

Architecture Description Language driven Functional Test Program Generation for Microprocessors using SMV

Prabhat Mishra
pmishra@cecs.uci.edu

Nikil Dutt
dutt@cecs.uci.edu

Architectures and Compilers for Embedded Systems (ACES)
Center for Embedded Computer Systems, University of California, Irvine, CA, USA

CECS Technical Report #02-26
Center for Embedded Computer Systems
University of California, Irvine, CA 92697, USA

September 13, 2002

Abstract

Formal techniques offer an opportunity to significantly reduce the cost of microprocessor verification. We propose a model checking based approach to automatically generate functional test programs for pipelined processors. We specify the processor architecture in an Architecture Description Language (ADL). The processor model is extracted from the ADL specification. Specific properties are applied to the processor model using SMV model checker to generate test programs. We applied this methodology on a single-issue DLX processor to demonstrate the usefulness of our approach.

Contents

1	Introduction	3
2	Our Approach	3
2.1	Classification of Testcases	4
2.2	Coverage Estimation	5
3	A Case Study	5
3.1	ADL Specification of the DLX Architecture	6
3.2	Generation of SMV Description for the DLX Processor	7
3.3	Specification of Properties	7
3.4	Functional Testcase Generation using SMV	7
3.5	Coverage Estimation and Specification of New Properties	8
4	Summary	9
5	Acknowledgments	9
A	The SMV Description of the DLX Architecture	12

List of Figures

1	Functional Test Program Generation Flow	4
2	The DLX architecture	6
3	A fragment of the DLX architecture	8

1 Introduction

Functional verification consumes a significant portion of the microprocessor design cycle time. Verification techniques can be broadly categorized into simulation-based approaches and formal techniques. Formal techniques have emerged as an alternative approach for ensuring the quality and correctness of hardware designs, overcoming some of the limitations of traditional simulation. However, simulation is still the most widely used form of microprocessor verification: millions of cycles are spent during simulation using random test cases in traditional design flow. Certain heuristics and design abstractions are used to generate directed random testcases. However, due to the bottom-up nature and localized view of these heuristics the generated testcases may not yield a good coverage. We propose a directed random test program generation scheme using behavioral knowledge of the pipelined architecture specified in an Architecture Description Language (ADL).

Several approaches for formal or semi-formal verification of pipelined processors have been developed in the past. Theorem proving techniques, for example, have been successfully adapted to verify pipelined processors ([5] [13] [15]). Burch and Dill presented a technique for formally verifying pipelined processor control circuitry [4]. The technique has been extended to handle more complex pipelined architectures by several researchers ([11] [14]). Ho et al. [6] extract controlled token nets from a logic design to perform efficient model checking. Hauke et al. [10] proposed a technique, called reverse engineering, which extracts the ISA model of a pipelined processor from its implementation model and compares the extracted ISA with user-specified ISA.

Traditionally, validation of a microprocessor has been performed by resorting to functional approaches based on exciting all the functions and resources described in its data-sheets [19]. Generation of effective test programs for the self-test of a processor has been studied by several researchers ([2] [18] [20] [21]). Ur and Yadin [23] presented a method for generation of assembler test programs that systematically probe the micro-architecture of a PowerPC processor. Iwashita et al. [3] use a FSM based processor modeling to automatically generate test programs. Aharon et al. [24] have proposed a new methodology and test program generator for functional verification of PowerPC processors in IBM.

In this report, we present an approach for automatic functional test program generation from an architectural specification using model checking. Similar techniques have been proposed in the past to validate software designs [12]. To the best of our knowledge, this technique has not been studied before in the context of pipelined processor verification. We applied model checking to automatically generate functional test programs using behavioral knowledge of the architecture specified in an ADL. Section 2 outlines our approach and the overall flow of our test program generation environment followed by a case study in Section 3. Section 4 concludes with a short summary and future work directions.

2 Our Approach

Figure 1 shows the flow in our approach. In our specification-driven test program generation scenario, the designer starts by specifying the microprocessor architecture in an Architecture Description Language (ADL). We verify the correctness of the ADL specification of the architecture

([7] [8] [9]). The verification engineers specify the properties that the architecture should satisfy. The processor model is generated from the architecture specification. Both the processor model and the properties are described using the SMV language. The properties are applied to this processor model using the SMV model checker [25] to generate testcases. We actually write the negation of the properties that we want to verify. For example, to generate a testcase for verifying a feedback path fp , we write a property that specifies that the feedback path fp is not exercised. The model checker produces a counter example (instruction sequence) that activates the feedback path fp . These counterexamples (instruction sequences) are converted into complete tests (instruction sequence followed by expected results) using a cycle-accurate structural simulator [17]. The simulator is generated automatically from the ADL specification of the architecture [16]. More properties are added if the coverage requirement is not satisfied.

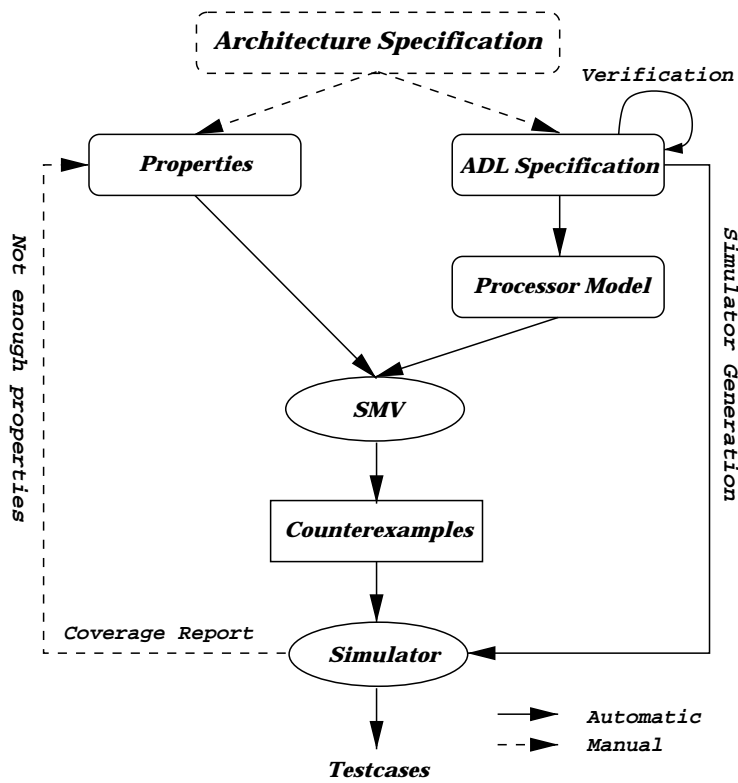


Figure 1. Functional Test Program Generation Flow

In the remainder of this section, we briefly mention the category of testcases we consider and our coverage estimation technique.

2.1 Classification of Testcases

We classify the testcases in several categories. Here, we briefly mention some of them.

- *Pipeline Flow*

- *Timing of each operation*: Issue only one valid operation and insert NOPs. Check the timing of the operation in the pipeline.
 - *Hazards*: Generate testcases that will cause different kinds of hazards (data, control, and structural). Check the committed result in each case to ensure that the pipeline works correctly in the presence of hazards.
 - *Stalls*: Stalls are generated due to hazards or exceptions. The goal is to create hazard/exception conditions such that a specific pipeline stage is stalled. Check to see whether dependent stages are stalled for correct number of cycles.
 - *Exceptions*: Create testcases that generate exceptions (e.g., cache miss, TLB miss etc.). Check to ensure that the exceptions are handled properly and appropriate pipeline stages are flushed.
- *Feedback Paths*: Generate testcases that exercise each feedback path in the pipeline.
 - *Branch Prediction*: Generate testcases that would cause branch mis-prediction, stall and flushing.
 - *Execution Style*: Generate testcases to verify the execution style of the pipeline. For example, if it is an in-order execution processor the testcase should validate or falsify it.
 - *Memory Controller*: Generate testcases to validate several features of the memory controller e.g., data forwarding from store queue to load queue, TLB miss etc.

2.2 Coverage Estimation

Measuring progress is one of the most important tasks in verification, and is the critical element that enables the designer to decide when to end the verification effort. Several coverage measures are commonly used: code coverage, toggle coverage, fault coverage etc. Unfortunately, neither of these measures described above has any direct relation to the functionality of the device, nor there is any correlation to common user applications. For example, none of these determine if all possible interactions of hazards, stalls and multiple exceptions are tested in a processor pipeline. We propose a coverage metric based on functional coverage of the specification. This allows the verification engineer to define exactly what functionality of the device should be monitored.

3 A Case Study

We applied our methodology on a single-issue DLX [22] architecture. Figure 2 shows the pipeline structure of the DLX architecture. The oval boxes represent pipeline latches, rectangular boxes represent functional units, solid lines represent pipeline edges, and dotted lines represent data-transfer edges. The pipeline latches are also called instruction registers (IR) since they contain instructions being executed in the pipeline. A pipeline edge transfers an instruction from a parent unit to a child unit using pipeline latches (instruction registers). A data-transfer edge is used to transfer data from a functional unit to a storage or from a storage to a functional unit.

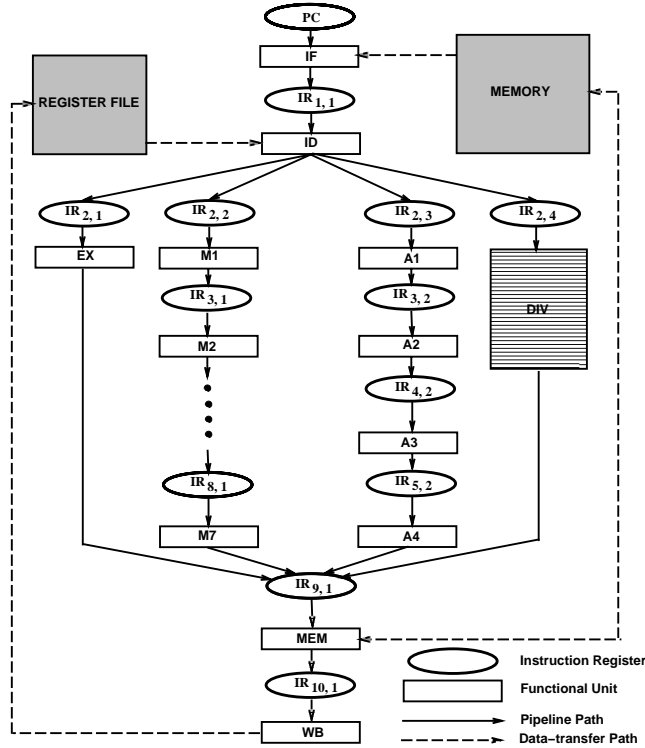


Figure 2. The DLX architecture

First, we describe how we capture the DLX architecture in the EXPRESSION ADL [1]. The SMV description of the DLX processor is generated automatically from this ADL specification. Next, we present how to specify the necessary properties, followed by an example of test program generation using SMV. Finally, we present a coverage estimation scenario followed by addition of necessary properties.

3.1 ADL Specification of the DLX Architecture

We use the EXPRESSION ADL [1] to specify the DLX architecture. Two very important concepts in the ADL are pipeline paths and data-transfer paths. A path from a root node (e.g., Fetch unit) to a leaf node (e.g., WriteBack unit) consisting of units and pipeline edges is called a *pipeline path*. Intuitively, a pipeline path denotes an execution flow in the pipeline taken by an operation. For example, one of the pipeline path is $\{IF, ID, DIV, MEM, WB\}$. A path from a unit to a storage or from a storage to a unit consisting of storages and data-transfer edges is called a *data-transfer path*. For example, $\{MEM, MEMORY\}$ is a data-transfer path.

The ADL captures the structure, behavior and mapping (between the structure and behavior) of the architecture pipelines. The structure is defined by its components (units, storages, ports, connections) and the connectivity (pipeline and data-transfer paths) between these components. Each component is defined by its attributes e.g., the list of opcodes it supports, execution timing for each supported opcode etc. The behavior of a processor is defined by its instruction set. Each

operation in the instruction-set is defined in terms of opcode, operands and the functionality of the operation. A set of mapping functions correlate the abstract, high-level behavioral model of the architecture to the structural model. For example, *unit-to-opcode* (*opcode-to-unit*) mapping is a bi-directional function that maps units in the structure to opcodes in the behavior. It defines, for each functional unit, the set of operations supported by that unit (and vice versa). For example, *unit-to-opcode* function maps the division unit to the opcodes $\{NOP, DIV\}$. The ADL also captures hazards, stalls, interrupts and exceptions.

3.2 Generation of SMV Description for the DLX Processor

We have developed a library of generic architectural components that can be used for validation. Each component is described using the SMV language at different levels of abstraction. For example, a simplified version of the instruction fetch unit (IF) is shown below:

```

module Fetch (PC, InstMemory, operation)
{
    input PC : integer;
    input InstMemory : memory;
    output operation : opType;

    init(operation.opcode) := NOP;
    next(operation) := InstMemory[PC];
}

```

The SMV description of the DLX architecture is generated automatically from the ADL specification using the component library. The SMV description of the DLX architecture has 354 lines of code using the pipeline and cycle-accurate components from the library as shown in Appendix A.

3.3 Specification of Properties

We have written properties for each category of the testcases mentioned in Section 2.1. Here we show an example to stall a particular unit. For example, the following property is used to stall the decode unit (ID). The *_stall* bit for the decode unit can be true due to data hazard or when one of the children is stalled.

```

hazard: assert G(ID._stall = 0);

```

3.4 Functional Testcase Generation using SMV

We apply the properties on the processor model using SMV. Since we write the negation of the properties we want to validate, the counter example (instruction sequence) generated by SMV can be used as a testcase. The expected result can be obtained by using the simulator. For example, to generate the counterexample for the property mentioned in Section 3.3 the system took 1.3 seconds on a 359 MHz Sun UltraSPARC-II with 2048M RAM. The instruction sequence is shown below. The read-after-write hazard sets the *_stall* bit in this scenario. The *ADD* operation is supported by integer ALU (EX) unit. The decode unit (ID) will be stalled in cycle 4 in this case.

Fetch Cycle	Opcode	Dest	Src1	Src2
1	NOP			
2	ADD	R3	R1	R2
3	ADD	R4	R3	R2

3.5 Coverage Estimation and Specification of New Properties

While analyzing the simulator coverage report we observed that one specific path in the division unit (DIV) is not exercised by any testcase. Further analysis revealed that it was necessary to initialize two internal registers to specific values to activate the path. Figure 3 shows a fragment of the DLX pipeline containing the internals of the division unit (DIV). The two internal input registers for DIV unit are A_{in} and B_{in} . The internal output register for DIV unit is C_{out} . The input instruction is in and the output result is out . In this particular scenario A_{in} and B_{in} receives data from the first and second source operands of the input instruction (in) i.e., $A_{in} = in.src1$ and $B_{in} = in.src2$; C_{out} returns the result of the division i.e., $C_{out} = A_{in} \div B_{in}$; finally the output is fed from C_{out} i.e., $out = C_{out}$. However, in general these could be any arbitrary functions f_1, f_2, f_3, f_4 such that $A_{in} = f_1(in), B_{in} = f_2(in), C_{out} = f_3(A_{in}, B_{in})$, and $out = f_4(C_{out})$. In effect, this is a controllability problem: how to assign specific values to these internal input registers at specific clock cycle using the primary inputs of the DLX processor?

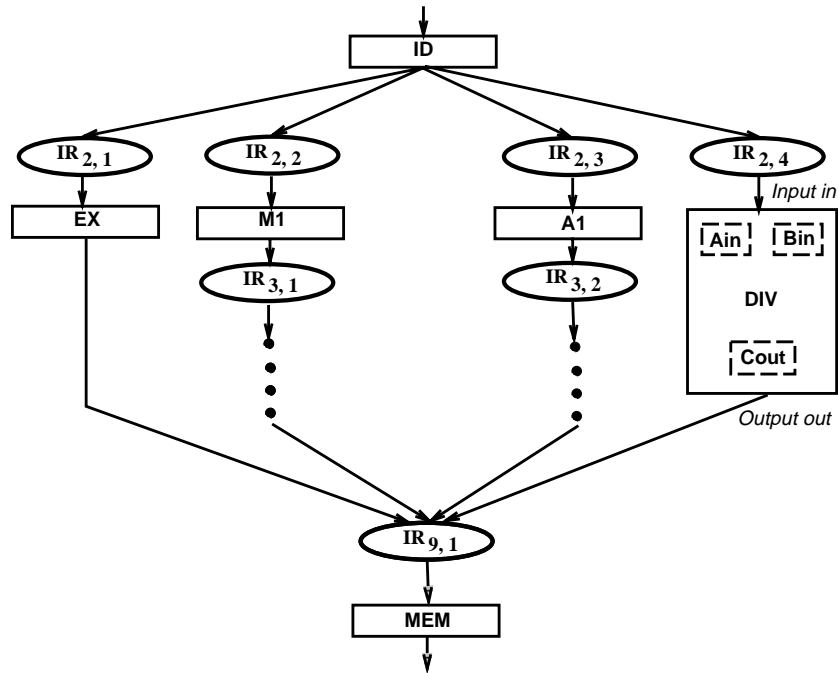


Figure 3. A fragment of the DLX architecture

We added the following property that generates instruction sequence to initialize A_{in} and B_{in} with values 2 and 3 respectively at clock cycle 9.

```
init: assert G((cycle = 8) -> X((DIV.Ain ~= 2) | (DIV.Bin ~= 3)));
```


The system took 75.4 seconds to come up with the counterexample on a 359 MHz Sun UltraSPARC-II with 2048M RAM. The instruction sequence is shown below. The DIV instruction will be in division unit (DIV) in cycle 9. The MOVI (move immediate) instruction is supported by integer ALU (EX) unit and DIV instruction is supported by division (DIV) unit. The A_{in} will get value from R4, which is 2 and B_{in} will get value from R5, which is 3 in this particular case.

Fetch Cycle	Opcode	Dest	Src1	Src2
1	NOP			
2	MOVI	R4,	#2	
3	MOVI	R5,	#3	
4	NOP			
5	NOP			
6	NOP			
7	DIV	R0,	R4,	R5

4 Summary

Functional verification consumes a significant portion of the microprocessor design cycle. Formal methods, typically used in the verification of microprocessor, offer an opportunity to reduce the cost of the validation phase. We pursued this path by applying model checking to the problem of functional test program generation for pipelined processors.

We generate a SMV description of the processor model automatically from the ADL specification of the architecture. The SMV description of the properties are written manually. We specify the negation of the properties that we want to verify in the architecture. The model checker generates counterexamples. The expected results are generated for each counterexample using a cycle-accurate structural simulator. The simulator is generated automatically from the ADL specification of the architecture. More properties can be added depending on the required coverage. We applied our methodology on a single-issue DLX architecture to demonstrate the feasibility of our approach.

Currently, we apply these tests on the cycle-accurate structural simulator of the architecture. We are working towards applying these tests on the RTL description of the processor. Currently, the properties are written by hand. Also, we rely on manual coverage analysis and addition of new properties. Our future work includes automatic coverage estimation and generation of properties from the ADL specification of the architecture. We are also investigating the use of SAT-based bounded model checkers to generate functional test programs.

5 Acknowledgments

This work was partially supported by grants from Motorola Inc., Hitachi Ltd., and NSF award CCR-0203813. We would like to thank Prof. Sandeep Shukla for his helpful comments and suggestions.

References

- [1] A. Halambi et al. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *DATE*, 1999.
- [2] L. Chen et al. DEFUSE: A Deterministic Functional Self-Test Methodology for Processors. *VTS*, 2000.
- [3] H. Iwashita et al. Automatic test pattern generation for pipelined processors. *ICCAD*, 1994.
- [4] J. Burch and D. Dill. Automatic verification of pipelined microprocessor control. *CAV*, 1994.
- [5] D. Cyrluk. Microprocessor verification in pvs: A methodology and simple example. Technical report, SRI-CSL-93-12, 1993.
- [6] P. Ho et al. Formal verification of pipeline control using controlled token nets and abstract interpretation. In *ICCAD*, 1998.
- [7] P. Mishra et al. Automatic Modeling and Validation of Pipeline Specifications driven by an Architecture Description Language. *ASP-DAC/VLSI Design*, 2002.
- [8] P. Mishra et al. Automatic Verification of In-Order Execution in Microprocessors with Fragmented Pipelines and Multicycle Functional Units. *DATE*, 2002.
- [9] P. Mishra et al. Modeling and Verification of Pipelined Embedded Processors in the Presence of Hazards and Exceptions. *DIPES*, 2002.
- [10] J. Hauke and J. Hayes. Microprocessor design verification using reverse engineering. In *HLDVT*, 1999.
- [11] M. Velev et al. Formal verification of superscalar microprocessors with multicycle functional units, exceptions, and branch prediction. *DAC*, 2000.
- [12] P. Ammann et al. Using Model Checking to Generate Tests from Specifications. *ICFEM* 1998.
- [13] J. Sawada et al. Trace table based approach for pipelined microprocessor verification. *CAV*, 1997.
- [14] J. Skakkebaek et al. Formal verification of out-of-order execution using incremental flushing. *CAV*, 1998.
- [15] M. Srivas et al. Formal verification of a pipelined microprocessor. *IEEE Software*, volume 7(5), pages 52–64, 1990.
- [16] P. Mishra et. al. Functional Abstraction driven Design Space Exploration of Heterogeneous Programmable Architectures. *ISSS*, pages 256–261, 2001.

- [17] A. Khare et al. V-SAT: A visual specification and analysis tool for system-on-chip exploration. *EUROMICRO*, 1999.
- [18] F. Corno et al. On the Test of Microprocessor IP Cores. *DATE*, 2001.
- [19] S. Thatte et al. Test Generation for Microprocessors. *IEEE Trans. on Computers*, Vol. C-29, pp. 429-441, June 1980.
- [20] K. Batcher et al. Instruction Randomization Self Test for Processor Cores. *VTS*, 1999.
- [21] J. Shen et al. Functional Verification of the Equator MAP1000 Microprocessor. *DAC*, 1999.
- [22] J. Hennessy and D. Patterson. *Computer Architecture: A quantitative approach*. Morgan Kaufmann Publishers Inc, San Mateo, CA, 1990.
- [23] S. Ur et al. Micro architecture coverage directed generation of test programs. *DAC*, 1999.
- [24] A. Aharon et al. Test Program Generation for Functional Verification of PowerPC Processors in IBM. *DAC*, 1995.
- [25] SMV. <http://www.cs.cmu.edu/~modelcheck>.

A The SMV Description of the DLX Architecture

```
/*
*****
**          The SMV description of the single-issue DLX processor          **
** Prabhat Mishra, Center of Embedded Computer Systems, November 13, 2001 **
**          Copyright Regents of the University of California, Irvine      **
*****
*/

#define maxInt 3
#define maxDbl 15
#define memSize 3
#define regSize 3
#define instSize 3
#define Alu 0
#define Mul 1
#define FAdd 2
#define Div 3
#define divCounter 3

typedef integer 0..maxInt;
typedef double 0..maxDbl;
typedef opcodes {NOP, ADD, SUB, MUL, DIV, FADD, MOVI};

typedef opType struct {
    opcode : opcodes;
    src1 : integer;
    src2 : integer;
    dest : integer;
}

typedef resType struct {
    dest : integer;
    value : integer;
}

typedef memory array 0..memSize of opType;
typedef register array 0..regSize of integer;
typedef vliw array 0..instSize of opType;
typedef results array 0..instSize of resType;
typedef busyRegisters array 0..regSize of boolean;

/** Fetch Unit, branch is not considered here **/
module Fetch (PC, InstMemory, operation) {
    input PC : integer;
    input InstMemory : memory;
    output operation : opType;

    init(operation.opcode) := NOP;
    next(operation) := InstMemory[PC];
}

```

```

/** Decode Unit */
module Decode (RegFile, BusyRegFile, operation, instruction)
{
    input operation : opType;
    output instruction : vliw;
    input RegFile : register;
    input BusyRegFile : busyRegisters;

    _stall : boolean;
    src1 : integer;
    src2 : integer;
    dest : integer;
    oper : opType;
    validOper : opType;

    init(_stall) := 0;

    /* Create a NOP operation */
    oper.opcode := NOP;
    oper.src1 := 0;
    oper.src2 := 0;
    oper.dest := 0;

    for (i=0; i<=instSize; i=i+1)
        init(instruction[i].opcode) := NOP;

    /* Read the operands */
    src1 := operation.src1;
    src2 := operation.src2;
    dest := operation.dest;

    validOper.dest := dest;
    validOper.opcode := operation.opcode;

    if (operation.opcode = MOVI) { /** Immediate retains their value */
        validOper.src1 := src1;
        validOper.src2 := src2;
    }
    else if (operation.opcode ~= NOP) {
        /** When children are busy it should be stalled as well.          **/
        /** Not just due to data hazard as considered below. In this case **/
        /** the Decode will issue DIV operation even when Divider is busy **/
        if ((BusyRegFile[src1] = 1) | (BusyRegFile[src2] = 1))
            next(_stall) := 1;
        if (BusyRegFile[src1] = 0)
            validOper.src1 := RegFile[src1];
        if (BusyRegFile[src2] = 0)
            validOper.src2 := RegFile[src2];
    }
}

```

```

/* Put the operation in the correct slot. */
if (operation.opcode = DIV) {
    next(instruction[Alu]) := oper;
    next(instruction[Mul]) := oper;
    next(instruction[FAdd]) := oper;
    next(instruction[Div]) := validOper;
} else if (operation.opcode = FADD) {
    next(instruction[Alu]) := oper;
    next(instruction[Mul]) := oper;
    next(instruction[FAdd]) := validOper;
    next(instruction[Div]) := oper;
} else if (operation.opcode = MUL) {
    next(instruction[Alu]) := oper;
    next(instruction[Mul]) := validOper;
    next(instruction[FAdd]) := oper;
    next(instruction[Div]) := oper;
} else {
    next(instruction[Alu]) := validOper;
    next(instruction[Mul]) := oper;
    next(instruction[FAdd]) := oper;
    next(instruction[Div]) := oper;
}
}

module IntAlu (operation, result) /** Integer ALU Unit **/
{
    input operation: opType;
    output result : resType;

    next(result.dest) := operation.dest;

    if (operation.opcode = NOP)
        next(result.value) := 0;
    else if (operation.opcode = ADD)
        next(result.value) := operation.src1 + operation.src2;
    else if (operation.opcode = SUB)
        next(result.value) := operation.src1 - operation.src2;
    else if (operation.opcode = MOVI)
        next(result.value) := operation.src1;
    else
        next(result.value) := 0;
}

/** Dummy Pipeline Stage **/
module Pass (opIn, opOut)
{
    input opIn : opType;
    output opOut : opType;

    next(opOut) := opIn;
}

```

```

/** Multiply Unit */
module Multiply (operation, result)
{
    input operation: opType;
    output result : resType;

    if (operation.opcode = MUL) {
        next(result.value) := operation.src1 * operation.src2;
        next(result.dest) := operation.dest;
    } else
        next(result.value) := 0;
}

/** Seven Stage Integer Multiplier */
module IntMul (operation, result)
{
    input operation: opType;
    output result : resType;
    op1 : opType;
    op2 : opType;
    op3 : opType;
    op4 : opType;
    op5 : opType;
    op6 : opType;

    M1: Pass (operation, op1);
    M2: Pass (op1, op2);
    M3: Pass (op2, op3);
    M4: Pass (op3, op4);
    M5: Pass (op4, op5);
    M6: Pass (op5, op6);
    M7: Multiply (op6, result);
}

/** Floating-point Adder Unit */
module FAdder (operation, result)
{
    input operation: opType;
    output result: resType;

    if (operation.opcode = FADD) {
        next(result.value) := operation.src1 + operation.src2;
        next(result.dest) := operation.dest;
    } else
        next(result.value) := 0;
}

```

```

/** Four Stage Floating-Point Adder **/
module FloatAdd (operation, result)
{
    input operation: opType;
    output result  : resType;

    op1 : opType;
    op2 : opType;
    op3 : opType;

    F1: Pass (operation, op1);
    F2: Pass (op1, op2);
    F3: Pass (op2, op3);
    F4: FAdder (op3, result);
}

/** Multi-cycle Divider Unit **/
module Divider (operation, result, busy)
{
    input operation: opType;
    output result  : resType;
    output busy   : boolean;
    count : integer;

    src1 : integer;
    src2 : integer;

    init (src1) := 0;
    init (src2) := 0;

    next(src1) := operation.src1;
    next(src2) := operation.src2;

    init (busy) := 1;
    init (count) := 0;

    if (operation.opcode = DIV) {
        next(busy) := 1;
        next(count) := 0;
    } else if (count = divCounter)
        next(busy) := 0;
    else if (busy = 1) {
        if (count < divCounter)
            next(count) := count + 1;
        else
            next(count) := count;
    }

    if (busy = 0) {
        next(result.dest) := operation.dest;
        next(result.value) := operation.src1 / operation.src2;
    }
}

```



```

    }
}

/** Execute Stage Consisting of Four Parallel Execution Units**/
module Execute (instruction, result)
{
    input instruction : vliw;
    output result : results;
    result1 : resType;
    result2 : resType;
    result3 : resType;
    result4 : resType;
    busy : boolean;

    IA: IntAlu (instruction[Alu], result1);
    FA: FloatAdd (instruction[FAdd], result3);
    IM: IntMul (instruction[Mul], result2);
    DV: Divider (instruction[Div], result4, busy);

    result[Alu] := result1;
    result[Mul] := result2;
    result[FAdd] := result3;
    result[Div] := result4;
}

/** WriteBack Unit **/
module WriteBack(RegFile, result)
{
    input result : results;
    output RegFile : register;

    wbInit : boolean;
    init(wbInit) := 1;
    next(wbInit) := 0;

    if (wbInit) {
        /** Initialize Register File **/
        for(i = 0; i <= regSize; i = i + 1)
            init(RegFile[i]) := 0;
    } else {
        /** Write Back the Result **/
        if (result[0].dest ~= 0)
            next(RegFile[result[0].dest]) := result[0].value;
        else if (result[1].dest ~= 0)
            next(RegFile[result[1].dest]) := result[1].value;
        else if (result[2].dest ~= 0)
            next(RegFile[result[2].dest]) := result[2].value;
        else if (result[3].dest ~= 0)
            next(RegFile[result[3].dest]) := result[3].value;
    }
}

```

```

module Hazard (BusyRegFile, operation, result) /** Hazard Detection Unit **/
{
    output BusyRegFile : busyRegisters;
    input operation : opType;
    input result : results;
    first : boolean;

    init(first) := 1;
    next(first) := 0;

    if (first) {
        for (i=0; i<=regSize; i=i+1)
            init(BusyRegFile[i]) := 0;
    } else if (operation.opcode ~= NOP) {
        /** Setting busy bit to 1 and resetting it to zero should be done **/
        /** independently. However, SMV gives re-initialization error and **/
        /** I use only one if-then-else although it is not correct **/
        if (operation.dest ~= 0)
            next(BusyRegFile[operation.dest]) := 1;
        else if (result[0].dest ~= 0)
            next(BusyRegFile[result[0].dest]) := 0;
        else if (result[1].dest ~= 0)
            next(BusyRegFile[result[1].dest]) := 0;
        else if (result[2].dest ~= 0)
            next(BusyRegFile[result[2].dest]) := 0;
        else if (result[3].dest ~= 0)
            next(BusyRegFile[result[3].dest]) := 0;
    }
}

module main () {
    operation : opType;
    instruction : vliw;
    result : results;
    PC : integer;
    InstMemory : memory;
    RegFile : register;
    BusyRegFile : busyRegisters;

    init(PC) := 0;

    FE: Fetch (PC, InstMemory, operation);
    DE: Decode (RegFile, BusyRegFile, operation, instruction);
    HZ: Hazard (BusyRegFile, operation, result);
    EE: Execute (instruction, result);
    WB: WriteBack (RegFile, result);

    /** The following property will generate test to stall the Decode unit **/
    hazard: assert G(DE._stall = 0);
}

```