MIPS-to-Verilog, Hardware Compilation for the eMIPS Processor

Karl Meier, Alessandro Forin
Microsoft Research

September 2007

Technical Report
MSR-TR-2007-128

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

MIPS-to-Verilog, Hardware Compilation for the eMIPS Processor

Karl Meier, Alessandro Forin
Microsoft Research

Abstract

The MIPS-to-Verilog (M2V) compiler translates blocks of MIPS machine code into a hardware design represented in Verilog. The design constitutes an Extension for the eMIPS processor, a dynamically extensible processor realized on the Virtex-4 XC4LX25 FPGA. The Extension interacts closely with the basic pipeline of the microprocessor and recognizes special Extension Instructions, instructions that are not part of the basic MIPS ISA. Each instruction is semantically equivalent to one or more blocks of MIPS code. The purpose of the M2V compiler is to automate the process of creating Extensions for the specific purpose of accelerating the execution of software programs.

M2V is a three-pass compiler that accepts as input basic blocks in the form generated by the eMIPS BB-Tools, a set of programs for the analysis and instrumentation of MIPS ELF images. Pass 1 of M2V generates a circuit graph that is semantically equivalent to the basic block that is being accelerated. Pass 2 schedules the operations in the circuit graph under the microarchitectural constraints of the eMIPS processor. Pass 3 emits synthesizable Verilog that constitutes the hardware accelerator that runs in the eMIPS extension slot.

The compiler was implemented from scratch in C++ and despite its current limitations it can already compile a few simple examples. The quality of the synthesizable Verilog that is generated by M2V compares favorably with hand-generated code for the same input. On a 64-bit division test M2V generates an Extension that performs at the same speed but uses half the area of the hand-generated version.

1 Introduction

An embedded system typically runs a small set of applications and has tight power, cost, and performance criteria. Using a general purpose CPU for these systems can help meet the performance goals, but inefficiency can result when specialized resources, such as a floating point unit (FPU), are present but seldom, if ever, used. A lower power processor with a reduced instruction set architecture (ISA) and without a FPU may suffer from poor performance.

Extensible processors try to compromise between general purpose CPUs and minimal RISC implementations. Extensible processors have a simple RISC pipeline and the ability to augment the ISA with custom instructions. The ISA can be augmented statically, at tape-out, or it can be augmented dynamically when applications are loaded. The eMIPS processor is an example of a dynamically extensible processor.

Extensible processors take advantage of the fact that a small amount of code takes the majority of execution time in a typical program. The code that executes most often is a candidate for hardware acceleration. The code must be identified by a special instruction that will initiate the accelerator.

Selection of the best code to accelerate is an active area of research. The eMIPS tool-chain restricts the code selection problem to the set of basic blocks in the application. Using the strict definition in [1], the basic block is a directed acyclic graph (DAG). The BB-Tools select the basic blocks to accelerate and patch the binary image with the special instructions for the accelerator. The M2V compiler automatically generates the hardware accelerator.

The accelerator can be statically loaded when eMIPS is reset or it can be dynamically loaded when an application is loaded using partial reconfiguration of the FPGA. By dynamically loading and unloading accelerators, programmable hardware can be minimized.

In previous versions of eMIPS, the accelerator blocks could be specified and given to a hardware designer to hand design the accelerator. While this can lead to an efficient implementation, it does not scale well as dynamically extensible processors are more widely used. The use of tools like M2V can expand the use of hardware acceleration.

M2V accelerates applications from their compiled machine code. This is the only option when an application's source code is unavailable, such as when a third-party writes the application and keeps the sources. Accelerating from binaries can also be advantageous if multiple front-ends are used for development. A developer could use C, LISP, Perl, or any other high-level

programming language for their development, but they all must be compiled to machine code.

The remainder of this document is structured as follows. Section 2 discusses related work, Section 3 gives an overview of the eMIPS hardware platform, Section 4 goes through the M2V data structures and algorithms in detail. Section 5 explains how to run the M2V tool when a BBW source file has been generated. Section 6 goes through the source code implementation for M2V and Section 7 explains the hardware that is generated from M2V in more detail. Section 8 discusses the experimental results, Section 9 gives future research directions, and Section 10 concludes the report. Appendix I contains figures that illustrate scheduling for the primary example in the report. Appendix II contains the Verilog output from M2V for the example in the report. Appendix III contains the BBW file for the example basic block and Appendix IV lists the output from M2V when using the example basic block as input.

2 Related Work

Work on extensible processors can be divided in several ways. One avenue of exploration is to define the underlying hardware. Chimaera [7] and GARP [8] are two examples of extensible hardware from the late 1990's. Commercial FPGA manufacturers today all provide examples of soft-cores, microprocessor designs that the customer can modify and extend for their application [15, 3, 13]. M2V uses the eMIPS processor [6] as its underlying hardware platform. eMIPS is the first design that is secure for general purpose multi-user loads, and the set of potential applications is therefore more open-ended than those found in the typical embedded system alone.

A common approach to generate code for an extensible processor is to modify an existing C compiler. Tensilica [14] regenerates a full GNU compilation system given the RTL of the new instruction. Ienne et al. [4] use the SUIF compiler. To the best of our knowledge, M2V is the first compiler that accepts as input binary machine code rather than source code. There are trade-offs between accelerating from source code in a high-level language or from binaries. One of the major advantages when accelerating from binaries is that any application can be accelerated, even applications where the source code is controlled by an outside party and not available to the system developer. A disadvantage is that some of the information that has been discarded must be reconstructed, and there are limits to this reversal process.

Another avenue of research in extensible processors is the identification of the instruction set extensions (ISE) that most benefit a given program, see for instance [5] for a recent overview. Bonzini [5] advocates generating the ISE from within the compiler, Tensilica [14] from profiling data. M2V currently follows the application profiling approach; it uses the BB-Tools and dynamic full-system simulation with Giano to select the candidate basic blocks. A possible extension to our work is to use M2V in concert with a high-level compiler. Once the ISE is identified from within the compiler, its definition could be output in the form of a BBW file.

A related area is the generation of HDL code from C, the so-called C-to-gates design flows [11, 12]. The input to M2V is binary code, but the target is similar.

3 eMIPS Hardware Overview

The extensible MIPS (eMIPS) processor [6] has been developed at Microsoft Research as an example of a RISC processor integrated with programmable logic. The programmable logic has many uses, such as: extensible on-line peripherals, zero overhead online verification of software, hardware acceleration of general-purpose applications, and in-process software debugging [2]. This report is concerned with automatically generating hardware accelerators.

The instruction set for the eMIPS processor is the instruction set for the R4000 MIPS processor [10]. The R4000 is an example of a classic RISC architecture. The eMIPS pipeline follows the classic RISC pipeline [9] consisting of five stages: instruction fetch (IF), instruction decode (ID), execute (EX), memory access (MA), and register write-back (WB).

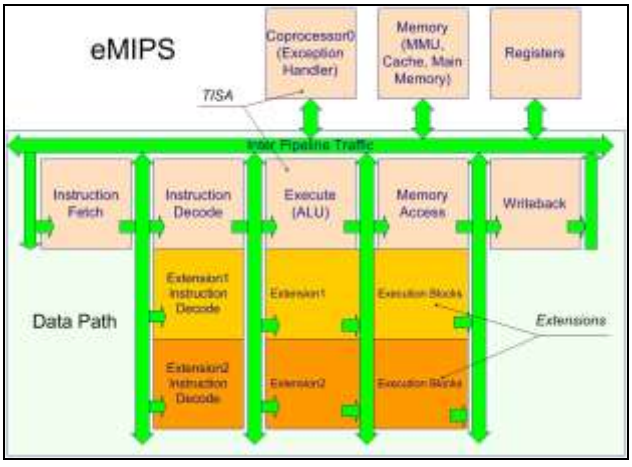

Figure 1: Block diagram of the eMIPS architecture.

The eMIPS processor departs from a standard RISC processor by adding an interface to programmable logic. The programmable logic is tightly integrated with the

RISC pipeline, it can synchronize with it and it can access the same resources as the RISC pipeline. Figure 1 shows a logical block diagram for the eMIPS processor, a physical view of the prototype is depicted in Figure A-8 and Figure A-9. The tight coupling of the pipeline and programmable logic creates a very low latency interface between the accelerator and the RISC pipeline.

Figure 2 (and a repeated, larger, version in Figure A-1) illustrates the pipelining of instructions through eMIPS. The decode logic in the extension logic is always an observer of the main pipeline and is trying to decode the instruction in the instruction decode (ID) phase of the pipeline. When the instruction is not an extension instruction, the extension fails to decode it and it is executed in the main pipeline. When the instruction is successfully decoded by the extension logic, the extension logic is activated and the hardware accelerator is used. Instructions flowing through the main RISC pipeline prior to the extension instruction complete normally. Instructions following the extension are stalled until the extension is near completion, in the EXn-1 cycle.

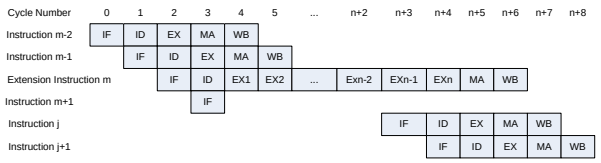

Figure 2: Instruction flow through eMIPS pipeline.

The RISC pipeline imposes microarchitectural constraints on the extension logic, for instance in the arbitration for access to the register file and other resources. The extension logic needs to read and write the register file and access the memory management unit (MMU). These accesses by the extension logic are scheduled by M2V so as not to conflict with the primary RISC pipeline.

The primary RISC pipeline uses two read ports on the register file when an instruction is in the ID stage, it uses one MMU port when in the MA stage, and it uses one write port on the register file when in the WB stage. The eMIPS register file has four ports which are multiplexed between four read ports and two write ports. The extension logic has the potential to use all of the eMIPS register file ports, but it must not conflict with the primary RISC pipeline. Thus, register writes must be delayed by the extension until previous instructions are retired and register reads must be finished a couple of cycles before trailing instructions get to the ID stage.

As a specific example, when the extension instruction is in the EX1 cycle of execution, instruction m-1 is in the MA pipeline stage and so instruction m-1 has access to the MMU. Instruction m-2 is in the WB pipeline stage and it has control of the register file write ports. The extension instruction does not have control of all the resources until stage EX3 when the previous instructions have been retired.

The eMIPS processor has been implemented on a Xilinx Virtex 4 FPGA using the ML401 evaluation board. The partial reconfiguration capabilities of this FPGA allow software to load dynamically the hardware for the instruction extensions.

4 The M2V Compiler

The M2V compiler is one element of a larger eMIPS tool-chain which is illustrated in Figure 3. The goal of the eMIPS tool-chain is to accelerate a pre-compiled application with the programmable extension logic in eMIPS. The goal of M2V is automatically generate the hardware which will be loaded into the extension unit.

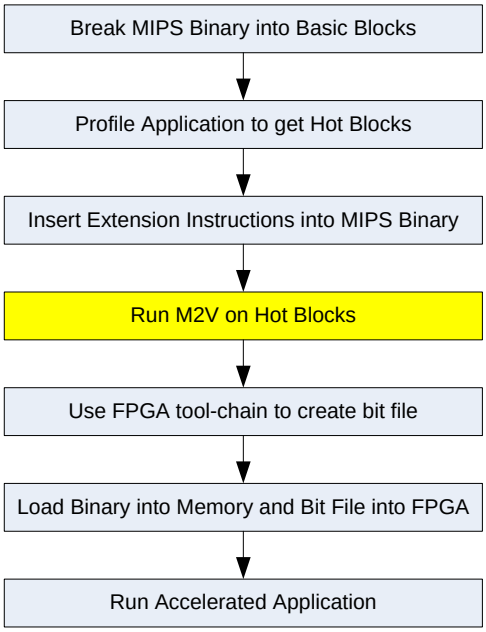

Figure 3: Tool chain for accelerating basic blocks on the eMIPS architecture.

The eMIPS tool-chain includes the BB-Tools. The BB-Tools identify the basic blocks and then profile them on the Giano simulator to get dynamic execution counts. The basic blocks with the highest execution time (the hot blocks) are selected for acceleration. The BB-Tools generate an instruction encoding for the hot block and write out the BBW file (see section 4.1.1). The BB-Tools also insert the extension instruction into the original binary and patch any branch and jump addresses.

The M2V compiler takes the description of the hot blocks in the BBW file and automatically generates synthesizable Verilog for the extension. The Verilog for the extension can be synthesized with the standard FPGA synthesis tool-chain, which eventually produces a BIT file. The BIT file contains all of the programming information for the FPGA to implement the hardware of the extension. The BIT file can be statically loaded at reset of the FPGA or the operating system loader can dynamically load the BIT file using partial reconfiguration when the application is loaded into main memory.

The M2V compiler makes three passes through the database as illustrated in Figure 4. The first pass processes the BBW file and consists of three major steps: map the encoding for the extension instruction to the basic block, analyze the MIPS instructions, and build a circuit graph. The second pass schedules the operations that are represented in the graph. The third pass emits the Verilog that will be synthesized and placed in the eMIPS FPGA.

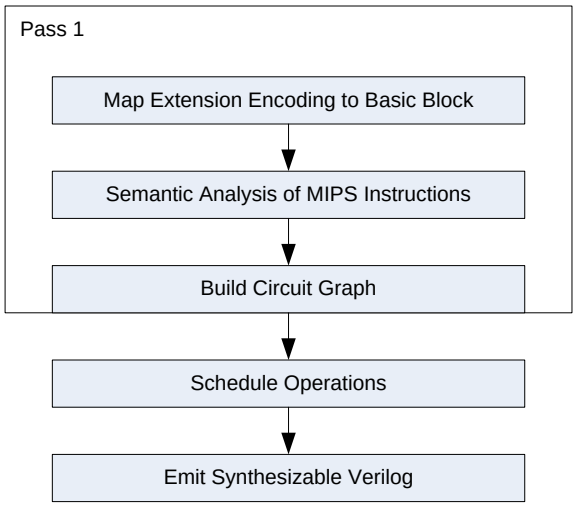

Figure 4: Steps in M2V.

As the algorithm is discussed, it is helpful to use an example to reinforce the concepts. The example that was selected for this purpose comes from the analysis of video games and real-time applications [6]. This basic block implements part of a 64-bit divide on the 32-bit architecture of eMIPS. The assembly code for the example is shown in Figure 5, where the bracketed number is the hexadecimal byte offset for the instruction:

```
[  0 ] ext0 r4, r2, offset0
[  4 ] sll r1, r1, 1
[  8 ] srl r3, r2, 31
[  c ] or r1, r1, r3
[ 10 ] sll r2, r2, 1
[ 14 ] srl r3, r4, 31
[ 18 ] or r2, r2, r3
[ 1c ] sll r4, r4, 1
[ 20 ] srl r3, r5, 31
[ 24 ] or r4, r4, r3
[ 28 ] sltu r3, r1, r6
[ 2c ] beq r0, r3, offset0
[ 30 ] sll r5, r5, 1
```

Figure 5: Example basic block.

The instruction at address 0, *ext0*, is the extension instruction that is inserted by the BB-Tools and is not part of the original basic block. The shift-left-logical instruction at address 30 is in the branch delay slot and it will be executed within the basic block before the instruction at the branch target is executed.

4.1 Pass 1 – Process BBW File

The BBW file is the interface between the BB-Tools and the M2V compiler. The components in the BBW file relevant to M2V are as follows: extension instruction encoding, canonical register relationships, canonical value relationships, sequential MIPS instruction stream in the basic block, and the code size of the basic block. The first pass in the M2V compiler is to parse and analyze the BBW file. The BBW file for the example in Figure 5 is shown in Appendix III.

4.1.1 Mapping Instruction Encoding to Basic Block

For the first revision of M2V, the extension instruction is encoded as a MIPS instruction of the "I" format. The instruction encoding is illustrated in Figure 6. In assembly, the "I" instruction is written:

Opcode_name rt, rs, immediate.

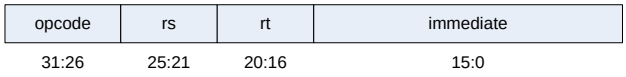

Figure 6: MIPS "I" instruction format used for the extension instruction.

The opcode that identifies the instruction must be unique or it will alias onto an existing instruction. The *rs* and *rt* fields are used to map two actual registers to two canonical registers. The *immediate* field is used for the branch relative address at the end of the basic block.

When the original MIPS executable binary file is broken into basic blocks, the registers are canonicalized. Thus the first actual register in the basic block is assigned

canonical register R1, the second actual register is assigned canonical register R2, etc. By using canonical registers, it is hoped that multiple basic blocks can be mapped to the same canonical basic block. This gives more opportunities to accelerate the application.

Basic blocks that are good candidates for acceleration will typically reference more than two registers. Since the instruction encoding only has enough bits to encode two unique registers, all other registers must be relative to these two registers. The relationships between *rs*, *rt*, and the other canonical registers are recorded in the BBW file and the relationships must be preserved in the hardware accelerator.

Likewise, there are only enough bits in the extension encoding to store one *immediate* value. If there are additional values in the basic block, they must be a constant for the basic block or the value must be relative to the encoded value. The BBW file and the hardware accelerator have the ability to use relative relationships between canonical values.

The relationships between the canonical registers and values are stored in a data structure that is later referenced when emitting Verilog. The data structure is used to generate the proper address for the register when the register file is accessed.

The code size for the basic block is not specifically part of the encoding, but the size is needed so that the program counter (PC) can be calculated for the fall-through case of the basic block.

4.1.2 Analyze MIPS Instructions

The MIPS instructions must be analyzed to create the dependency graph, request pipeline resources, and emit the equivalent Verilog. There are three instruction formats in the MIPS I and MIPS II instruction set architectures supported by M2V: "I," "R," and "J".

The MIPS "I" format is illustrated in Figure 6. "I" instructions use a 16-bit immediate field as one of the operands. Examples of "I" instructions are ADDI, ANDI, BEQ, LW, SW, XORI, etc. Within the "I" instructions, ALU operations, branches, loads, and stores must be distinguished so that the correct dependencies can be built.

The MIPS "R" format is illustrated in Figure 7. "R" instructions read operands from registers and write results back to registers. Examples of "R" instructions are ADD, AND, SLL, etc. In assembly, the "R" instruction is written:

Opcode_name rd, rs, rt.

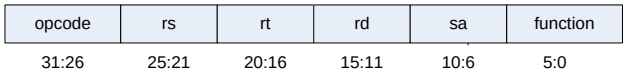

Figure 7: MIPS "R" instruction format.

The MIPS "J" format is illustrated in Figure 8. "J" instructions modify the program counter (PC) using the address field in the instruction encoding. Examples of "J" instructions are J and JAL. In assembly, the "J" instruction is written:

Opcode_name address.

Figure 8: MIPS "J" instruction format.

Analysis of the instructions provides the register and constant operands, the function of the instruction, and the output register for the instruction. The cost of the function is also part of the analysis. The cost is an estimate of how much time the combinatorial logic in an instruction will take. The cost is used when scheduling to determine when paths in the hardware need to be pipelined.

4.1.3 Build the Circuit Graph

The circuit graph is the core data structure for the second and third passes of M2V. It is constructed using the analysis of the MIPS instructions to build dependencies between the instructions. The graph is built in a single pass, as each instruction is sequentially read from the BBW file. There are two types of nodes in the graph: register nodes and instruction nodes.

Instruction nodes represent a single MIPS instruction. Edges into the instruction node are the operands for the instruction. An edge out of the instruction node represents the result of the instruction being streamed to a register value.

Register nodes represent a value that has been read or calculated in the basic block. When the register node is a root of the circuit graph, it represents the basic block's initial read from the register file. To minimize contention on the register file, each register is read at most once from the register file and the value is stored locally in the accelerator. There can be up to 31 roots in the circuit graph representing each of the MIPS registers. An edge into the register node is a value coming from an instruction. An edge out of the register node represents the value being used as an operand to an instruction.

In hardware, the register node may be realized with a pipeline stage from one operation to the next, or it may be a bus without a latch. The scheduling pass determines whether the register node is pipelined or not.

4.1.3.1 The Register Table

The register table is an additional data structure that is not part of the circuit graph, but it is used to eliminate redundant register nodes and to determine which register nodes must be written back to the register file. The register table has an entry for each MIPS register. The table entry is initialized to be invalid. When an instruction uses a MIPS register, the register table entry for that MIPS register is read. If the entry is invalid then the value must be read from the register file and the entry will be updated with a pointer to the current register node. If the entry points to an existing register node, then an edge will be added from the register node to the instruction. When an instruction calculates a value and writes it to a register node, the register table entry for that MIPS register is updated with a pointer to the register node.

The register table therefore always contains the most current value for the MIPS registers. When all instructions have been processed by M2V, a final scan of the register table will indicate the register nodes that contain the basic block's final value for the MIPS registers. Each of these register nodes must be written back to the register file. A maximum of 31 writes, one for each MIPS register, will be sent to register file.

The circuit graph is the final collection of instruction nodes, register nodes, and edges between these nodes.

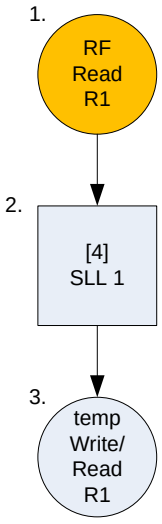

Figure 9: Graph for: sll r1, r1, 1.

4.1.3.2 Examples

The procedure for building the circuit graph will now be illustrated with some examples.

The instruction: [4] sll r1, r1, 1, is mapped to the graph illustrated in Figure 9. The constant shift value, 1, is an operand to the SLL instruction but it is not a vertex on the graph. Since M2V is producing custom hardware, a constant value can be easily optimized. The instruction uses MIPS register R1 both as an operand and as a result register. This creates two separate register nodes in the circuit graph. When the R1 operand is read, the register table entry is invalid so the register file must be accessed. The register table is updated to point to register node 1. When the result from instruction node 2 is written back, the register table entry is written with a pointer to register node 3.

The instruction: [c] or r1, r1, r3, is mapped to the graph illustrated in Figure 10. When the MIPS register R1 operand is read, the register table entry points to register node 3. This register node was generated from the sll instruction at address 4. The two instructions can share the register node. Since the register node is local to the hardware accelerator, the register file does not need to be accessed. Reducing the register file bottleneck is one area where the accelerator improves performance.

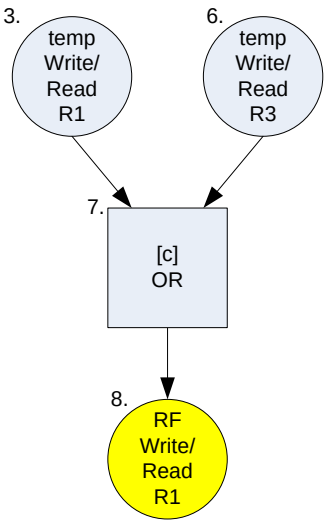

Figure 10: Graph for: or r1, r1, r3.

The circuit graph resulting from the two example instructions is illustrated in Figure 11. The sharing of register node 3 is more explicit in this figure.

The circuit graph for the entire example basic block can be found in Figure A-2 in Appendix I.

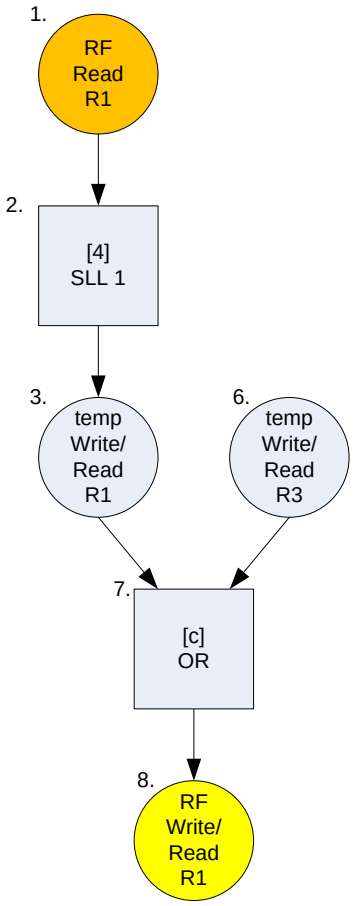

Figure 11: Circuit graph for example instructions.

4.2 Pass 2 - Schedule Operations

After a circuit graph has been generated in pass 1 of M2V, the nodes in the graph can be scheduled. There are two major constraints that the scheduler must consider. First, the cycle time for the chip must be met and no combinatorial path can exceed the cycle budget. Second, the microarchitectural constraints of eMIPS must be respected. There are limited read and write ports on the register file and limited ports on the memory management unit. Also, the memory controller and the register file ports may not be accessible in every cycle due to the progress of instructions before and after the extension instruction through the RISC pipeline.

Figure 2 and Figure A-1 illustrate the progression of instructions through the eMIPS pipeline. In a regular unaccelerated instruction, the register file read port is accessible in the ID pipeline stage, the register file write port is accessible in the WB pipeline stage, and the MMU is accessible in the MA pipeline stage. The extension logic respects these constraints and resources are blocked from within the accelerator when they would conflict with an instruction in the main RISC pipeline. A set of tables store the register file read ports, register file write ports, and MMU ports available to the extension logic on a cycle by cycle basis until steady-state is achieved. Steady-state is when the extension logic has control over all pipeline resources. Another set of tables store the resources that must be free at the end of the extension to prevent conflicts with trailing instructions.

There are four main data structures that are used in the scheduling pass of M2V: the register read list (RRL), the next temporary register queue (NQ), the temporary register queue (TRQ), and the register write queue (RWQ).

The register read list (RRL) was built as the circuit graph was created. The RRL contains all roots of the circuit graph and therefore represents all of the values that must be read from the register file.

The next temporary register queue (NQ) is empty at the beginning of each cycle. As the scheduling algorithm progresses, nodes are added to the NQ if the node has exceeded the cycle-time budget or if the node has unmet dependencies. The NQ represents all nodes that may need to be pipelined before the next cycle.

The temporary register queue (TRQ) is initially empty. At the beginning of each cycle, the NQ from the previous cycle is copied to the TRQ.

The register write queue (RWQ) is initially empty. A register value is placed on the RWQ when it has been calculated and it needs to be written back to the register file.

At the beginning of a clock cycle, the register file read ports, the register file write ports, and the MMU ports are calculated based on the values in the look up tables and the current cycle. The register read ports for this cycle dictate the number of nodes that can be removed from the RRL in this cycle. The register write ports for this cycle dictate the number of nodes that can be removed from the RWQ in this cycle. All nodes can be removed from the TRQ in a cycle because the nodes are temporary calculations that are local to the accelerator.

Nodes are scheduled by covering the circuit graph using a depth-first traversal from the nodes that are available from the TRQ, the RRL, and the RWQ. The traversal continues until a node is encountered with an unmet dependency or until the worst-case combinatorial delay from the last register exceeds the cycle budget. An unmet dependency is defined as an operand that has not

been read or calculated yet. A node is placed on the NQ when its traversal cannot continue.

The cycle ends when the TRQ is empty and the available nodes from the RRL and RWQ have been traversed as far as possible. The NQ is copied to the TRQ, the cycle is incremented, new resource constraints are calculated and the procedure repeats until the TRQ, the RRL, the RWQ, and the NQ are all empty.

Nodes on the TRQ may need to be pipelined or they may get covered later in the same cycle. Thus, when beginning the traversal of a node from the TRQ, the nodes should be examined to see if the original unmet dependency still exists. If the dependency exists the node will need to be pipelined, otherwise the node can be dropped.

The M2V scheduler is greedy: it allocates resources to the nodes at the heads of the RRL and the RWQ. A more optimal solution could potentially be found if different traversals were considered, although this could be more computationally expensive.

Appendix I contains a complete example showing how the basic block in Figure 5 is scheduled. Figure A-3 shows the circuit graph in cycle 1, the ID stage for the accelerator. Figure A-4 shows the circuit graph in cycle 2, the EX1 stage for the accelerator. Figure A-5 shows the circuit graph in cycle 3, the EX2 stage for the accelerator. Figure A-6 shows the circuit graph in cycle 4, the MA stage for the accelerator. Figure A-7 shows the circuit graph in cycle 5, the WB stage for the accelerator.

The unscheduled circuit graph is illustrated in Figure A-2. We will reference the nodes within the graph by their sequence numbers, which are located to the upper left of the node. At the beginning of the scheduling algorithm, all of the register read nodes (1, 4, 11, 18, 23, and 26) are on the RRL and the TRQ, NQ, and RWQ are empty.

Figure A-3 shows the circuit graph in cycle 1, the accelerator's ID stage. Register file reads during this cycle are limited to the two read ports accessed by the RISC pipeline since the accelerator is snooping the results from the RISC pipeline. The register read nodes 4 and 11 are covered in this cycle. Nodes 4 and 11 were removed from the RRL and placed on the NQ (the reads in the ID stage are placed on the NQ as a special case). The TRQ and RWQ are still empty.

Figure A-4 shows the circuit graph for cycle 2, the accelerator's EX1 stage. Two registers (nodes 1 and 18) were taken from the RRL and read from the register file, ten instructions (nodes 2, 5, 7, 9, 12, 14, 16, 19, 21, and 28) are calculated, and four MIPS register values (nodes 8, 15, 22, and 29) are calculated and placed on the RWQ. The NQ from cycle 1 was moved to the TRQ at the beginning of cycle 2. The NQ at the end of cycle 2 contains nodes 6, 8, 10, and 17. Nodes 6, 10, and 17 were placed on the NQ speculatively because the operand nodes 3, 13, and 20 had not been read yet.

Figure A-5 shows the circuit graph for cycle 3, the accelerator's EX2 stage. The final read nodes (23, 26) are taken from the RRL and read from the register file, the final instructions (24, 27) are calculated, and register write node 25 is placed on the RWQ. The NQ from cycle 2 was copied to the TRQ at the beginning of cycle 3. The speculative nodes (6, 10, 17) on the TRQ are pruned when we notice that the successor instruction is already covered. Node 8 is removed from the TRQ, but it is placed on the NQ since the TRQ is emptied before any nodes on the RRL can be removed. The NQ also contains node 25 at the end of cycle 3. One register write (15) is removed from the RWQ and is written back to the register file. All nodes have been covered in cycles 1 through cycles 3, but the NQ and RWQ are not empty so the scheduling continues in the next cycle.

Figure A-6 shows the circuit graph for cycle 4, the accelerator's MA stage. The NQ from cycle 3 is copied to the TRQ at the beginning of cycle 4. Both nodes on the TRQ are pruned because the successor instruction node is covered. Two nodes (8, 22) are removed from the RWQ and written back to the register file. The RRL, TRQ, and NQ are empty. The RWQ has two nodes left and so the scheduling algorithm continues for another cycle.

Figure A-7 shows the circuit graph for cycle 5, the accelerator's WB stage. The final two nodes (25, 29) are removed from the RWQ and written back to the register file. All queues are empty and the scheduling phase is complete.

4.3 Pass 3 - Emit Verilog

The third and final pass in M2V is emitting Verilog. There are four contributions to the final Verilog file: the eMIPS invariant code, the BBW dependent code, the circuit graph dependent code, and the cycle dependent code.

The eMIPS invariant code, or boilerplate code, is the same for all eMIPS extensions. This Verilog code contains the interface to the primary eMIPS pipeline, routing through the bus-macros for partial reconfigurability, and the logic that is common for all extensions.

The BBW dependent code is Verilog that is specific to the instruction encoding and the register and value relationships encoded in the BBW file. During pass 1, the BBW information was extracted and stored in a data structure. The decode logic for the extension instruction

is generated using the opcode stored in the BBW file. The canonical register is sufficient as an identifier for temporary variables, but the actual register addresses must be used when accessing the register file. A table lookup maps between canonical register addresses and actual MIPS register addresses.

The circuit graph dependent code is generated with a final walk covering all the nodes in the circuit graph. As nodes on the circuit graph were generated and scheduled, the nodes were decorated with information that is now used to generate the final Verilog code. The circuit graph dependent code is: variable declarations for all register nodes, register file address and data logic, pipeline logic for register nodes that need it, and combinatorial logic for the instruction nodes.

The cycle dependent code is code that is generated for the state machines in the extension. This code is only dependent on the number of cycles in the extension and the general shape of the circuit graph is not important.

The Verilog code emitted for the entire example basic block in Figure 5 can be found in Appendix II. Additional description for the Verilog implementation can be found in Section 7.

5 Running the M2V Compiler

The makefile supplied with the BB-Tools will also compile the M2V tool. The command, "nmake", will compile all of the BB-Tools including m2v.exe using the visual C++ compiler. Alternatively, "nmake m2v.exe", will compile just m2v and its dependencies.

The command line for m2v is:

m2v.exe [-v] infile [outfile].

Typing m2v by itself will give the usage for the command. The optional verbose option, -v, sends information to standard out about how the compile is progressing. Details about the extension encoding, construction of the circuit graph, and scheduling information are output. The verbose option will also add comments to the combinatorial logic in the Verilog output so that a clear mapping between the MIPS binary and the Verilog can be seen.

The *infile* argument is the name of the BBW (.bbw) file that is used as input to the m2v compiler. The BBW file is generated by the BB-Tools. The optional *outfile* argument is the file name for the Verilog output. If an output file is not specified, the output is written to *a.v*.

The synthesizable Verilog generated by M2V can be simulated in the eMIPS infrastructure and it can be synthesized with the standard FPGA design tools. Detailed instructions on how to synthesize an eMIPS Extension bit file are included in the eMIPS release documentation.

6 M2V Implementation Details

In this section we will discuss the structure of the M2V source code in more detail. The majority of the code for M2V was written in C++. The exception is the semantic analyzer, mips_dissect.c, which was written in C. The main routine is in m2v.cpp. Other files of interest are: Circuit.cpp, Instruction.cpp, Register.cpp, and RegTable.cpp. Each of these files has a header file, *.h, associated with it as well.

The *main* function in m2v calls the routines that step through each compilation pass. The first pass scans the BBW file which details the encoding of the extension instruction and enables the building of the circuit graph. The second pass schedules the operations using *Circuit.assignCycle* and the third pass emits the Verilog using *Circuit.emitVerilog*.

In pass 1, the *parseEncoding* procedure parses the extension instruction encoding from the BBW file. The encoding gives the extended op-code for the instruction, the canonical registers in the RS and RT fields, and the canonical value in the immediate field. There are not enough bits in the extension instruction to encode every canonical register in the basic block. To work around this problem, the basic block definition in the BBW file defines the fixed relationships between the registers that are actually encoded in the instruction and all the other canonical registers. For example, RS may hold canonical register R2, and canonical R1 could be defined as R1 = R2 + 1. The classes *RegEncoding* and *ValEncoding* (in m2v.h) store a single relationship and the *BbwData* class encapsulates all of the relationships which are needed when emitting Verilog in pass 3.

The semantic analyzer in pass 1 is implemented in the *MipsDissect* procedure. This procedure takes the binary encoding of a single MIPS instruction and produces the information needed to build the circuit graph and to emit the Verilog for this instruction in pass 3. To build the circuit graph we must determine the register operands and the destination register for the instruction. To emit the Verilog we need to record the function (semantic) of the instruction and the value of any immediate operands or constants. This information is stored in the *disRecord* structure (mips_dissect.h), accessible via a pointer in the instruction node. This procedure creates one such structure for each instruction in the MIPS binary.

The *Circuit* class contains the data structures and procedures for building, manipulating, and using the circuit graph. The *dis2nodes* method completes pass 1 of the compilation. It uses the analysis from *MipsDissect* to build *Register* and *Instruction* nodes and connect them into a circuit graph. The *assignCycle* and *emitVerilog* methods are used for compiler passes 2 and 3.

The *dis2nodes* method uses the analysis information in *disRecord* to further divide a single MIPS instruction into an *Instruction* node and a few *Register* nodes, building (a fragment of) the circuit graph. *Register* nodes are needed for an instruction's operands and to store its result. When a *Register* node is created for the first read from the register file, *dis2nodes* pushes the node onto the register read queue, *regRdVec*, which is an entry point into the circuit graph. The edges of the circuit graph represent the dependencies between instructions and data and are stored within the *Instruction* and the *Register* nodes.

The *Register* class represents an intermediate value for a MIPS register. A value may come directly from the register file or it may be the result of an instruction. The value may need to be written back to the register file and it may be used as the operand to one or more instructions. The class maintains pointers to the source of the value and the destinations that use the value. The value in a *Register* node may be pipelined to make the cycle-time or it could be strictly combinatorial. A recursive walk method is used in pass 2 of the compilation to traverse the circuit graph when scheduling operations.

The *Instruction* class stores the results of the analysis in *MipsDissect* and maintains pointers to operand and destination data. A recursive walk method is used in the scheduling pass (pass 2) to traverse the circuit graph.

The *RegisterTable* class is a compile time resource that stores the current location of a MIPS register value. There is an entry in the table for each MIPS register. The entry is initially marked invalid. When an instruction within the basic block uses a MIPS register for the first time, a *Register* node is created and the *RegisterTable* entry for that register number is modified to point to the new node. Subsequent reads of the same register can then come from the *Register* node rather than from the register file. When an instruction writes to a register, a pointer to the instruction's result is stored in the *RegisterTable* entry. Maintaining the *RegisterTable* allows instructions to share *Register* nodes when the instructions use the same value and it minimizes accesses to the register file. The *setWriteBacks* method is called at the beginning of the second pass to determine which registers have been written during the basic block's execution. These Register nodes are marked and will be the only values written back to the register file.

Pass 2 of the compiler takes the circuit graph and generates a schedule for the operations. The top-level code for pass 2 is the *assignCycle* method in the *Circuit* class. The steps in pass 2 are to initialize the cycle-by-cycle resource restrictions, determine the MIPS registers that need to be written back, walk the entire circuit graph to assign cycles, and potentially add extra cycles to the extension to maintain proper pipeline behavior.

Access to the register file and to the memory controller is constrained because instructions in the eMIPS pipeline before or after the extension may own those resources. Three classes of variables define the resources available in each cycle. The *init** arrays define the number of read ports, write ports, and load/store ports available in each cycle as the extension instruction starts. The *ss** variables define the maximum number of resources available during steady-state execution. The *fin** variables define the resources available as the extension is completing.

The array values used for register access are defined in the Circuit.h file. The Circuit.h file contains the architecture specific constants for eMIPS. By changing these constants, a different architecture could be supported. For revision 1.0 of M2V, there are 2 reads allowed in each cycle of the extension. There are 0 writes allowed in cycles 1 and 2, 1 write allowed in cycle 3, and 2 writes allowed in steady-state. Architectural explorations are possible by changing the constants in Circuit.h.

The walk of the circuit graph takes several iterations to complete, where each iteration represents a cycle in the hardware accelerator. During a cycle, all nodes will be removed from the TRQ, *regTmpVec*, some nodes will be removed from the RRL, *regRdVec*, and some nodes may be removed from the RWQ, *regWrVec*. The cycle number and the array values described above define how many nodes are removed from the RRL and RWQ.

The walk of the graph begins in a register node and continues depth first until an instruction with an unmet dependency is found or until the cycle budget is exceeded. The *walk* methods in the *Register* and *Instruction* classes perform the traversal through the dependents until there is an unmet dependency, the cost function is exceeded, or there is no successor to the node.

The cost is calculated by taking the cost from the previous node and adding the cost for this node. The maximum cost defined in Circuit.h is roughly the estimated logic levels from the last register. By keeping the logic levels within the cost function, the cycle budget should be met. For an instruction node, the cost is calculated by taking the highest cost from all of its operands and adding the incremental cost for the instruction. Since a register has a single entry point, the

cost is calculated by adding the incremental cost of the register node to the cost entering the node. The incremental cost is a function of fan-out since high fan-out will increase the wiring delay to subsequent nodes.

The walk ends when the return code from the dependent node indicates failure due to excessive cost or an unmet dependency. When the node receives a failure code from its dependent, the node is pushed to the NQ, *nxtQ*.

The register and instruction nodes are decorated with cycle, cost, and pipeline information during the walk. The sum of the decorations creates the final schedule and they are used to emit the correct Verilog in the final pass of the circuit graph.

When the allowed nodes have been removed from the TRQ, RRL, and RWQ in a given cycle, the next cycle can begin. Every cycle begins by copying the NQ from the previous cycle to the TRQ for this cycle. The scheduling pass is complete when the NQ, TRQ, RRL, and RWQ are empty.

The final pass of the circuit graph emits the Verilog for the hardware accelerator. The top-level method for this pass is *Circuit.emitVerilog*. This method combines invariant, BBW dependent, circuit graph dependent, and cycle dependent code in the correct order to produce the Verilog code for the hardware accelerator. The methods called by *emitVerilog* to produce the Verilog are described in Section 7.

The invariant part of the extension is stored in three Verilog files. The m2v_mod_bp.v file is the wrapper logic for the extension which contains the basic interface to the rest of eMIPS. The m2v_ex_bp.v file contains the module inputs and outputs between the extension logic wrapper and the logic for the execution stage in the extension. The m2v_state_mc.v file contains the declarations and logic for the read, write, and branch state machines that are the same for every extension.

7 Hardware Implementation Details

Appendix II lists the entire Verilog code that is generated by M2V for the basic block in Figure 5. As discussed in Section 4.3, there are four contributions to the final Verilog file: the eMIPS invariant code, the BBW dependent code, the circuit graph dependent code, and the cycle dependent code.

Lines 1-540 of the Verilog are the first lines of invariant code in the accelerator definition. Lines 1-300 define the extension's top-level module, lines 301-425 define the bus macros for the execution-to-write-back interface, and lines 426-540 define the bus macros for the instruction-decode-to-execution interface. Lines 1-300 are simply copied from m2v_mod_bp.v at runtime.

The extension's top-level module defines the interface signals between the extension and the rest of the eMIPS design. It contains multiplexor logic for the shared data busses to the register file and the program counter update logic. It also instantiates four modules that make up the core of the extension: the instruction decode logic, the execution logic, and the two bus macro modules.

The bus macros provide connectivity between the extension logic and the primary eMIPS logic. They represent physical routing locations and are required for partial reconfiguration.

The instruction decode logic defined in lines 541-600 is BBW dependent code. This logic decodes the instruction in parallel with the primary RISC pipeline. If the opcode of the instruction matches the opcode of the extension, the logic will assert the RI signal so that the extension logic can take control from the RISC pipeline. The fall-through address for the basic block is sent to the program counter. The fields within the instruction are decoded and sent to the execution logic. The first revision of M2V hardcodes the extension instruction to the MIPS "I" format. The *Circuit.emit_decode* method generates this code.

The extension execution logic is defined in lines 601-1038. The execution logic is composed of invariant code, BBW dependent code, circuit graph dependent code, and cycle dependent code.

Lines 601-666 define the interface signals between the execution logic and the rest of eMIPS. It is invariant for every extension and is copied from m2v_ex_bp.v at runtime.

Lines 667-695 define the Verilog registers that are used later in the execution logic. This code is circuit graph and cycle dependent. The registers for the register node values follow a convention to create an identifiable mapping between the generated logic and the circuit graph. The format is rX_Y[_r], where X is the actual MIPS register, Y is the sequence number of the register node, "_r" indicates that the value comes directly from a register, and the absence of "_r" indicates that the value comes from combinatorial logic. Thus, r9_3 is a combinatorial value for MIPS register 9 that corresponds to the register node with sequence number 3. The *Circuit.emitVarDecl* method generates this code.

Lines 696-891 define the state machines that interface with the register file and the program counter logic. This code is invariant and is copied from m2v_state_mc.v at runtime. These state machines are eMIPS-specific.

Lines 892-924 define the register file and program counter usage for each cycle in the extension. This information is used by the state machines defined in lines 696-891. This code is generated by the *Circuit.emitCycState* method.

Lines 925-959 define the register file interface logic. Since there are limited ports on the register file, the read and write addresses are scheduled onto the register file address lines. Likewise, read data from the register file must be routed to the correct register node, and write data to the register file must come from the correct calculation. The *Circuit.emitRFLogic* method generates this code.

Lines 960-980 define the pipeline registers that are needed by the extension logic. When a calculation must be pipelined, it is latched at the end of the calculation cycle and held for the remainder of the extension's execution. The *Circuit.emitPipeReg* method generates this code.

Lines 981-1022 define the combinatorial logic for the instruction nodes. This code is generated by the *Circuit.emitInstLogic* method.

Lines 1023-1038 define the primary extension state machine. The state machine is 1-hot encoded with one state representing one cycle in the schedule so the states can be directly used as control signals. The machine is idle until an extension is successfully decoded and it steps through each cycle in the extension. The *Circuit.emitESM* method generates this code.

8 Experimental Results

The compiler is at its very early stages of development, but nonetheless the first simple test we ran gave very positive indications. We used the basic block of Figure 5, for which we already had both a hand-written version of the eMIPS Extension and a test program that exercised it. The test program executes some 500+ 64-bit division tests, validating the results against the tabulated ones. It is one of the standard basic validation tests for the Microsoft Invisible Computing RTOS.

	Hand-coded	M2V generated
Minimum Period	5.729 ns	5.886 ns
Flip-Flops	755	494
Slices	867	448
4-Input LUTs	1542	810

Table 1: Synthesis results.

The hand-coded accelerator and the M2V-generated accelerator were synthesized and verified on the Virtex-4 XC4LX25 FPGA, using Xilinx ISE v8.2i. The synthesis results are summarized in Table 1.

There are twelve instructions in the original basic block in Figure 5. With a CPI of 1, it takes twelve cycles to complete the unaccelerated basic block. M2V was able to accelerate the block such that it only needed five cycles resulting in a 2.4 speed-up, under idealized CPI conditions.

The actual speed-up of the application is dependent on the number of times that the basic block is executed over the course of the application and on the actual CPI. Notice that memory does not need to be fetched for instructions in the accelerated block, whereas it does for the unaccelerated one. On the ML401 board the SRAM chips have a worst-case latency of 3 cycles and a pipelined latency of 1 cycle, but only for batch-mode fetches (e.g. cache refill). Since eMIPS does not currently have a cache each instruction fetch costs 3 cycles. This gives a speed-up of 12 from reduced I-fetches alone, and a projected speed-up of 4.5 for the basic block in isolation.

The output and timing results from running the test program are shown in Appendix I, Figure A-14. Both versions obtained an overall speed-up of about 2.3 for the overall test program, over the unaccelerated version.

Of particular interest on eMIPS is the cost of an Extension in terms of its area utilization. The FPGA chip used on the ML401 board has fairly limited resources, being as it is the second-smallest chip in the Virtex-4 family of devices. The area we can devote to an Extension is consequently also limited. Figure A-8 is a rendition of the floor plan used for eMIPS, as depicted by the PlanAhead tool from Xilinx. On the left-hand side of the picture is the rectangle for the Extension logic. Figure A-9 shows a detail of the Extension floor plan, indicating the area used for the bus macros – the inter-connection points between the TISA and the Extension. There are many signals in the interface, and consequently a high-price to pay for them.

Per-iteration speedup, CPI=1 (idealized)	2.4
Speedup from eliminated I-fetches	12.0
Per-iteration speedup, CPI=3 (actual)	4.5
Application speedup (actual)	2.3
Area reduction factor	2
Maximum frequency reduction	2.8%

Table 2: Summary of the performance results.

Figure A-10 shows a few selected statistics generated by the PlanAhead tool for the resource utilizations of the

hand-generated test extension. Of notice is the 58% utilization of the LUTs available in the Extension area. Figure A-11 shows the same statistics for the M2V-generated version, with a LUT utilization of only 30%.

The synthesis reports (before place and route) from the Xilinx ISE v8.2-PR are shown in Figure A-12 for the hand-generated version and Figure A-13 for the M2V-generated version, respectively. The reports confirm the PlanAhead indications of a factor of 2 area reduction. The timing reports show that the hand-generated version can potentially run at a higher frequency of about 175 MHz against 170 MHz for M2V. Both are well beyond the clock frequency of 100 MHz used on the ML401 board for eMIPS. Table 2 summarizes the various performance indicators discussed above.

9 Limitations and Future Directions

The extension logic in eMIPS has direct access to the memory management unit (MMU), but M2V does not currently implement load and store instructions. Load and store operations can be both supported and optimized in the M2V compiler. Bandwidth to and from the memory can be increased by making the data path to memory wider than a conventional MIPS instruction would allow. For instance, an entire cache line could be read or written in a single cycle. Reads and writes can be reordered in the instruction stream to better hide latencies.

M2V must become aware of potential translation faults. The general scheme in eMIPS is to keep the original binary code in the executable, so that execution can be restarted from within it. M2V could keep a virtual program counter, associated with the register writebacks, to indicate where execution should restart in case of an MMU exception. This approach also solves the problem of handling external interrupts in a timely fashion.

M2V currently generates Verilog that executes in the minimum number of cycles for the given resource constraints. This can result in values being computed before they are needed. An additional pass through the graph could delay calculation of results until they are actually needed, a just in time approach. This would allow for idle hardware to be reused and could result in better overall utilization of the programmable logic.

Multiple basic blocks could be combined into a single accelerator. The state machines to control the control flow are simple, but multiple register tables might be needed to account for conditional branch instructions.

M2V is the first implementation of the general idea of taking a binary executable image and converting it to hardware. It understands MIPS instructions currently, but it could be extended so that the executable code could in fact be x86, ARM, or any other instruction set. Rather than targeting synthesizable Verilog, M2V could be modified to generate gates using a standard cell library or a number of other hardware elements. Note that if gates were to be generated directly, M2V would need additional optimization steps to improve the quality of the output. Currently, M2V takes advantage of the FPGA synthesis tools to perform a number of optimizations.

10 Conclusions

By realizing M2V we have demonstrated that it is possible to automatically generate efficient hardware accelerators, starting directly from binary code. In contrast to existing approaches, M2V does not require any compiler modifications and therefore it supports any programming language and even cases where the code was directly written in assembly, or the sources are not available.

The implementation shows that the execution acceleration is based on a number of factors. Register file access is improved by scheduling register operations as soon and as efficiently as possible and by increasing the number of ports on the register file. Also, temporary writes and reads to the register file can be avoided as all intermediate results are kept in the accelerated logic.

Parallelism in the code is automatically extracted from the dependency graph. Instructions are built to proceed in parallel, as soon as their operands are ready. There are more operands available at any given time, because temporary variables are streamed directly to the instruction(s) that need them and there are more ports on the register file.

Multiple sequential instructions are executed in a single cycle when they fit within the cycle budget of the accelerator. For example, the SLL instruction can be executed in zero time on the hardware because it is only redefining the numbering of the bits within the bus.

The pressure on the memory bus and/or on the cache is greatly reduced. Memory bandwidth is freed up because the instructions in the accelerated basic block do not need to be fetched. In future revisions of M2V, loads and stores can be scheduled to optimize bandwidth resources and wider data-paths to the memory can also help.

References

[1] Aho, A. V.., Lam, M. S., Sethi, R., Ullman, J. D. *Compilers: Principles, Techniques, and Tools.* Addison Wesley Publishers, Boston, MA. 2007.

[2] Almeida, O., Forin, A., Garcia, P., Helander , J., Khantal, N., Lu, H., Meier, K., Mohan, S., Nielson, H., Pittman, R. N., Serg, R., Sukhwani, B., Veanes, M., Zorn, B., Berry, S., Boyce, C., Chaszar, D., Culrich, B., Kisin, M., Knezek, G., Linam-Church, W., Liu, S., Stewart, M., Toney, D. *Embedded Systems Research at DemoFest'07.* Microsoft Research Technical Report MSR-TR-2007-94, July 2007.

[3] Altera Corp. *Excalibur Embedded Processor Solutions*, 2005. .http://www.altera.com/products/devices/excalibur/excindex.html,

[4] Biswas, P., Banerjee, S., Dutt, N., Ienne, P., Pozzi, L. *Performance and Energy Benefits of Instruction Set Extensions in an FPGA Soft Core* VLSID'06, pag. 651-656

[5] Bonzini, P., Pozzi, L. *Code Transformation Strategies for Extensible Embedded Processors* CASES'06, pagg. 242-252.

[6] Forin, A., Lynch, N., L., Pittman, R. N. *eMIPS,A Dynamically Extensible Processor.* Microsoft Research Technical Report MSR-TR-2006-143, October 2006.

[7] Hauck, S. et al. *The Chimaera Reconfigurable Functional Unit*. IEEE VLSI, 2004.

[8] Hauser, J. R., Wawrzynek, J. *Garp: A MIPS Processor with a Reconfigurable Coprocessor*. FCCM'97 pagg 12-21, April 1997.

[9] Hennessy, J. L., Patterson, D.A. *Computer Organization and Design: The Hardware/Software Interface.* Morgan Kaufmann Publishers, San Francisco, CA. 1998.

[10] Kane, G., Heinrich, J. *MIPS RISC Architecture*. Prentice Hall, Upper Saddle River, NJ. 1992.

[11] Kastner, R., Kaplan, A., Ogrenci Memik, S. Bozorgzadeh, E. *Instruction generation for hybrid reconfigurable systems* TODAES vol. 7, no. 4, pagg. 605-632, October 2002.

[12] Lau, D., Pritchard, O., Molson, P. *Automated Generation of Hardware Accelerators with Direct Memory Access from ANSI/ISO Standard C Functions. FCCM'06, pagg. 45-54, April 2006.*

[13] Stretch, Inc. http://www.stretchinc.com 2006.

[14] Tensilica, Inc. http://www.tensilica.com, 2006.

[15] Xilinx Inc. *Virtex 4 Family Overview*. Xilinx Inc., June 2005. Available at http://direct.xilinx.com/bvdocs/publications/ds112.pdf

Appendix I – Additional Figures

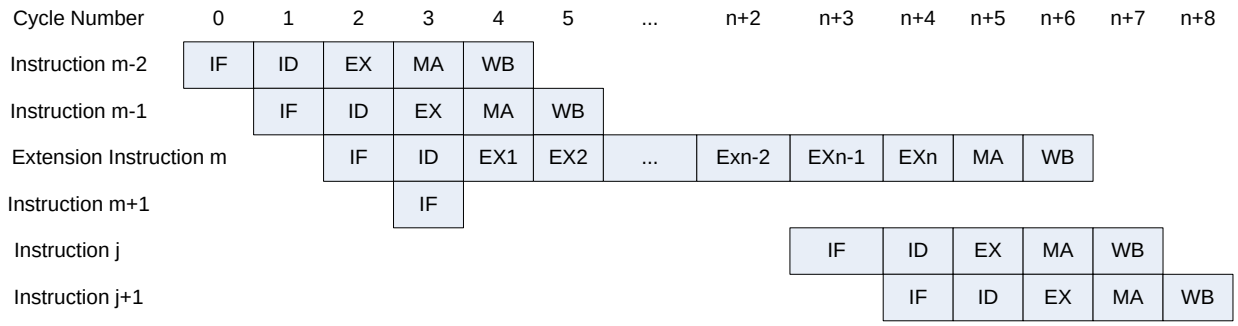

Figure A-1. Extension Instruction and interaction with MIPS Pipeline.

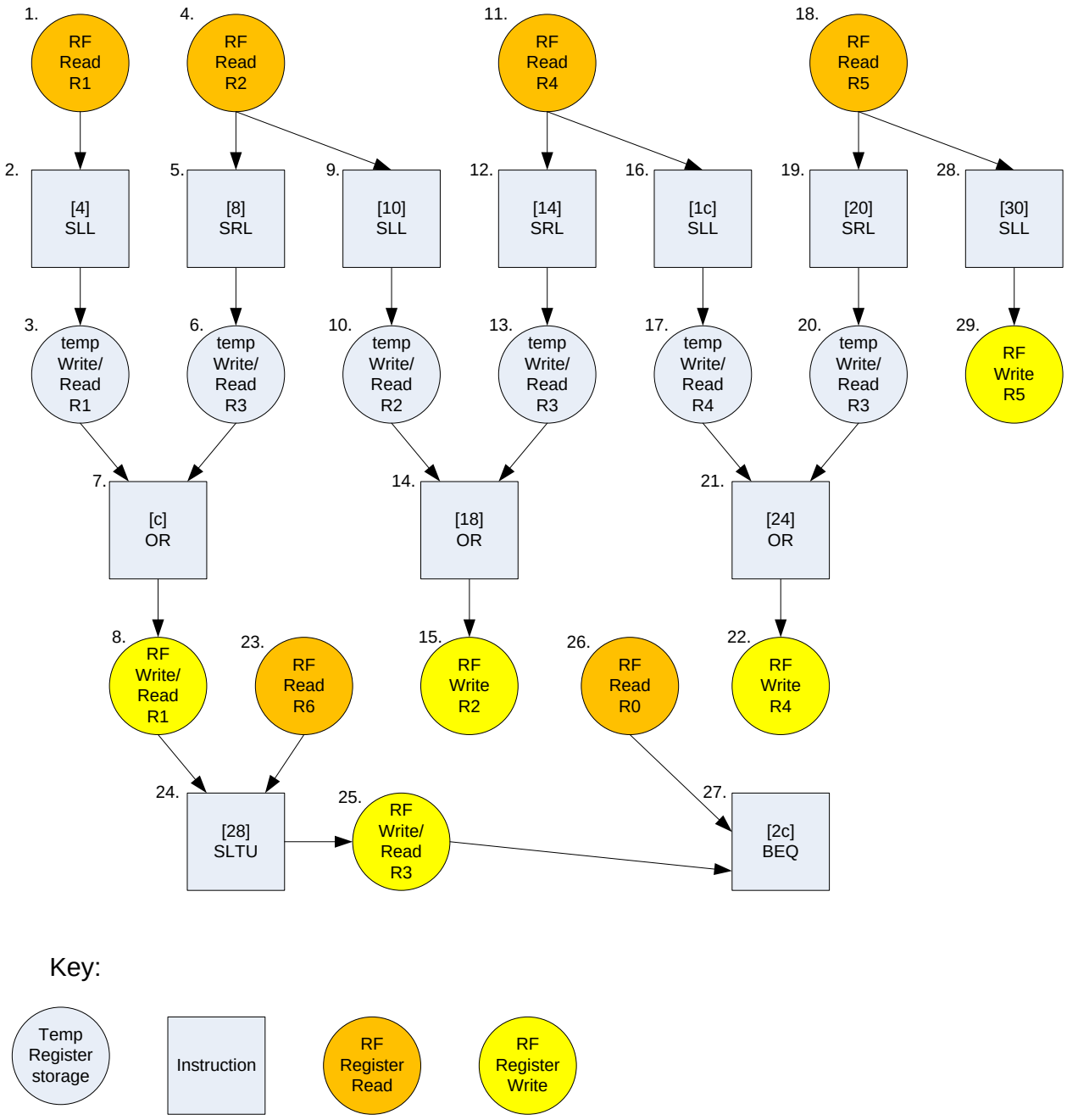

Figure A-2. Circuit Graph for Example Basic Block.

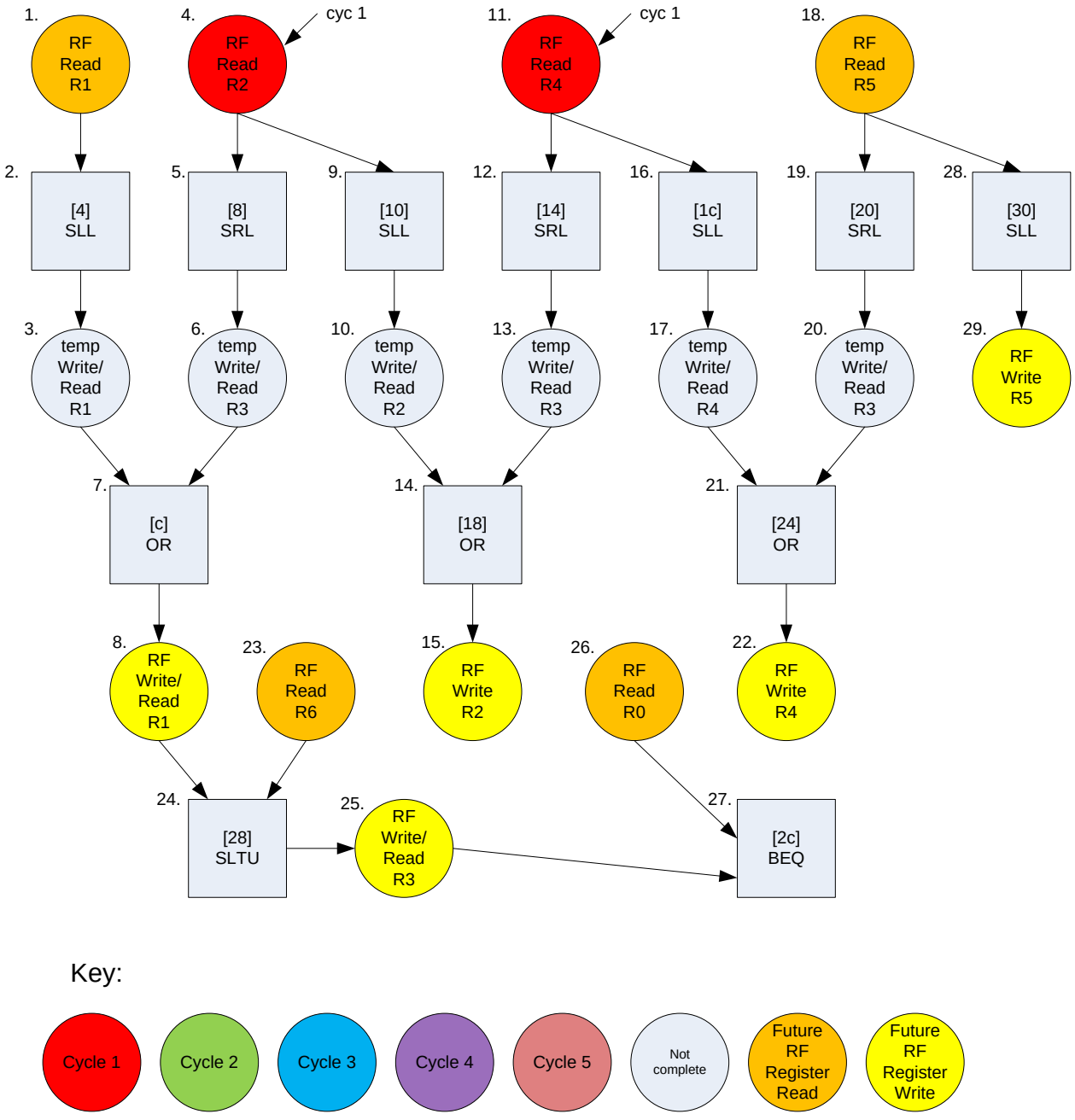

Figure A-3. Circuit Graph for Example Basic Block in Cycle 1.

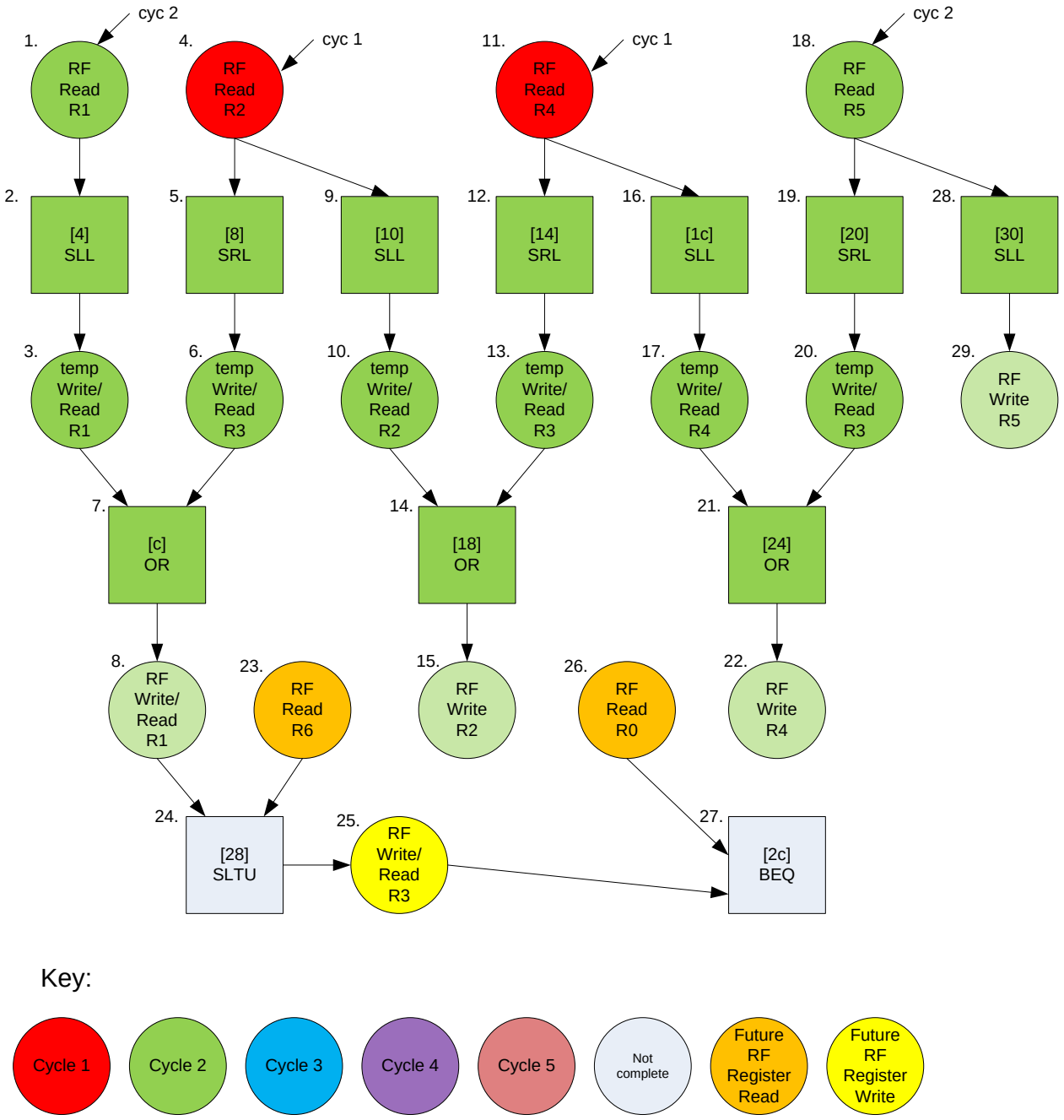

Figure A-4. Circuit Graph for Example Basic Block in Cycle 2.

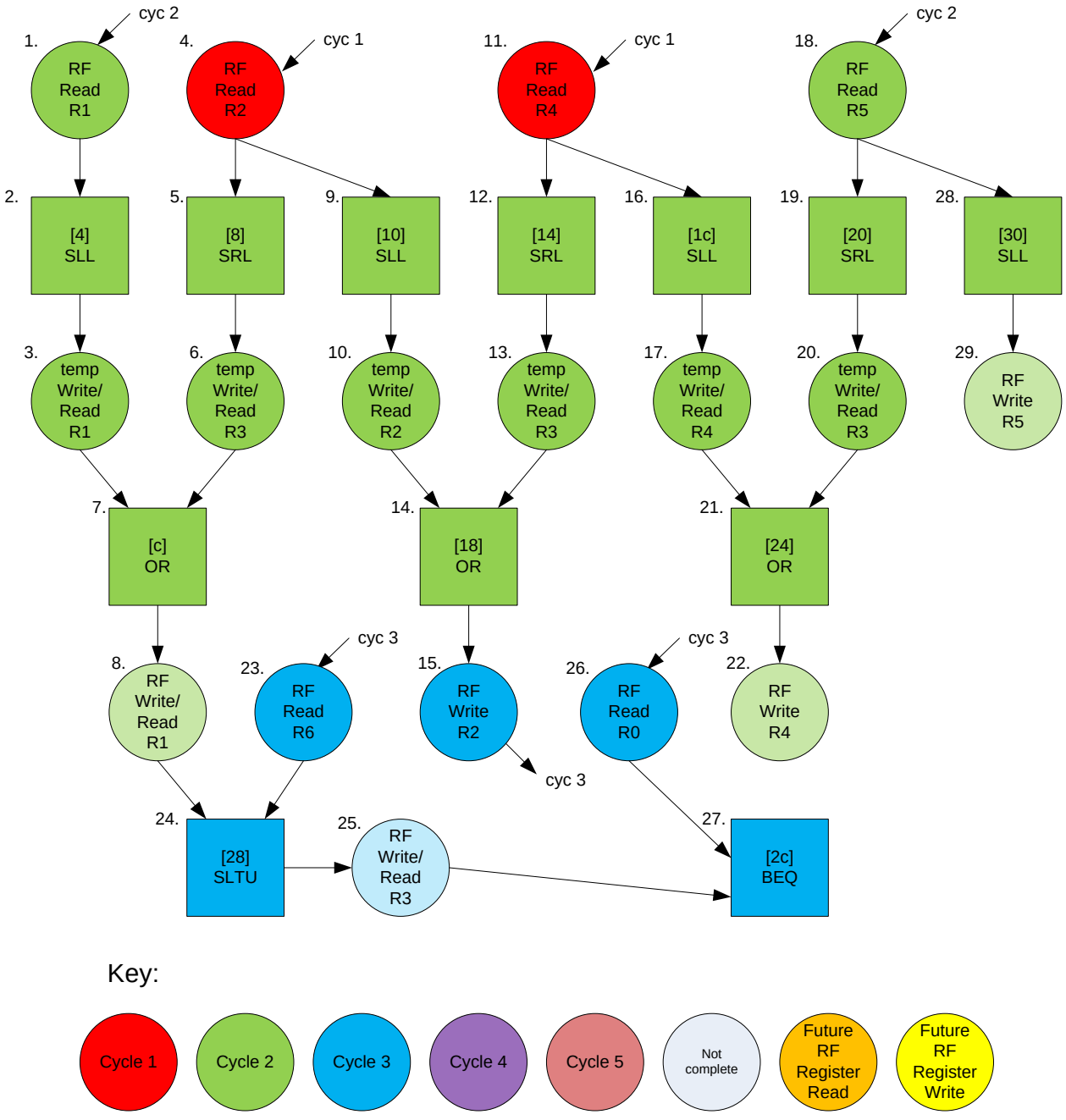

Figure A-5. Circuit Graph for Example Basic Block in Cycle 3.

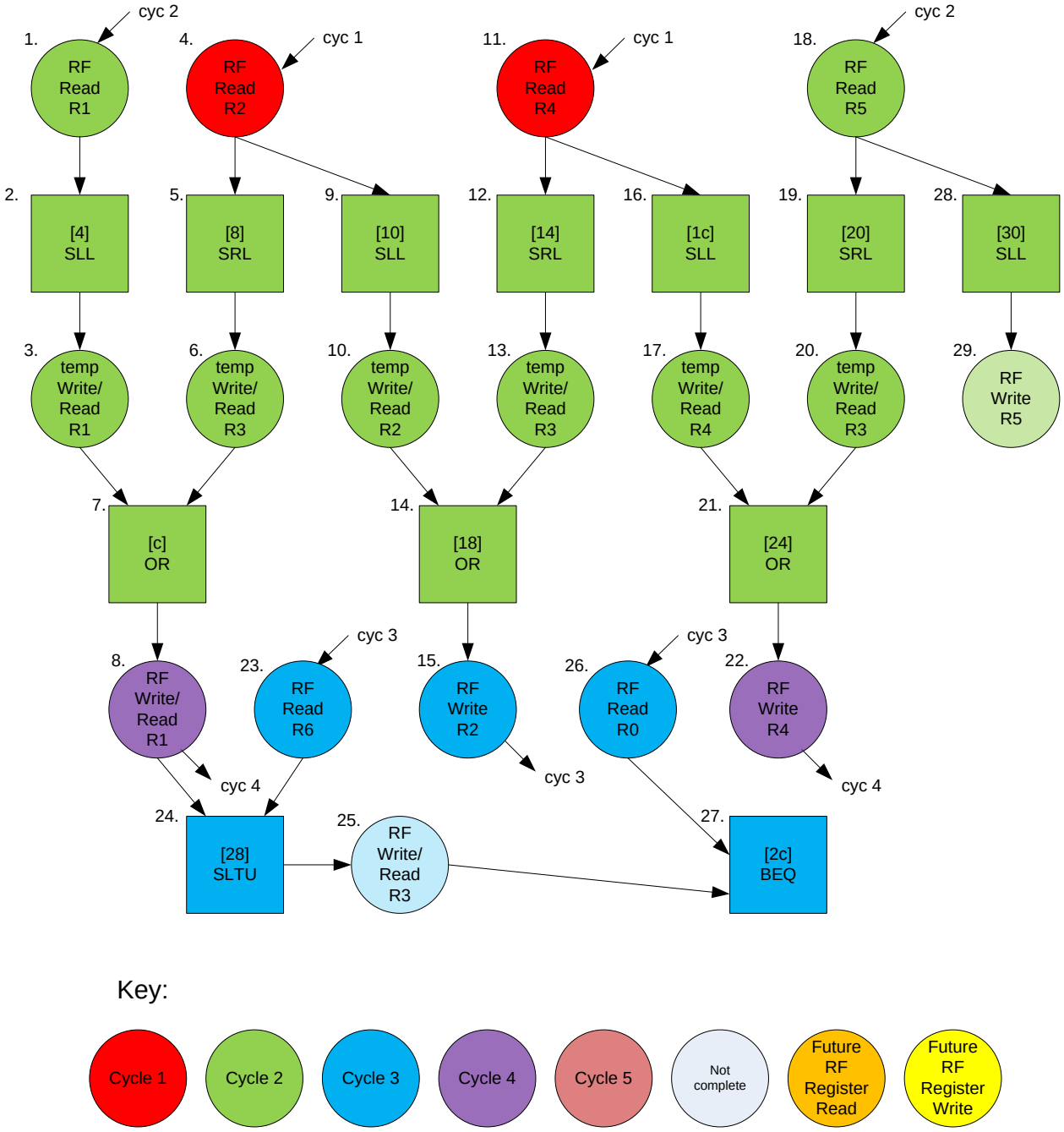

Figure A-6. Circuit Graph for Example Basic Block in Cycle 4.

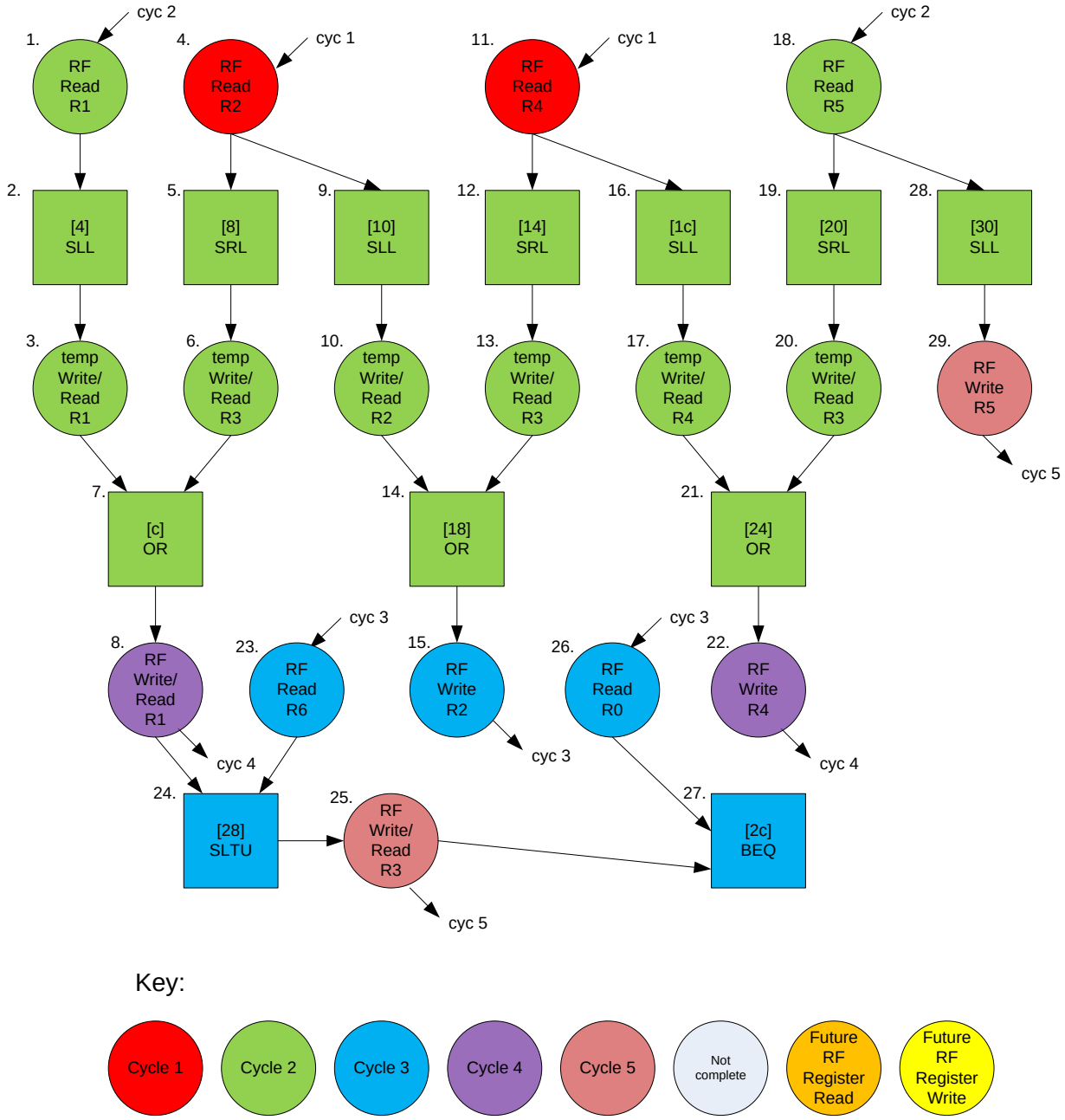

Figure A-7. Circuit Graph for Example Basic Block in Cycle 5.

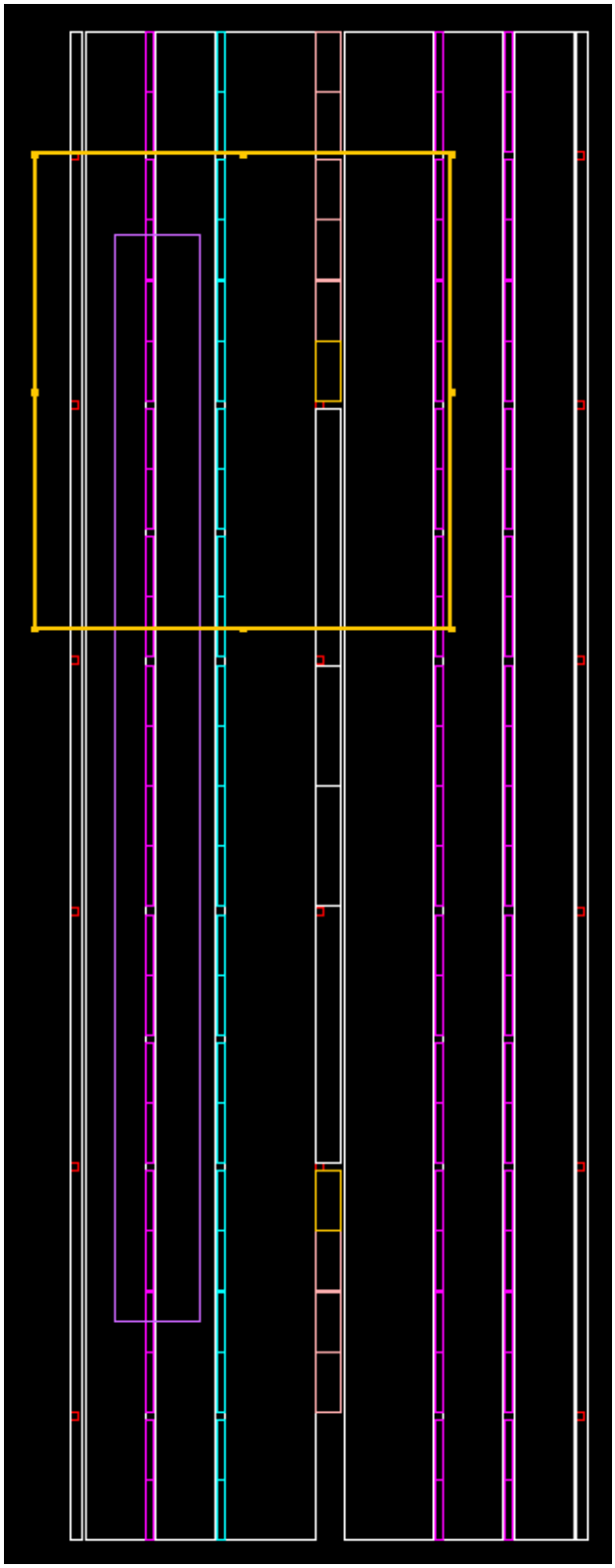

Figure A-8. Floor-plan of the eMIPS processor. The area for the extension slot is the elongated purple vertical rectangle on the left side of the chip.

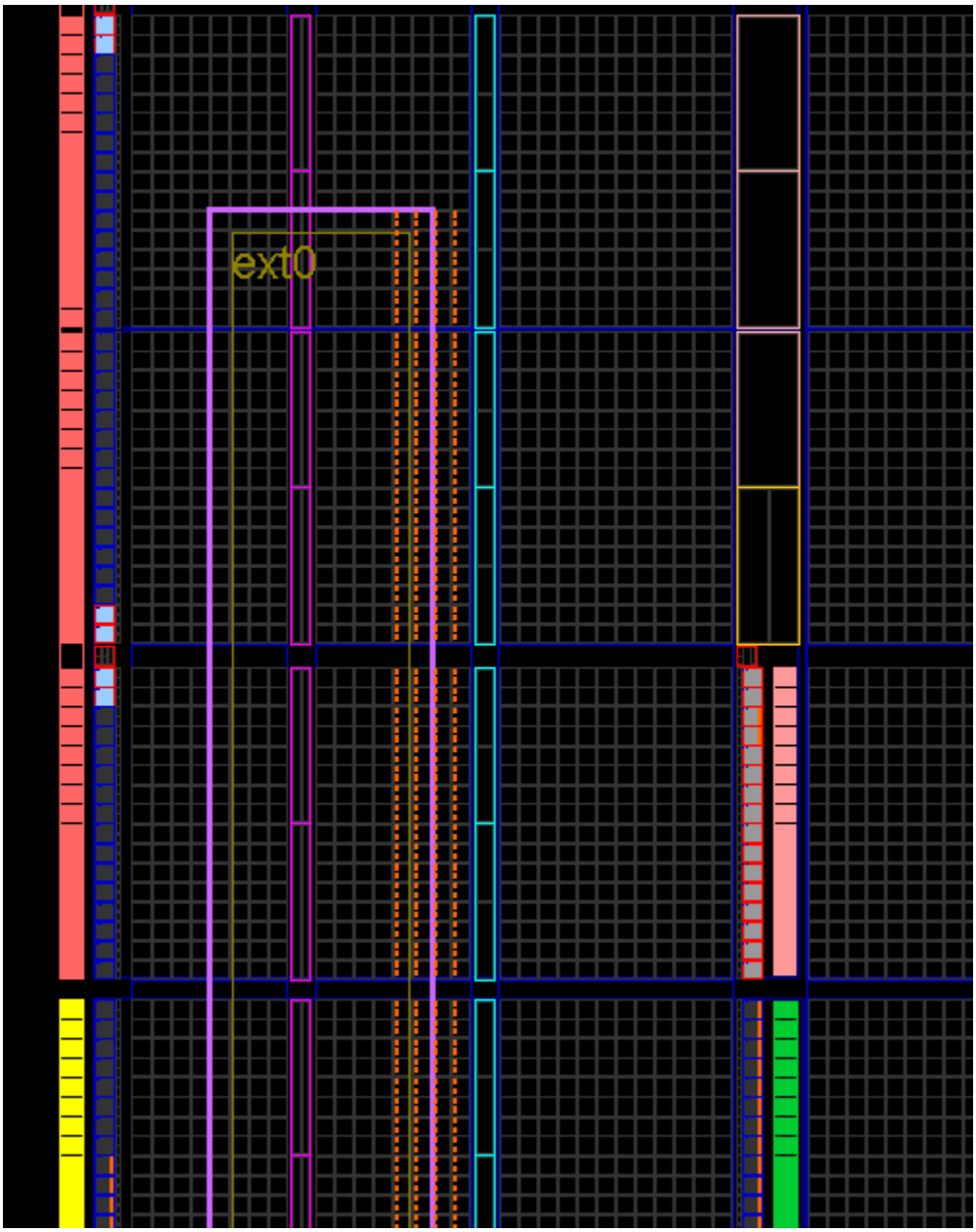

Figure A-9. Detailed view of the top portion of the Extension area. Bus macros are visible (orange dots) on the right side of the extension.

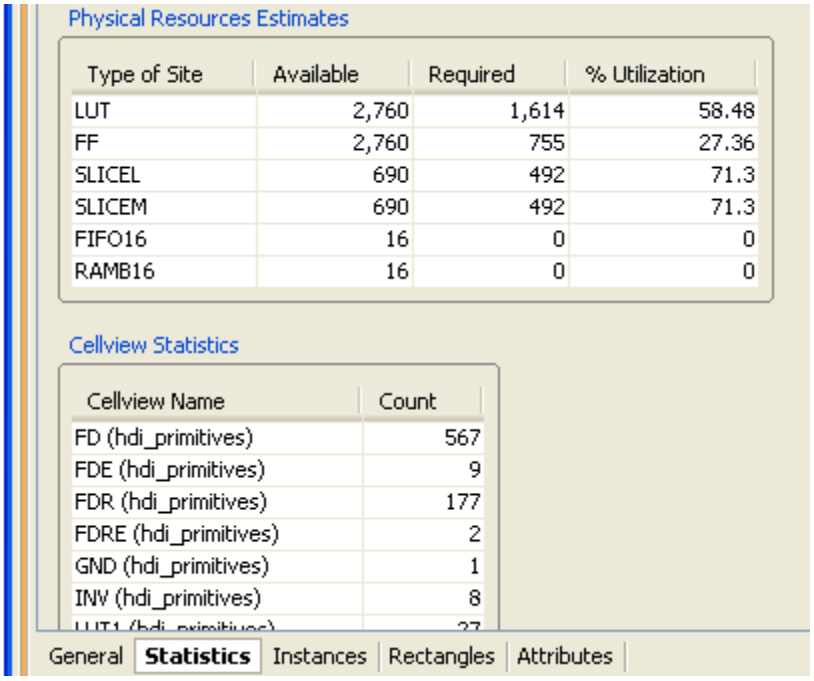

Figure A-10. Statistics from PlanAhead for the hand-generated version of the mmldiv64 extension.

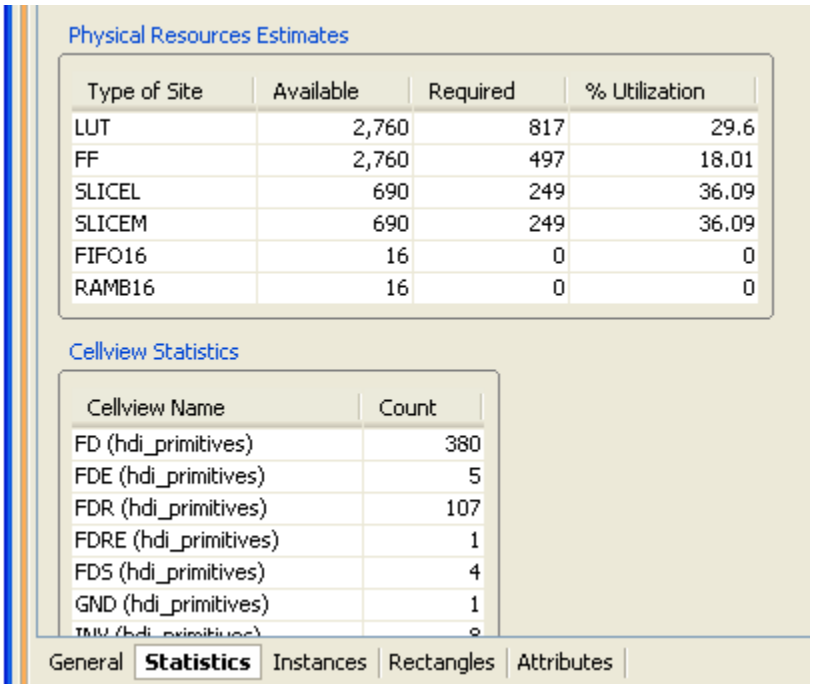

Figure A-11. Statistics from PlanAhead for the M2V-generated version of the mmldiv64 extension.

```
Macro Statistics
# Registers                      : 755
 Flip-Flops                     : 755

Device utilization summary:
Selected Device : 4vlx25ff668-10

 Number of Slices:              867  out of  10752    8%
 Number of Slice Flip Flops:    755  out of  21504    3%
 Number of 4 input LUTs:       1542 out of  21504    7%
 Number of IOs:                324
 Number of bonded IOBs:          0  out of    448    0%

Timing Summary:
Speed Grade: -10

  Minimum period: 5.729ns (Maximum Frequency: 174.557MHz)
  Minimum input arrival time before clock: 3.270ns
  Maximum output required time after clock: 3.971ns
  Maximum combinational path delay: No path found
```

Figure A-12. Synthesis report for the hand-generated version of the mmldiv64 extension.

```
Macro Statistics
# Registers                      : 494
 Flip-Flops                     : 494

Device utilization summary:
Selected Device : 4vlx25ff668-10

 Number of Slices:              448  out of  10752    4%
 Number of Slice Flip Flops:    494  out of  21504    2%
 Number of 4 input LUTs:        810  out of  21504    3%
 Number of IOs:                324
 Number of bonded IOBs:          0  out of    448    0%

Timing Summary:
Speed Grade: -10

  Minimum period: 5.886ns (Maximum Frequency: 169.894MHz)
  Minimum input arrival time before clock: 5.029ns
  Maximum output required time after clock: 6.351ns
  Maximum combinational path delay: 5.494ns
```

Figure A-13. Synthesis report for the M2V-generated version of the mmldiv64 extension.

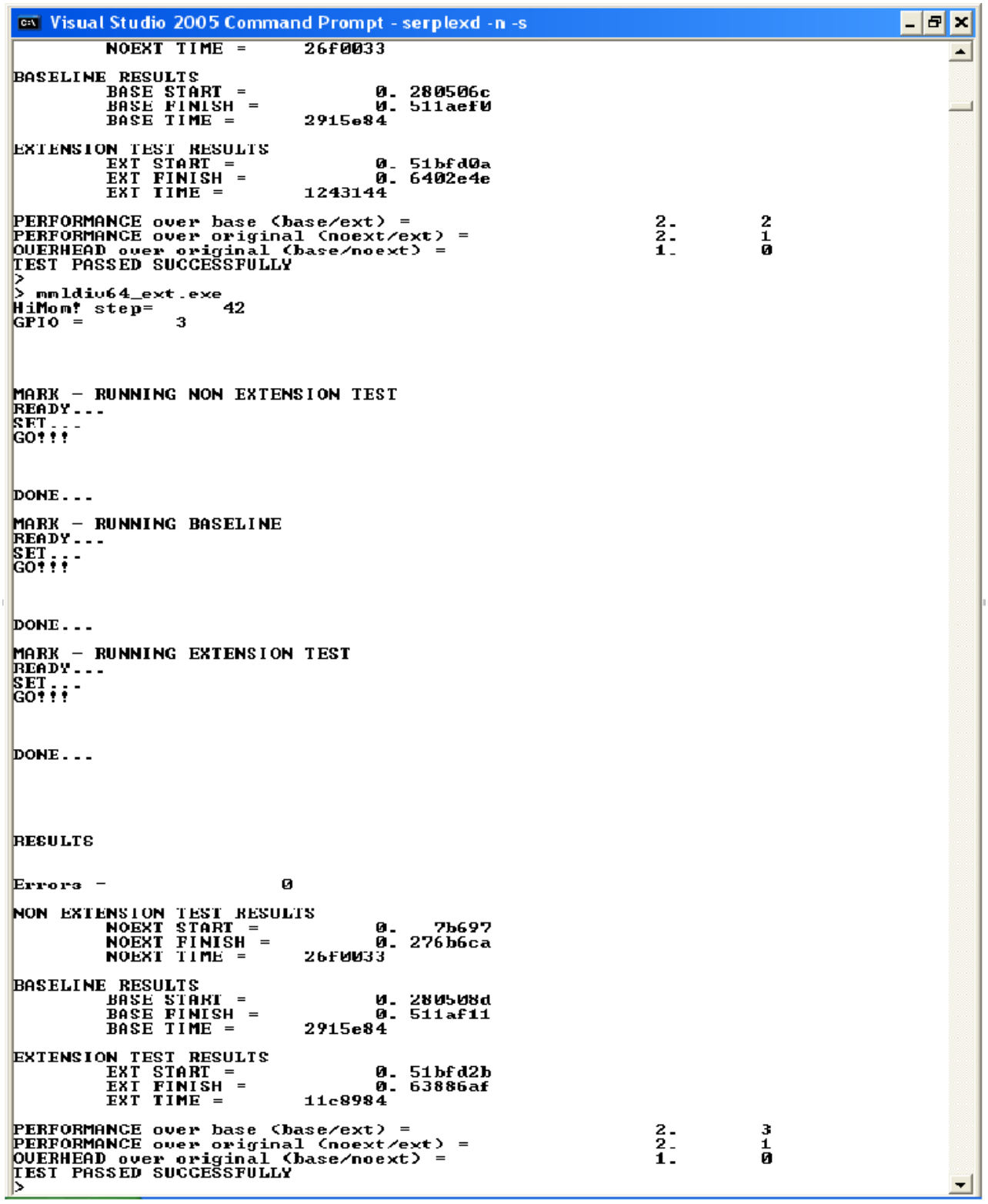

Figure A-14. Execution time results for the 64-bit division test program. The hand-generated version was run first, then the M2V-generated one

Appendix II – Verilog Output for Example Basic Block

```
// a.v
// auto-generated by m2v revision 1 on Wed Sep 19 14:47:36 2007
//
// INFO: reading from m2v_mod_bp.v
//
// m2v_mod_bp.v
// 8/15/07
// Karl Meier, Neil Pittman
//
// MIPS to Verilog (m2v) module (_mod) boilerplate (_bp)
//
// Copyright (c) Microsoft Corporation. All rights reserved.

`timescale 1ns / 1ps

module mmlite_div64 (
/*****Ports*******************************************************************/
        /* INPUT PORTS */
        input           CLK,            /* System Clock 50 - 100 MHZ */
        input           EN,             /* Enable */
        input           EXCEXT,         /* Exception Flush */
        input           EXTNOP_MA,      /* Extension Bubble in Memory Access Phase */
        input           GR,             /* Grant Pipeline Resources */
        input [31:0]    INSTR,          /* Current Instruction */
        input [31:0]    PC,             /* Current PC External */
        input           PCLK,           /* Pipeline Clock */
        input [31:0]    RDREG1DATA,     /* Register Read Port 1 Register Data */
        input [31:0]    RDREG2DATA,     /* Register Read Port 2 Register Data */
        input [31:0]    RDREG3DATA,     /* Register Read Port 3 Register Data */
        input [31:0]    RDREG4DATA,     /* Register Read Port 4 Register Data */
        input           REGEMPTY,       /* Register Write Buffer Empty */
        input           REGFULL,        /* Register Write Buffer Full */
        input           REGRDY,         /* Register Write Buffer Ready */
        input           RESET,          /* System Reset */
        /* OUTPUT PORTS */
        output          ACK,            /* Enable Acknowledged */
        output [31:0]   EXTADD,         /* Extension Address */
                                        /* Multiplexed: */
                                        /*      Next PC */
                                        /*      Exception Address */
                                        /*      PC Memory Access Phase */
        output          PCNEXT,         /* Conditional PC Update */
        output [4:0]    RDREG1,         /* Register Read Port 1 Register Number */
                                        /* Multiplexed: */
                                        /*      Read Port 1 Register Number */
                                        /*      Write Port 1 Register Number */
                                        /*      Write Register Memory Access Phase */
        output [4:0]    RDREG2,         /* Read Port 2 Register Number */
                                        /* Multiplexed: */
                                        /*      Register Read Port 2 Register Number */
                                        /*      Register Write Port 2 Register Number */
                                        /*      <0> Register Write Enable Memory Access Phase */
                                        /*      <1> Memory to Register Memory Acess Phase */
        output [4:0]    RDREG3,         /* Register Read Port 3 Register Number */
                                        /* Multiplexed: */
                                        /*      Register Read Port 3 Register Number */
        output [4:0]    RDREG4,         /* Register Read Port 4 Register Number Internal */
                                        /* Multiplexed: */
                                        /*      Register Read Port 4 Register Number */
                                        /*      <1:0> Data Address [1:0] Memory Access Phase */
                                        /*      <2> Right/Left Unaligned Load/Store Memory Access Phase */
                                        /*      <3> Byte/Halfword Load/Store Memory Acess Phase */
        output          REGWRITE1,      /* Register Write Port 1 Write Enable */
        output          REGWRITE2,      /* Register Write Port 2 Write Enable */
        output          REWB,           /* Re-enter at Writeback */
```

```
68              output          RI,                     /* Reserved/Recognized Instruction */
69              output [31:0]   WRDATA1,                /* Register Write Port 1 Data Internal */
70                                                      /* Multiplexed: */
71                                                      /*      Register Write Port 1 Data */
72                                                      /*      ALU Result Memory Access Phase */
73              output [31:0]   WRDATA2                 /* Register Write Port 2 Data Internal */
74                                                      /* Multiplexed: */
75                                                      /*      Register Write Port 2 Data */
76                                                      /*      Memory Data Out Memory Access Phase */
77              );
78
79      /*****Signals*************************************************************/
80
81              wire [31:0]ALURESULT_WB;        /* ALU Result to Writeback Phase */
82              wire            BHLS_WB;                /* Byte/Halfword Load/Store to Writeback Phase */
83              wire [31:0]CJMPADD;             /* Conditional Jump address to offset from Current PC */
84              wire [15:0]DIMM_EX;             /* Data Immediate Execute Phase */
85              wire [15:0]DIMM_ID;             /* Data Immediate Instruction Decode Phase */
86              wire [1:0]      DMADD_WB;               /* Least Significant Bits of Data Address to Writeback Phase */
87              wire [31:0]DMDATAOUT_WB;        /* Memory Data Out to Writeback Phase */
88              wire            DNE;                    /* Execution Done */
89              wire            EN_EX;                  /* Enable Execute Phase */
90              wire [31:0]JMPADD;              /* Jump address to end of basic block */
91              wire            MEMTOREG_WB;            /* Memory to Register to Writeback Phase */
92              wire [31:0]PC_EX;               /* PC Execute Phase */
93              wire [31:0]PC_WB;               /* PC to Writeback Phase */
94              wire [4:0]      RD_EX;                  /* Destination Register Execution Phase */
95              wire [4:0]      RDREG1_EX;              /* Register Read Port 1 Register Number Execute Phase */
96              wire [31:0]RDREG1DATA_EX;       /* Register Read Port 1 Register Data Execute Phase */
97              wire [4:0]      RDREG2_EX;              /* Register Read Port 2 Register Number Execute Phase */
98              wire [31:0]RDREG2DATA_EX;       /* Register Read Port 2 Register Data Execute Phase */
99              wire [4:0]      RDREG3_EX;              /* Register Read Port 3 Register Number Execute Phase */
100             wire [4:0]      RDREG4_EX;              /* Register Read Port 4 Register Number Execute Phase */
101             wire            REGWRITE_EX;            /* Register Write Execute Phase */
102             wire            REGWRITE_ID;            /* Register Write Instruction Decode Phase */
103             wire            REGWRITE_WB;            /* Register Write to Writeback Phase */
104             wire            RESET_EX;               /* Reset Execute Phase */
105             wire [31:0]RESULT_EX;           /* Result Execution Phase */
106             wire            RNL_WB;                 /* Right/Left Unaligned Load/Store to Writeback Phase */
107             wire [4:0]      RS_EX;                  /* Operand Register 1 Execute Phase */
108             wire [4:0]      RS_ID;                  /* Operand Register 1 Instruction Decode Phase */
109             wire [4:0]      RT_EX;                  /* Operand Register 2 Execute Phase */
110             wire [4:0]      RT_ID;                  /* Operand Register 2 Instruction Decode Phase */
111             wire            SLL128_EX;              /* Shift Left Logical 128 bits Execute Phase */
112             wire            SLL128_ID;              /* Shift Left Logical 128 bits Instruction Decode Phase */
113             wire [31:0]WRDATA1_EX;          /* Register Write Port 1 Data Execute Phase */
114             wire [31:0]WRDATA2_EX;          /* Register Write Port 2 Data Execute Phase */
115             wire [4:0]      WRREG_WB;               /* Write Register Number to Writeback Phase */
116             wire [4:0]      WRREG1_EX;              /* Register Write Port 1 Register Number Execute Phase */
117             wire [4:0]      WRREG2_EX;              /* Register Write Port 2 Register Number Execute Phase */
118
119     /*****Registers***********************************************************/
120
121             reg en_reg;             /* Enable */
122             reg gr_reg;/* Grant Pipeline Resources */
123
124     /*****Initialization******************************************************/
125     /*
126             initial
127             begin
128                     en_reg = 1'b0;
129                     gr_reg = 1'b0;
130             end
131     */
132
133     /************************************************************************/
134
135             assign EXTADD   =       (en_reg)?  JMPADD:
```

```
                                        (PCNEXT)? CJMPADD:
                                        (REWB)?  PC_WB:
                                                 32'hffffffff;
        assign RDREG1       =           (gr_reg & REGWRITE1)? WRREG1_EX:
                                        (REWB & gr_reg)?    WRREG_WB:
                                        (gr_reg)?   RDREG1_EX:
                                                 5'b11111;
        assign RDREG2       =           (gr_reg & REGWRITE2)? WRREG2_EX:
                                        (REWB & gr_reg)?    {3'b0,MEMTOREG_WB,REGWRITE_WB}:
                                        (gr_reg)?   RDREG2_EX:
                                                 5'b11111;
        assign RDREG3       =           (REWB & gr_reg)?    5'b0:
                                        (gr_reg)?   RDREG3_EX:
                                                 5'b11111;
        assign RDREG4       =           (REWB & gr_reg)?    {1'b0,BHLS_WB,RNL_WB,DMADD_WB}:
                                        (gr_reg)?   RDREG4_EX:
                                                 5'b11111;
        assign WRDATA1 =  (gr_reg & REGWRITE1)?         WRDATA1_EX:
                                (REWB)?  ALURESULT_WB:
                                         32'hffffffff;
        assign WRDATA2 =  (gr_reg & REGWRITE2)?         WRDATA2_EX:
                                (REWB)?  DMDATAOUT_WB:
                                         32'hffffffff;


    //
    // instantiate the instruction decode module for the extension instruction
    //   - the instruction decode module is auto generated and appended to the
    //     end of the verilog file (a.v unless redefined)
    //

        ext_id id (
                .CLK(CLK),
                .DIMM(DIMM_ID),
                .EN(EN),
                .JMPADD(JMPADD),
                .INSTR(INSTR),
                .PC(PC),
                .REGWRITE(REGWRITE_ID),
                .RESET(RESET),
                .RI(RI),
                .RS(RS_ID),
                .RT(RT_ID),
                .SLL128(SLL128_ID)
                );

/*****Instruction Decode -> Execute*****************************************************************/

        mmldiv64_toex to_ex(
                .ACK(ACK),
                .CLK(CLK),
                .DIMM_EX(DIMM_EX),
                .DIMM_ID(DIMM_ID),
                .EN_EX(EN_EX),
                .EN_ID(EN),
                .EXCEXT(EXCEXT),
                .PC_EX(PC_EX),
                .PC_ID(PC),
                .PCLK(PCLK),
                .RDREG1DATA_EX(RDREG1DATA_EX),
                .RDREG1DATA_ID(RDREG1DATA),
                .RDREG2DATA_EX(RDREG2DATA_EX),
                .RDREG2DATA_ID(RDREG2DATA),
                .REGWRITE_EX(REGWRITE_EX),
                .REGWRITE_ID(REGWRITE_ID),
                .RESET(RESET),
                .RESET_EX(RESET_EX),
                .RS_EX(RS_EX),
```

```
                        .RS_ID(RS_ID),
                        .RT_EX(RT_EX),
                        .RT_ID(RT_ID),
                        .SLL128_ID(SLL128_ID),
                        .SLL128_EX(SLL128_EX)
                        );


    //
    // instantiate the execution module for the extension instruction
    //   - the execution module is auto generated and appended to the
    //     end of the verilog file (a.v unless redefined)
    //

            ext_ex ex(
                        .ACK(ACK),
                        .DIMM(DIMM_EX),
                        .DNE(DNE),
                        .CLK(CLK),
                        .CJMPADD(CJMPADD),
                        .EN(EN_EX),
                        .EXTNOP_MA(EXTNOP_MA),
                        .GR(GR),
                        .PC(PC_EX),
                        .PCLK(PCLK),
                        .PCNEXT(PCNEXT),
                        .RD(RD_EX),
                        .RDREG1(RDREG1_EX),
                        .RDREG1DATA(RDREG1DATA),
                        .RDREG1DATA_ID(RDREG1DATA_EX),
                        .RDREG2(RDREG2_EX),
                        .RDREG2DATA(RDREG2DATA),
                        .RDREG2DATA_ID(RDREG2DATA_EX),
                        .RDREG3(RDREG3_EX),
                        .RDREG3DATA(RDREG3DATA),
                        .RDREG4(RDREG4_EX),
                        .RDREG4DATA(RDREG4DATA),
                        .REGEMPTY(REGEMPTY),
                        .REGFULL(REGFULL),
                        .REGRDY(REGRDY),
                        .REGWRITE1(REGWRITE1),
                        .REGWRITE2(REGWRITE2),
                        .RESET(RESET_EX),
                        .RESULT(RESULT_EX),
                        .RS(RS_EX),
                        .RT(RT_EX),
                        .SLL128(SLL128_EX),
                        .WRDATA1(WRDATA1_EX),
                        .WRDATA2(WRDATA2_EX),
                        .WRREG1(WRREG1_EX),
                        .WRREG2(WRREG2_EX)
                        );

/*****Execute -> to Writeback*****************************************************************/

            mmldiv64_topipe_wb to_wb(
                        .ACK(ACK),
                        .ALURESULT_WB(ALURESULT_WB),
                        .BHLS_WB(BHLS_WB),
                        .CLK(CLK),
                        .DMADD_WB(DMADD_WB),
                        .DMDATAOUT_WB(DMDATAOUT_WB),
                        .DNE(DNE),
                        .EN_EX(EN_EX),
                        .EXCEXT(EXCEXT),
                        .EXTNOP_MA(EXTNOP_MA),
                        .PC_EX(PC_EX),
                        .PC_WB(PC_WB),
```

```
272                     .PCLK(PCLK),
273                     .MEMTOREG_WB(MEMTOREG_WB),
274                     .RD_EX(RD_EX),
275                     .REGWRITE_EX(REGWRITE_EX),
276                     .REGWRITE_WB(REGWRITE_WB),
277                     .RESET(RESET),
278                     .RESULT_EX(RESULT_EX),
279                     .REWB(REWB),
280                     .RNL_WB(RNL_WB),
281                     .WRREG_WB(WRREG_WB)
282                     );
283
284 /*********************************************************************/
285
286         always@(posedge CLK)
287         begin
288                 if (RESET == 1'b0)
289                 begin
290                         en_reg <= 1'b0;
291                         gr_reg <= 1'b0;
292                 end
293                 else
294                 begin
295                         en_reg <= EN;
296                         gr_reg <= GR;
297                 end
298         end
299
300 endmodule
301
302
303
304 /*****Execute -> to Writeback*****************************************************************/
305
306 module mmldiv64_topipe_wb(
307 /*****Ports*****************************************************************/
308         /* INPUT PORTS */
309         input           ACK,                    /* Enable Acknowledged */
310         input           CLK,                    /* System Clock 50 - 100 MHZ */
311         input           DNE,                    /* Execution Done */
312         input           EN_EX,                  /* Enable Execute Phase */
313         input           EXCEXT,                 /* Exception Flush */
314         input           EXTNOP_MA,              /* Extension Bubble in Memory Access Phase */
315         input [31:0]    PC_EX,                  /* Current PC Execute Phase */
316         input           PCLK,                   /* Pipeline Clock */
317         input [4:0]     RD_EX,                  /* Destination Register Execution Phase */
318         input           REGWRITE_EX,            /* Register Write Execute Phase */
319         input           RESET,                  /* System Reset */
320         input [31:0]    RESULT_EX,              /* Result Execution Phase */
321         /* OUTPUT PORTS */
322         output [31:0]   ALURESULT_WB,           /* ALU Result to Writeback Phase */
323         output          BHLS_WB,                /* Byte/Halfword Load/Store to Writeback Phase */
324         output [1:0]    DMADD_WB,               /* Least Significant Bits of Data Address to Writeback Phase */
325         output [31:0]   DMDATAOUT_WB,           /* Memory Data Out to Writeback Phase */
326         output          MEMTOREG_WB,            /* Memory to Register to Writeback Phase */
327         output [31:0]   PC_WB,                  /* Current PC to Writeback Phase */
328         output          REGWRITE_WB,    /       * Register Write to Writeback Phase */
329         output          REWB,                   /* Re-enter at Writeback */
330         output          RNL_WB,                 /* Right/Left Unaligned Load/Store to Writeback Phase */
331         output [4:0]    WRREG_WB                /* Write Register Number to Writeback Phase */
332         );
333
334 /*****Signals*****************************************************************/
335
336         wire            EN_WB;                  /* Enable to Writeback Phase */
337         wire            RESET_WB;               /* Reset to Writeback Phase */
338
339 /*****Registers*****************************************************************/
```

```
340
341             reg [70:0] ex_wb;               /* Execute -> to Writeback Pipeline Register */
342             reg [1:0] pclkcnt;              /* Pipeline Clock edge detection */
343             reg reset_reg;                  /* Reset to Writeback Phase */
344             reg rewb_reg;                   /* Re-enter at Writeback */
345
346     /*****Initialization*************************************************************/
347     /*
348             initial
349             begin
350                     ex_wb = 71'b0;
351                     pclkcnt = 2'b0;
352                     rewb_reg = 1'b0;
353                     reset_reg = 1'b0;
354             end
355     */
356     /*******************************************************************/
357
358             assign RESET_WB         =       reset_reg;
359             assign REWB                     =       rewb_reg & EN_WB;
360             assign EN_WB                    =       ex_wb[70];                      //EN_EX;
361             assign REGWRITE_WB      =       ex_wb[69];                      //REGWRITE_EX;
362             assign MEMTOREG_WB      =       1'b0;
363             assign RNL_WB                   =       1'b0;
364             assign BHLS_WB          =       1'b0;
365             assign DMADD_WB         =       2'b0;
366             assign WRREG_WB         =       ex_wb[68:64];           //RD_EX;
367             assign ALURESULT_WB     =       ex_wb[63:32];           //RESULT_EX;
368             assign DMDATAOUT_WB     =       32'b0;
369             assign PC_WB                    =       ex_wb[31:0];            //PC_EX;
370
371     /*******************************************************************/
372
373             always@(posedge CLK)
374             begin
375                     /* Pipeline Clock edge detection */
376                     pclkcnt = {pclkcnt[0],PCLK};   // karl, 9/19, change to non-blocking to
377                                                     // match Neil
378             end
379
380             always@(posedge CLK)
381             begin
382                     case(pclkcnt)
383                     2'b01:  begin
384                                     /* Synchronize Reset to Pipeline Clock */
385                                     reset_reg <= RESET;
386                             end
387                     default:        begin
388                             end
389                     endcase
390             end
391
392             always@(posedge CLK)
393             begin
394                     /* Execute -> to Memory Access Pipeline Register */
395                     casex({pclkcnt,RESET_WB,EXTNOP_MA,rewb_reg,ACK,DNE,EXCEXT})
396                     8'bxx0xxxx:             begin
397                                                     /* Reset */
398                                                     rewb_reg <= 1'b0;
399                                                     ex_wb <= 71'b0;
400                                             end
401                     8'b011xxxx1:            begin
402                                                     /* Exception in Pipeline, Flush */
403                                                     rewb_reg <= 1'b0;
404                                                     ex_wb <= 71'b0;
405                                             end
406                     8'bxx1x0110:            begin
407                                                     /* Latch Data and Control after Execution Finishes */
```

```
408                                             ex_wb <= {EN_EX,REGWRITE_EX,RD_EX,RESULT_EX,PC_EX};
409                                     end
410                     8'b101100x0:    begin
411                                             /* Raise REWB at next Negedge of PCLK after ACK Lowers */
412                                             rewb_reg <= 1'b1;
413                                     end
414                     8'b011x1xx0:    begin
415                                             /* Lower REWB at next Posedge and reset register */
416                                             rewb_reg <= 1'b0;
417                                             ex_wb <= 71'b0;
418                                     end
419                     default:        begin
420                                             /* NOP */
421                                     end
422                     endcase
423             end
424     endmodule
425
426
427     /*****Instruction Decode -> Execute*****************************************************************/
428
429     module mmldiv64_toex(
430     /*****Ports*****************************************************************/
431             /* INPUT PORTS */
432             input                   ACK,                    /* Enable Acknowledged */
433             input                   CLK,                    /* System Clock 50 - 100 MHZ */
434             input [15:0]            DIMM_ID,                /* Data Immediate Instruction Decode Phase */
435             input                   EN_ID,                  /* Enable Instruction Decode Phase */
436             input                   EXCEXT,                 /* Exception Flush */
437             input [31:0]            PC_ID,                  /* Current PC Decode Phase */
438             input                   PCLK,                   /* Pipeline Clock */
439             input [31:0]            RDREG1DATA_ID,          /* Register Read Port 1 Register Data Instruction Decode Phase */
440             input [31:0]            RDREG2DATA_ID,          /* Register Read Port 2 Register Data Instruction Decode Phase */
441             input                   REGWRITE_ID,            /* Register Write Instruction Decode Phase*/
442             input                   RESET,                  /* System Reset */
443             input [4:0]             RS_ID,                  /* Operand Register 1 Instruction Decode Phase */
444             input [4:0]             RT_ID,                  /* Operand Register 2 Instruction Decode Phase */
445             input                   SLL128_ID,              /* Shift Left Logical 128 bits Instruction Decode Phase */
446             /* OUTPUT PORTS */
447             output [15:0]           DIMM_EX,                /* Data Immediate Execute Phase */
448             output                  EN_EX,                  /* Enable Execute Phase */
449             output [31:0]           PC_EX,                  /* Current PC Instruction Decode Phase */
450             output [31:0]           RDREG1DATA_EX,          /* Register Read Port 1 Register Data Execute Phase */
451             output [31:0]           RDREG2DATA_EX,          /* Register Read Port 2 Register Data Execute Phase */
452             output                  REGWRITE_EX,            /* Register Write Execute Phase*/
453             output                  RESET_EX,               /* Reset Execute Phase */
454             output [4:0]            RS_EX,                  /* Operand Register 1 Execute Phase */
455             output [4:0]            RT_EX,                  /* Operand Register 2 Execute Phase */
456             output                  SLL128_EX               /* Shift Left Logical 128 bits Execute Phase */
457             );
458
459     /*****Registers*****************************************************************/
460
461             reg [124:0]id_ex;               /* Instruction Decode -> Execute Pipeline Register */
462             reg [1:0]               pclkcnt;                /* Pipeline Clock edge detection */
463             reg                     reset_reg;              /* Reset Execute Phase */
464
465     /*****Initialization*****************************************************************/
466
467     /*
468             initial
469             begin
470                     id_ex = 125'b0;
471                     pclkcnt = 2'b0;
472                     reset_reg = 1'b0;
473             end
474     */
475
```

```
476  /*********************************************************************/
477
478          assign RESET_EX         =       reset_reg;
479          assign EN_EX                    =       id_ex[124];                     //EN_ID;
480          assign SLL128_EX        =       id_ex[123];                     //SLL128_ID;
481          assign REGWRITE_EX      =       id_ex[122];                     //REGWRITE_ID;
482          assign RS_EX                    =       id_ex[121:117];  //RS_ID;
483          assign RT_EX                    =       id_ex[116:112];  //RT_ID;
484          assign DIMM_EX          =       id_ex[111:96];          //DIMM_ID;
485          assign PC_EX                    =       id_ex[95:64];           //PC_ID;
486          assign RDREG1DATA_EX    =       id_ex[63:32];           //RDREG1DATA_ID;
487          assign RDREG2DATA_EX    =       id_ex[31:0];            //RDREG2DATA_ID
488
489  /*********************************************************************/
490
491          always@(posedge CLK)
492          begin
493                  /* Pipeline Clock edge detection */
494                  pclkcnt = {pclkcnt[0],PCLK};   // karl, 9/19, change to non-blocking to
495                                                  // match Neil
496          end
497
498          always@(posedge CLK)
499          begin
500                  case(pclkcnt)
501                  2'b01:  begin
502                                  /* Synchronize Reset to Pipeline Clock */
503                                  reset_reg <= RESET;
504                          end
505                  default:        begin
506                          end
507                  endcase
508          end
509
510          always@(posedge CLK)
511          begin
512                  /* Instruction Decode -> Execute Pipeline Register */
513                  casex({pclkcnt,RESET_EX,ACK,EXCEXT})
514                  5'bxx0xx:  begin
515                                  /* Reset */
516                                  id_ex <= 109'b0;
517                          end
518                  5'b011x1:  begin
519                                  /* Exception in Pipeline, Flush */
520                                  id_ex <= 109'b0;
521                          end
522                  5'bxx110:  begin
523                                  /* Hold state during Execute Phase */
524                          end
525                  5'b01100:  begin
526                                  /* Clocking the Pipeline */
527                                  id_ex <=
528  {EN_ID,SLL128_ID,REGWRITE_ID,RS_ID,RT_ID,DIMM_ID,PC_ID,RDREG1DATA_ID,RDREG2DATA_ID};
529                          end
530                  default:        begin
531                                  /* NOP */
532                          end
533                  endcase
534          end
535  endmodule
536
537
538  //
539  // INFO: finished reading from m2v_mod_bp.v
540  //
541
542  //
543  // extension instruction decode
```

```
//
module ext_id(
  input        CLK,
  input        EN,
  input [31:0]  INSTR,
  input [31:0]  PC,
  input        RESET,

  output reg [15:0] DIMM,
  output reg [31:0] JMPADD,
  output reg       REGWRITE,
  output reg       RI,
  output reg [4:0]  RS,
  output reg [4:0]  RT,
  output reg       SLL128
  );

  reg [31:0] jmpadd_c;
  reg en_r;
  reg [5:0] op_r;
  reg [31:0] pc_r;
  reg opcode_match;

  // combinatorial logic for instruction decode
  always @ (*) begin
    jmpadd_c = pc_r + 48 + 4;
    opcode_match = (op_r == 30);
  end

  // sequential logic for instruction decode
  always @ (posedge CLK) begin
    if (!RESET) begin
      DIMM    <= 16'h0;
      op_r    <= 6'h0;
      RS      <= 5'h0;
      RT      <= 5'h0;
      en_r    <= 1'h0;
      pc_r    <= 32'h0;
      JMPADD  <= 32'h0;
      RI      <= 1'h1;
      SLL128  <= 1'h0;
      REGWRITE <= 1'h0;
    end else begin
      DIMM    <= INSTR[15:0];
      op_r    <= INSTR[31:26];
      RS      <= INSTR[25:21];
      RT      <= INSTR[20:16];
      en_r    <= EN;
      pc_r    <= PC;
      JMPADD  <= jmpadd_c;
      RI      <= ~opcode_match;
      SLL128  <= en_r & opcode_match;
      REGWRITE <= en_r & opcode_match;
    end
  end
endmodule

//
// INFO: reading from m2v_ex_bp.v
//
// m2v_ex_bp.v
// 8/15/07
// Karl Meier, Neil Pittman
//
// MIPS to Verilog (m2v) execution (_ex) boilerplate (_bp)
//
// Copyright (c) Microsoft Corporation. All rights reserved.
//
```

```
612
613     module ext_ex (
614     /*****Ports*******************************************************************/
615                 /* INPUT PORTS */
616                 input               CLK,                /* System Clock 50 - 100 MHZ */
617                 input [15:0]        DIMM,               /* Data Immediate */
618                 input               EN,                 /* Enable  */
619                 input               EXTNOP_MA,          /* Extension Bubble in Memory Access Phase */
620                 input               GR,                 /* Grant Pipeline Resources */
621                 input [31:0]        PC,                 /* Current PC */
622                 input               PCLK,               /* Pipeline Clock */
623                 input [31:0]        RDREG1DATA,         /* Register Read Port 1 Register Data */
624                 input [31:0]        RDREG1DATA_ID,      /* Register Read Port 1 Register Data Instruction Decode Phase */
625                 input [31:0]        RDREG2DATA,         /* Register Read Port 2 Register Data */
626                 input [31:0]        RDREG2DATA_ID,      /* Register Read Port 2 Register Data Instruction Decode Phase */
627                 input [31:0]        RDREG3DATA,         /* Register Read Port 3 Register Data */
628                 input [31:0]        RDREG4DATA,         /* Register Read Port 4 Register Data */
629                 input               REGEMPTY,           /* Register Write Buffer Empty */
630                 input               REGFULL,            /* Register Write Buffer Full */
631                 input               REGRDY,             /* Register Write Buffer Ready */
632                 input               RESET,              /* System Reset */
633                 input [4:0]         RS,                 /* Operand Register 1 */
634                 input [4:0]         RT,                 /* Operand Register 2 */
635                 input               SLL128,             /* Shift Left Logical 128 bits */
636
637                 /* OUTPUT PORTS */
638                 output reg          ACK,                /* Enable Acknowledged */
639                 output reg [31:0]   CJMPADD,            /* Conditional Jump address to offset from Current PC */
640                 output reg          DNE,                /* Execution Done */
641                 output reg          PCNEXT,             /* Conditional PC Update */
642                 output reg [4:0]    RD,                 /* Destination Register */
643                 output reg          REGWRITE1,          /* Register Write Port 1 Write Enable */
644                 output reg          REGWRITE2,          /* Register Write Port 2 Write Enable */
645                 output reg [4:0]    RDREG1,             /* Register Read Port 1 Register Number */
646                 output reg [4:0]    RDREG2,             /* Register Read Port 2 Register Number */
647                 output reg [4:0]    RDREG3,             /* Register Read Port 3 Register Number */
648                 output reg [4:0]    RDREG4,             /* Register Read Port 4 Register Number */
649                 output reg [31:0]   RESULT,             /* Result */
650                 output reg [31:0]   WRDATA1,            /* Register Write Port 1 Data */
651                 output reg [31:0]   WRDATA2,            /* Register Write Port 2 Data */
652                 output reg [4:0]    WRREG1,             /* Register Write Port 1 Register Number */
653                 output reg [4:0]    WRREG2              /* Register Write Port 2 Register Number */
654                 );
655
656       // tie off outputs that are not used in the automated accelerator
657       always @ (posedge CLK) begin
658         RD <= 0;
659         RESULT <= 0;
660       end
661
662     /*****************************************************************************/
663
664     //
665     // INFO: finished reading from m2v_ex_bp.v
666     //
667
668       // parameters for extension execution block
669       parameter MAX_STATE = 6;
670       parameter REG_READ_WAIT_STATES = 5;
671
672       // declarations for extension state machine
673       reg[MAX_STATE:1] state_r;
674       reg[5:1] branch_state_r;
675       reg[5:1] write_state_r;
676       reg[5:1] read_state_r;
677
678       // declarations for register read variables
679       reg[31:0] r9_1;
```

```
680         reg[31:0] r8_4, r8_4_r;
681         reg[31:0] r4_11, r4_11_r;
682         reg[31:0] r5_18;
683         reg[31:0] r6_23;
684         // declarations for register temp variables
685         reg[31:0] r9_3;
686         reg[31:0] r11_6;
687         reg[31:0] r9_8, r9_8_r;
688         reg[31:0] r8_10;
689         reg[31:0] r11_13;
690         reg[31:0] r8_15, r8_15_r;
691         reg[31:0] r4_17;
692         reg[31:0] r11_20;
693         reg[31:0] r4_22, r4_22_r;
694         reg[31:0] r11_25, r11_25_r;
695         reg[31:0] r5_29, r5_29_r;
696       //
697       // INFO: reading from m2v_state_mc.v
698       //
699         // m2v_state_mc.v
700         //
701         // Karl Meier
702         // 8/15/07
703         //
704         // invariant state machine logic for the read, write, and branch state
705         // machines
706         //
707
708         reg [1:0] pclk_del_r;
709         reg pclk_rise, pclk_fall;
710         reg en_r, sll128_r, gr_r, regrdy_r, regfull_r, regempty_r, extnop_ma_r;
711         reg clr_dne, DNE_c, ACK_c;
712         reg done_state, done_state_r;
713         reg wsm_idle, wsm_idle_r, wsm_pulse, wsm_pulse_r, wsm_wait, wsm_wait_r;
714         reg write_this_state, wsm_done;
715         reg rsm_idle, rsm_idle_r, rsm_latch, rsm_latch_r;
716         reg rsm_wait, rsm_wait_r, rsm_wait2, rsm_wait2_r;
717         reg [3:0] rsm_count, rsm_count_r;
718         reg read_this_state, rsm_done;
719         reg bsm_idle, bsm_idle_r, bsm_calc, bsm_calc_r;
720         reg bsm_wait, bsm_wait_r, bsm_waitpf, bsm_waitpf_r;
721         reg bsm_waitpr, bsm_waitpr_r;
722         reg branch_this_state, bsm_done;
723         reg fsm_idle, fsm_idle_r, fsm_wait2, fsm_wait2_r, fsm_wait, fsm_wait_r;
724         reg final_state, fsm_done;
725         reg take_branch, take_branch_r;
726
727         // state machine logic for compiled extension
728         always @ (*) begin
729           pclk_rise = (pclk_del_r == 2'b01);
730           pclk_fall = (pclk_del_r == 2'b10);
731
732           // start the extension instruction
733           clr_dne = state_r[1] & en_r & sll128_r;
734
735           // state machine for read logic
736           read_this_state = (| (state_r & read_state_r));
737           rsm_wait = read_this_state &
738                   (rsm_idle_r & gr_r) |
739                   (rsm_wait_r & (rsm_count_r != REG_READ_WAIT_STATES));
740           rsm_latch = rsm_wait_r & (rsm_count_r == REG_READ_WAIT_STATES);
741           rsm_wait2 = (rsm_wait2_r | rsm_latch_r) & ~done_state;
742           rsm_idle = ~rsm_wait & ~rsm_wait2 & ~rsm_latch;
743           rsm_count = rsm_idle_r ? 4'h0 : (rsm_count_r + 1);
744           rsm_done = ~read_this_state |
745                   (read_this_state & (rsm_latch_r | rsm_wait2_r));
746
747           // state machine for write logic
```

```
748         write_this_state = (| (state_r & write_state_r));
749         wsm_pulse = wsm_idle_r & write_this_state & gr_r & regrdy_r & ~regfull_r;
750         wsm_wait = (wsm_pulse_r & ~done_state) |
751                 (wsm_wait_r  & ~done_state);
752         wsm_idle = ~wsm_pulse & ~wsm_wait;
753         wsm_done = ~write_this_state |
754                 (write_this_state & (wsm_pulse_r | wsm_wait_r));
755
756         // state machine for branch logic
757         branch_this_state = (| (state_r & branch_state_r));
758         bsm_calc = bsm_idle_r & branch_this_state;
759         bsm_waitpf = (bsm_calc_r & take_branch_r) |
760                  (bsm_waitpf_r & ~pclk_fall);
761         bsm_waitpr = (bsm_waitpf_r & pclk_fall) |
762                  (bsm_waitpr_r & ~pclk_rise);
763         bsm_wait = (bsm_calc_r & ~take_branch_r & ~done_state) |
764                 (bsm_waitpr_r & pclk_rise & ~done_state) |
765                 (bsm_wait_r & ~done_state);
766         bsm_idle = ~bsm_calc & ~bsm_wait & ~bsm_waitpr & ~bsm_waitpf;
767         bsm_done = ~branch_this_state |
768                 (branch_this_state &
769                   ((bsm_calc_r & ~take_branch_r) |
770                    (bsm_waitpr_r & pclk_rise) |
771                     bsm_wait_r));
772
773         // state machine to finish up the extension instruction
774         final_state = state_r[MAX_STATE];
775         fsm_wait = final_state & rsm_idle_r |
776                 (fsm_wait_r & ~(gr_r & regempty_r & extnop_ma_r));
777         fsm_wait2 = (fsm_wait_r & gr_r & regempty_r & extnop_ma_r) |
778                  (fsm_wait2_r & ~en_r);
779         fsm_idle = ~fsm_wait & ~fsm_wait2;
780         fsm_done = final_state & fsm_wait2_r & ~en_r;
781
782         // clear DNE as the extension instruction is entered
783         // set DNE as the extension instruction is exited
784         DNE_c = (DNE | (fsm_wait_r & gr_r & regempty_r & extnop_ma_r)) & ~clr_dne;
785         ACK_c = (ACK | (~DNE & ~ACK)) & ~(ACK & DNE & pclk_rise);
786       end
787
788       always @ (*) begin
789         // true when all conditions for a state have been satisfied
790         done_state = clr_dne |
791                 (~state_r[1] & bsm_done & rsm_done & wsm_done);
792       end
793
794
795       // state to determine rising and falling edges of pclk
796       always @ (posedge CLK) begin
797         pclk_del_r <= {pclk_del_r[0], PCLK};
798       end
799
800       // buffer signals that may be heavily loaded or come from a distance
801       //   - is this needed?  this is present to maintain compatibility with Neil
802       always @ (posedge CLK) begin
803         if (!RESET) begin
804           en_r <= 1'h0;
805           sll128_r <= 1'h0;
806           gr_r <= 1'h0;
807           regrdy_r <= 1'h0;
808           regfull_r <= 1'h0;
809           regempty_r <= 1'h0;
810           extnop_ma_r <= 1'h0;
811         end else begin
812           en_r <= EN;
813           sll128_r <= SLL128;
814           gr_r <= GR;
815           regrdy_r <= REGRDY;
```

```
816             regfull_r <= REGFULL;
817             regempty_r <= REGEMPTY;
818             extnop_ma_r <= EXTNOP_MA;
819           end
820         end
821
822         // misc control for the extension
823         always @ (posedge CLK) begin
824           if (!RESET) begin
825             ACK <= 1'h0;
826             DNE <= 1'h1;
827             done_state_r <= 1'b0;
828
829             wsm_idle_r <= 1'b1;
830             wsm_pulse_r <= 1'b0;
831             wsm_wait_r <= 1'b0;
832
833             rsm_idle_r <= 1'b1;
834             rsm_latch_r <= 1'b0;
835             rsm_wait_r <= 1'b0;
836             rsm_wait2_r <= 1'b0;
837             rsm_count_r <= 4'b0;
838
839             bsm_idle_r <= 1'b1;
840             bsm_calc_r <= 1'b0;
841             bsm_wait_r <= 1'b0;
842             bsm_waitpr_r <= 1'b0;
843             bsm_waitpf_r <= 1'b0;
844             take_branch_r <= 1'h0;
845
846             fsm_idle_r <= 1'b1;
847             fsm_wait_r <= 1'b0;
848             fsm_wait2_r <= 1'b0;
849
850           end else begin
851             /*
852             // clear ack
853             if (ACK & DNE & pclk_rise)
854               ACK <= 1'h0;
855             // set ack
856             else if (~DNE & ~ACK)
857               ACK <= 1'h1;
858             */
859
860             ACK <= ACK_c;
861             DNE <= DNE_c;
862             done_state_r <= done_state;
863
864             wsm_idle_r <= wsm_idle;
865             wsm_pulse_r <= wsm_pulse;
866             wsm_wait_r <= wsm_wait;
867
868             rsm_idle_r <= rsm_idle;
869             rsm_latch_r <= rsm_latch;
870             rsm_wait_r <= rsm_wait;
871             rsm_wait2_r <= rsm_wait2;
872             rsm_count_r <= rsm_count;
873
874             bsm_idle_r <= bsm_idle;
875             bsm_calc_r <= bsm_calc;
876             bsm_wait_r <= bsm_wait;
877             bsm_waitpr_r <= bsm_waitpr;
878             bsm_waitpf_r <= bsm_waitpf;
879             // if take_branch_r is ever used outside of the branch state machine,
880             // it may need to be cleared at the end of the branch operation
881             take_branch_r <= bsm_calc ? take_branch : take_branch_r;
882
883             fsm_idle_r <= fsm_idle;
```

```
884             fsm_wait_r <= fsm_wait;
885             fsm_wait2_r <= fsm_wait2;
886           end
887         end
888
889       //
890       // INFO: finished reading from m2v_state_mc.v
891       //
892
893
894         // registers that contain state about this cycle
895         always @ (posedge CLK) begin
896           if (~RESET) begin
897             branch_state_r[1] <= 1'b0;
898             write_state_r[1]  <= 1'b0;
899             read_state_r[1]   <= 1'b1;
900
901             branch_state_r[2] <= 1'b0;
902             write_state_r[2]  <= 1'b0;
903             read_state_r[2]   <= 1'b1;
904
905             branch_state_r[3] <= 1'b1;
906             write_state_r[3]  <= 1'b1;
907             read_state_r[3]   <= 1'b1;
908
909             branch_state_r[4] <= 1'b0;
910             write_state_r[4]  <= 1'b1;
911             read_state_r[4]   <= 1'b0;
912
913             branch_state_r[5] <= 1'b0;
914             write_state_r[5]  <= 1'b1;
915             read_state_r[5]   <= 1'b0;
916
917           end else begin
918             branch_state_r <= branch_state_r;
919             write_state_r  <= write_state_r;
920             read_state_r   <= read_state_r;
921           end
922         end
923
924
925         // combinatorial logic to/from the register file
926         always @ (*) begin
927           // combinatorial logic for register reads
928           // use read ports 3 & 4 to prevent write conflicts
929           RDREG1 = 0;
930           RDREG2 = 0;
931           r9_1 =  RDREG3DATA;
932           r8_4 =  RDREG2DATA_ID;
933           r4_11 =  RDREG1DATA_ID;
934           r5_18 =  RDREG4DATA;
935           r6_23 =  RDREG3DATA;
936           RDREG3 =  ({5{state_r[2]}} & (RT + 1))
937                   | ({5{state_r[1]}} & RT)
938                   | ({5{state_r[3]}} & (RS + 2));
939           RDREG4 =  ({5{state_r[1]}} & RS)
940                   | ({5{state_r[2]}} & (RS + 1));
941
942           // combinatorial logic for register writes
943           WRREG1 =  ({5{state_r[3]}} & RT)
944                   | ({5{state_r[4]}} & (RT + 1))
945                   | ({5{state_r[5]}} & (RS + 1));
946           WRDATA1 = ({32{state_r[3]}} & r8_15_r)
947                   | ({32{state_r[4]}} & r9_8_r)
948                   | ({32{state_r[5]}} & r5_29_r);
949           REGWRITE1 = wsm_pulse_r & (state_r[3]
950                   | state_r[4]
951                   | state_r[5]);
```

```
    WRREG2 =  ({5{state_r[4]}} & RS)
          | ({5{state_r[5]}} & 11);
    WRDATA2 = ({32{state_r[4]}} & r4_22_r)
          | ({32{state_r[5]}} & r11_25_r);
    REGWRITE2 = wsm_pulse_r & (state_r[4]
          | state_r[5]);
  end

  // internal pipeline logic
  always @ (posedge CLK) begin
    if (~RESET) begin
      r8_4_r <= 32'h0;
      r4_11_r <= 32'h0;
      r9_8_r <= 32'h0;
      r8_15_r <= 32'h0;
      r4_22_r <= 32'h0;
      r11_25_r <= 32'h0;
      r5_29_r <= 32'h0;
    end else begin
      r8_4_r <=  state_r[1] ? r8_4 : r8_4_r;
      r4_11_r <=  state_r[1] ? r4_11 : r4_11_r;
      r9_8_r <=  state_r[2] ? r9_8 : r9_8_r;
      r8_15_r <=  state_r[2] ? r8_15 : r8_15_r;
      r4_22_r <=  state_r[2] ? r4_22 : r4_22_r;
      r11_25_r <=  state_r[3] ? r11_25 : r11_25_r;
      r5_29_r <=  state_r[2] ? r5_29 : r5_29_r;
    end
  end

  // combinatorial logic for the instruction nodes
  always @ (*) begin
    // [0x0]0x10840    sll          r9, r9, 1
    r9_3 = r9_1 << 1;

    // [0x4]0x21fc2    srl          r11, r8, 31
    r11_6 = r8_4_r >> 31;

    // [0x8]0x230825  or          r9, r9, r11
    r9_8 = r9_3 | r11_6;

    // [0xc]0x21040    sll          r8, r8, 1
    r8_10 = r8_4_r << 1;

    // [0x10]          0x41fc2    srl          r11, r4, 31
    r11_13 = r4_11_r >> 31;

    // [0x14]          0x431025  or          r8, r8, r11
    r8_15 = r8_10 | r11_13;

    // [0x18]          0x42040    sll          r4, r4, 1
    r4_17 = r4_11_r << 1;

    // [0x1c]          0x51fc2    srl          r11, r5, 31
    r11_20 = r5_18 >> 31;

    // [0x20]          0x832025  or          r4, r4, r11
    r4_22 = r4_17 | r11_20;

    // [0x24]          0x26182b  sltu          r11, r9, r6
    r11_25 = ({1'b0, r9_8_r} < {1'b0, r6_23}) ? 1 : 0;

    // [0x28]          0x10030005          beq          r0, r11, 20
    take_branch = (32'h0 == r11_25);
    CJMPADD = take_branch ? (PC + 4 + {{16{DIMM[15]}},DIMM}) : PC;
    PCNEXT = state_r[3] & bsm_waitpr & take_branch;

    // [0x2c]          0x52840    sll          r5, r5, 1
    r5_29 = r5_18 << 1;
```

```
1020
1021        end
1022
1023        // primary extension state machine
1024        always @ (posedge CLK) begin
1025          if (~RESET) begin
1026            state_r <= 1;
1027          end else begin
1028            if (en_r) begin
1029              if (done_state)
1030                state_r <= {state_r[MAX_STATE-1:1], 1'b0};
1031            end
1032            else begin
1033              state_r <= 1;
1034            end
1035          end
1036        end
1037
1038      endmodule
```

Appendix III – BBW File for Example Basic Block

```
[bbname __ull_div]
MIPSBE
[encoding]
[r1=r2+1;r3=r0+11;r5=r4+1;r6=r5+1]b26.6:c011110;b21.5:r4;b16.5:r2;b0.16:v0;
[code 48]
40080100
c21f0200
25082300
40100200
c21f0400
25104300
40200400
c21f0500
25208300
2b182600
5000310
40280500
[disasm]
sll     r1,r1,1
srl     r3,r2,31
or      r1,r1,r3
sll     r2,r2,1
srl     r3,r4,31
or      r2,r2,r3
sll     r4,r4,1
srl     r3,r5,31
or      r4,r4,r3
sltu    r3,r1,r6
beq     r0,r3,40
sll     r5,r5,1
[registers 7]
0,9,8,11,4,5,6
[valuess 1]
{40,11,5}
```

Appendix IV – Verbose Output from M2V for Example Basic Block

```
c:\fpga\bb2\m2v.exe -v small.bbw
Verbose output is enabled

Basic Block Dump:
Regs: 0 9 8 11 4 5 6
Values: {28,b,5}
Code:    [  0]    10840 sll   r1,r1,1
         [  4]    21fc2 srl   r3,r2,31
         [  8]   230825 or    r1,r1,r3
         [  c]    21040 sll   r2,r2,1
         [ 10]     41fc2 srl  r3,r4,31
         [ 14]   431025 or    r2,r2,r3
         [ 18]    42040 sll   r4,r4,1
         [ 1c]    51fc2 srl   r3,r5,31
         [ 20]   832025 or    r4,r4,r3
         [ 24]   26182b sltu            r3,r1,r6
         [ 28] 10030005 beq             r0,r3,40
         [ 2c]    52840 sll   r5,r5,1
encoding of the extension instruction:
  [r1=r2+1;r3=r0+11;r5=r4+1;r6=r5+1]b26.6:c011110;b21.5:r4;b16.5:r2;b0.16:v0;

Parsing register relationships (pre-conditions)
  r1 = r2 + 1
  r3 = r0 + 11
  r5 = r4 + 1
  r6 = r5 + 1
Parsing instruction encoding
  extension is decoded as opcode = 30
  RS register = r4
  RT register = r2
  immediate value in encoding = v0
  extension is encoded as: 0x78820000
Finished parsing instruction encoding

Generating IL and Dependency Graph...
.............

Assigning cycles to Instructions and RegFile access.....

**** Cycle 1 ****

INFO: reading register RT
NODE 4: Register
  [0x4]   Read r2 (canonical) from the RF
    used by: [0xc]
INFO: reading register RS
NODE 11: Register
  [0x10] Read r4 (canonical) from the RF
    used by: [0x18]

INFO: cycle 1 state: RF reads = 2, RF writes = 0, LS accesses = 0, may branch = 0
```

**** Cycle 2 ****

NODE 4 is being pipelined
NODE 5: Instruction
  [0x4] 0x21fc2 srl r11, r8, 31
cost = 4
NODE 6: Register
  [0x4] Write r3 (canonical)
    used by: [0x8]
NODE 9: Instruction
  [0xc] 0x21040 sll r8, r8, 1
cost = 4
NODE 10: Register
  [0xc] Write r2 (canonical)
    used by: [0x14]
NODE 11 is being pipelined
NODE 12: Instruction
  [0x10] 0x41fc2 srl r11, r4, 31
cost = 4
NODE 13: Register
  [0x10] Write r3 (canonical)
    used by: [0x14]
NODE 14: Instruction
  [0x14] 0x431025 or r8, r8, r11
cost = 5
NODE 15: Register
  [0x14] Write r2 (canonical) to the RF
NODE 16: Instruction
  [0x18] 0x42040 sll r4, r4, 1
cost = 4
NODE 17: Register
  [0x18] Write r4 (canonical)
    used by: [0x20]

INFO: reading register (RT + 1)
NODE 1: Register
  [0x0] Read r1 (canonical) from the RF
NODE 2: Instruction
  [0x0] 0x10840 sll r9, r9, 1
cost = 4
NODE 3: Register
  [0x0] Write r1 (canonical)
    used by: [0x8]
NODE 7: Instruction
  [0x8] 0x230825 or r9, r9, r11
cost = 5
NODE 8: Register
  [0x8] Write r1 (canonical) to the RF
    used by: [0x24]
INFO: reading register (RS + 1)
NODE 18: Register
  [0x1c] Read r5 (canonical) from the RF
    used by: [0x2c]
NODE 19: Instruction

```
  [0x1c] 0x51fc2 srl         r11, r5, 31
cost = 4
NODE 20: Register
  [0x1c] Write r3 (canonical)
    used by: [0x20]
NODE 21: Instruction
  [0x20] 0x832025          or          r4, r4, r11
cost = 5
NODE 22: Register
  [0x20] Write r4 (canonical) to the RF
NODE 28: Instruction
  [0x2c] 0x52840 sll         r5, r5, 1
cost = 4
NODE 29: Register
  [0x2c] Write r5 (canonical) to the RF

INFO: cycle 2 state: RF reads = 2, RF writes = 0, LS accesses = 0, may branch = 0

**** Cycle 3 ****

NODE 8 is being pipelined

INFO: reading register (RS + 2)
NODE 23: Register
  [0x24] Read r6 (canonical) from the RF
NODE 24: Instruction
  [0x24] 0x26182b          sltu        r11, r9, r6
cost = 13
NODE 25: Register
  [0x24] Write r3 (canonical) to the RF
    used by: [0x28]
INFO: reading register 0
NODE 26: Register
  [0x28] Read r0 (canonical) from the RF
NODE 27: Instruction
  [0x28] 0x10030005        beq         r0, r11, 20
cost = 18

NODE 15 is being pipelined
NODE 15 written back to RF in cycle 3
INFO: cycle 3 state: RF reads = 1, RF writes = 1, LS accesses = 0, may branch = 1

**** Cycle 4 ****


NODE 8 written back to RF in cycle 4
NODE 22 is being pipelined
NODE 22 written back to RF in cycle 4
INFO: cycle 4 state: RF reads = 0, RF writes = 2, LS accesses = 0, may branch = 0

**** Cycle 5 ****

NODE 29 is being pipelined
NODE 29 written back to RF in cycle 5
```

```
NODE 25 is being pipelined
NODE 25 written back to RF in cycle 5
INFO: cycle 5 state: RF reads = 0, RF writes = 2, LS accesses = 0, may branch = 0

INFO: Extension requires 5 cycles

Writing Verilog module, a, to a.v
.........
PASS: m2v completed successfully
```