6         R7         R8         R9         R10        R11
[00000015] [FFFFE73D] [00000400] [00100054] [BBBBBBBB] [BBBBBBBB]
 R12        R13        R14        R15        R16        R17
[BBBBBBBB] [BBBBBBBB] [BBBBBBBB] [BBBBBBBB] [BBBBBBBB] [BBBBBBBB]
 R18        R19        R20        R21        R22        R23
[000000FF] [0000FFFF] [BBBBBBBB] [BBBBBBBB] [BBBBBBBB] [BBBBBBBB]
 R24        R25        R26        R27        R28        R29
[BBBBBBBB] [BBBBBBBB] [00100042] [0010004C] [FFFFE728] [00100040]
 R30 (ep)   R31 (lp)   PC         PSW        (NEISCOSZ)
[FFFFDA54] [00000038] [00000064] [00000021] [00100001]
 EIPC       EIPSW      (NEISCOSZ)  FEPC       FEPSW
[000010FC] [00000000] [00000000]  [00000000] [00000000]
 (NEISCOSZ)   ECR          CYCLES
[00000000]   [00000000]   [0000007237]
```

```
Frame  Event  Time   Probe  Fetch  Address   OP-code   Asm src
04960         06949  0000   M1     00000046  00000741  ST.B     R0, 0x0000[F
04961         06950  0000   M1     0000004A  FDB50A41  ADD      0x0001, R1
04962         06951  0000   M1     0000004C  0000FDB5  BR       ?seg_clear
04963         06954  0000   BRM1   00000042  29E129E1  CMP      R1, R5
04964         06955  0000   M1     00000044  0741FDE2  BZ       $-0x04
04965         06956  0000   M1     00000046  00000741  ST.B     R0, 0x0000[F
04966         06957  0000   M1     0000004A  FDB50A41  ADD      0x0001, R1
04967         06958  0000   M1     0000004C  0000FDB5  BR       ?seg_clear
04968         06961  0000   BRM1   00000042  29E129E1  CMP      R1, R5
04969         06962  0000   M1     00000044  0741FDE2  BZ       $-0x04
04970         06963  0000   M1     00000046  00000741  ST.B     R0, 0x0000[F
04971         06964  0000   M1     0000004A  FDB50A41  ADD      0x0001, R1
04972         06965  0000   M1     0000004C  0000FDB5  BR       ?seg_clear
04973         06968  0000   BRM1   00000042  29E129E1  CMP      R1, R5
04974         06969  0000   M1     00000044  0741FDE2  BZ       $-0x04
04975         06970  0000   M1     00000046  00000741  ST.B     R0, 0x0000[F
04976         06971  0000   M1     0000004A  FDB50A41  ADD      0x0001, R1
04977         06972  0000   M1     0000004C  0000FDB5  BR       ?seg_clear
04978         06975  0000   BRM1   00000042  29E129E1  CMP      R1, R5
04979         06976  0000   M1     00000044  0741FDE2  BZ       $-0x04
04980         06977  0000   M1     00000046  00000741  ST.B     R0, 0x0000[F
04981         06978  0000   M1     0000004A  FDB50A41  ADD      0x0001, R1
04982         06979  0000   M1     0000004C  0000FDB5  BR       ?seg_clear
04983         06982  0000   BRM1   00000042  29E129E1  CMP      R1, R5
04984         06983  0000   M1     00000044  0741FDE2  BZ       $-0x04
04985         06984  0000   M1     00000046  00000741  ST.B     R0, 0x0000[F
04986         06985  0000   M1     0000004A  FDB50A41  ADD      0x0001, R1
04987         06986  0000   M1     0000004C  0000FDB5  BR       ?seg_clear
04988         06989  0000   BRM1   00000042  29E129E1  CMP      R1, R5
04989         06990  0000   M1     00000044  0741FDE2  BZ       $-0x04
04990         06991  0000   M1     00000046  00000741  ST.B     R0, 0x0000[F
04991         06992  0000   M1     0000004A  FDB50A41  ADD      0x0001, R1
04992         06993  0000   M1     0000004C  0000FDB5  BR       ?seg_clear
04993         06996  0000   BRM1   00000042  29E129E1  CMP      R1, R5
04994         06997  0000   M1     00000044  0741FDE2  BZ       $-0x04
04995         06998  0000   M1     00000046  00000741  ST.B     R0, 0x0000[F
04996         06999  0000   M1     0000004A  FDB50A41  ADD      0x0001, R1
04997         07000  0000   M1     0000004C  0000FDB5  BR       ?seg_clear
04998         07003  0000   BRM1   00000042  29E129E1  CMP      R1, R5
04999         07004  0000   M1     00000044  0741FDE2  BZ       $-0x04
05000         07005  0000   M1     00000046  00000741  ST.B     R0, 0x0000[F
05001         07006  0000   M1     0000004A  FDB50A41  ADD      0x0001, R1
05002         07007  0000   M1     0000004C  0000FDB5  BR       ?seg_clear
05003         07010  0000   BRM1   00000042  29E129E1  CMP      R1, R5
05004         07011  0000   M1     00000044  0741FDE2  BZ       $-0x04
05005         07012  0000   M1     00000046  00000741  ST.B     R0, 0x0000[F
05006         07013  0000   M1     0000004A  FDB50A41  ADD      0x0001, R1
05007         07014  0000   M1     0000004C  0000FDB5  BR       ?seg_clear
05008         07017  0000   BRM1   00000042  29E129E1  CMP      R1, R5
05009         07018  0000   M1     00000044  0741FDE2  BZ       $-0x04
05010         07019  0000   M1     00000046  00000741  ST.B     R0, 0x0000[F
05011         07020  0000   M1     0000004A  FDB50A41  ADD      0x0001, R1
05012         07021  0000   M1     0000004C  0000FDB5  BR       ?seg_clear
05013         07024  0000   BRM1   00000042  29E129E1  CMP      R1, R5
```

```
Create and/or bring the calls window to the front        Ln 11, Col 38   10/19/00   14:46:11
```

Figure 4-1: User interface of IAR C-Spy[7].

4.2.1 Discussion on Emulator vs. Real Chip

Considering that we used the emulator for verifying and clarifying information found in the chip documentation, questions on emulator accuracy are inevitable. We have not been able to experiment with real hardware chip during our work, thus we have no way to measure or quantify any potential incorrectness of the emulator. To handle the gap between emulator and real hardware chip, we have made the following assumption: *The emulator is always accurate and corresponds correctly to the real hardware.*

This means that whenever we encountered divergence between documentation and emulator, we have considered the information given by the emulator as correct.

Nevertheless, some test scenarios have been found in which execution results presented by the emulator are very strange and inexplicable. We have denoted these cases as “remarkable” throughout this paper. In order to maintain consistency on our emulator assumption, we have chosen to include them as relevant criteria for our final test code generation. We do not know if these exceptions indicate an incorrect emulator or an accurate emulator, emulating a real chip with odd design and undocumented features (or even bugs, which is quite common even in hardware design and implementations).

4.3. Test Program Generator

Each test code pattern has been derived and specified manually but the pattern specifications were translated into final test programs by a simple code-generating program (written in SML).

Since emulator programs must be fully executable, they include instruction sequences for initiating data memory and registers. Execution time for this overhead is avoided by letting dedicated instructions trigger the clock-cycle counters. The memory address of every instruction must be kept synchronised with the corresponding simulator program so that no uncontrolled misalignment scenarios occur during execution, and thus the emulator code is loaded to the same address as the simulator uses.

Due to the design of the simulator, simulator test programs must be structured into basic block. Within a basic block, each individual instruction is encoded together with its memory address [8].
Each code pattern instance is translated into a basic block when the pattern ends with a branching instruction. Otherwise, all pattern instances can be aggregated into one large basic block.
Test code wrapping sequences are always encoded into separate start and end blocks.
The simulator also needs a description of the basic block execution order. This file is also produced by the code generator.

An example of equivalent emulator and simulator programs is in Appendix A: Test Programs of Test Case F0_1.

Most patterns have been designed so that the total execution time for the final test programs can be expected as multiples of the number of pattern instances when executed on the emulator i.e. we can ascribe a certain number of cycles to each iteration of the pattern.
Exceptions are misalignment test programs, which are designed to trigger a misalignment scenario every other pattern instances. This design is chosen since it allows start address 0x2000 to be used for all test programs.

5. V850E Pipeline Analysis

The first step in our analyses involves a closer look at the instruction execution environment and its characteristics. All information is gathered from the V850E User's Manual.
This chapter gives a brief overview of the NEC V850E pipeline configuration. This also includes descriptions and discussion on parallel execution, so called dual issue scenarios.

5.1. V850E Pipeline Configuration

Unlike earlier models of V850 family, the V850E CPU core features both a five-stage Master Pipeline and a special four-stage pipeline dedicated for short load (short address mode) and branch instructions. In this Additional Pipeline, address calculation is performed during instruction decode stage (ID), thus no executing stage (EX) is required.

The normal data path of the Master Pipeline is split after EX. Instructions that do not access memory, such as arithmetic and logic instructions, utilise the "original" data path. Execution is then transferred by a data forwarding stage (DF) to final write back stage (WB).

Memory accessing instructions use the memory stage (MEM) and WB of the Additional Pipeline. Thus a non-blocking behaviour of memory accessing instruction is achieved. We will however not be able to investigate and test this option since data memory accesses are completed in a single clock cycle in all our test configurations.

The instruction fetch stage (IF) is shared by the Master and the Additional Pipeline. IF is designed to process at most 32 bits of instruction data per clock cycle. This implies that instructions longer than 32 bits should always cause some pipeline delay.

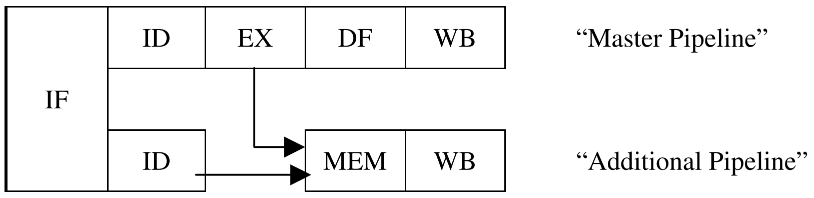


Figure 5-1. The V850E Pipeline Configuration [6]

5.2. Instruction Prefetch

Before instructions are processed in IF they have to be fetched from instruction memory. This is performed by a dedicated Prefetch Unit, which reads four consecutive bytes from instruction memory, always from a 32-bit aligned address. If only the first 16 bits can be processed by IF, e.g. first two bytes are a 16-bit instruction and remaining two bytes are first half of a following 32-bit instruction, then remaining two bytes are stored in an instruction buffer queue.

5.3. Parallel Instruction Execution (Dual issue)

Dual pipelines with independent data paths also bring the possibility of two instructions executing simultaneously. Assuming that IF holds two 16-bit instructions which together constitute a possible dual issue pair, and that preceding instructions have not caused the pipelines to stall, then both instructions can be issued together during next clock cycle.

Dual issue scenarios may have a significant impact on the execution time of programs and instruction sequences. Typically, a suitable sequence of dual-issued 16 bit instructions have a CPI as low as 0.5, while a sequence of 32-bit instruction never can reach below 1.0 CPI. This improvement raises the performance of the CPU by at least 10 %, sometimes as high as up to 30 %.

5.4. In-order Issue and Execution

The older members of the V850 family implemented a strict in-order issuing policy for instruction execution. This regulates the order in which instructions are allowed to access resources of the pipeline. Interlock mechanisms allows instructions with very long execution times or "messy" pipeline behaviour, e.g. divide or bit manipulation instructions, to stall the pipeline while they execute. Succeeding instructions are never allowed to overtake executing instructions at any point.

The V850E core has inherited this pipeline policy but the introduction of parallel pipelines with dual issue has made several execution scenarios more complex and harder to predict. For instance, a dual issue scenario implies that the logical issuing order between two instructions is arbitrary. Consecutive instructions can also utilise different pipelines during execution. This may require other types of interlocking and control mechanisms than found in the V850 core.

5.5. Pipeline Analysis Results

Instruction length is clearly significant for instruction fetch timing as well as the possibility for 16-bit instructions to participate in dual issue scenarios. This is an indication of an individual instruction parameter and we investigate this in more detail during instruction set analysis, presented in the next chapter.

However, we have one execution parameter that depends on the execution environment itself. Assume a branch instruction is causing the instruction buffer queue to be invalidated and that branch destination instruction is 32 bits and misaligned, i.e. least significant address byte: 0x02, 0x06, 0x0A or 0x0E. Since the prefetch unit follows a strict word alignment policy when reading instruction data, prefetch will then need an extra read-cycle since only first half of the instruction is read during the first clock cycle. This will then cause an execution delay of one clock cycle.

A similar delay will occur when a branch destination instruction, also misaligned, constitutes the first 16-bit instruction of a dual issue pair. These two 16-bit instructions will not be issued together since they do not execute in IF during the one and same clock cycle.

This discovered execution parameter is denoted EP0. Since it is not a parameter of individual instruction, we can not use it for instruction classification. However, our test suite must include misalignment scenarios where this behaviour is triggered and tested.

- **EP0: Instruction prefetch misalignment after changed program execution flow.**

6. V850E Instruction Set Analysis

This chapter gives our execution analysis of the V850E instruction set. The analysis aims at finding:

- Execution parameters (EP) of instructions and instruction variant. Execution parameters with assigned values are later used for equivalence classification and instruction set reduction. Discovered execution parameter are shown as:
  - ❑ **EP<no>: <description>** **(<value range>)**
- Interesting execution scenarios. We are especially interested in finding scenarios where instruction interference may cause additional execution delays. We also want to find all possible dual issue scenarios. This is not trivial since the V850E documentation does not specify such scenarios at all.
- The "propagation" of instruction interference, i.e. assuming an instruction sequence of instructions A, B, and C, can there be scenarios where instruction A does not interfere directly with its successor B, but instead interfere directly with next instruction C? This is important to investigate since we want our test code patterns to be as short and as few as possible, yet including all kinds of instruction interference scenarios.

The instructions are already grouped in the V850E User's Manual [6], regarding their type and execution behaviour. We have used this basic classification for structuring our analyses.

During pipeline configuration analysis, we already found that instruction length is a significant execution parameter. The V850E instruction set includes instruction of length: 16, 32, 48, and 64 bits. The majority is 16 and 32 bits, the only exceptions are 48-bit `MOVimm32` together with the 48 and 64 bit `PREPARE` variants.

❑ **EP1: Instruction length. (16/32/48/64)**

6.1. Analysis of individual instructions

Our basic strategy has been to begin by analysing the simplest instructions, how they execute in the pipeline and how they interfere with each other. With "simplest" we mean an instruction with relatively well-known and easily understood executing behaviour, for example the 32-bit "addition with immediate" instruction (`ADDI`). We then move on to load/store instructions, branch/jump instructions, instructions with pipeline loops, and so on.

6.1.1 No operation instruction

Analysed instruction: `NOP`

The simplest instruction is NOP. NOP does not do anything, and thus no execution parameters are found. The Execution Flow Diagram (EFD) of the User's Manual shows execution only in the IF and the ID stage. These are probably the stages of Master Pipeline; the EFD's are not always clear on which pipeline is used.

6.1.2 Arithmetic and logical instructions

Analysed instructions: `MOV`, `MOVEA`, `MOVHI`, `ADD`, `ADDI`, `CMP`, `SUB`, `SUBR`, `SETF`, `SASF`, `CMOV`, `ZXB`, `ZXH`, `SXB`, `SXH`, `BSH`, `BSW`, `HSW`, `NOT`, `OR`, `ORI`, `XOR`, `XORI`, `AND`, `ANDI`, `TST`, `SHR`, `SAR`, `SHL`, `SATADD`, `SATSUB`, `SATSUBI`, and `SATSUBR`

Includes 16 and 32-bit arithmetic, logical and saturating arithmetic instructions with simple and easily analysed pipeline execution behaviours. Multiply and divide instructions are not included in this group. Documentation indicates the following:

- Execution in Master Pipeline only.
- Read at most two general registers (GR) during ID stage
- `CMOV, SASF` and `SETF` read Program Status Word (PSW) during ID stage

- Most instructions write to PSW, presumably during EX stage
- Execution in EX stage during one clock cycle.
- Write back to at most one register in WB

Reading and writing general registers or PSW is a potential source of pipeline execution delays, due to data hazards [9]. We must consider both data hazards and structural hazards. Structural hazards will result if the register file has too few read or write ports. Delays due to read-after-write hazards (RAW) are usually avoided since the V850E pipeline architecture includes forwarding techniques (an example shown in Figure 6-1), but there are cases when they occur.

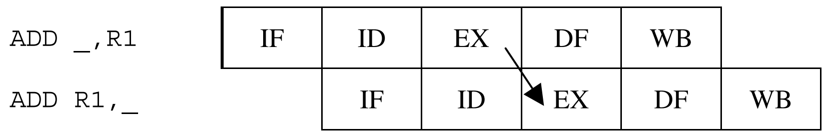


Figure 6-1: Forwarding (arrow) eliminates RAW hazard delay.

Emulator tests verify that the type of operand (immediate or register) show no impact on execution times. This indicates that at least two registers can be read in ID during a single clock cycle.
Tests also verify that in any case of RAW hazards among this group of instructions, forwarding eliminates delay cycles. We expect however RAW hazards on GR or PSW to cause delays in cases such as a load instruction followed by an arithmetic instruction.

❑ **EP2: Instruction reading one or two GR during ID stage. (Yes/No)**
❑ **EP3: Instruction writing to GR in WB stage. (Yes/No)***

Value range of EP3 will be redefined in section 6.1.8

❑ **EP4: Instruction reading PSW during ID stage. (Yes/No)**
❑ **EP5: Instruction writing PSW during EX stage. (Yes/No)**

6.1.3 Enable and disable interrupt instructions

Analysed instructions: `EI` and `DI`

Sets or resets bit in Program Status Word. The EFD shows execution terminates after EX stage. This is also an indication that PSW writing occurs during EX stage. Emulator tests show no extra delays at any time due to RAW hazards on PSW.

6.1.4 System Register load and store instructions

Analysed instructions: `LDSR` and `STSR`

Uses the Master Pipeline. The EFD shows normal execution in five clock cycles without causing any pipeline delays for the next instruction. Moving system registers (SR) to and from general registers may cause delay cycles. The User's Manual claims that such delays occur if system registers EIPC or FEPC are loaded by Load System Register (`LDSR)` and then immediate stored to a GR by Store System Register (`STSR`). However, our emulator tests do not show these exceptions to cause any delays at all.

❑ **EP6: Instruction reading SR, other than PSW. (Yes/No)**
❑ **EP7: Instruction writing SR, other than PSW. (Yes/No)**

6.1.5 Store and load instructions

Analysed instructions: `LD`, `ST`, `SLD`, and `SST`

32-bit Store (`ST`), 16-bit Short format Store (`SST`) and 32-bit Load (`LD`) all execute their first three stages in the Master Pipeline. The memory address is probably calculated in the EX stage. Memory access is then performed in the MEM stage of the Additional Pipeline. Total execution time is expected to be five clock cycles in isolation since we have assumed that data memory accesses take a single clock cycle. Executing flow of `LD`, `ST`, and `SST` is shown in Figure 6-2a.
We also assume that the size of accessed data does not impact access times, as long as data alignment is respected. Tests in emulator confirm these assumptions.

The EFD indicates that a RAW hazard between `LD` and succeeding instruction will cause a single clock cycle delay. This is also verified by emulator tests.

16-bit Short format Load (`SLD`) executes in the four-stage Additional Pipeline. Address calculation is performed instantly during the ID stage. Total execution time in isolation is thereby only four clock cycles. Executing flow of `SLD` is shown in Figure 6-2b.

IF	ID	EX	MEM	WB

Figure 6-2a: Execution flow of LD, ST, and SST.

IF	ID	MEM	WB

Figure 6-2b: Execution flow of SLD.

Since `SLD` uses the MEM stage one cycle "in advance", forwarding normally takes care of any RAW hazard, eliminating extra delays. However, if the preceding instruction also executes in the MEM stage, a structural hazard can occur and delay `SLD` memory access. This has impact on total execution time if a RAW hazard occurs between `SLD` and it's succeeding instruction. Such scenario is shown in Figure 6-3.

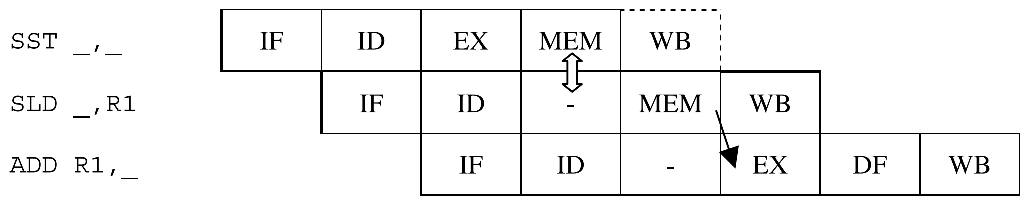


Figure 6-3: SST and SLD conflict in MEM stage (block arrow). ADD becomes one cycle delayed.

Emulator experiments verify the normal `SLD` execution time given by the User's Manual. Testing also confirm the `SLD/ADD` RAW delay shown in picture.

❑ **EP8: Instruction accessing memory in MEM stage. (Yes/No)***

Value range of EP8 will be redefined in section 6.1.10

6.1.6 Unconditional jump/branch instructions

Analysed instructions: `JR`, `JARL`, `JMP`, and `BR`

Jumps and branches change instruction flow by overwriting Program Counter (PC) unconditionally. This behaviour always causes a pipeline execution delay.

The EFD shows the instructions as terminated after the ID stage, implying that the PC is changed during execution of this stage. The succeeding instruction is also invalidated in its IF stage during this clock cycle and the branch destination instruction is fetched and executed in the IF stage during the next cycle. A static branch delay of a single clock cycle is thereby derived from the EFD, and emulator tests verify this for `JR, JARL` and `BR`.

However, emulator tests show the static branch delay of `JMP` to be two clock cycles. This corresponds well to Table 5-10 of the User's Manual while the Execution Flow Diagram shows only a single delay cycle, i.e. the documentation is inconsistent. We assume that `JMP` modifies PC during execution of the EX stage, which is not shown in the EFD. This may also indicate that `JMP` execute in the Master Pipeline while `JR`, `JARL` and `BR` use the Additional Pipeline.

- ❑ **EP9: Instruction modifying PC. (Uncond. with 1 cycle static delay/ Uncond. with 2 cycle static delay)***

Value range of EP9 will be augmented in section 6.1.7

6.1.7 Conditional branch instructions

Analysed instructions: all variants of `Bcond`

Conditional branch instructions (`Bcond`) read PSW and PC. If the PSW satisfies the branch condition, then the displacement given by the operand is added to PC value and written back to the PC. All these operations are performed in the ID stage in a single clock cycle and the `Bcond` instruction terminates after this. During this cycle, the succeeding instruction in the instruction memory executes in the IF stage. All `Bcond` execute in the Additional Pipeline according to the User's Manual.

The EFD shows two types of execution scenarios: branch not taken and branch taken.

Branch not taken: Since the branch condition is not satisfied, the succeeding instruction is valid and can continue execution. No branch delays occur. The `Bcond` simply becomes a `NOP`, except for reading PSW.

Branch taken: Similar to unconditional branch. The succeeding instruction is invalidated and the change of PC causes a branch delay of a single clock cycle.

If an instruction immediate preceding `Bcond` writes to PSW (e.g. an addition instruction), `Bcond` is delayed in the IF stage for one extra clock cycle due to a RAW hazard on PSW. This happens only when `Bcond` eventually decides branch taken. This scenario is shown in Figure 6-4. When branch not is taken, correct instruction is already executing and no pipeline delays occur.

We believe this execution behaviour to be reasonable, but it is not correctly documented in the User's Manual. Table 5-10 of the User's Manual indicates no delay for the PSW RAW-delay scenario, which is completely wrong.

Emulator tests verify one and two cycle delays for branch taken (depending on PSW read/write) and no delays for branch not taken scenarios, regardless of PSW read.

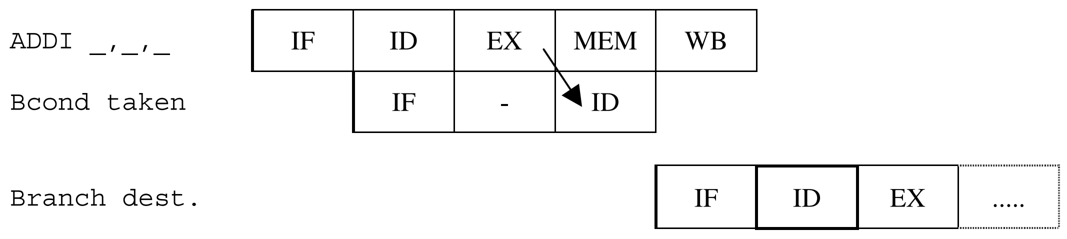


Figure 6-4: `ADDI` *writes PSW in EX stage. The branch destination instruction becomes delayed one extra cycle. The example assumes that the branch destination instruction is a 16-bit instruction or a word-aligned 32-bit instruction.*

We augment the value range of EP9:

- **EP9: Instruction modifying PC.** **(Uncond. with 1 cycle static delay/ Uncond. with 2 cycle static delay/ Cond.)**

6.1.8 Multiply instructions

Analysed instructions: `MULH`, `MULHI`, `MUL`, and `MULU`

All multiply instructions execute in two consecutive and fully pipelined EX stages. These stages are not shown in the general pipeline configuration figure in the User's Manual, but we assume that the second EX stage can be viewed as corresponding to the DF stage of Master Pipeline.
Emulator tests confirm that RAW hazards between multiply instructions and their succeeding instructions cause a delay of a single clock cycle. An example is shown in Figure 6-5.

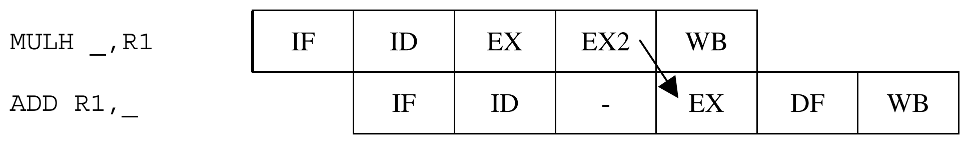


Figure 6-5: Execution of `MULH` *with special EX2 stage. RAW hazard on R1 causes a single cycle delay of* `ADD`*. (Arrow shows forwarding)*

32-bit `MUL` and `MULU` potentially write to two GR. Emulator tests show that these instructions always cause a delay clock cycle, independent of any data hazard. This delay is not shown in the Execution Flow Diagram but indicated in Table 5-10 of the User's Manual.

- **EP10: Instruction execution in EX stage more than once.** **(Number of EX stage cycles)**

Redefinition of EP3 value range:

- **EP3: Instruction writing to GR in WB stage.** **(Number of GR's)**

6.1.9 Divide instructions

Analysed instructions: `DIVH`, `DIV`, and `DIVU`

Like other arithmetic instructions, divide instructions execute in the Master Pipeline. The execution loops in the EX stage, 33 loops for unsigned divide instructions and 34 for signed. All documentation claims a static pipeline delay of 33 and 34 clock cycles for next executing instruction. This is confirmed by emulator tests.

6.1.10 Bit manipulation instructions

Analysed instructions: `SET1`, `CLR1`, and `TST1`

This set of instructions is typical for microcontroller chips, but not present on desktop RISC designs. Their purpose is set or reset specific bits in data memory. The operation requires the instructions to loop from MEM stage back to EX stage during execution.

EFD shows that `SET1`, `CLR1` and `NOT1` perform two rounds in EX and MEM stage and then terminate, while `TST1` terminates immediately after executing the second EX stage since it does not have to write back to memory.

The succeeding instruction is always delayed for two clock cycles due to the extra execution round. `SET1`, `CLR1` and `NOT1` will cause

The second execution in MEM stage for `SET1`, `CLR1` and `NOT1` will also cause a structural hazard if the next instruction is a `SLD`. This will have impact on total execution time if a RAW hazard occur between the `SLD` and its succeeding instruction. (Similar to the scenario described in section 6.1.6) This is not a fact for `TST1` since it do not execute twice in MEM stage.

Emulator tests verify timings including delays for all scenarios above.

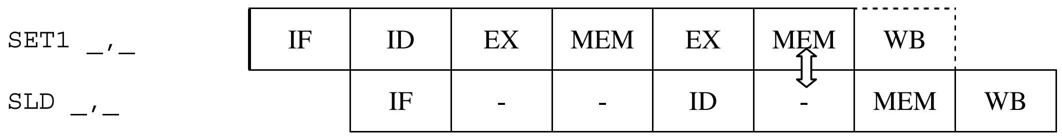


Figure 6-6: Execution of SET1 followed by SLD. The two first cycle delays of SLD are statically delays, i.e. independent of type of instruction following SET1, but the third cycle delay is caused by a structural conflict on MEM stage (block arrow).

- ❑ **EP11: Instruction executes EX stage after MEM stage (Yes/No)**

Redefinition of EP8 value range:

- ❑ **EP8: Instruction accessing memory in MEM stage. (Number of clock cycles)**

6.1.11 TRAP and RETI

Analysed instructions: `TRAP` and `RETI`

The execution of the `TRAP` instruction is shown in the Execution Flow Diagram as a six-stage pipeline with two successive ID stages. The shown usage of two ID stages is hard to interpret. It could mean that the ID stage of the Master Pipeline is used twice, but also that execution is performed in the ID stages of both Master and Additional Pipeline.

The EFD also shows execution termination after WB stage. This is strange since `TRAP` only read and write system registers, ending by writing PC, and the documentation indicates that all other instructions performing system register access does this in ID or EX stage.

`RETI` works as a return instruction and restores PC and PSW from two system registers. Execution is described in the EFD with a similar six-stage pipeline as `TRAP`. Execution is terminated after EX stage, however.

Both `TRAP` and `RETI` cause a two-cycle delay to the target instruction according to all documentation. This is confirmed by emulator tests. Emulator tests where adjacent instructions manipulate the system registers render no additional delays.

6.1.12 CALLT and CTRET

Analysed instructions: `CALLT` and `CTRET`

`CALLT` is a table indirect branch instruction. Its execution behaviour, as given by the EFD, is very hard to interpret. Execution is most likely initiated in the Additional Pipeline since a memory address is first calculated and then used in the MEM stage, which follows directly after the ID stage. Address calculation can be done in the ID stage of the Additional Pipeline even though the User's Manual generally claims that this pipeline is dedicated for `SLD` and branch instructions. After MEM stage, execution continues back to the EX stage of the Master Pipeline and then terminates.

All documentation in the User's Manual states a static delay of four clock cycles. Emulator tests render however a static delay of six cycles. Readings from emulator log-files indicate that `CALLT` suspends itself for two extra cycles. This may be related to MEM stage where internal ROM is accessed and not data memory (internal RAM). The User's Manual states internal ROM access time in the MEM stage to three clock cycles, which correspond well to this assumption. Similar additional delay also occurs when testing `SWITCH` (see below), which also access internal ROM in the MEM stage.

Since the MEM stage is used immediately after the ID stage, a structural hazard in the MEM stage will occur if, for example, a `ST` instruction precedes `CALLT`. The emulator confirms an extra delay cycle for `CALLT` in this scenario.

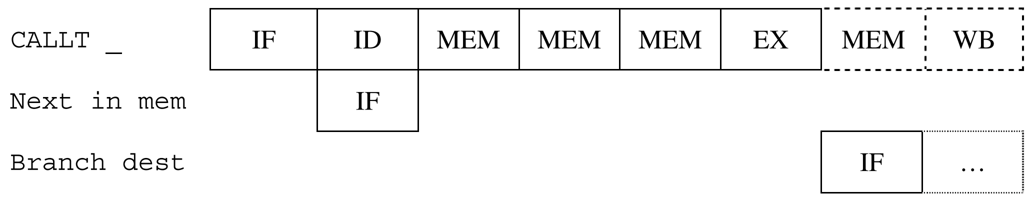


Figure 6-7: Execution of CALLT. Internal ROM is accessed in MEM stage (three clock cycles).

The indirect-branch instruction `CTRET` is used for returning from a `CALLT` call. `CTRET` reads the return address and PSW value from the system registers CTPC and CTPSW. It is not clear when these registers are read. Emulator tests show no delays if system registers CTPC or CTPSW are written by `LDSR` directly before CTRET executes.

`CTRET` itself always causes a static two-cycle delay, similar to `JMP`. As for `JMP`, this is correctly documented in Table 5-10 of the User's Manual, but not in the Execution Flow Diagram.

6.1.13 SWITCH

Analysed instruction: `SWITCH`

The table indirect branch instruction `SWITCH` executes an additional EX stage after MEM stage, similar to instruction `TST1`. `SWITCH` ends its operations by writing PC in this second EX stage and then terminates. This causes a static delay of four clock cycles according to all documentation in the UM. Since internal ROM is accessed in MEM stage, an additional pipeline delay of two clock cycles occurs when testing `SWITCH` execution time in the emulator.

6.1.14 Move Word instruction

Analysed instruction: `MOV (imm32)`

The `MOV imm32` instruction transfers a 32-bit immediate value, given by its operand, to a GR. Since this instruction is 48 bits long and the IF stage only handles 32 bits during a single clock cycle, a pipeline delay must occur. The EFD describes execution as taking two cycles in EX stage, thereby causing the next instruction to be delayed for one cycle. We can not interpret this diagram any further but the execution times, including delay cycle, given by the EFD and Table 5-10 of the User's Manual are confirmed by emulator tests. Assuming a delay in both IF and EX stage gives one delay cycle too much. No other delays, caused by adjacent instructions or data hazards, are found in emulator tests.

6.1.15 PREPARE and DISPOSE

Analysed instructions: all variants of `PREPARE` and `DISPOSE`

These are another example of complex instructions typical for microcontroller architectures. They are normally used when entering or returning from sub-routine calls, for saving and restoring register contents on a memory stack.

The instruction length for `PREPARE` varies from 32 to 64 bits depending on instruction operands.

The isolated execution times for PREPARE and DISPOSE is always dependent upon the operands, typically the numbers of GR to be saved or restored.

PREPARE reads the contents of up to 12 GR and writes this to memory. Other registers, such as Stack and Element Pointer (sp, ep) are also manipulated. For all these operations, the EFD shows only a looping behaviour in MEM stage during execution. We can not analyse this execution behaviour any further since the documentation and the emulator test traces do not provide us with enough detailed information. This also embraces the execution behaviour of DISPOSE.

Emulator tests confirm execution times given in the User's Manual for all PREPARE and DISPOSE flavours. These tests also show no indications for additional pipeline delays due to adjacent instructions.

6.1.16 HALT

NOT analysed instruction: HALT

The HALT instruction is not a subject of analysis since its "execution time" is completely arbitrary.

6.2. Analysis of Parallel Execution Scenarios (Dual Issue)

A dual issue scenario may occur when two adjacent 16-bit instructions are available to be processed in the IF stage simultaneous, e.g. fetched together by the 32-bit prefetch unit. The instructions must also execute separately in the Master and Additional Pipeline.

Dual issue is not well documented in the User's Manual. There are no clear specifications of possible dual issue pairs or under which circumstances they are valid and can execute in parallel. Only a single Execution Flow Diagram in the User's manual shows an example of a dual issue scenario, with an ADD and an unspecified "branch" instruction as dual issue pair. Our analysis, including emulator tests, indicates that this EFD is only partially correct and only if the "branch" instruction is the instruction BR. Furthermore, the "branch destination instruction" will in this case be delayed one more clock cycle than indicated in the EFD.

The given pipeline configuration in the User's Manual (reproduced in Figure 5-1) indicates SLD, Bcond instructions and BR as candidates for second instruction in dual issue pairs. We used emulator tests to search for possible companions, and we found four cases of dual issue scenarios.

6.2.1 Dual issue case 1: * / SLD

By testing each 16-bit instruction together with SLD we found that all arithmetic and logical operation instruction trigger a dual issue scenario. This includes 16-bit multiply instruction MULH and also NOP. Divide instruction DIVH is not qualified, probably due to the in-order issue/execution requirement of V850 family (see chapter 5: Pipeline Analysis), and neither are any branch/jump instructions or memory accessing instructions (i.e. SST and SLD).

We also discovered some very strange execution behaviours when valid dual issue pairs are repeated in our test codes. The instructions of the second pair and all subsequent pairs become separately issued one instruction per each clock cycle. By experimenting, we found that some instructions enable a new pair to be dual issued if such an instruction executes before the dual issue pair. Another group of instructions shares the property of disabling an already enabled dual issue scenario if executed after enabling instruction but before the dual issue pair.

These two instruction groups are very heterogeneous, i.e. instructions with very different semantics and normal execution properties. Thus we have not discovered the common factors that cause these behaviours. Instead, we strongly suspect this to be symptoms of emulator incorrectness, or plain bugs in the core pipeline.

To be consistent, we have added enabling and disabling `*/SLD` dual issue to our list of execution parameters.

- ❑ **EP12: Instruction enables `*/SLD` dual issue. (Yes/No)**
- ❑ **EP13: Instruction disables `*/SLD` dual issue. (Yes/No)**

6.2.2 Dual issue case 2: `* / Bcond`

`Bcond` instructions often become delayed due to PSW RAW hazards, which was found earlier. Emulator tests show clearly that instructions writing PSW can never be dual issued with `Bcond`. This holds even if branch is not taken. Dual issue may also be disabled if the immediately preceding instruction writes PSW. This invalidates the `ADD/Bcond` example given in the User's Manual.

Emulator experiments also show that `SST` and `SLD` are valid dual issue instruction together with `Bcond`. This is not expected for `SLD` since the User's Manual claims that `SLD` share the ID stage of the Additional Pipeline with `Bcond`.

6.2.3 Dual issue case 3: `* / BR`

`BR` is not sensitive to PSW accesses. Emulator tests show that all other 16-bit instructions can be dual issued together with `BR`. Exceptions are `DIVH` and branch/jump instructions. `SLD` is once again valid, which is remarkable.

6.2.4 Dual issue case 4: `MOV / *`

When examining the assembler code from several benchmark programs and real world applications, all compiled with the IAR compiler, we noticed a quite common instruction sequence:

```
 MOV  <reg1>, <reg3>
 ADD  <reg2>, <reg3>
```

The V850 family lacks a three-address addition instruction, i.e. that adds content from register #1 and register #2 and writes result to register #3 without corrupting registers #1 and #2. The sequence above mimics the behaviour of such a three-address `ADD`.

Running this sequence in the emulator render a surprising dual issue scenario. The `ADD` simply disappears from the instruction trace. We can not tell if this is a totally undocumented chip feature where the two instructions perhaps are translated into a "virtual" three-address `ADD`, or if it just is a very odd emulator bug.

Emulator tests show that operand relations must be as in the sequence above. This dual issue scenario also holds for all other 16-bit arithmetic and logical instruction variants with register read and write-semantic identical to `ADD reg, reg`.

This concludes the dual issue analysis. The result is presented in Appendix B: Table B-1. In this table, each 16-bit instruction is classified according to how they participate in valid dual issue scenarios. The dual issue classification is also added to the list of execution parameters.

- ❑ **EP14: Dual issue classification**. **(Classification number)**

6.3. Analysis of Instruction Execution Interference.

Execution scenarios where an executing instruction interferes directly with a third or even fourth instruction without interfering with the intervening instruction are not described or clearly indicated in the V850E documentation. If such a scenario exists, then it probably must be included in our test suite, rendering more complex test code patterns. Otherwise, test code patterns can be simplified and shortened, since they only have to be designed with regard to instruction interference between adjacent instructions.

6.3.1 Single Issue Scenarios

We first analyse scenarios where each instruction is issued individually.

We make the following fundamental assumptions:
- We have a sequence of three arbitrary, consecutively execution instructions.
- Data memory access time in MEM stage is one clock cycle.
- Instruction memory access time in MEM stage (reading data from instruction memory) is three clock cycles.
- Static pipeline delay caused by `JMP` and `CALLT` is three clock cycles.
- All other static pipeline delays shown in the Execution Flow Diagrams (EFD) of the User's Manual are correct, as confirmed by our emulator tests.
- Reading/writing PSW or other SR does never cause RAW hazard delays. Exception: `Bcond` reading PSW.

6.3.1.1 Structural Conflict Interference

Assume that the second instruction finishes its ID stage at time T, the EFD's shows that all instructions with long execution times always will cause the second instruction to be delayed, presumably in IF stage. Hence, the first instruction has at most two stages left to execute at time T:
- DF and WB stages of Master Pipeline.
- MEM and WB stages of Additional Pipeline.
- A second EX stage (EX2) and probably WB stage of Master Pipeline. (Multiply instructions only).

First instruction will then always enter WB at T+1 and terminate at T+2. Since we assume single issue, third instruction can not finish ID stage before T+1 and never enter DF, MEM or EX2 before T+2. Only exception is `SLD`, which will enter MEM at T+1 and WB at T+2.

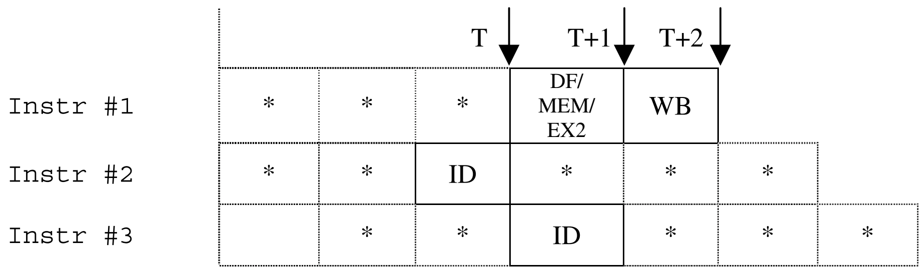


Figure 6-8. Instruction #2 finish ID stage at time T. Instruction #3 finish ID stage at earliest possible time, T+1.

`CALLT` and `SWITCH` access instruction memory and may require several MEM stage cycles. Assuming `CALLT` or `SWITCH` as first instruction, the EFD indicates that such MEM stage loops are executed during the pipeline delay of second instruction when it is in IF stage. Thus `CALLT` or `SWITCH` never directly delays a third instruction.

`TRAP` and `RETI` have an extra ID stage. Assuming `TRAP` or `RETI` as the second instruction, they always cause the third instruction to stall rendering the third instruction to enter DF, MEM or EX2 no earlier than at T+4. Exception is SLD, which enters MEM at T+3 in this scenario. Since the first instruction terminates at T+2, it can not interfere directly with the third instruction.

6.3.1.2 Data Conflict Interference

Figure 6-8 shows that the first instruction has finished its data calculation no later then at time T+1. It is also clear that the third instruction never requires this data before time T+1. Hereby, forwarding can eliminate data conflict interference.

We know that `Bcond` can be delayed if preceding instruction write PSW. However, assuming Bcond as the third instruction, `Bcond` will never read PSW until after time T when it enters the ID stage. There are also no scenarios where the first instruction writes PSW after time T. Thus, the first instruction can never interfere directly with `Bcond` in this scenario.

Conclusion: There are no single-issue scenarios where data or structural conflicts cause the first instruction interfere directly with a third instruction without interfering without intervening instructions. This means that test code patterns with single-issue scenarios only have to be designed with regard to instruction interference between adjacent instructions.

6.3.2 Dual Issue Scenarios

We now look at the dual issue scenarios identified above. In these cases, the second or third instruction is issued together with preceding instruction, rendering an earlier execution of each pipeline stage.
As mentioned before (chapter 5: Pipeline Analysis), dual issue scenarios imply that the logical issue order between two instructions is arbitrary. Thus, the impacts of structural or data conflicts will become equal to scenarios where the third instruction is issued and executed immediately after the first instruction, with no intervening instruction.

Analysing the dual issue cases gives some typical scenarios where the first instruction interferes directly with third instruction.

Interference delay is measured for each scenario by emulator tests. Emulator results are presented as trace logs. Timestamps indicate that dual issue pairs are sometimes broken up. Other examples show that the dual issue pair is delayed as a pair.

From Dual issue case 1:

Assembler code:

```
LD.W     0[ep], R10
ADD      R11, R12
SLD.W    R13, 4[ep]          (Dual issue pair)
```

Manual analysis:	Emulator trace:		Conclusion:
Structural hazard in MEM stage between `LD` and `SLD`.	Instr.	Timestamp	`ADD` and `SLD` are issued separately.
	`LD`	C	
	`ADD`	C+1	
	`SLD`	C+2	

Assembler code:

```
MULH    R10, R11
SLD.W   0[ep], R12          (Dual issue pair)
ADDI    1, R11, R13
```

Manual analysis:	Emulator trace:		Conclusion:
RAW hazard on R11 between MULH and ADDI.	Instr.	Timestamp	MULH/SLD are issued together and delays ADDI by one cycle (from MULH write).
	MULH/SLD	C	
	ADDI	C+2	

From Dual issue case 2:

Assembler code:

```
SST.W   R10, 4[ep]
BV      <label>   ;Bcond not taken     (Dual issue pair)
SLD.W   0[ep], R11
ADD     R11, R12
```

Manual analysis:	Emulator trace:		Conclusion:
Structural hazard in MEM stage between SST and SLD. RAW hazard on R11 between SLD and ADD is normally not causing execution of ADD to be delayed.	Instr.	Timestamp	SST is causing SLD to access memory one cycle later than usual. ADD becomes one cycle delayed.
	SST/BV	C	
	SLD	C+1	
	ADD	C+3	

Assembler code:

```
ADDI    1, R10, R11
SST.W   R12, 0[ep]
BV      <label>  ;Bcond not taken     (Dual issue pair)
```

Manual analysis:	Emulator trace:		Conclusion:
RAW hazard on PSW between ADDI and Bcond. This scenario was mentioned in unconditional branch analysis.	Instr.	Timestamp	SST and BV are issued separately.
	ADDI	C	
	SST	C+1	
	BV	C+2	

From Dual issue case 3:
No scenarios can be found since BR always causes an additional single cycle delay. Hence, the presence of BR will not be completely overlapped, and the instruction preceding BR will not interfere with the instruction succeeding BR.

From Dual issue case 4:

Assembler code:

```
LD.W     0[sp], R10
MOV      R11, R12
ADD      R10, R12          (Dual issue pair)
```

Manual analysis:	Emulator trace:		Conclusion:
RAW hazard on R10 between LD and ADD	Instr.	Timestamp	MOV/ADD are issued together but delayed one cycle. Seems like MOV/ADD cannot be separated.
	LD	C	
	MOV/ADD	C+2	

This is an even more extreme scenario invoking two consecutive dual issue pair. First instruction interfere now directly with forth instruction:

Assembler code:

```
MULH     R11, R10
SLD.W    0[ep], R12        (Dual issue pair)
MOV      R13, R14
ADD      R10, R14          (Dual issue pair)
```

Manual analysis:	Emulator trace:		Conclusion:
RAW hazard on R10 between MUL and ADD.	Instr.	Timestamp	MULH/SLD are issued together. MOV/ADD are also issued together but delayed one cycle.
	MULH/SLD	C	
	MOV/ADD	C+2	

Conclusion: The examples above show that there are several dual issue scenario where data or structural conflicts cause a first instruction to interfere directly with the third or forth instruction, without interfering with the intervening instructions. Test code patterns including dual issue scenarios must be designed with regard to such instruction interference.

6.4. The Equivalence Classes

Each instruction and instruction variant is applied to all found instruction execution parameters and instructions with equal set of (EP, value)-tuples are classified together. This classification is in Appendix B: Tables B-2, B-3, and B-4.

An instruction is selected arbitrary from each equivalence class and included in the reduced instruction set, which holds all instruction used in our test cases. The selected instructions are in Appendix B: Table B-5.

7. Our Test Cases

This chapter describes the structure and design of our test suite. All individual test code patterns are in Appendix A: Test Case Specifications.

A simple solution to improve the measurement and comparison accuracy is also given.
We conclude by briefly describing the framework and requirements regarding emulator and simulator test program generation.

7.1. Test Case Families

We used the instruction grouping of the Execution Flow Diagrams of the User's Manual as the main structure of our instruction set analysis. This grouping is also used for deriving the basic test case families and structure of the final test suite.

An exception is `CMOV`, which was given a dedicated family. Even though analysis and emulator tests showed `CMOV` not to be sensitive to adjacent instructions, `CMOV` performs many complex operations during very few clock cycles. It is not trivial for the simulator to model all these operations correctly. We need several test cases to verify the simulator model of `CMOV`; hence it is given a dedicated test case family. Test case families for each dual issue case were also inserted.

The following test case families were derived:

Name:	Targeting:
F0	`NOP`
F1	Arithmetic, logical etc. instructions with well-behaving Master Pipeline execution
F2	Load/Store instructions
F3	Multiply instructions
F4	Divide instructions
F5	`MOV imm32`
F6	`CMOV`
F7	Bit manipulation instructions
F8	System Register Load/Store instructions
F9	Simple Jump/Branch instructions including `Bcond`
F10	Dual Issue, Case 1
F11	Dual Issue, Case 2
F12	Dual Issue, Case 3
F13	Dual Issue, Case 4
F14	`PREPARE` and `DISPOSE`
F15	`SWITCH`
F16	`CALLT` and `CTRET`
F17	`TRAP` and `RETI`

7.2. Test Code Patterns

The derivation of test case patterns has mostly been based on the results from the preceding hardware analyses. The test patterns can roughly be divided into two scenario types.

- Possible instruction interference scenarios. Includes RAW and structural hazards and also misalignment scenarios.
- "Nice" scenarios. No hazards. Only static pipeline delays are expected or no delays at all.

The simulator should be tested with as little bias as possible, and thus patterns with possible instruction interference scenarios are included, even if all documentation and emulator tests show that no delays are expected to occur (due to data forwarding). Such test case scenarios can indicate whether forwarding etc. is modelled correctly in the simulator

In “nice” scenarios, data hazards such as RAW hazards are avoided. These tests act as a complement to the interference scenario test cases. The “nice” scenarios may show that the “basic” execution modelling for each instruction is correct while, interference scenarios give the exceptions to the fluent-flow of instructions. “Nice” scenario test cases also indicate that the simulator correctly models the static pipeline delays.

Figure 7-1 shows an example of a test case with a possible instruction interference scenario and the complementary “nice” scenario.

```
{                                {
 DIVH R15, R16                    DIVH R15, R16
 ADD  R16, R17                    ADD  R17, R18
)                                }
```

Figure 7-1. Test code patterns F4_1 (RAW hazard on register R16) and F4_2 (no hazards).

The test code patterns are typically two or three instructions long and repeated 500 or 1000 times in the final test code programs. All pattern instances are executed consecutively, even for patterns including branch instructions. Hence, branching instructions are allowed only at the end of a pattern.

To control execution flow when testing `Bcond,` we use `BNV` (“branch if no overflow”) for branch taken and `BV` (“branch if overflow”) for branch not taken. Figure 7-2 shows a `Bcond` code pattern where branch is taken. “Overflow” is chosen since it is easy to control. To guarantee these conditions when executing code on the emulator, registers are initialised and used in such manner that overflow never occur during execution.

```
{
 NOP
 ADDI 1, R15, R16
 BNV  Label<I>
Label<I>:
}
```

Figure 7-2. Test code pattern F9_4. A NOP is inserted to eliminate misalignment scenarios after branch is taken. <I> is translated to current pattern sequence number in final test program.

7.3. Improving Measurement and Comparison Accuracy

In chapter 3: Method Overview, we specified some problems concerning measuring and comparing execution time on the simulator and the emulator.

We solve the problem of uncertain initial state of the emulator and the simulator by inserting a sequence of six `NOP` instructions in the beginning of each test code. We assume that flushing the pipeline with this number of `NOP`’s is sufficient to ensure a stable and “clean” initial execution environment, i.e. a *warm pipeline* with a known state.

As for the problem of finding the state of test code termination, our solution benefits from the V850 in-order issue constrains. Our analyses on dual issue scenarios have shown that in-order issue may not be strictly implemented in the V850E CPU core. However, by also ending the test code with a sequence of six `NOP` instructions, we may assume that every instruction of the test code has terminated when all ending `NOP`’s has been issued and the measuring timer is stopped.

By using identical sequences of 6 + 6 `NOP` instructions as wrappings for every test code, we obtain that the test code can be equivalently measured on the simulator and on the emulator. The execution time result for an test invoking only the wrapping sequences, i.e. the twelve `NOP`'s, can be subtracted from the timing results for the ordinary test codes in order improve the comparison accuracy.

By always subtracting the total execution time for the twelve `NOP`'s, we can say that we do not measure and compare absolute execution times (T). Instead, we compare *changes* in execution time (ΔT) due to insertion of the code pattern sequences. We believe these values to be more stable and improve the accuracy of execution time comparison since any difference in how the emulator and the simulator start and stop their timers is removed.

8. Validation Test Results

This chapter presents the results from comparing the test code execution times produced by the emulator to the execution times calculated by the original simulator and its improved versions. These results are shown by comparing the execution times for a set of benchmark programs, between the emulator and each simulator version.

8.1. Validation and Debugging Iterations

Initial validation was performed on a previously untested (regarding timing) version of the simulator, denoted here as simulator version 0.1.
It was assumed that this version contained several faults in form of erroneous abstraction, modelling and implementation bugs. Comparing simulator version 0.1 results to the validation reference, given by the emulator, showed the simulator miscalculating execution time in 62 % of the test cases.
Table 8.1 shows some examples of analysis of the comparison results together with the actual bugs found in the simulator implementation.

We do not list all simulator bugs, why they occurred and how they where fixed. It is not the intention of this thesis to cover simulator design and implementation issues. The important is that all bug where discovered directly or indirectly from analysing the results from execution our validation test programs, and we want to give a flavour of the kinds of bugs found. Many simulator errors found were due to the fact that the simulator was based on an earlier version of the V850E documentation that did not detail the Additional Pipeline.

Comparison outcome	Debugging
Unexpected addition of one extra clock cycle to every other pattern instances in many test cases, including some relatively trivial tests in family F1. All such test cases invoke a mix of 16 and 32-bit instructions and this indicates an incorrect modelling of the prefetch unit. This is a very serious fault, which was not taken into consideration when the test cases where designed. F1_7, F1_8 and F1_9 were added to clarify this error.	All misaligned 32-bit instructions take two cycles to fetch in the simulator, which is wrong, due to a logical error in prefetch simulation. Modelling bug.
Signed and unsigned divide instruction tests have mismatched execution times. (F4)	The simulator had swapped execution times for signed and unsigned division, due to a implementation bug
Most simple jump/branch instructions modelled with too long static delays. (F9)	All branch instructions are one cycle faster in new documentation. Major change to simulator structure.
One variant of the multiplication instruction, writing to two GR, was too optimistic simulated. (F3_3, F3_4)	The extra delay was not modelled since it was not stated in the documentation.
Several PREPARE and DISPOSE tests miscalculated. (F14)	PREPARE with three operands modelled too optimistic, DISPOSE with return too pessimistic. PREPARE not writing ep one clock cycle too long due to implementation bug.
Dual issue scenarios not implemented at all in the simulator (F10, F11, F12, F13)	Additional Pipeline not implemented. Major simulator design bug, due to old documentation.

Table 8-1. Examples of errors found on simulator version 0.1

After simulator debugging, we rerun all the test cases on the simulator, and analysed the errors again. This process was repeated until comparison outcomes showed that every significant divergence between simulator results and validation reference was eliminated.

Table 8-2 gives the numbers of correct and incorrect test cases for each simulator version. Note that as a result of debugging the simulator, some earlier correct test cases become incorrect. The reduction of incorrect test cases after debugging is also shown as a bar chart in Figure 8-1.

Sim. version	# Test cases	# Correct test cases	# Incorrect test cases	
			old	new
v 0.1	97	37	60	-
v 1.1	97	81	9	7
v 1.2	97	92	4	1
v 1.3	97	94	3	0

Table 8-2. Number of correct and incorrect test cases.

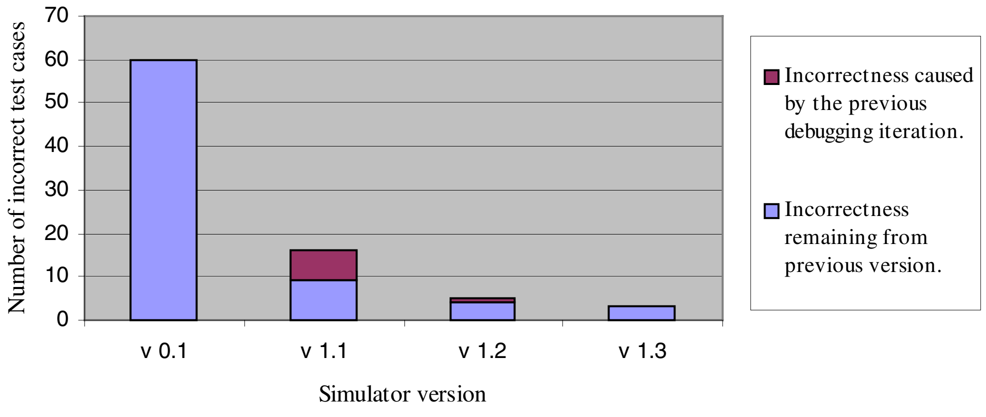


Figure 8-1. The reduction of incorrect test cases.

8.1.1 Omitted SLD Test Cases

There are some exceptions from the validation condition of perfect agreement. These test cases (F10_3, F11_3, and F12_3) include the instruction SLD and they render strange execution timing results when run on the emulator. It has been very hard to find an simulator model that produce the same results as rendered by the emulator, hence these test cases was omitted.

We strongly suspect that the instruction `SLD` suffer from hardware bugs, which may have effected the emulator results. Furthermore, we have during our work received information on suspected `SLD` hardware bugs from NEC in Japan and Germany. It seems that `SLD` does not check for write-after write hazards, which can occur, since `SLD` is run in parallel to the Master Pipeline.

8.2. Benchmark Evaluation of Simulator Improvement

To test the improvements of the simulator accuracy and measure the effectiveness of our method, we have executed a set of benchmark programs on the emulator and compared the execution times to the results of simulator version 0.1 and version 1.3. The benchmark programs are modified to execute with a static worst-case behaviour and have been used for WCET experiments [3]. The benchmarks are listed and described in Table 8-1.

We used a modified IAR V850E compiler to produce equivalent simulator and emulator test programs from each benchmark source code. The only difference is the output format, since the simulator has its own input format.

Table 8-2 shows resulting execution times and differences between the emulator and simulator version 0.1 and version 1.3. All execution times (*Total*) are given in clock cycles.

Program	Brief description
fibcall	Simple iterative fibonacci calculation. Calculate fib(30).
insertsort	Insertion sort on a reversed array of length 10.
matmult	Matrix multiplication of two 20 x 20 matrices.
duff	Using "Duff's device" to copy a 43 byte array.
fir	DSP algorithm; finite impulse response filter.
jfdctint	Discrete-cosine transformation on an 8x8 pixel block.

Table 8-1: Description of the benchmark programs.

Program	Emulator	Simulator v 0.1			Simulator v 1.3		
			Divergence			Divergence	
	Total	Total	Total	%	Total	Total	%
fibcall	325	286	-39	-12,0	313	-12	-3,7
insertsort	956	1123	167	17,5	1047	91	9,5
matmult	222236	239526	17290	7,8	231422	9186	4,1
duff	1094	1193	99	9,0	1083	-11	-1,0
fir	348105	323277	-24828	7,1	348097	-8	0,0
jfdctint	4686	5414	728	15,5	4893	207	4,4
			Average:	11,5		Average:	3,8

Table 8-2. Execution times for benchmarks programs.

The benchmark tests indicate a significant improvement in simulator accuracy. The final results from *fibcall*, *duff* and *fir* are satisfying. Emulator measurement conditions during these benchmark tests have not been as controllable as during validation tests, thus minor divergences in total execution time are probably unavoidable.

Final results of *insertsort, matmult* and *jfdctint* clearly show improvement but also that the accuracy still is unacceptably poor. This indicates that we have missed something significant in our validation analyses and tests, or it may even point out that our method may not be a feasible strategy for reaching acceptable cycle correctness in the simulator.

Closer investigations of emulator and simulator trace logs from the benchmark programs *insertsort*, *matmult*, and, *jfdctint* gave us some answers.

1. An additional valid dual issue case 4 (`MOV/ *`) scenario was discovered. Our specifications on dual issue case 4 were not correct. The analysis had discovered too few instruction combinations and the case restrictions were derived too hastily. This is clearly a problem when investigating totally undocumented hardware behaviours.

2. Our analysis and validation tests show that the multiply instructions `MUL` and `MULU`, which write two general registers, always cause an pipeline delay. This is not true if one or both of these registers is R0, which was not discovered during our analysis. This exception is also not documented and is probably due to the special property of R0 as "read as zero". I.e. treated as a special case, compared to other registers.

The benchmark comparison test results after modifying the simulator to handle these new specifications are given in Table 8-3.

Program	Emulator	Simulator v 1.4		
			Divergence	
	Total	Total	Total	%
fibcall	325	313	-12	-3,7
insertsort	956	939	-17	-1,8
matmult	222236	221822	-414	-0,2
duff	1094	1083	-11	-1,0
fir	348105	348095	-10	0,0
jfdctint	4686	4733	47	1,0
			Average:	1,3

Table 8-3. Execution times for benchmarks programs on emulator and simulator version 1.4

To judge what are acceptable divergences, we need to look at the *absolute* divergences ("Total") and not just at the *relative* divergences ("%"). A divergence of 10 clock cycles may be explained by measurement inaccuracy, in which case the real error might be much smaller. Indeed, the larger benchmarks show better accuracy (relative) than the smaller, supporting this theory.

The time frame of this work did not permit a closer investigation of the last set of divergences.

9. Conclusions

In this thesis, we have investigated a new strategy for validation and debugging of cycle-accurate CPU simulators. This validation method is based on comparing the execution time for small micro-benchmark programs between a CPU simulator and real hardware.

Our work has been in the form of a case study, in which a NEC V850E CPU simulator taken from the Uppsala/Paderborn WCET research project has been compared to a NEC V850E CPU emulator. The work has included studies of V850E documentation as well as analysis of the correlation between the hardware documentation and the real-world results given by the emulator.

From the results of the micro-benchmark programs, all derived from our hardware analysis, we have been able to significantly improve the cycle-accuracy of the simulator. Comparison tests invoking several larger benchmark programs shows that average incorrectness in cycle-accuracy has dropped from 11,5 % down to 1,3 %, and from 62% of tests yielding errors down to 3%.

We believe that these results indicate that our method of validation can be a valid strategy, at least regarding CPU architectures that are not too complex. Other enabling restrictions on the hardware complexity are simplified memory interface models with well-known memory access times.

During hardware analysis, we have encountered several obscurities in the V850E documentation and also several cases of divergence between the documentation and emulator tests results. These examples illustrate a significant problem for independent compiler and simulator vendors, since they are often limited to hardware documentation and emulators for deriving application specifications. The requirements for correct and detailed hardware documentation, provided by the hardware manufacturer, is definitely unavoidable.

10. References

[1] J.Engblom and A.Ermedahl. Overview of WCET Analysis System. Internal design document. Jan. 2000.

[2] B.Black and J.P.Shen. Calibration of Microprocessor Performance Models, Computer, Vol. 31, No. 5, May 1998

[3] J.Engblom, A.Ermedahl, and F.Stappert. Validating a Worst-Case Execution Analysis Method for an Embedded Processor. IEEE RTSS Work-in-progress session, Orlando, Florida, Dec. 2000

[4] R.H.Saavedra, R.S. Gaines, and M.J. Carlton. Micro Benchmark Analysis of the KSR1. Supercomputing T93, November 1993

[5] P.Puschner and R.Nossal. Testing the Results of Static Worst-Case Execution-Time Analysis. Proc. of RTSS98.

[6] Preliminary User's Manual, V850E/MS1 - Architecture, U12197EJ3V0UM00, NEC Corporation, Japan

[7] V850 C-Spy User Guide, First edition: January 1999, IAR Systems

[8] J.Engblom, A.Ermedahl, and F.Stappert. Specializing the Generic WCET System to the NEC V850E. Internal design document. Jan. 2000.

[9] J.L.Hennessy and D.A.Pattersson. Computer Architecture – a Quantitative Approach, Morgan Kuafman Publisher Inc, 2nd edition: 1996

11. Appendix A

11.1. Test Case Specifications

This list specifies every individual test code pattern. The patterns are organised into test case families. Most families target on closely related instruction where the relations are defined by execution properties associated with the instructions. Special Dual issue families target on separate dual issue conditions. In all other test case families, dual issue is avoided.
Test cases of initial F0 family are used to measure execution timing of test bed overhead.

Each test pattern has a short description for clarifying its characteristics.

Modification notes:

Modified 000515 ADDITIONAL: F1_7-10 and F3_7 for clarification purpose
Modified 000522 ADDITIONAL F9_12-13and F14_10 for clarification purpose
Forced to introduce MOVHI in combination with MOVEA to avoid MOV32.
Modified 000601 ADDITIONAL FAMILIES AF7, AF10, and AF13 for clarification purpose.

****
Family: NOP (extended with empty test case)
Number: F0
Target Instructions: NOP
Remark:
NOP instruction is tested to verify assumption of well-defined execution behavior.
****

F0_1: Empty test sequence. Only test code wrapping instructions are executed.
```
{
}
```

F0_2: Only NOP instructions
```
{
 NOP
}
```

****
Family: Arithmetic, logical etc. instructions with well-behaving Master Pipeline execution
Number: F1
Target Instructions: ADD (reg, reg), ADD (imm, reg), CMP, MOV (reg, reg), SXB, ADDI, DI
and MOVEA
Remark:
All instructions in these tests utilise Master Pipeline only, they individually fit well into a five – stage pipeline model, implying that no execution delays due to structural conflicts or data hazards are excepted.
****

F1_1: Only 32-bit instructions. No data hazards.
```
{
 MOVEA 1,R15, R16
 ADDI  1, R17, R18
}
```

F1_2: Only 32-bit instructions. Maximum data hazards

```
{
 ADDI 1, R15, R15
}
```

F1_3: Only 16-bit instructions. No data hazards.

```
{
 MOV  R15, R16
 ADD  1, R18
}
```

F1_4: Only 16-bit instructions. Maximum data hazards.

```
{
 ADD  R15, R15
}
```

F1_5: Mix of 16 and 32-bit instructions. No data hazards between adjacent instructions.

```
{
 DI
 MOV   R15, R16
 MOVEA 1, R17, R18
 ADD   R19, R15
 SXR   R17
 ADDI  1, R18, R16
 ADD   1, R15
 CMP   R17, R18
}
```

F1_6: Mix of 16 and 32-bit instructions. Maximum data hazards.

```
{
 DI
 MOV   R17, R16
 MOVEA 0, R16, R17
 ADD   R17, R17
 SXB   R17
 ADDI  1, R17, R17
 ADD   1, R17
 CMP   R16, R17
}
```

F1_7: Every other: one 16-bit and one 32-bit instructions. ADDITIONAL 000515

```
{
 ADD   R15, R16
 ADDI  1, R17, R18
 ADD   R15, R16
 ADDI  1, R17, R18
}
```

F1_8: Every other: two 16-bit and two 32-bit instructions. ADDITIONAL 000515

```
{
 ADD   R15, R16
 ADD   R17, R18
 ADDI  1, R15, R16
 ADDI  1, R17, R18
}
```

F1_9: one 16-bit followed by two 32-bit instructions, repeated. ADDITIONAL 000515

```
{
 ADD   R15, R16
 ADDI  1, R17, R18
 ADDI  1, R19, R20
}
```

F1_10: two 16-bit followed by one 32-bit instruction, repeated. ADDITIONAL 000515

```
{
 ADD   R15, R16
 ADD   R17, R18
 ADDI  1, R19, R20
}
```

****

Family: Load /Store instructions
Number: F2
Target Instructions: SLD.W, SST.W, LD.W and ST.W
Remark:
Note conceivable delays due to data hazards and structural conflict.

****

F2_1: 32-bit load. No data hazards.

```
{
 LD.W 0[sp], R15
 ADDI 1, R16, R17
}
```

F2_2: 32-bit load. Data hazard (R15).

```
{
 LD.W 0[sp], R15
 ADDI 1, R15, R17
}
```

F2_3: 16-bit short address mode load. No data hazards.

```
{
 SLD.W 0[ep], R15
 ADDI  1, R16, R17
}
```

F2_4: 16-bit short address mode load. Data hazard (R15).

```
{
 SLD.W 0[ep], R15
 ADDI  1, R15, R17
}
```

F2_5: 32-bit store. Data hazard (R17).

```
{
 ADD  R16, R17
 ST.W R17, 0[sp]
}
```

F2_6: Structural conflict (MEM stage) between 32-bit store and 16-bit short load. Data hazard between short load and addition instruction (R15). This combination should render an extra delay of the succeeding ADD instruction.

```
{
 ST.W  R15, 0[sp]
 SLD.W 0[ep], R16
 ADD   R16, R17
}
```

F2_7: Data hazard (ep). May delay 16-bit short load instruction?

```
{
 MOVEA 0, sp, ep
 SLD.W 0[ep], R16
 ADD   R16, R17
}
```

F2_8: Data hazard (sp). May delay 32-bit load instruction?
```
{
 MOVEA 0, ep, sp
 LD.W  0[sp], R16
 ADD   R16, R17
}
```

****
Family: Multiply instructions
Number: F3
Target Instructions: MULH, MULHI and MUL

Remark:
Note conceivable impact of data hazards. Note impact of instruction writing to two general registers.
****

F3_1: 32-bit. Writing to single GR. No data hazards.
```
{
 MULHI 1, R16, R17
 ADD   R18, R19
}
```

F3_2: 32-bit. Writing to single GR. Data hazard (R17).
```
{
 MULHI 1, R16, R17
 ADD   R17, R17
}
```

F3_3: 32-bit. Writing to two GR. No data hazards.
```
{
 MUL  R15, R16, R17
 ADD  R18, R19
}
```

F3_4: 32-bit. Writing to two GR. Data hazards (R16, R17)
```
{
 MUL  R15, R16, R17
 ADD  R16, R17
}
```

F3_5: 16-bit. Writing to single GR. No data hazards.
```
{
 MULH R15, R16
 ADD  R17, R18
}
```

F3_6: 16-bit. Writing to single GR. Data hazard (R16).
```
{
 MULH R15, R16
 ADD  R16, R17
}
```

F3_7: 32-bit. Writing to two GR. Immediate succeeding (and preceding) instruction do not write to any register. ADDITIONAL 000515
```
{
 MUL  R15, R16, R17
 NOP
}
```

****

Family: Divide instructions
Number: F4
Target Instructions: DIVH, DIVU and DIV

Remark:
Note how instructions with long execution block the entire pipeline. Note conceivable impact of writing to two GR and data hazards.

****

F4_1: Writing to single GR. No data hazards.
```
{
 DIVH R15, R16
 ADD  R17, R18
}
```

F4_2: Writing to single GR. Data hazard (R16).
```
{
 DIVH R15, R16
 ADD  R16, R17
}
```

F4_3: Writing to two GR. No data hazards.
```
{
 DIV  R15, R16, R17
 ADD  R18, R19
}
```

F4_4: Writing to two GR. Data hazards (R16, R17).
```
{
 DIV  R15, R16, R17
 ADD  R16, R17
}
```

F4_5: Writing to two GR. No data hazards. Unsigned.
```
{
 DIVU R15, R16, R17
 ADD  R18, R19
}
```

****

Family: MOV imm32
Number: F5
Target Instruction: MOV (imm32)

Remark:
Long 48-bit instruction. Separate tests of invoking memory alignment interference. Possible interference of data hazards is also tested.

****

F5_1: Memory alignment test. Only every other move instruction located on word alignment.
```
{
 MOV  55555555, R15
}
```

F5_2: Memory alignment test. Every move instruction located on word alignment.
```
{
 MOV  55555555, R15
 NOP
}
```

F5_3: No data hazards. Every move instruction located on word alignment.

```
{
 MOV  55555555, R15
 ADDI 1, R16, R17
 NOP
}
```

F5_4: Data hazard (R15). Every move instruction located on word alignment.

```
{
 MOV  55555555, R15
 ADDI 1, R15, R17
 NOP
}
```

****

Family: CMOV

Number: F6

Target Instruction: CMOV

Remark:

Conceivable impact of data hazards (PSW and GR) is tested.

Condition test: 0 = “if Overflow”. Will never be true.

****

F6_1: Data hazard on PSW, since preceding instruction is writing PSW. No other data hazards.

```
{
 ADD  R15, R16
 CMOV 0, R17, R18, R19
}
```

F6_2: No hazard on PSW, since preceding instruction does not write PSW.

```
{
 NOP
 CMOV 0, R17, R18, R19
}
```

F6_3: No data hazard on PSW. Data hazard (R17).

```
{
 CMOV 0, R15, R16, R17
 MOV  R17, R18
}
```

F6_4: Data hazard (R16).

```
{
 ADD   R15, R19
 MOV   R18, R16
 CMOV  0, R15, R16, R17
}
```

F6_5: No data hazard. ADD writes PSW.

```
{
 ADD   R15, R19
 MOV   R18, R19
 CMOV  0, R15, R16, R17
}
```

****
Family: Bit manipulation instructions
Number: F7
Target Instructions: SET1, TST1

Remark:
These instructions have a long execution and thereby block the pipeline. Data hazards of base address registers are tested. Also note impact of structural conflicts (MEM stage). Note that SET1 executes in MEM stage twice, TST1 only once.
****

F7_1: No data hazards or other conflicts.
```
{
 CLR1 R15, [sp]
 NOP
}
```

F7_2: No data hazards or other conflicts.
```
{
 TST1 R15, [sp]
 NOP
}
```

F7_3: Data hazards on address register (R16).
```
{
 MOV  sp, R16
 CLR1 R15, [R16]
 NOP
}
```

F7_4: CLR1 execute MEM stage twice, rendering a structural conflict (MEM stage) between CLR1 and SLD. Data hazard (R17) will cause ADD instruction to be delayed.
```
{
 CLR1  R15, [sp]
 SLD.W 0[ep], R17
 ADD   R17, R18
}
```

F7_5: TST1 only execute MEM stage once. Hence there will be no additional structural conflicts and ADD instruction will not be delayed.
```
{
 TST1  R15, [sp]
 SLD.W 0[ep], R17
 ADD   R17, R18
}
```

****
Family: System Register Load/Store instructions
Number: F8
Target Instruction: LDSR, STSR
Remark:
Tests if access to system registers require additional time.
****

F8_1: Fast "Download - Modify – Upload". Exception/Interrupt Program Status Word (EIPSW)

```
{
 STSR 1, R15
 MOV  R15, R16
 LDSR R15, 1
 NOP
}
```

F8_2: Continuos “Download – Upload – Download...” (EIPSW)

```
{
 STSR 1, R15
 LDSR R15, 1
}
```

F8_3: Continuos “Download – Upload – Download...” Exception/Interrupt Program Counter (EIPC). According to “V850E User’s Manual [1], chapter 8 p. 154”, each pattern transition in this test will cause some extra delay.

```
{
 STSR 0, R15
 LDSR R15, 0
}
```

****
Family: Simple Jump/Branch instructions
Number: F9
Target Instructions: Bcond (BV, BNV), BR, JMP, JARL, JR

Remark:
Note impact of condition hazard (fixed delays). Note delays when instruction writing PSW precede conditional branch instruction.
BV (branch if overflow) and BNV (branch if not overflow) is chosen as Bcond, in order to maintain full control of test execution.
A 16-bit instruction is also inserted first in sequences, to avoid execution delays due to misalignment.
All jumps are performed forward in test code.
****

F9_1: Jump relative

```
{
 NOP
 ADDI 1, R15, R16
 JR   label<I>
Label<I>:
}
```

F9_2: Jump register

```
{
 NOP
 MOV  Label<I>, R10    (MOV imm32)
 JMP  [R10]
Label<I>:
}
```

F9_3: Jump and Register Link

```
{
 NOP
 ADDI R15, R16
 JARL Label<I>, R10
Label<I>:
}
```

F9_4: Conditional branch (taken). Preceding instruction is writing PSW.

```
{
 NOP
 ADDI 1, R15, R16
 BNV  Label<I>
Label<I>:
}
```

F9_5: Conditional branch (not taken). Preceding instruction is writing PSW
```
{
 NOP
 ADDI 1, R15, R16
 BV   Label<I>
Label<I>:
}
```

F9_6: Conditional branch (taken). NO preceding instruction is writing PSW.
```
{
 NOP
 MOVEA 1, R15, R16
 BNV   Label<I>
Label<I>:
}
```

F9_7: Conditional branch (not taken). ADD instruction is writing PSW but not MOVEA.
```
{
 ADD   R15, R16
 MOVEA 1, R15, R16
 BV    Label<I>
Label<I>:
}
```

F9_8: Conditional branch (not taken). NO preceding instruction is writing PSW
```
{
 NOP
 MOVEA 1, R15, R16
 BV    Label<I>
Label<I>:
}
```

F9_9: Unconditional branch. Preceding instruction is writing PSW.
```
{
 NOP
 ADDI 1, R15, R16
 BR   Label<I>
Label<I>:
}
```

F9_10: Unconditional branch. NO preceding instruction is writing PSW.
```
{
 NOP
 MOVEA 1, R15, R16
 BR    Label<I>
Label<I>:
}
```

F9_11: Misalignment test. Misalignment of 32 - bit instruction in every other sequence.
```
{
 ADDI 1, R15, R16     (misalignment in every other seq.)
 BR   Label<I>
Label<I>:
}
```

F9_12: Jump register. ADDITIONAL 000522. Avoid MOV32.
```
{
 NOP
 MOVHI hwrd(Label<I>), R0, R10
 MOVEA lwrd(Label<I>), R10, R10
 JMP  [R10]
Label<I>:
}
```

F9_13: Jump register. ADDITIONAL 000522. Avoid MOV32. Avoid Data dependency (R10)
```
{
 NOP
 MOVHI hwrd(Label<I>), R0, R10
 MOVEA lwrd(Label<I>), R10, R10
 NOP
 JMP  [R10]
Label<I>:
}
```

****
Family: Dual issue case 1
Number: F10
Target Instructions: * / SLD.W

Remark:
Note impact of instruction preceding * / SLD pair (DD Condition Enabler/Disabler). Very unclear if this is an emulator bug or not.
****

F10_1: A NOP inserted to enable dual issue.
```
{
 NOP
 ADD   R15, R16
 SLD.W 0[ep], R17
}
```

F10_2: A NOP inserted to enable dual issue.
```
{
 NOP
 MULH  R15, R16
 SLD.W 0[ep], R17
}
```

F10_3: No such enabling instruction.
```
{
 ADD   R15, R16
 SLD.W 0[ep], R17
}
```

F10_4: No such enabling instruction.
```
{
 MULH  R15, R16
 SLD.W 0[ep], R17
}
```

F10_5: Enabling instruction inserted but not a valid dual issue scenario.
```
{
 NOP
 DIVH  R15, R16
 SLD.W 0[ep], R17
}
```

F10_6: Enabling instruction followed by a disabling instruction.
```
{
 NOP
 SST.T R16, 0[ep]
 MULH  R15, R16
 SLD.W 0[ep], R17
}
```

****
Family: Dual issue case 2
Number: F11
Target Instructions: */ Bcond

Remark:
Note impact of preceding instruction(s) writing PSW.
BNV chosen as Bcond instruction in order to render tests where branch is always taken.
A 16-bit instruction is inserted first in sequence to avoid misalignment execution delays.
****

F11_1: No preceding instructions write PSW.
```
{
 NOP
 MOV  R15, R16
 BNV  Label<I>
Label<I>:
}
```

F11_2: No preceding instructions write PSW.
```
{
 NOP
 SST.W R15, 0[ep]
 BNV   Label<I>
Label<I>:
}
```

F11_3: No preceding instructions write PSW.
```
{
 SLD.W 0[ep], R15
 SLD.W 0[ep], R16
 BNV   Label<I>
Label<I>:
}
```

F11_4: First instruction in dual issue pair is writing PSW.
```
{
 NOP
 CMP  R15, R16
 BNV  Label<I>
Label<I>:
}
```

F11_5: Instruction preceding dual issue pair is writing PSW.
```
{
 NOP
 ADDI 1, R15, R16
 MOV  R17, R18
 BNV  Label<I>
Label<I>:
}
```

****
Family: Dual issue case 3
Number: F12
Target Instructions: * / BR

Remark:
Note impact of preceding instruction(s) writing PSW (No impact at all?).
"Cross – branch data hazard" test.
A 16-bit instruction is inserted first in sequence to avoid misalignment execution delays.
Similar to Case 2 tests.
****

F12_1: No preceding instructions are writing PSW
```
{
 NOP
 MOV  R15, R16
 BR   Label<I>
Label<I>:
}
```

F12_2: No preceding instructions are writing PSW
```
{
 NOP
 SST.W      R15, 0[ep]
 BR   Label<I>
Label<I>:
}
```

F12_3: No preceding instructions are writing PSW
```
{
 SLD.W      0[ep], R15
 SLD.W      0[ep], R16
 BR   Label<I>
Label<I>:
}
```

F12_4: First instruction in dual issue pair is writing PSW
```
{
 NOP
 ADD  R15, R16
 BR   Label<I>
Label<I>:
}
```

F12_5: "Cross-branch data hazard test". Data hazard (R16) between each multiplication instruction.
```
{
 MULH R15, R16
 BR   Label<I>
Label<I>:
}
```

****
Family: Dual issue case 4
Number: F13
Target Instructions: MOV / *

Remark:
Note instruction format and data hazard as Dual issue Conditions. Not many tests implemented, since this behaviour has not been found in any V850E documentation and it is very unclear if it is a chip feature or an emulator bug.
****

F13_1: Conditions are satisfied.
```
{
 MOV  R15, R16
 ADD  R17, R16
}
```

F13_2: Data hazard condition is not satisfied.
```
{
 MOV  R15, R16
 ADD  R17, R18
}
```

F13_3: Operand format condition is not satisfied.
```
{
 MOV  R15, R16
 ADD  1, R16
}
```

****
Family: PREPARE and DISPOSE
Number: F14
Target Instructions: PREPARE, DISPOSE

Remark:
Note impact of instruction length (PREPARE only) and register list length (operand). Instruction to restore origin stack pointer value inserted .
****

F14_1: List - operand includes only one register.
```
{
 DISPOSE 0, {R20}
 NOP
 MOV  R7, sp
}
```

F14_2: List - operand includes eight registers.
```
{
 DISPOSE 0, {R20, R21, R22, R23, R24, R26, R27, R28}
 NOP
 MOV R7, sp
}
```

F14_3: List - operand includes eight registers. Invokes unconditional branch.
```
{
 NOP
 MOV R7, sp
 MOV   Label<I>, R15         (MOV imm32)
 DISPOSE 0, {R20, R21, R22, R23, R24, R26, R27, R28}, [R15]
 Label<I>:
}
```

F14_4: 32-bit instruction. List - operand includes only one register.

```
{
 PREPARE {R20}, 0
 NOP
 MOV R7, sp
}
```

F14_5: 32-bit instruction. List - operand includes eight registers.

```
{
 PREPARE {R20, R21, R22, R23, R24, R26, R27, R28}, 0
 NOP
 MOV  R7, sp
}
```

F14_6: 32-bit instruction. List - operand includes eight registers.

```
{
 PREPARE {R20, R21, R22, R23, R24, R26, R27, R28}, 0, sp
 NOP
 MOV  R7, sp
}
```

F14_7: 48-bit instruction. List - operand includes eight registers.

```
{
 PREPARE {R20, R21, R22, R23, R24, R26, R27, R28}, 0, 5555
 NOP
 MOV  R7, sp
}
```

F14_8: 64-bit instruction. List - operand includes only one register.

```
{
 PREPARE {R20}, 0, 55555555
 NOP
 MOV  R7, sp
}
```

F14_9: 64-bit instruction. List - operand includes eight registers.

```
{
 PREPARE {R20, R21, R22, R23, R24, R26, R27, R28}, 0, 55555555
 NOP
 MOV  R7, sp
}
```

F14_10: ADDITIONAL 000522. Variant of F14_3. Avoid MOV32. Avoid data dependency (R15)

```
{
 NOP
 MOV R7, sp
 MOVHI hwrd(Label<I>), R0, R15
 MOVEA lwrd(Label<I>), R15, R15
 NOP
 DISPOSE 0, {R20, R21, R22, R23, R24, R26, R27, R28}, [R15]
 Label<I>:
}
```

****
Family: SWITCH
Number: F15
Target Instruction: SWITCH

Remark:
Single test exploring fixed delays. Base address to case table in R11
****

F15_1: Simple test of nested switch - statement.
```
{
 SWITCH R11
 DH   1
}
```

****
Family: CALLT and CTRET
Number: F16
Target Instructions: CALLT, CTRET

Remark:
No nested invocations. Note impact of structural conflict (MEM stage). Test F16_3 and F16_4 explores possible interference of adjacent instructions reading and writing system register.
****
Call Routine A: “Nice”

```
 MOV R15, R16
 CTRET
```

Call Routine B: STSR reads CTPSW, LDSR writes CTPSW and RETI reads CTPSW.

```
 STSR 17, R15    (17 = CTPSW)
 NOP
 LDSR R15, 17
 CTRET
```

F16_1: No structural conflicts. Call Routine A.
```
{
 MOV   R16, R17
 MOV   R18, R19
 CALLT 0
}
```

F16_2: Structural conflict (MEM stage) between ST and CALLT. Call Routine A.
```
{
 MOV   R15, R16
 ST.W  R17, 0[sp]
 CALLT 0
}
```

F16_3: Instruction preceding CALLT write CTBP, CALLT always read CTBP. Call Routine A.
```
{
 NOP
 STSR 20, R15    (20 = CTBP)
 NOP
 LDSR R15, 20
 CALLT 0
}
```

F16_4: Invokes Call Routine B. Attention: CALLT always writes CTPSW.
```
{
 MOV R17, R18
 CALLT 0
}
```

****
Family: TRAP and RETI
Number: F17
Target Instructions: TRAP, RETI

Remark:
Test for checking statically delay. Adjacent System Register Usage test, similar to test in F16.
****
Trap Routine A. “Nice”:
```
{
 ADD  R15, R17
 MOV  R15, R16
 RETI
}
```

Trap Routine B. STSR reads EIPSW, LDSR writes EIPSW and RETI reads EIPSW:
```
{
 STSR 1, R15   (1 = EIPSW)
 NOP
 LDSR R15, 1
 RETI
}
```

F17_1: No possible conflicts on system register EIPSW. Trap routine A.
```
{
 MOV R17, R18
 TRAP 0
}
```

F17_2: Invokes Trap Routine B. Attention: TRAP always writes EIPSW.
```
{
 MOV R17, R18
 TRAP 0
}
```

****
Family: ADDITIONAL Bit Manipulation instructions
Number: AF7
Target Instructions: SET1, TST1

Remark:
These instructions have a long execution and thereby block the pipeline. Data hazards of base address registers are tested. Also note impact of structural conflicts (MEM stage). Note that SET1 executes in MEM stage twice, TST1 only once.
****

AF7_1: Possible conflict on sp.
```
{
 CLR1 R15, [sp]
 LD.W 0[sp], R17
 NOP
}
```

AF7_2: No register conflicts.
```
{
 CLR1 R15, [sp]
 LD.W 0[ep], R17
 NOP
}
```

AF7_3: Same as AF7_1 but with TST1.
```
{
 TST1 R15, [sp]
 LD.W 0[sp], R17
 NOP
}
```

AF7_4: Same as AF7_2 but with TST1
```
{
 TST1  R15, [sp]
 LD.W 0[ep], R17
 NOP
}
```

AF7_5: Same as AF7_4 but with SLD
```
{
 TST1  R15, [sp]
 SLD.W 0[ep], R17
 NOP
}
```

****
Family: ADDITIONAL Dual issue, Case 1
Number: AF10
Target Instructions: * / SLD.W

Remark: Introduce SUB
****

AF10_1:
```
{
 NOP
 SUB  R13, R14
 SLD.W 0[ep], R15
 ADD  R11, R14
}
```

****
Family: ADDITIONAL Dual issue, Case 4
Number: AF13
Target Instructions: MOV / *

Remark: Some more cases. All 16 bit instructions
****

AF13_1:
```
{
 MOV  R15, R16
 ADD  1, R16
}
```

AF13_2:
```
{
 MOV  R15, R16
 SUB  R17, R16
}
```

AF13_3:
```
{
 MOV  R15, R16
 SUBR R17, R16
}
```

AF13_4:
```
{
 MOV  R15, R16
 AND  R17, R16
}
```

AF13_5:
```
{
 MOV  R15, R16
 OR   R17, R16
}
```

AF13_6:
```
{
 MOV  R15, R16
 XOR  R17, R16
}
```

AF13_7:
```
{
 MOV  R15, R16
 NOT  R17, R16
}
```

AF13_8:
```
{
 MOV  R15, R16
 SATADD     R17, R16
}
```

11.2.Test Programs of Test Case F0_1

This example shows corresponding simulator and emulator programs generated by the test program generator

F0_1 Emulator Test Program

```
. ;*************************************************
   NAME testfile

   PUBLIC main
   PUBLIC ?C_EXIT

;*** DATA DEF ***
startd DEFINE 0ABCDh
stopd  DEFINE 0DCBAh
sSize  DEFINE 64

;*** RAM ***
   RSEG DATA
stack DS32 sSize  ; some data memory space (64x4 bytes)
tags  DS32  2  ; space for timer tags

;*** ROM ***

   ASEG 0
   JR main
;------------------------
;TRAP AREA:
;-----------------------
;CALLT AREA:
;------------------------

   ASEG 1000h

main

   MOV stack,R20
   MOV sSize,R21
   MULH 4,R21
   MOV stack,R22
   ADD R21,R22
   MOV R0,R21
   MOV 4,R21

clean
   CMP R20,R22
   BLE done

   ST.W R0,0[R22]
   SUB R21,R22
   JR clean

done
   MOV stack,sp
   ADDI 100,sp,sp
   MOV sp,R7
```

```
  MOV tags,R6
  MOV 1,R10
  MOV startd,R11
  MOV stopd,R12

  ST.W R10,0[sp]
  ST.W R11,0[R6]
  ST.W R12,4[R6]

  MOV 1,R15
  MOV 1,R16
  MOV 1,R17
  MOV 1,R18
  MOV 1,R19
  MOV 1,R20

  MOV 0,R10
  MOV 0,R11

  MOV sp,ep
  JR go

; R6 - timer tags base address
; R7 - origin sp address
; R25 - tag carrier


  ASEG 1FFCh

go
  LD.W 0[R6],R25  ; start timer
;************************************************

  NOP
  NOP
  NOP
  NOP
  NOP
  NOP


  NOP
  NOP
  NOP
  NOP
  NOP
  NOP

;**************************************************
  LD.W 4[R6],R25   ; stop timer
  NOP
  NOP
  NOP
?C_EXIT
  END main
;*************************************************
```

F0_1 Simulator Test Program

Basic Block Definition File:

```
begin block label_start:
  pred  ;
  succ label_end ;
  code
       2000 : 2 : Nop ;
       2002 : 2 : Nop ;
       2004 : 2 : Nop ;
       2006 : 2 : Nop ;
       2008 : 2 : Nop ;
       2010 : 2 : Nop ;
  end code
end block


begin block label_end:
  pred label_start ;
  succ  ;
  code
       2012 : 2 : Nop ;
       2014 : 2 : Nop ;
       2016 : 2 : Nop ;
       2018 : 2 : Nop ;
       2020 : 2 : Nop ;
       2022 : 2 : Nop ;
  end code
end block
```

Basic Block Trace File:

```
label_start
label_end
```

12. Appendix B

12.1. List of the Execution Parameters

Used in Table B-2, B-3, B-4, and B-5:

EP1:	Instruction length
EP2:	Instruction reading one or two GR during ID stage.
EP3:	Instruction writing to GR in WB stage.
EP4:	Instruction reading PSW during ID stage.
EP5:	Instruction writing PSW during EX stage.
EP6:	Instruction reading SR (other than PSW).
EP7:	Instruction writing SR (other than PSW).
EP8:	Instruction accessing memory in MEM stage.
EP9:	Instruction modifying PC.
EP10:	Instruction execution in EX stage more than once
EP11:	Instruction executes EX stage after MEM stage
EP12:	Instruction enables */SLD dual issue.
EP13:	Instruction disables */SLD dual issue.
EP14:	Dual issue classification

12.2. List of Dual Issue Scenario

Used in Table B-1:

Case **1**: **M**aster / `SLD.X`
Case **2**: **M**aster / `Bcond`
Case **3**: **M**aster / `BR`
Case **4**: `MOV r1,r2` / **S**lave

12.3. Table B-1: Dual issue Classification (EP14)

Instr	Opds	C 1	C 2	C 3	C 4	EP14
MOV	r1, r2	M	M	M	**M**	D1
MOV	imm5, r2	M	M	M		D2
MULH	r1, r2	M	M	M		
MULH	imm5, r2	M	M	M		
NOP		M	M	M		
SXB	r1	M	M	M		
SXH	r1	M	M	M		
ZXB	r1	M	M	M		
ZXH	r1	M	M	M		
ADD	r1, r2	M		M	S	D3
AND	r1, r2	M		M	S	
OR	r1, r2	M		M	S	
SATADD	r1, r2	M		M	S	
SATSUB	r1, r2	M		M	S	
SATSUBR	r1, r2	M		M	S	
SUB	r1, r2	M		M	S	
SUBR	r1, r2	M		M	S	
XOR	r1, r2	M		M	S	
ADD	imm5, r2	M		M		D4
CMP	r1, r2	M		M		
CMP	imm5, r2	M		M		
NOT	r1, r2	M		M		
SAR	imm5, r2	M		M		
SATADD	imm5, r2	M		M		
SHL	imm5, r2	M		M		
SHR	imm5, r2	M		M		
TST	r1, r2	M		M		
SST.X	r2, disp[ep]		M	M		D5
SLD.X	disp[ep], r2	**S**	M	M		D6
Bcond	disp9		**S**			D7
BR				**S**		D8
DIVH	r1, r2					D9
JMP	[r1]					
CALLT	imm6					

12.4. **Table B-2: 16-bit Instructions with Constant Timing Behaviour**

EC: Equivalence Class

Instr	Opds	EP2	EP3	EP4	EP5	EP6	EP7	EP8	EP9	EP10	EP11	EP12	EP13	EP14	EC
ADD	r1, r2	Yes	1		Yes									D3	16-01
AND	r1, r2	Yes	1		Yes									D3	
OR	r1, r2	Yes	1		Yes									D3	
SATADD	r1, r2	Yes	1		Yes									D3	
SATSUB	r1, r2	Yes	1		Yes									D3	
SATSUBR	r1, r2	Yes	1		Yes									D3	
SUB	r1, r2	Yes	1		Yes									D3	
SUBR	r1, r2	Yes	1		Yes									D3	
XOR	r1, r2	Yes	1		Yes									D3	
ADD	imm5, r2	Yes	1		Yes									D4	16-02
NOT	r1, r2	Yes	1		Yes									D4	
SAR	imm5, r2	Yes	1		Yes									D4	
SATADD	imm5, r2	Yes	1		Yes									D4	
SHL	imm5, r2	Yes	1		Yes									D4	
SHR	imm5, r2	Yes	1		Yes									D4	
CMP	r1, r2	Yes			Yes							Yes		D4	16-03
CMP	imm5, r2	Yes			Yes							Yes		D4	
TST	r1, r2	Yes			Yes							Yes		D4	
DIVH	r1, r2	Yes	1		Yes					34		Yes		D9	16-04
MOV	r1, r2	Yes	1											D1	16-05
MOV	imm5, r2	Yes	1											D2	16-06
SXB	r1	Yes	1											D2	
SXH	r1	Yes	1											D2	
ZXB	r1	Yes	1											D2	
ZXH	r1	Yes	1											D2	
NOP												Yes		D2	16-07
MULH	r1, r2	Yes	1							2[1]		Yes		D2	16-08
MULH	imm5, r2	Yes	1							2[1]		Yes		D2	
SLD.X	disp[ep], r2	Yes	1					1					Yes	D6	16-09
SST.X	r2, disp[ep]	Yes						1					Yes	D5	16-10
Bcond	disp9			Yes					Cond			Yes		D7	16-11
BR	disp9								UnCond			Yes		D8	16-12
JMP	[r1]	Yes							UnCond			Yes		D9	16-13
CALLT	imm6			Yes		Yes	Yes	1[2]	UnCond		Yes			D9	16-14

Note 1: Fully pipelined.
Note 2: Instruction memory access.

12.5. Table B-3: 32 and 48-bit Instructions with Constant Timing Behaviour

EC: Equivalence Class

Instr	Opds	EP1	EP2	EP3	EP4	EP5	EP6	EP7	EP8	EP9	EP10	EP11	EP12	EP13	EC
ADDI	imm16, r1, r2	32	Yes	1		Yes									32-01
ANDI	imm16, r1, r2	32	Yes	1		Yes									
BSH	r1, r2	32	Yes	1		Yes									
BSW	r1, r2	32	Yes	1		Yes									
HSW	r1, r2	32	Yes	1		Yes									
ORI	imm16, r1, r2	32	Yes	1		Yes									
SAR	r1, r2	32	Yes	1		Yes									
SATSUBI	im16, r1, r2	32	Yes	1		Yes									
SHL	r1, r2	32	Yes	1		Yes									
SHR	r1, r2	32	Yes	1		Yes									
XORI	imm16, r1, r2	32	Yes	1		Yes									
DI		32				Yes									32-02
EI		32				Yes									
CLR1	b#3, disp16[r1]	32	Yes			Yes			2		2	Yes		Yes	32-03
CLR1	r1, [r2]	32	Yes			Yes			2		2	Yes		Yes	
NOT1	b#3, disp16[r1]	32	Yes			Yes			2		2	Yes		Yes	
NOT1	r2, [r1]	32	Yes			Yes			2		2	Yes		Yes	
SET1	b#3, disp16[r1]	32	Yes			Yes			2		2	Yes		Yes	
SET1	r2, [r1]	32	Yes			Yes			2		2	Yes		Yes	
TST1	b#3, disp16[r1]	32	Yes			Yes			1		2	Yes	Yes		32-04
TST1	r2, [r1]	32	Yes			Yes			1		2	Yes	Yes		
DIVHU	r1, r2, r3	32	Yes	2		Yes					**34**		Yes		32-05
DIVU	r1, r2, r3	32	Yes	2		Yes					34		Yes		
DIV	r1, r2, r3	32	Yes	2		Yes					35		Yes		32-06
DIVH	r1, r2, r3	32	Yes	2		Yes					35		Yes		
MOVEA	imm16, r1, r2	32	Yes	1											32-07
MOVHI	imm16, r1, r2	32	Yes	1											
CMOV	cccc, r1, r2, r3	32	Yes	1	Yes										32-08
CMOV	cccc,imm5,r1,r2	32	Yes	1	Yes										
SASF	cccc, r2	32	Yes	1	Yes										
SETF	cccc, r2	32	Yes	1	Yes										
MULHI	imm16, r1, r2	32	Yes	1							2 [1]				32-09
MUL	r1, r2, r3	32	Yes	2							2 [1]				32-10
MUL	imm9, r2, r3	32	Yes	2							2 [1]				
MULU	r1, r2, r3	32	Yes	2							2 [1]				
MULU	imm9, r2, r3	32	Yes	2							2 [1]				
LD	disp16[r1], r2	32	Yes	1					1					Yes	32-11
ST	r2, disp16[r1]	32	Yes						1					Yes	32-12
JARL	disp22, r2	32	Yes	1						UnCond			Yes		32-13
JR	disp22	32								UnCond			Yes		32-14
CRET	imm6	32				Yes[2]			1 [3]	UnCond		Yes			32-15
RETI		32								UnCond					32-16
SWITCH	r1	32	Yes						1 [3]	UnCond	2	Yes			32-17
TRAP	vector	32				Yes[2]				UnCond					32-18
LDSR	r1,regID	32		1	op.dep		op.dep								32-19
STSR	regID,r1	32	Yes			op.dep		op.dep							32-20
MOV	imm32, r1	48		1									Yes		48-1

Note 1: Fully pipelined.
Note 2: Will not effect branch destination instruction or succeding instructions.
Note 3: Instruction memory access.

12.6. Table B-4: Instructions with Operand Dependent Timing Behaviour

EC: Equivalence Class

Instr	Opds	EP1	EP2	EP3	EP8	EP9	EP12	EP13	EC
DISPOSE	imm5, list12	32	Yes[1]	op.dep[2]	op.dep.			Yes	32-21S
DISPOSE	imm5,list12, [r1]	32	Yes[1]	op.dep[2]	op.dep.	UnCond	Yes		32-22S
PREPARE	list, imm5	32	Yes[1]	sp[2]	op.dep.			Yes	32-23S
PREPARE	list, im5, sp	32	Yes[1]	sp/ep[2]	op.dep.			Yes	32-24S
PREPARE	list, imm5, imm16	48	Yes[1]	sp/ep[2]	op.dep.			Yes	48-02S
PREPARE	list, imm5, imm32	64	Yes[1]	sp/ep[2]	op.dep.			Yes	64-01S

Note 1: Unclear at which points in time GR readings occur (incl. sp).
Note 2: Unclear at which points in time GR writings occur (incl. sp and ep).

12.7. Table B-5: Selected Instructions

Instr	Opds	EP1	EP2	EP3	EP4	EP5	EP6	EP7	EP8	EP9	EP10	EP11	EP12	EP13	E.C
ADD	r1, r2	16	Yes	1		Yes									16-01
ADD	imm5, r2	16	Yes	1		Yes									16-02
CMP	r1, r2	16	Yes			Yes							Yes		16-03
DIVH	r1, r2	16	Yes	1		Yes					34		Yes		16-04
MOV	r1, r2	16	Yes	1											16-05
SXB	r1	16	Yes	1											16-06
NOP		16											Yes		16-07
MULH	r1, r2	16	Yes	1							2[1]		Yes		16-08
SLD.X	disp[ep], r2	16	Yes	1					1					Yes	16-09
SST.X	r2, disp[ep]	16	Yes						1					Yes	16-10
Bcond	disp9	16			Yes					Cond			Yes		16-11
BR	disp9	16								UnCond			Yes		16-12
JMP	[r1]	16	Yes							UnCond			Yes		16-13
CALLT	imm6	16			Yes		Yes	Yes	1[2]	UnCond		Yes			16-14
ADDI	imm16, r1, r2	32	Yes	1		Yes									32-01
DI		32				Yes									32-02
CLR1	r1, [r2]	32	Yes			Yes			2		2	Yes		Yes	32-03
TST1	r2, [r1]	32	Yes			Yes			1		2	Yes	Yes		32-04
DIVU	r1, r2, r3	32	Yes	2		Yes					34		Yes		32-05
DIV	r1, r2, r3	32	Yes	2		Yes					35		Yes		32-06
MOVEA	imm16, r1, r2	32	Yes	1											32-07
CMOV	cccc, r1, r2, r3	32	Yes	1	Yes										32-08
MULHI	imm16, r1, r2	32	Yes	1							2[1]				32-09
MUL	r1, r2, r3	32	Yes	2							2[1]				32-10
LD	disp16[r1], r2	32	Yes						1					Yes	32-11
ST	r2, disp16[r1]	32	Yes	1					1					Yes	32-12
JARL	disp22, r2	32	Yes							UnCond			Yes		32-13
JR	disp22	32		1						UnCond			Yes		32-14
CTRET		32				Yes	Yes			UnCond			Yes		32-15
RETI		32			Yes	Yes	Yes			UnCond			Yes		32-16
SWITCH	r1	32	Yes						1[2]	UnCond	2	Yes	Yes		32-17
TRAP	vector	32			Yes	Yes		Yes		UnCond			Yes		32-18
LDSR	r1, regID	32		1	op.dep.		op.dep.								32-19
STSR	regID, r2	32	Yes			op.dep.		op.dep.							32-20
MOV	imm32, r1	48		1									Yes		48-01
DISPOSE	imm5, list12	32	Yes[3]	op.dep.[4]					op.dep.					Yes	32-21S
DISPOSE	imm5,list12, [r1]	32	Yes[3]	op.dep.[4]					op.dep.	UnCond			Yes		32-22S
PREPARE	list, imm5	32	Yes[3]	sp[4]					op.dep.					Yes	32-23S
PREPARE	list, im5, sp	32	Yes[3]	sp/ep[4]					op.dep.					Yes	32-24S
PREPARE	list, imm5, imm16	48	Yes[3]	sp/ep4					op.dep.					Yes	48-02S
PREPARE	list, imm5, imm32	62	Yes[3]	sp/ep4					op.dep.					Yes	64-01S

Note 1: Fully pipelined.
Note 2: Instruction memory access.
Note 3: Unclear at which points in time GR readings occur (incl. sp).
Note 4: Unclear at which points in time GR writings occur (incl. sp and ep).