# M2V – Automatic Hardware Generation from Software Binaries

Ruirui Gu, Alessandro Forin
*Microsoft Research*

August 2009

Technical Report
MSR-TR-2009-106

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

# M2V – Automatic Hardware Generation from Software Binaries

Ruirui Gu, Alessandro Forin
*Microsoft Research*

## Abstract

*The MIPS-to-Verilog (M2V) compiler translates blocks of MIPS machine code into a hardware design represented in Verilog. M2V is part of a tool-chain that automatically accelerates software applications on hardware platforms such as the eMIPS processor, a dynamically extensible processor realized on the Virtex-4 XC4LX25 FPGA.*

*The M2V compiler can accelerate blocks that include load and stores, supports interrupts and TLB misses, and automatically encodes extended instructions for minimal latency. The new version supports the composition of multiple basic blocks using four basic control patterns. The optimization techniques make use of path-based scheduling algorithms derived from dataflow static scheduling, and from control-flow state machines. Simulation results indicate a factor of 22 in performance improvement in the case of a simple self-looped basic block.*

## 1 Introduction

The goal of the M2V project is to automatically generate hardware accelerators from software binaries. Figure 1 shows the overall tool chain that includes M2V. The input to the tool chain is an ordinary executable binary file, runnable on all MIPS platforms. The output is an extended executable that includes a configuration bitfile for the (FPGA-based) accelerator. The tool-chain restricts the code selection problem for the accelerator to the set of most frequently executed basic blocks in the application. Each basic block is a directed acyclic graph (DAG), a set of machine instructions that do not contain branches and are branched-to only at the very first instruction. The best candidate blocks are those blocks which require a lot computation and execute most frequently in the application. These candidates are identified by executing the application using the Giano full-system simulator[22], in concert with the data obtained via static analysis of the application binary. The BBTools select the basic blocks to accelerate and patch the binary image with the special instructions for the accelerator. The M2V automatically generates the design for the hardware accelerator, which is then synthesized onto programmable logic such as FPGA boards.

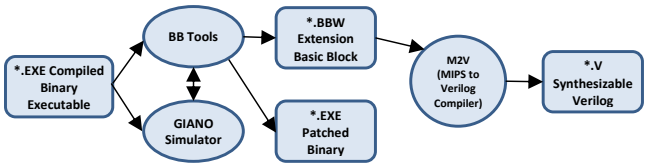


**Figure 1: Tool-chain for automatically generating Verilog coding from executable binary files. Giano is a full-system simulator that executes the application and extracts basic block profiles. The BB Tools are a series of tools that identify the basic blocks in an image, patch the image with the insertion of the special instructions for the accelerator, and finally merge the patched image and the bitfile for the hardware design. The hardware design is synthesized using the vendor's tools.**

The tool-chain applies to any programmable logic attached to a tightly coupled pipeline, e.g. with minimal latency between the accelerator and the RISC pipeline. The extensible MIPS (eMIPS) processor is such a platform that is being developed at Microsoft research as an example of a RISC processor integrated with programmable logic. It should be possible to target more loosely coupled architectures, but we have not attempted to do so.

The eMIPS platform consists of a standard MIPS pipeline and an extension unit (EU). The EU contains programmable logic that is used for extensions to the MIPS instruction set. The EU recognizes special "extended instructions", typically used to accelerate the execution of an application. The machine opcode for the extended instruction is inserted before the accelerated basic block in the MIPS binary. When the extended instruction completes, program execution will proceed at the address following the basic block or at the address of a branch target.

The goal of the M2V compiler is to automatically create the logic for eMIPS extensions using a .bbw file as the hardware specification. The M2V compiler generates synthesizable Verilog which is synthesized using the standard Xilinx place and route tools to create a bit file that can be loaded onto the eMIPS platform. In an embedded platform, the extension can be loaded at power-up. In a

more general purpose system, the extension can be dynamically loaded when a binary image is loaded. Dynamic loading of the extension requires partial reconfiguration of the programmable logic. By dynamically loading and unloading accelerators, the area of the programmable hardware can be used more efficiently. It is worth to note that the original code is preserved so that execution can fall back to software if/when necessary.

Graph based methods are powerful tools to analyze data dependencies and explore optimization strategies. We propose to use these tools in a compiler that automatically generates hardware accelerators from software. The work described in this paper uses a special kind of compiler for hardware acceleration. Rather than using hardware description languages [18] or developing a hardware compiler for some specific language such as C [5] or Scheme [23], the MIPS-to-Verilog (M2V) compiler [23] translates blocks of MIPS machine code into a hardware design represented in Verilog. In a sense, M2V is a hardware assembler, usable for the last translation phase of any high-level language compiler. The optimizations we applied to M2V use only four basic control patterns to assemble arbitrary sets of basic blocks. We tested M2V within an automatic tool chain that generates Verilog from software binaries, using a 64-bit divider, accelerating only three small basic blocks. Simulations demonstrate a factor of 22 in performance improvement for a simple self-looped basic block case, and a 16% performance improvement for an infrequent branch-away case.

Extending M2V based on the previous version involved the following tasks:

- Add support for "J" instructions. M2V now recognizes jump instructions, and merges the target block.
- Add support for multiple basic blocks. The previous version of M2V only handled one basic block at a time. The new version can handle multiple blocks, where control structure falls into one of four kinds: self-loop, sequential, branch and join.
- Improve the execution of self-looped basic blocks on the extension, switching from a pipeline-based implementation to a control-flow-based implementation.
- Add support for code elimination, code motion, and sharing of expensive arithmetic logic.

## 2 Related Work

eMIPS [1] is a dynamically extensible processor architecture based on the MIPS R4000 [2] instruction set. The hardware chip is composed of a hard fabric and a soft programmable logic, which is often a FPGA. The configurable logic portion is called the extension unit (EU). One attractive property of the EU is that it can access all the resources of the main processor, provided it is in the proper pipeline state.

There is a long history for developing techniques for compilers. Aho [3] et al. elaborates the principles and techniques of compilers in details. Verilog is a hardware design language (HDL) supported by multiple companies, used to synthesize circuits on FPGAs [6]. There is quite a bit of research on automatically generating Verilog code. Koutsougeras [7] et al. provide an automatic Verilog HDL model generator (AVMG), which is used to automatically generate circuit graphs and then Verilog code based on captured printed circuit board (PCB) images. Husueh [4] et al. propose a design method which can automatically generate Verilog code for an 8-bit RISC microcontroller with a user-defined instruction set. Soderman [5] proposes implementing C algorithms in Reconfigurable Hardware using C2VeriZog. McFarland et al. [8] present high-level synthesis that takes an abstract behavioral specification of a digital system and finds a register-transfer level structure that realizes the given behavior. Stephen [9] proposes a new language, Esterel, to abstract at a higher level than RTL, and then automatically generate Verilog code. Karpuzcu's work [10] investigates the automatic generation of Verilog code, representing digital circuits through Grammatical Evolution (GE). Our work on M2V automatically generates Verilog Code by integrating methodology taken from compilers.

Graphs are often used to assist Verilog code generation, and dataflow modeling is attracting more attention. Synchronous dataflow (SDF) is streamlined for efficient representation and benefits from static scheduling [25]. In order to increase the expressiveness to a wider range of applications, a variety of dynamic dataflow modeling techniques have been developed, including the Functional DIF [26] and the CAL actor language [27]. Our work on M2V utilizes both control flow representations and dataflow static scheduling by judiciously integrating compiler techniques with architecture-specific insights.

## 3 M2V Compiler Implementation

The M2V compiler is a four-pass compiler, as shown in Figure 2.

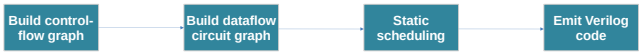


**Figure 2: The four passes of M2V, a compiler that translates MIPS instruction sequences into circuits.**

The first pass builds the control flow graph based on the relationship between basic blocks. Besides single block

cases, we deal with four kinds of basic control patterns among basic blocks: Sequential, Self-loop, Branch and Join, as shown in Figure 3. Any graph can be decomposed into a combination of these basic patterns. In a Sequential pattern, at the end of block *B1* there is an unconditional branch pointing to block *B2*. We deal with this case by combining both blocks together and re-cannibalizing the global register sets. In a Self-loop pattern, at the end of the basic block *B1* there is a conditional branch, and one of the targeted addresses is the beginning of the block. A "FOR" loop in software is one of such example. In a Branch pattern, at the end of *B1* there is a conditional branch targeting either *B2* or *B3*. In a Join pattern, there are two entry blocks *B1* and *B2* and both blocks are followed by the same block *B3*. B3 is actually executed after only one of the two blocks. After analyzing the control flow between basic blocks, we conduct different strategies for optimized implementations.

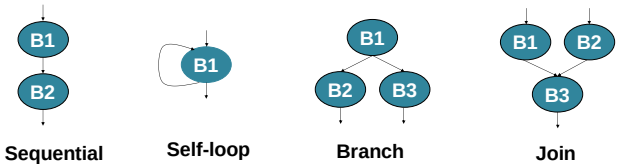


**Figure 3: M2V recognizes four kinds of basic control patterns among basic blocks. Each $B_i$ represents a different basic block *i*, *i = 1,2,3*. An arbitrary graph is decomposed as a combination of these four patterns.**

The second pass parses the .BBW file, semantically analyzes the MIPS instructions, and builds and connects nodes in a dataflow graph revealing the data dependencies. There are two types of nodes in the graph: register nodes and instruction nodes. The semantic analysis provides the cost for each instruction and the function of register dependencies. The register nodes represent a register access which could go to the register file or to a temporary storage location in the EU. The register table tracks whether a register has already been read from the register file, where the last update to the register is stored, and whether the register needs to be written back to the register file. When multiple instructions read the same unchanged register value, the register table provides the information so that only a single register node is created. Register nodes may or may not result in an actual clocked hardware register depending on specific control patterns. The final schedule for the extension determines when pipeline stages are added and whether a register node will result in a hardware register.

A major challenge for the M2V compiler is to constrain the EU such that it does not interfere with instructions flowing through the eMIPS pipeline before and after the extended instruction executes. eMIPS uses a standard five stage RISC pipeline, with IF, ID, EX, MA, and WB stages. This pipeline is tightly integrated with the EU. An extended instruction will take multiple cycles to execute since it is semantically equivalent to all of the MIPS instructions in a basic block. During ID, the extension will snoop the register reads that are visible to the primary eMIPS pipeline. If the instruction is an extended instruction, the EU will claim the instruction and stall the instructions behind it while it executes. Instructions before the extended instruction complete normally and must have access to the same resources that they would normally use. Figure 4 shows how the instructions proceed through the eMIPS pipeline, where instruction *m* is the extended instruction executed on EU. Here *m* is a single extended instruction, corresponding to one or more basic blocks.

During cycle 3 in Figure 4, the EU will decode and claim the extended instruction, snoop the reads from the register file, store the register reads, and prepare to stall the trailing instructions in cycle 4. The instruction fetch in cycle 3 does not perform useful work since this instruction is the first instruction of the accelerated basic block. During cycle 4, instruction *m-2* has control of the register write-back logic, instruction *m-1* has control of the memory access logic, and the extended instruction begins stage *EX1*. In *EX1*, reads to the register file are controlled by the EU since future instructions are stalled. In *EX2*, the EU can read from the register file and access memory through the main memory unit. The EU is in steady-state from *EX3* until *EXn-2* and it can control all ports on the register file and access to the memory logic. In stage *EXn-1*, instruction *j* must be fetched and so any branch conditions and branch addresses must be resolved by this stage. In cycle n+4, the EU must relinquish control of the register read ports to instruction *j* which is in the ID stage. In cycle n+5, the EU performs its last memory access and can also write to the register file. In cycle n+6, the EU performs its last write to the register file and the extended instruction is complete.

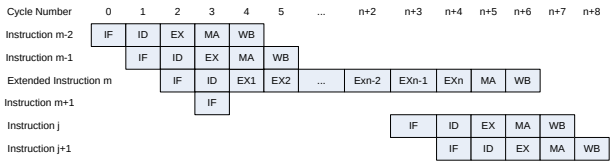


**Figure 4: eMIPS pipeline with a mix of regular and extended instructions. The Extended instruction *m* represents an entire block of instructions, and it is executed on the EU.**

The third pass of the M2V compiler creates the schedule for register and memory accesses by doing a constrained depth-first traversal of the dependency graph created in the second pass of compilation. The traversal begins at the

register nodes and continues until a dependency cannot be met. When the node cannot be completed, it is placed on a queue to be traversed in the next cycle. The nodes with unmet dependencies at the end of a cycle mark where pipeline stages will be inserted. Constraints on the schedule are the register and memory resources available in a given cycle and the delay through a sequential stream of operations. The register-to-register delay is estimated from the complexity of the instruction and the fan-out of the register nodes. The semantic analyzer provides the complexity of the instruction and the dependency graph yields the fan-out from each node. When the delay exceeds the cycle-time threshold, a pipeline stage is added. In order to distinguish with traditional meaning of "pipeline stage", we denote *state* as one intermediate stage in which a set of instructions are executed in a parallel way. Each state may contain several cycles if the cost of some instruction is more than one cycle. During static scheduling of dataflow graph, one important factor is determination of two registers, *rs* and *rt*, are encoded in the extension instruction and are available directly from the decode stage of the MIPS pipeline. The strategy to choose these two registers determines the roots of the scheduling tree, and therefore affects the execution time of the extension. In M2V we determine these two registers based on the parameters of fan-out and depth.

Based on the dependency graph, the fourth pass generates the synthesizable Verilog, which is executed on the EU.

# 4 Combination Of Control Flow Graph And Data Flow Scheduling

In M2V, we take advantage of both control-flow-based analysis and dataflow-based static scheduling.

## 4.1 Determine the Graph Type

There are two types of graphs: graphs of single basic block and graphs of multiple basic blocks. As mentioned in Section 3, currently we deal with four kinds of control structures: self-loop, sequential, branch and join. Self-loop basic block is classified as a graph containing a single block. There are therefore three types of graphs containing multiple basic blocks.

In the current version of M2V, we assume in each directed graph, there is only one entry block. By default, the entry block is indexed as *node 0*. There could be multiple exit blocks. We determine the type of graphs based on the *depth first search* (DFS) algorithm. The paths between entry and exit blocks are first calculated and recorded. We then determine the graph type, using the number of exit blocks and number of paths. For example, if there is one exit block and two paths, the graph belongs to the join case.

Detection of graph type remains to be an NP hard problem when the connection in the graph is complicated. In the class of *TopGraph*, the member function of "determineGraphType" is reserved for future use.

## 4.2 Path based Static Scheduling Trees

In order to improve time efficiency, we apply the path-based static scheduling strategy to each control flow graph. The strategy is composed of three phases: path detection, static scheduling and control-flow implementation.

In the "path detection" phase we detect different paths based on previously determined entry and exit blocks. In the current version, we assume there is only one entry block, and possible multiple exit blocks. It is possible to extend the application by choosing from multiple entry blocks. Multiplexer in the input side is a possible solution.

In the "static scheduling" phase, we apply static scheduling trees applied to a single basic block to the combined blocks. There is space to improve time efficiency by analyzing relationship between instructions from different blocks, such as code elimination and code motion.

## 4.3 Control-flow Based Self Loop Basic Block

In the previous version of M2V each basic block was executed only once, and then the resource was return back to the processor. The processor would determine the program instruction being executed next. A self-loop was implemented as a pipeline-based procedure, as shown in Figure 5. Every time one loop is finished, the resource and program counter (PC) are returned back to the processor, and the processor determines based on the PC weather to execute the extension again or not. This is not very efficient, for the processor has to start another pipeline to load and execute the extension again. Although we use the same programmable logic in the same extension, the time between executions of two loop iterations is not optimal.

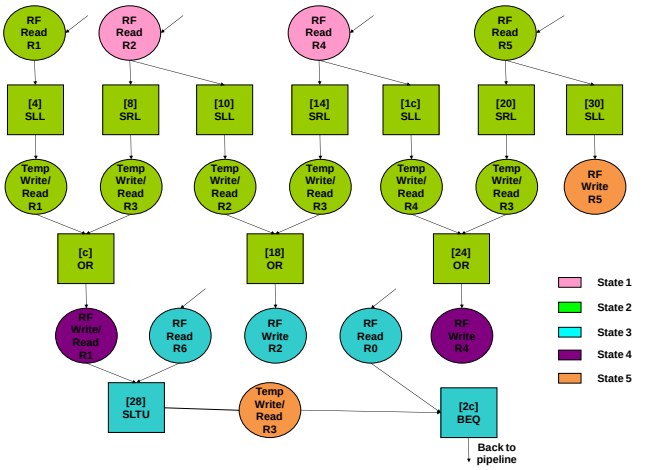


**Figure 5: Circuit graph for a pipeline-based realization of a self-looping basic block. RF indicates a register file access. The hex number in [] is the index of the instruction in the input file. In this case we start from 4. Different colors represents different state and each state may execute for one or more than one cycle. "Back to pipeline" represents the resource being returned to the processor.**

The new version of M2V uses the control-flow based scheme shown in Figure 6. In this implementation, the extension executes all the loop iterations before returning the resource back to the processor. At the end of one loop, the branch condition is calculated. If the target PC points to the beginning of the block, the extension will execute the same logic again. The processor is not involved into the execution on the extension, so all the computation of self-loop is finished in one pipeline with more cycles assigned to EX stage.

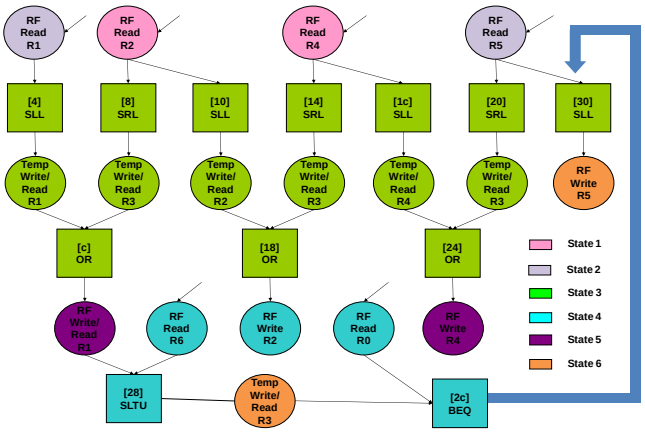


**Figure 6: Circuit graph for a control-flow-based implementation of self-looping basic block. The blue arrow represents the transition from state 4 to state 1.**

### 4.4 Mechanism for write-back

For the same register, different path may update different values. Finally we have to choose one path to execute, and then commit the write back values along this path.

## 5 Optimizations in the M2V Compiler

### 5.1 Register Sharing Among Multiple Blocks

In the input BBW file, each basic block has a distinguished mapping between canonical registers and real registers. When we combine multiple blocks together, we must correlate the registers used in the two blocks in a coherent way. The best approach is to share the registers directly. For this purpose we index the global registers as a mapping table in the "TopGraph" class. Using this mapping table, for each basic block, there is a unique mapping between old canonical registers and new canonical registers, which we call "re-canonicalization".

### 5.2 Code Elimination

Since the input of the M2V compiler is a fully-optimized MIPS program fragment, we would expect a relatively small amount of redundancy among instructions inside each basic block. However, when we combine different basic blocks together, it is possible to further eliminate some more redundant instructions. For instance, consider the instructions that are only meant to combine two blocks together. One such example is an unconditional jump. Another is a register copy.

Another kind of code elimination is redundant operation to the same set of registers without any actual modification to the output registers.

### 5.3 Code Motion

Code motion matters for performance reasons, especially the execution time performance. We use two kinds of code motion.

The first is probability-based path moving. Previous passes have determined and recorded the paths to be statically scheduled. However, when we put those paths together, the instructions are listed in a sequential order as follows:

Entry Block

Intermediate Blocks in Path 1

Exit Block of Path 1

…..

Intermediate Blocks in Path n

Exit Block of Path n

In this way, the path whose instructions are found first has the higher priority to be scheduled, since the whole statically scheduling tree is traversed from top to bottom. In M2V, we give each path a parameter of "*statProb*". The range of "*statProb*" is between 0 and 1, and the sum of all "*statProb*"s of all paths is 1. We arrange the order of paths in the list based on the value of "*statProb*". So the path with the highest "*statProb*" is put right below the entry block, which is the first place to be scheduled. By default, we assume the "*statProb*" of each path equals to each other, so we list the paths just as indexed.

The second case of code motion is to move upward a conditional jump. Since all other paths except one path will be discontinued, the execution of unused paths may result in waste of clock cycles, especially when the cost of some instruction is as big as several clock cycles. In this case, we try to move the instructions related to the conditional jump as early as possible. Once we determine the exact path to be executed, we can stop other paths as early as possible.

### 5.4 Expensive Arithmetic Logic Sharing

When assigning states, we determine in advance the combinatorial logic which can be shared in different states.

The sharing of combinatorial logic is meaningful if the resource is expensive and used in many places. This is the case of a multiplication or division.

## 6 Evaluation

For evaluation we used simulations in the combined environment of the Xilinx ISE 10.1 and ModelSim 6.4. The test program is run using Giano to simulate the full system, and ModelSim to simulate the eMIPS processor with reconfigurable extensions. We selected a 64-bit divider "mmldiv" in video games from Microsoft Xbox applications as an example. The divider is composed of 8 basic blocks, as shown in Figure 7. We simulated two scenarios: with (ACCEL) and without accelerator (NO_ACCEL). In the NO_ACCEL scenario we execute the entire mmldiv software on the eMIPS processor. In the ACCEL scenario we execute just the relevant instructions in hardware, while the rest remains on the eMIPS processor. Two different circuits realize either the branch-away case or the whole self-loop block on the accelerator.

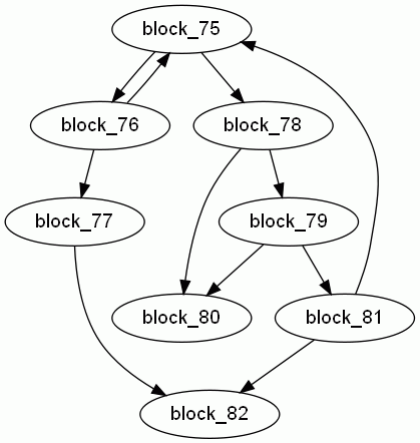


**Figure 7: Control flow graph for the MMLDIV example. This functions computes a 64-bit division in software.**

The test program for the branch-away case is run 60 times with different inputs, and the self-loop case is run 30 times. The average wall time for the simulation across all runs is shown in Table 1. The branch-away case shows only a 16% improvement. There are total of 31 instructions in mmldiv, and only 8 instructions (Blocks 78-80 in Figure 7) account for the branch-away case. The mmldiv example spends most of the time in Blocks 75-76 instead, which are covered by the self-loop case. With an average of 32 repetitions per activation, this block provides a more significant performance improvement; the overall simulation time without acceleration is 22.3 times higher. The previous version of M2V was able to speed up this application only by 2.3x.

**Table 1: Average Simulation Time Comparison (Sec)**

| Simulation case | NO_ACCEL | ACCEL |
|---|---|---|
| Branch-away | 2773 | 2337 |
| Single self-loop block | 1027 | 46 |

## 7 Notes and Future Work

Currently M2V supports any graph of basic blocks with one entry block and one exit block, which implies that there is only one PC passed from/to the processor. In case there are more than one exit blocks, there is competition from different branches, and therefore different PCs. It is better to design an integrated interface for multiple exit blocks.

The current version of M2V is not scalable in terms of control flow implementations. On one hand, the generated Verilog code utilizes a flat state machine to control the state transition of all the blocks. It is efficient and convenient when the number of paths is small and the number of states

is not large. However, when the number of states is large, it is difficult to debug the state transition between states. An implementation mechanism of hierarchical state machine is a promising future direction.

Another note is about instructions supported by current M2V. Although all the demos provided by current M2V works fine, there are still some instruction not recognized and supported by M2V. These bugs can be fixed when different examples are applied.

One possible and interesting extension from current M2V is to extend the library of control structure. Theoretically M2V works on any complicated graphs by using combination of four basic structures. However, there is still space if we treat some complicated control structure as a single graph, instead of a combination of graphs.

## 8 Conclusions

M2V, as part of tool chain shown in Figure 1, provides the solution to automatically generate hardware accelerators from software binaries. By taking advantage of both control flow analysis and dataflow scheduling, the Verilog Code automatically generated from M2V efficiently implemented dynamic applications, such as branch, join and self-loop blocks.

Currently M2V supports the eMIPS platform by providing hardware solution for the reconfigurable logic of the extensions. Furthermore, M2V can be applied to a wider range of FPGA systems, provided the issue of partial reconfiguration is addressed.

## References


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

[2] Heinrich, J. MIPS R4000 Microprocessor User's Manual. 1994.

[3] Aho, A. V., Lam, M. S., Sethi, R. Compiler: Principles, Techniques, and Tools. Addison Wesley Publishers, Boston, MA. 2007.

[4] Husueh, Y. T., Chang, W. C., Lai, J. M. Automatic Verilog code generation of an 8-bit RISC micro-controller. *ASIC, 2002. Proceedings. 2002 IEEE Asia-Pacific Conference on* , vol., no., pp. 327-330, 2002.

[5] Soderman, D. and Panchul, Y. Implementing C Algorithms in Reconfigurable Hardware Using C2Verilog. In *Proceedings of the IEEE Symposium on FPGAs For Custom Computing Machines* (April 15 - 17, 1998). FCCM. IEEE Computer Society, Washington, DC, 339.

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

[7] Koutsougeras, C., Bourbakis, N. G., and Gallardo, V., Reverse engineering of real PCB level design using VERILOG HDL. *Intl. Journal of Engineering Intelligent Systems*, V10, No 2, pp. 63-68, June 2002.

[8] McFarland, M. C., Parker, A. C., and Camposano, R. Tutorial on high-level synthesis. In *Proceedings of the 25th ACM/IEEE Design Automation Conference.* Annual ACM IEEE Design Automation Conference. IEEE Computer Society Press, Los Alamitos, CA, pp. 330-336.

[9] Edwards, S. A. High-level Synthesis from the Synchronous Language Esterel. In *Proceedings of the International Workshop on Logic and Synthesis (IWLS).* New Orleans, Louisiana, June, 2002.

[10] Karpuzcu, U. R. Automatic Verilog Code Generation through Grammatical Evolution. In *Proceedings of the 2005 Workshops on Genetic and Evolutionary Computation (GECCO '05)*. ACM, New York, NY, pp. 394-397.

[11] Safari, S., Jahangir, A. H., and Esmaeilzadeh, H. A parameterized graph-based framework for high-level test synthesis. *Integr. VLSI J.* 39, 4 (Jul. 2006), pp. 363-381.

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

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

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

[15] 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*, pp. 651-656.

[16] Mittal, G., Zaretsky, D.C., Xiaoyong Tang, Banerjee, P. Automatic translation of software binaries onto FPGAs. *In Proceedings of Design Automation Conference (DAC)*, 2004.

[17] Mittal, G., Zaretsky, D.C., Xiaoyong Tang, Banerjee, P. An overview of a compiler for mapping software binaries to hardware. *IEEE VLSI*, 2007.

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

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

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

[21] Chandrasekhar, V., Forin, A. Mining Sequential Programs for Coarse-grained Parallelism using Virtualization, *MSR-TR-2008-113*, Microsoft Research, WA, August 2008.

[22] Microsoft Giano, http://research.microsoft.com/en-us/projects/giano/

[23] Bacharach, J., Qumsiyeh, D., Tobenkin, M. Hardware Scriptin in Gel. *FCCM'08*, pp. 13-22, April 2008.

[24] Meier, K., Forin, A. MIPS-to-Verilog, Hardware Compilation for the eMIPS Processor, *16th Symposium on Field-Programmable Custom Computing Machines*, Stanford, CA, April 2008.

[25] Lee, E. A. and Messerschmitt, D. G. Synchronous dataflow, *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[26] Plishker, W., Sane, N., Kiemb, M., Anand, K., Bhattacharyya, S. S. Functional DIF for rapid prototyping, *in Proceedings of the International Symposium on Rapid System Prototyping*, Monterey, California, June 2008, pp. 17–23.

[27] Eker, J. and Janneck, J. W. CAL language report, language version 1.0 — document edition 1, Electronics Research Laboratory, University of California at Berkeley, *Tech. Rep. UCB/ERL M03/48*, December 2003.

[28] Mittal, G., Zaretsky, D.C., Tang, X. and Banerjee, P. An overview of a compiler for mapping software binaries to hardware, *IEEE VLSI*, 2007.

## Appendix I – Verilog Output for the Self-loop Basic Block

```
1   // test_loop.v
2   // auto-generated by m2v revision 1 on Mon Jul 13 11:09:45 2009
3   // MIPS to Verilog (m2v) module (_mod) boilerplate (_bp)

4   // Copyright (c) Microsoft Corporation. All rights reserved.

5   `timescale 1ns / 1ps

6   module mmlite_div64 (
7   //
8   // INFO: reading from m2v_mod_bp_ports.v
9   //
10  /*****Ports*************************************************************/
11  /* INPUT PORTS */
12  input           CLK,            /* System Clock 50 - 100 MHZ */
13  input           EN,             /* Enable */
14  input           EXCEXT,             /* Exception Flush */
15  input           EXTNOP_MA,              /* Extension Bubble in Memory Access Phase */
16  input           GR,             /* Grant Pipeline Resources */
17  input [31:0]    INSTR,          /* Current Instruction */
18  input [31:0]    PC,             /* Current PC External */
19  input           PCLK,               /* Pipeline Clock */
20  input [31:0]    RDREG1DATA,         /* Register Read Port 1 Register Data */
21  input [31:0]    RDREG2DATA,         /* Register Read Port 2 Register Data */
22  input [31:0]    RDREG3DATA,         /* Register Read Port 3 Register Data */
23  input [31:0]    RDREG4DATA,         /* Register Read Port 4 Register Data */
24  input           REGEMPTY,           /* Register Write Buffer Empty */
25  input           REGFULL,            /* Register Write Buffer Full */
26  input           REGRDY,             /* Register Write Buffer Ready */
27  input           RESET,          /* System Reset */
28  input [31:0]    MDATA_IN,       /* Memory Data In */
29  /* Multiplexed: */
30  /*    Memory Data In */
31  /*    Peripheral Memory Data In */
32  /*    Memory Data Monitor */
33  input       MDATA_VLD_IN,   /* Memory Data Valid */

34  /* OUTPUT PORTS */
35  output      ACK,            /* Enable Acknowledged */
36  output [31:0] EXTADD,           /* Extension Address */
37  /* Multiplexed: */
38  /*    Next PC */
39  /*    Exception Address */
40  /*    PC Memory Access Phase */
41  output      PCNEXT,         /* Conditional PC Update */
42  output [4:0]    RDREG1,             /* Register Read Port 1 Register Number */
43  /* Multiplexed: */
44  /*    Register Read Port 1 Register Number */
45  /*    Register Write Port 1 Register Number */
46  /*    Write Register Memory Access Phase */
47  output [4:0]    RDREG2,             /* Register Read Port 2 Register Number */
48  /* Multiplexed: */
49  /*    Register Read Port 2 Register Number */
50  /*    Register Write Port 2 Register Number */
51  /*        <0> Register Write Enable Memory Access Phase */
52  /*    <1> Memory to Register Memory Acess Phase */
53  output [4:0]    RDREG3,             /* Register Read Port 3 Register Number */
54  /* Multiplexed: */
55  /*    Register Read Port 3 Register Number */
56  output [4:0]    RDREG4,             /* Register Read Port 4 Register Number Internal */
57  /* Multiplexed: */
58  /*    Register Read Port 4 Register Number */
59  /*    <1:0> Data Address [1:0] Memory Access Phase */
60  /*        <2> Right/Left Unaligned Load/Store Memory Access Phase */
61  /*    <3> Byte/Halfword Load/Store Memory Acess Phase */
```

```
62  output          REGWRITE1,          /* Register Write Port 1 Write Enable */
63  output          REGWRITE2,          /* Register Write Port 2 Write Enable */
64  output          REWB,               /* Re-enter at Writeback */
65  output          RI,                 /* Reserved/Recognized Instruction */
66  output [31:0] WRDATA1,              /* Register Write Port 1 Data Internal */
67  /* Multiplexed: */
68  /*    Register Write Port 1 Data */
69  /*    ALU Result Memory Access Phase */
70  output [31:0] WRDATA2,              /* Register Write Port 2 Data Internal */
71  /* Multiplexed: */
72  /*    Register Write Port 2 Data */
73  /*    Memory Data Out Memory Access Phase */
74  output          BLS_OUT,            /* Byte Load/Store */
75  output        HLS_OUT,            /* Halfword Load/Store */
76  output          RNL_OUT,            /* Memory Right/Left Unaligned Load/Store */
77  output [31:0] MADDR_OUT,          /* Memory Address */
78  output [31:0] MDATA_OUT,          /* Memory Data Out */
79  /* Multiplexed: */
80  /*    Memory Data Out */
81  /*    Peripheral Memory Data Out */
82  output          MOE_OUT,            /* Memory Output Enable */
83  output          MWE_OUT           /* Memory Write Enable */
84  );

85  /*****Signals*********************************************************************/

86  wire [31:0]   ALURESULT_WB;       /* ALU Result to Writeback Phase */
87  wire            BHLS_WB;            /* Byte/Halfword Load/Store to Writeback Phase */
88  wire [31:0]   CJMPADD;            /* Conditional Jump address to offset from Current PC */
89  wire [15:0]   DIMM_EX;            /* Data Immediate Execute Phase */
90  wire [15:0]   DIMM_ID;            /* Data Immediate Instruction Decode Phase */
91  wire [1:0] DMADD_WB;          /* Least Significant Bits of Data Address to Writeback Phase */
92  wire [31:0]   DMDATAOUT_WB;         /* Memory Data Out to Writeback Phase */
93  wire            DNE;                /* Execution Done */
94  wire            EN_EX;              /* Enable Execute Phase */
95  wire [31:0]   JMPADD;             /* Jump address to end of basic block */
96  wire            MEMTOREG_WB;        /* Memory to Register to Writeback Phase */
97  wire [31:0]   PC_EX;              /* PC Execute Phase */
98  wire [31:0]   PC_WB;              /* PC to Writeback Phase */
99  wire [4:0] RD_EX;             /* Destination Register Execution Phase */
100 wire [4:0] RDREG1_EX;          /* Register Read Port 1 Register Number Execute Phase */
101 wire [31:0]   RDREG1DATA_EX;       /* Register Read Port 1 Register Data Execute Phase */
102 wire [4:0] RDREG2_EX;          /* Register Read Port 2 Register Number Execute Phase */
103 wire [31:0]   RDREG2DATA_EX;       /* Register Read Port 2 Register Data Execute Phase */
104 wire [4:0] RDREG3_EX;          /* Register Read Port 3 Register Number Execute Phase */
105 wire [4:0] RDREG4_EX;          /* Register Read Port 4 Register Number Execute Phase */
106 wire            REGWRITE_EX;      /* Register Write Execute Phase */
107 wire            REGWRITE_ID;      /* Register Write Instruction Decode Phase */
108 wire            REGWRITE_WB;        /* Register Write to Writeback Phase */
109 wire            RESET_EX;           /* Reset Execute Phase */
110 wire [31:0]   RESULT_EX;          /* Result Execution Phase */
111 wire            RNL_WB;             /* Right/Left Unaligned Load/Store to Writeback Phase */
112 wire [4:0] RS_EX;             /* Operand Register 1 Execute Phase */
113 wire [4:0] RS_ID;             /* Operand Register 1 Instruction Decode Phase */
114 wire [4:0] RT_EX;             /* Operand Register 2 Execute Phase */
115 wire [4:0] RT_ID;             /* Operand Register 2 Instruction Decode Phase */
116 wire            SLL128_EX;          /* Shift Left Logical 128 bits Execute Phase */
117 wire            SLL128_ID;          /* Shift Left Logical 128 bits Instruction Decode Phase */
118 wire [31:0]   WRDATA1_EX;         /* Register Write Port 1 Data Execute Phase */
119 wire [31:0]   WRDATA2_EX;         /* Register Write Port 2 Data Execute Phase */
120 wire [4:0] WRREG_WB;          /* Write Register Number to Writeback Phase */
121 wire [4:0] WRREG1_EX;         /* Register Write Port 1 Register Number Execute Phase */
122 wire [4:0] WRREG2_EX;         /* Register Write Port 2 Register Number Execute Phase */
123 wire [31:0]   VIRPC;

124 /*****Registers*******************************************************************/

125 reg en_reg;   /* Enable */
126 reg gr_reg;   /* Grant Pipeline Resources */
```

```
/*****Initialization*****************************************************************/
/*
initial
begin
en_reg = 1'b0;
gr_reg = 1'b0;
end
*/

/***********************************************************************/

assign EXTADD    =  (EXCEXT)?                  VIRPC:
(en_reg)?                JMPADD:
(PCNEXT)?                   CJMPADD:
(REWB)?                     PC_WB:
32'hffffffff;
/*
The rest cannot be zero'ed out as the extension state machine
might still have to be completed till a particular transaction ends
*/
assign RDREG1 =  (gr_reg & REGWRITE1)?  WRREG1_EX:
//(EXCEXT)?                 5'b0:
(REWB & gr_reg)?        WRREG_WB:
(gr_reg)?                RDREG1_EX:
5'b11111;
assign RDREG2 =  (gr_reg & REGWRITE2)?  WRREG2_EX:
//(EXCEXT)?                 5'b0:
(REWB & gr_reg)?        {3'b0,MEMTOREG_WB,REGWRITE_WB}:
(gr_reg)?                RDREG2_EX:
5'b11111;
assign RDREG3 =  (REWB & gr_reg)?          5'b0:
//(EXCEXT)?                 5'b0:
(gr_reg)?                RDREG3_EX:
5'b11111;
assign RDREG4 =  (REWB & gr_reg)?          {1'b0,BHLS_WB,RNL_WB,DMADD_WB}:
//(EXCEXT)?                 5'b0:
(gr_reg)?                RDREG4_EX:
5'b11111;
assign WRDATA1 = (gr_reg & REGWRITE1)?    WRDATA1_EX:
//(EXCEXT)?                 32'b0:
(REWB)?                  ALURESULT_WB:
32'hffffffff;
assign WRDATA2 = (gr_reg & REGWRITE2)?    WRDATA2_EX:
//(EXCEXT)?                 32'b0:
(REWB)?                  DMDATAOUT_WB:
32'hffffffff;

//
// INFO: finished reading from m2v_mod_bp_ports.v
//

//
// 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)
//
mmlite_div64_ext_id id (
//
// INFO: reading from instantiate_ext_id.v
//
.CLK(CLK),
.DIMM(DIMM_ID),
.EN(EN),
.JMPADD(JMPADD),
.INSTR(INSTR),
.PC(PC),
.REGWRITE(REGWRITE_ID),
.RESET(RESET),
```

```
191  .RI(RI),
192  .RS(RS_ID),
193  .RT(RT_ID),
194  .SLL128(SLL128_ID)
195  );
196  //
197  // INFO: finished reading from instantiate_ext_id.v
198  //

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

200  mmlite_div64_toex to_ex (
201  //
202  // INFO: reading from instantiate_to_ex.v
203  //
204  .ACK(ACK),
205  .CLK(CLK),
206  .DIMM_EX(DIMM_EX),
207  .DIMM_ID(DIMM_ID),
208  .EN_EX(EN_EX),
209  .EN_ID(EN),
210  .EXCEXT(EXCEXT),
211  .PC_EX(PC_EX),
212  .PC_ID(PC),
213  .PCLK(PCLK),
214  .RDREG1DATA_EX(RDREG1DATA_EX),
215  .RDREG1DATA_ID(RDREG1DATA),
216  .RDREG2DATA_EX(RDREG2DATA_EX),
217  .RDREG2DATA_ID(RDREG2DATA),
218  .REGWRITE_EX(REGWRITE_EX),
219  .REGWRITE_ID(REGWRITE_ID),
220  .RESET(RESET),
221  .RESET_EX(RESET_EX),
222  .RS_EX(RS_EX),
223  .RS_ID(RS_ID),
224  .RT_EX(RT_EX),
225  .RT_ID(RT_ID),
226  .SLL128_ID(SLL128_ID),
227  .SLL128_EX(SLL128_EX)
228  );

229  //
230  // INFO: finished reading from instantiate_to_ex.v
231  //


232  //
233  // instantiate the execution module for the extension instruction
234  //   - the execution module is auto generated and appended to the
235  //     end of the verilog file (a.v unless redefined)
236  //
237  mmlite_div64_ext_ex ex(
238  //
239  // INFO: reading from instantiate_ex.v
240  //
241  .ACK(ACK),
242  .DIMM(DIMM_EX),
243  .DNE(DNE),
244  .CLK(CLK),
245  .CJMPADD(CJMPADD),
246  .EN(EN_EX),
247  .EXTNOP_MA(EXTNOP_MA),
248  .GR(GR),
249  .PC(PC_EX),
250  .PCLK(PCLK),
251  .PCNEXT(PCNEXT),
252  .RD(RD_EX),
253  .RDREG1(RDREG1_EX),
254  .RDREG1DATA(RDREG1DATA),
```

```
255  .RDREG1DATA_ID(RDREG1DATA_EX),
256  .RDREG2(RDREG2_EX),
257  .RDREG2DATA(RDREG2DATA),
258  .RDREG2DATA_ID(RDREG2DATA_EX),
259  .RDREG3(RDREG3_EX),
260  .RDREG3DATA(RDREG3DATA),
261  .RDREG4(RDREG4_EX),
262  .RDREG4DATA(RDREG4DATA),
263  .REGEMPTY(REGEMPTY),
264  .REGFULL(REGFULL),
265  .REGRDY(REGRDY),
266  .REGWRITE1(REGWRITE1),
267  .REGWRITE2(REGWRITE2),
268  .RESET(RESET_EX),
269  .RESULT(RESULT_EX),
270  .RS(RS_EX),
271  .RT(RT_EX),
272  .SLL128(SLL128_EX),
273  .WRDATA1(WRDATA1_EX),
274  .WRDATA2(WRDATA2_EX),
275  .WRREG1(WRREG1_EX),
276  .WRREG2(WRREG2_EX),
277  .MDATA_IN(MDATA_IN),
278  .MDATA_VLD_IN(MDATA_VLD_IN),
279  .BLS_OUT(BLS_OUT),
280  .HLS_OUT(HLS_OUT),
281  .RNL_OUT(RNL_OUT),
282  .MOE_OUT(MOE_OUT),
283  .MWE_OUT(MWE_OUT),
284  .MADDR_OUT(MADDR_OUT),
285  .MDATA_OUT(MDATA_OUT),
286  .EXCEXT(EXCEXT),
287  .VIRPC(VIRPC)
288  );
289  //
290  // INFO: finished reading from instantiate_ex.v
291  //


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

293  mmlite_div64_topipe_wb to_wb(
294  //
295  // INFO: reading from instantiate_wb.v
296  //
297  .ACK(ACK),
298  .ALURESULT_WB(ALURESULT_WB),
299  .BHLS_WB(BHLS_WB),
300  .CLK(CLK),
301  .DMADD_WB(DMADD_WB),
302  .DMDATAOUT_WB(DMDATAOUT_WB),
303  .DNE(DNE),
304  .EN_EX(EN_EX),
305  .EXCEXT(EXCEXT),
306  .EXTNOP_MA(EXTNOP_MA),
307  .PC_EX(PC_EX),
308  .PC_WB(PC_WB),
309  .PCLK(PCLK),
310  .MEMTOREG_WB(MEMTOREG_WB),
311  .RD_EX(RD_EX),
312  .REGWRITE_EX(REGWRITE_EX),
313  .REGWRITE_WB(REGWRITE_WB),
314  .RESET(RESET),
315  .RESULT_EX(RESULT_EX),
316  .REWB(REWB),
317  .RNL_WB(RNL_WB),
318  .WRREG_WB(WRREG_WB)
319  );
320  //
```

```
// INFO: finished reading from instantiate_wb.v
//


/*********************************************************************/
always@(posedge CLK)
begin
if (RESET == 1'b0)
begin
en_reg <= 1'b0;
gr_reg <= 1'b0;
end
else
begin
en_reg <= EN;
gr_reg <= GR;
end
end

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

module mmlite_div64_topipe_wb(
//
// INFO: reading from module_topipe_wb.v
//
/*****Ports*************************************************************************/
/* INPUT PORTS */
input           ACK,            /* Enable Acknowledged */
input           CLK,            /* System Clock 50 - 100 MHZ */
input           DNE,            /* Execution Done */
input           EN_EX,          /* Enable Execute Phase */
input           EXCEXT,             /* Exception Flush */
input           EXTNOP_MA,          /* Extension Bubble in Memory Access Phase */
input [31:0]    PC_EX,          /* Current PC Execute Phase */
input           PCLK,               /* Pipeline Clock */
input [4:0]     RD_EX,          /* Destination Register Execution Phase */
input               REGWRITE_EX, /* Register Write Execute Phase */
input           RESET,          /* System Reset */
input [31:0]    RESULT_EX,      /* Result Execution Phase */
/* OUTPUT PORTS */
output [31:0] ALURESULT_WB,  /* ALU Result to Writeback Phase */
output          BHLS_WB,        /* Byte/Halfword Load/Store to Writeback Phase */
output [1:0]    DMADD_WB,     /* Least Significant Bits of Data Address to Writeback Phase */
output [31:0] DMDATAOUT_WB,    /* Memory Data Out to Writeback Phase */
output          MEMTOREG_WB,  /* Memory to Register to Writeback Phase */
output [31:0] PC_WB,         /* Current PC to Writeback Phase */
output          REGWRITE_WB,    /* Register Write to Writeback Phase */
output          REWB,               /* Re-enter at Writeback */
output          RNL_WB,             /* Right/Left Unaligned Load/Store to Writeback Phase */
output [4:0]    WRREG_WB            /* Write Register Number to Writeback Phase */
);

/*****Signals***********************************************************************/

wire EN_WB;        /* Enable to Writeback Phase */
wire RESET_WB;  /* Reset to Writeback Phase */

/*****Registers*********************************************************************/

reg [70:0] ex_wb;       /* Execute -> to Writeback Pipeline Register */
reg [1:0] pclkcnt;    /* Pipeline Clock edge detection */
reg reset_reg;          /* Reset to Writeback Phase */
reg rewb_reg;               /* Re-enter at Writeback */

/*****Initialization****************************************************************/
/*
initial
begin
```

```
ex_wb = 71'b0;
pclkcnt = 2'b0;
rewb_reg = 1'b0;
reset_reg = 1'b0;
end
*/
/*************************************************************************/

assign RESET_WB     =  reset_reg;
assign REWB         =  rewb_reg & EN_WB;
assign EN_WB        =  ex_wb[70];      //EN_EX;
assign REGWRITE_WB  =  ex_wb[69];      //REGWRITE_EX;
assign MEMTOREG_WB  =  1'b0;
assign RNL_WB       =  1'b0;
assign BHLS_WB      =  1'b0;
assign DMADD_WB     =  2'b0;
assign WRREG_WB     =  ex_wb[68:64]; //RD_EX;
assign ALURESULT_WB =  ex_wb[63:32]; //RESULT_EX;
assign DMDATAOUT_WB =  32'b0;
assign PC_WB        =  ex_wb[31:0];   //PC_EX;

/*************************************************************************/

always@(posedge CLK)
begin
/* Pipeline Clock edge detection */
pclkcnt <= {pclkcnt[0],PCLK};   // here Neil suggest to change from = to <=

end

always@(posedge CLK)
begin
case(pclkcnt)
2'b01       :   begin
/* Synchronize Reset to Pipeline Clock */
reset_reg <= RESET;
end
default :   begin
end
endcase
end

always@(posedge CLK)
begin
/* Execute -> to Memory Access Pipeline Register */
casex({pclkcnt,RESET_WB,EXTNOP_MA,rewb_reg,ACK,DNE,EXCEXT})
8'bxx0xxxxx :   begin
/* Reset */
rewb_reg <= 1'b0;
ex_wb <= 71'b0;
end
8'b011xxxx1 :   begin
/* Exception in Pipeline, Flush */
rewb_reg <= 1'b0;
ex_wb <= 71'b0;
end
8'bxx1x0110 :   begin
/* Latch Data and Control after Execution Finishes */
ex_wb <= {EN_EX,REGWRITE_EX,RD_EX,RESULT_EX,PC_EX};
end
8'b101100x0 :   begin
/* Raise REWB at next Negedge of PCLK after ACK Lowers */
rewb_reg <= 1'b1;
end
8'b011x1xx0 :   begin
/* Lower REWB at next Posedge and reset register */
rewb_reg <= 1'b0;
ex_wb <= 71'b0;
end
```

```
444 default :       begin
445 /* NOP */
446 end
447 endcase
448 end
449 endmodule
450 //
451 // INFO: finished reading from module_topipe_wb.v
452 //


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

454 module mmlite_div64_toex (
455 //
456 // INFO: reading from module_toex.v
457 //
458 /*****Ports*************************************************************************/
459 /* INPUT PORTS */
460 input           ACK,            /* Enable Acknowledged */
461 input           CLK,            /* System Clock 50 - 100 MHZ */
462 input [15:0]    DIMM_ID,        /* Data Immediate Instruction Decode Phase */
463 input           EN_ID,          /* Enable Instruction Decode Phase */
464 input           EXCEXT,         /* Exception Flush */
465 input [31:0]    PC_ID,          /* Current PC Decode Phase */
466 input           PCLK,           /* Pipeline Clock */
467 input [31:0]    RDREG1DATA_ID, /* Register Read Port 1 Register Data Instruction Decode Phase */
468 input [31:0]    RDREG2DATA_ID, /* Register Read Port 2 Register Data Instruction Decode Phase */
469 input           REGWRITE_ID,  /* Register Write Instruction Decode Phase*/
470 input           RESET,          /* System Reset */
471 input [4:0]     RS_ID,          /* Operand Register 1 Instruction Decode Phase */
472 input [4:0]     RT_ID,          /* Operand Register 2 Instruction Decode Phase */
473 input           SLL128_ID,      /* Shift Left Logical 128 bits Instruction Decode Phase */
474 /* OUTPUT PORTS */
475 output [15:0] DIMM_EX,        /* Data Immediate Execute Phase */
476 output          EN_EX,          /* Enable Execute Phase */
477 output [31:0] PC_EX,          /* Current PC Instruction Decode Phase */
478 output [31:0] RDREG1DATA_EX,/* Register Read Port 1 Register Data Execute Phase */
479 output [31:0] RDREG2DATA_EX,/* Register Read Port 2 Register Data Execute Phase */
480 output          REGWRITE_EX, /* Register Write Execute Phase*/
481 output          RESET_EX,       /* Reset Execute Phase */
482 output [4:0]    RS_EX,          /* Operand Register 1 Execute Phase */
483 output [4:0]    RT_EX,          /* Operand Register 2 Execute Phase */
484 output          SLL128_EX       /* Shift Left Logical 128 bits Execute Phase */
485 );

486 /*****Registers*************************************************************************/

487 reg [124:0]    id_ex;    /* Instruction Decode -> Execute Pipeline Register */
488 reg [1:0]   pclkcnt;      /* Pipeline Clock edge detection */
489 reg         reset_reg;    /* Reset Execute Phase */

490 /*****Initialization*************************************************************************/

491 /*
492 initial
493 begin
494 id_ex = 125'b0;
495 pclkcnt = 2'b0;
496 reset_reg = 1'b0;
497 end
498 */

499 /*********************************************************************************/

500 assign RESET_EX     =   reset_reg;
501 assign EN_EX        =   id_ex[124];         //EN_ID;
502 assign SLL128_EX    =   id_ex[123];         //SLL128_ID;
503 assign REGWRITE_EX  =   id_ex[122];         //REGWRITE_ID;
```

```
504  assign RS_EX            =  id_ex[121:117];   //RS_ID;
505  assign RT_EX            =  id_ex[116:112];   //RT_ID;
506  assign DIMM_EX          =  id_ex[111:96];    //DIMM_ID;
507  assign PC_EX            =  id_ex[95:64];     //PC_ID;
508  assign RDREG1DATA_EX  =  id_ex[63:32];     //RDREG1DATA_ID;
509  assign RDREG2DATA_EX  =  id_ex[31:0];    //RDREG2DATA_ID

510  /*********************************************************************/

511  always@(posedge CLK)
512  begin
513  /* Pipeline Clock edge detection */
514  pclkcnt = {pclkcnt[0],PCLK};   // karl, 9/19, change to non-blocking to
515  // match Neil
516  end

517  always@(posedge CLK)
518  begin
519  case(pclkcnt)
520  2'b01       :   begin
521  /* Synchronize Reset to Pipeline Clock */
522  reset_reg <= RESET;
523  end
524  default :   begin
525  end
526  endcase
527  end

528  always@(posedge CLK)
529  begin
530  /* Instruction Decode -> Execute Pipeline Register */
531  casex({pclkcnt,RESET_EX,ACK,EXCEXT})
532  5'bxx0xx  :   begin
533  /* Reset */
534  id_ex <= 109'b0;
535  end
536  5'b011x1  :   begin
537  /* Exception in Pipeline, Flush */
538  id_ex <= 109'b0;
539  end
540  5'bxx110  :   begin
541  /* Hold state during Execute Phase */
542  end
543  5'b01100  :   begin
544  /* Clocking the Pipeline */
545  id_ex <= {EN_ID,SLL128_ID,REGWRITE_ID,RS_ID,RT_ID,DIMM_ID,PC_ID,RDREG1DATA_ID,RDREG2DATA_ID};
546  end
547  default :   begin
548  /* NOP */
549  end
550  endcase
551  end
552  endmodule
553  //
554  // INFO: finished reading from module_toex.v
555  //


556  //
557  // extension instruction decode
558  //
559  module mmlite_div64_ext_id(
560  input       CLK,
561  input       EN,
562  input [31:0]  INSTR,
563  input [31:0]  PC,
564  input       RESET,

565  output reg [15:0] DIMM,
```

```
566  output reg [31:0] JMPADD,
567  output reg        REGWRITE,
568  output reg        RI,
569  output reg [4:0]  RS,
570  output reg [4:0]  RT,
571  output reg        SLL128
572  );

573  reg [31:0] jmpadd_c;
574  reg en_r;
575  reg [5:0] op_r;
576  reg [31:0] pc_r;
577  reg opcode_match;

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

583  // sequential logic for instruction decode
584  always @ (posedge CLK) begin
585  if (!RESET) begin
586  DIMM     <= 16'h0;
587  op_r     <= 6'h0;
588  RS       <= 5'h0;
589  RT       <= 5'h0;
590  en_r     <= 1'h0;
591  pc_r     <= 32'h0;
592  JMPADD   <= 32'h0;
593  RI       <= 1'h1;
594  SLL128   <= 1'h0;
595  REGWRITE <= 1'h0;
596  end else begin
597  DIMM     <= INSTR[15:0];
598  op_r     <= INSTR[31:26];
599  RS       <= INSTR[25:21];
600  RT       <= INSTR[20:16];
601  en_r     <= EN;
602  pc_r     <= PC;
603  JMPADD   <= jmpadd_c;
604  RI       <= ~opcode_match;
605  SLL128   <= en_r & opcode_match;
606  REGWRITE <= en_r & opcode_match;
607  end
608  end
609  endmodule

610  module mmlite_div64_ext_ex (
611  //
612  // INFO: reading from m2v_ex_bp_ports.v
613  //
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  input [31:0]   MDATA_IN,      /* Memory Data In */
637  /* Multiplexed: */
638  /*    Memory Data In */
639  /*    Peripheral Memory Data In */
640  /*    Memory Data Monitor */
641  input          MDATA_VLD_IN,   /* Memory Data Valid */
642  input    EXCEXT,      /* Exception Signal */

643  /* OUTPUT PORTS */
644  output reg             ACK,           /* Enable Acknowledged */
645  output reg [31:0] CJMPADD,            /* Conditional Jump address to offset from Current PC */
646  output reg             DNE,           /* Execution Done */
647  output reg             PCNEXT,            /* Conditional PC Update */
648  output reg [4:0]  RD,              /* Destination Register */
649  output reg             REGWRITE1,     /* Register Write Port 1 Write Enable */
650  output reg             REGWRITE2,     /* Register Write Port 2 Write Enable */
651  output reg [4:0]  RDREG1,          /* Register Read Port 1 Register Number */
652  output reg [4:0]  RDREG2,          /* Register Read Port 2 Register Number */
653  output reg [4:0]  RDREG3,          /* Register Read Port 3 Register Number */
654  output reg [4:0]  RDREG4,          /* Register Read Port 4 Register Number */
655  output reg [31:0] RESULT,          /* Result */
656  output reg [31:0] WRDATA1,             /* Register Write Port 1 Data */
657  output reg [31:0] WRDATA2,             /* Register Write Port 2 Data */
658  output reg [4:0]  WRREG1,          /* Register Write Port 1 Register Number */
659  output reg [4:0]  WRREG2,          /* Register Write Port 2 Register Number */
660  output  reg    BLS_OUT,         /* Byte Load/Store */
661  output reg HLS_OUT,          /* Halfword Load/Store */
662  output  reg    RNL_OUT,         /* Memory Right/Left Unaligned Load/Store */
663  output reg [31:0]    MADDR_OUT,        /* Memory Address */
664  output reg [31:0]    MDATA_OUT,        /* Memory Data Out */
665  /* Multiplexed: */
666  /*    Memory Data Out */
667  /*    Peripheral Memory Data Out */
668  output  reg    MOE_OUT,           /* Memory Output Enable */
669  output  reg    MWE_OUT,           /* Memory Write Enable */
670  output reg [31:0] VIRPC   /* Virtual PC for interrupt support */
671  );

672  // tie off outputs that are not used in the automated accelerator
673  always @ (posedge CLK) begin
674  RD <= 0;
675  RESULT <= 0;
676  end

677  /*****************************************************************************/

678  //
679  // INFO: finished reading from m2v_ex_bp_ports.v
680  //

681  // parameters for extension execution block
682  parameter MAX_STATE = 8;
683  parameter REG_READ_WAIT_STATES = 5;

684  // declarations for extension state machine
685  reg[MAX_STATE:1] state_r;
686  reg[7:1] branch_state_r;
687  reg[7:1] write_state_r;
688  reg[7:1] read_state_r;

689  // declarations for the extension memory state machine
690  reg[7:1] mem_write_state_r;
691  reg[7:1] mem_read_state_r;
```

```
692  // declarations for memory data in variable
693  reg[31:0] mdata_in;

694  // declarations for register read variables
695  reg[31:0] r9_1;
696  reg[31:0] r8_4, r8_4_r;
697  reg[31:0] r4_11, r4_11_r;
698  reg[31:0] r5_18;
699  reg[31:0] r6_23;
700  // declarations for register temp variables
701  reg[31:0] r9_3;
702  reg[31:0] r11_6;
703  reg[31:0] r9_8, r9_8_r;
704  reg[31:0] r8_10;
705  reg[31:0] r11_13, r11_13_r;
706  reg[31:0] r8_15, r8_15_r;
707  reg[31:0] r4_17, r4_17_r;
708  reg[31:0] r11_20;
709  reg[31:0] r4_22, r4_22_r;
710  reg[31:0] r11_25, r11_25_r;
711  reg[31:0] r5_29, r5_29_r;
712  // declarations for memory address temp variables

713  // declarations for transaction model support
714  reg tran_state_done;
715  reg [31:0] virpc_tr0, virpc_tr1, virpc_tr2;
716  reg [7:1] tran_end_state_r;
717  reg transaction_end_this_state;

718  //
719  // INFO: reading from m2v_state_mc_selfloop.v
720  //
721  // m2v_state_mc.v
722  //
723  // Karl Meier
724  // 8/15/07
725  //
726  // invariant state machine logic for the read, write, and branch state
727  // machines
728  //

729  reg [1:0] pclk_del_r;
730  reg pclk_rise, pclk_fall;
731  reg en_r, sll128_r, gr_r, regrdy_r, regfull_r, regempty_r, extnop_ma_r;
732  reg clr_dne, DNE_c, ACK_c;
733  reg done_state, done_state_r;
734  reg wsm_idle, wsm_idle_r, wsm_pulse, wsm_pulse_r, wsm_wait, wsm_wait_r;
735  reg write_this_state, wsm_done;
736  reg rsm_idle, rsm_idle_r, rsm_latch, rsm_latch_r;
737  reg rsm_wait, rsm_wait_r, rsm_wait2, rsm_wait2_r;
738  reg [3:0] rsm_count, rsm_count_r;
739  reg read_this_state, rsm_done;
740  reg bsm_idle, bsm_idle_r, bsm_calc, bsm_calc_r;
741  reg bsm_wait, bsm_wait_r, bsm_waitpf, bsm_waitpf_r;
742  reg bsm_waitpr, bsm_waitpr_r;
743  reg branch_this_state, bsm_done;
744  reg fsm_idle, fsm_idle_r, fsm_wait2, fsm_wait2_r, fsm_wait, fsm_wait_r;
745  reg final_state, fsm_done;
746  reg take_branch, take_branch_r;

747  reg [1:0] mdata_vld_r;
748  reg mdata_vld_rise, mdata_vld_fall;
749  reg mem_read_this_state, mrs_done;
750  reg mem_write_this_state, mws_done;
751  reg [1:0] mem_this_state, mdne, mdne_c;

752  // State machine logic for MMU access instructions
753  // memory access control for the extension
754  always @ (*) begin
```

```
755  mdata_vld_rise = (mdata_vld_r == 2'b01);
756  mdata_vld_fall = (mdata_vld_r == 2'b10);

757  // Place memory read/write request on rising edge of PCLK
758  casex ({RESET, mem_this_state})
759  3'b0xx : begin
760  // Reset stage
761  RNL_OUT <= 1'b0;
762  BLS_OUT <= 1'b0;
763  HLS_OUT <= 1'b0;
764  MOE_OUT <= 1'b0;
765  MWE_OUT <= 1'b0;

766  mdata_in <= 32'b0;
767  mem_this_state <= 2'b00;
768  mdne_c <= 0;
769  end
770  3'b100 : begin
771  casex ({mem_read_this_state, mem_write_this_state})
772  2'b10 : begin
773  if (pclk_del_r == 2'b01) begin
774  /* Memory read state */
775  RNL_OUT <= 1'b0;
776  BLS_OUT <= 1'b0;
777  HLS_OUT <= 1'b0;
778  MOE_OUT <= 1'b1;
779  MWE_OUT <= 1'b0;

780  mem_this_state <= 2'b01;   // next state for read operation
781  end
782  end

783  2'b01 : begin
784  if (pclk_del_r == 2'b01) begin
785  /* Memory write state */
786  RNL_OUT <= 1'b0;
787  BLS_OUT <= 1'b0;
788  HLS_OUT <= 1'b0;
789  MOE_OUT <= 1'b0;
790  MWE_OUT <= 1'b1;

791  mem_this_state <= 2'b10;
792  end
793  end

794  default : begin
795  /* No memory access this state */
796  RNL_OUT <= 1'b0;
797  BLS_OUT <= 1'b0;
798  HLS_OUT <= 1'b0;
799  MOE_OUT <= 1'b0;
800  MWE_OUT <= 1'b0;

801  mdata_in <= 32'b0;
802  mem_this_state <= 2'b00;
803  end
804  endcase
805  end
806  3'b101 : begin
807  // Remove memory read request after the falling edge of MDATA_VLD_IN
808  if (mdata_vld_fall) begin
809  MOE_OUT <= 1'b0;
810  end
811  if(~MOE_OUT & mdata_vld_rise) begin // Look for MDATA_VLD_IN signal only after initiating the request
812  // and after the falling edge of PCLK
813  mdata_in <= MDATA_IN;  // Assign the input data to the result register

814  // Latch the memory done signal (MDATA_VLD_IN)
815  mdne_c <= 1;
```

```
816  mem_this_state <= 2'b00;
817  end
818  end
819  3'b110 : begin
820  // Remove memory write request after the falling edge of MDATA_VLD_IN
821  if (mdata_vld_fall) begin
822  MWE_OUT <= 1'b0;
823  end
824  if(~MWE_OUT & mdata_vld_rise) begin// Look for MDATA_VLD_IN signal only after initiating the request
825  // and after the falling edge of PCLK

826  // Latch the memory done signal (MDATA_VLD_IN)
827  mdne_c <= 1;
828  mem_this_state <= 2'b00;
829  end
830  end
831  default : begin
832  end
833  endcase
834  end

835  // state machine logic for compiled extension
836  always @ (*) begin
837  pclk_rise = (pclk_del_r == 2'b01);
838  pclk_fall = (pclk_del_r == 2'b10);

839  // start the extension instruction
840  clr_dne = state_r[1] & en_r & sll128_r;

841  // state machine for read logic
842  read_this_state = (| (state_r & read_state_r));
843  rsm_wait = read_this_state &
844  (rsm_idle_r & gr_r) |
845  (rsm_wait_r & (rsm_count_r != REG_READ_WAIT_STATES));
846  rsm_latch = rsm_wait_r & (rsm_count_r == REG_READ_WAIT_STATES);
847  rsm_wait2 = (rsm_wait2_r | rsm_latch_r) & ~done_state;
848  rsm_idle = ~rsm_wait & ~rsm_wait2 & ~rsm_latch;
849  rsm_count = rsm_idle_r ? 4'h0 : (rsm_count_r + 1);
850  rsm_done = ~read_this_state |
851  (read_this_state & (rsm_latch_r | rsm_wait2_r));

852  // state machine for write logic
853  write_this_state = (| (state_r & write_state_r));
854  wsm_pulse = wsm_idle_r & write_this_state & gr_r & regrdy_r & ~regfull_r;
855  wsm_wait = (wsm_pulse_r & ~done_state) |
856  (wsm_wait_r  & ~done_state);
857  wsm_idle = ~wsm_pulse & ~wsm_wait;
858  wsm_done = ~write_this_state |
859  (write_this_state & (wsm_pulse_r | wsm_wait_r));

860  // state machine for Memory read logic
861  mem_read_this_state = (| (state_r & mem_read_state_r)) & ~mdne;
862  mrs_done = ~mem_read_this_state |
863  (mem_read_this_state & mdne);

864  // state machine for Memory write logic
865  mem_write_this_state = (| (state_r & mem_write_state_r)) & ~mdne;
866  mws_done = ~mem_write_this_state |
867  (mem_write_this_state & mdne);

868  // state machine for branch logic
869  branch_this_state = (| (state_r & branch_state_r));
870  bsm_calc = bsm_idle_r & branch_this_state;
871  bsm_waitpf = (bsm_calc_r & take_branch_r) |
872  (bsm_waitpf_r & ~pclk_fall);
873  bsm_waitpr = (bsm_waitpf_r & pclk_fall) |
874  (bsm_waitpr_r & ~pclk_rise);
875  bsm_wait = (bsm_calc_r & ~take_branch_r & ~done_state) |
876  (bsm_waitpr_r & pclk_rise & ~done_state) |
```

```
877  (bsm_wait_r & ~done_state);
878  bsm_idle = ~bsm_calc & ~bsm_wait & ~bsm_waitpr & ~bsm_waitpf;
879  bsm_done = ~branch_this_state |
880  (branch_this_state &
881  ((bsm_calc_r & ~take_branch_r) |
882  (bsm_waitpr_r & pclk_rise) |
883  bsm_wait_r));

884  // state machine to finish up the extension instruction
885  final_state = state_r[MAX_STATE];
886  fsm_wait = final_state & rsm_idle_r |
887  (fsm_wait_r & ~(gr_r & regempty_r & extnop_ma_r));
888  fsm_wait2 = (fsm_wait_r & gr_r & regempty_r & extnop_ma_r) |
889  (fsm_wait2_r & ~en_r);
890  fsm_idle = ~fsm_wait & ~fsm_wait2;
891  fsm_done = final_state & fsm_wait2_r & ~en_r;

892  // clear DNE as the extension instruction is entered
893  // set DNE as the extension instruction is exited
894  //ruirui: make DNE keep down until the loop is over
895  DNE_c = (DNE | (fsm_wait_r & gr_r & regempty_r & extnop_ma_r & ~take_branch_r)) & ~clr_dne;
896  ACK_c = ((ACK | (~DNE & ~ACK)) & ~(ACK & DNE & pclk_rise) & EN) | take_branch_r;
897  // & EN -> to bring down ACK when it loses control/resources in case of an interrupt
898  end

899  always @ (*) begin
900  // true when all conditions for a state have been satisfied
901  done_state = clr_dne |
902  (~state_r[1] & bsm_done & rsm_done & wsm_done & mrs_done & mws_done & tran_state_done);
903  end


904  // state to determine rising and falling edges of pclk
905  always @ (posedge CLK) begin
906  pclk_del_r <= {pclk_del_r[0], PCLK};
907  // rise and falling edge of MDATA_VLD_IN
908  mdata_vld_r <= {mdata_vld_r[0], MDATA_VLD_IN};
909  end

910  // buffer signals that may be heavily loaded or come from a distance
911  //   - is this needed?  this is present to maintain compatibility with Neil
912  always @ (posedge CLK) begin
913  if (!RESET) begin
914  en_r <= 1'h0;
915  sll128_r <= 1'h0;
916  gr_r <= 1'h0;
917  regrdy_r <= 1'h0;
918  regfull_r <= 1'h0;
919  regempty_r <= 1'h0;
920  extnop_ma_r <= 1'h0;
921  end else begin
922  en_r <= EN;
923  sll128_r <= SLL128;
924  gr_r <= GR;
925  regrdy_r <= REGRDY;
926  regfull_r <= REGFULL;
927  regempty_r <= REGEMPTY;
928  extnop_ma_r <= EXTNOP_MA;
929  end
930  end

931  // misc control for the extension
932  always @ (posedge CLK) begin
933  if (!RESET) begin
934  ACK <= 1'h0;
935  DNE <= 1'h1;
936  done_state_r <= 1'b0;

937  wsm_idle_r <= 1'b1;
```

```
938  wsm_pulse_r <= 1'b0;
939  wsm_wait_r <= 1'b0;

940  rsm_idle_r <= 1'b1;
941  rsm_latch_r <= 1'b0;
942  rsm_wait_r <= 1'b0;
943  rsm_wait2_r <= 1'b0;
944  rsm_count_r <= 4'b0;

945  bsm_idle_r <= 1'b1;
946  bsm_calc_r <= 1'b0;
947  bsm_wait_r <= 1'b0;
948  bsm_waitpr_r <= 1'b0;
949  bsm_waitpf_r <= 1'b0;
950  take_branch_r <= 1'h0;

951  fsm_idle_r <= 1'b1;
952  fsm_wait_r <= 1'b0;
953  fsm_wait2_r <= 1'b0;

954  end else begin
955  /*
956  // clear ack
957  if (ACK & DNE & pclk_rise)
958  ACK <= 1'h0;
959  // set ack
960  else if (~DNE & ~ACK)
961  ACK <= 1'h1;
962  */

963  ACK <= ACK_c;
964  DNE <= DNE_c;
965  done_state_r <= done_state;

966  wsm_idle_r <= wsm_idle;
967  wsm_pulse_r <= wsm_pulse;
968  wsm_wait_r <= wsm_wait;

969  rsm_idle_r <= rsm_idle;
970  rsm_latch_r <= rsm_latch;
971  rsm_wait_r <= rsm_wait;
972  rsm_wait2_r <= rsm_wait2;
973  rsm_count_r <= rsm_count;

974  bsm_idle_r <= bsm_idle;
975  bsm_calc_r <= bsm_calc;
976  bsm_wait_r <= bsm_wait;
977  bsm_waitpr_r <= bsm_waitpr;
978  bsm_waitpf_r <= bsm_waitpf;
979  // if take_branch_r is ever used outside of the branch state machine,
980  // it may need to be cleared at the end of the branch operation
981  take_branch_r <= bsm_calc ? take_branch : take_branch_r;

982  fsm_idle_r <= fsm_idle;
983  fsm_wait_r <= fsm_wait;
984  fsm_wait2_r <= fsm_wait2;
985  end
986  end

987  //
988  // INFO: finished reading from m2v_state_mc_selfloop.v
989  //

990  // state machine for transaction model
991  always @ (*) begin
992  transaction_end_this_state = (| (state_r & tran_end_state_r));

993  virpc_tr0 = PC;
994  virpc_tr1 = PC + 28;
```

```
995   virpc_tr2 = CJMPADD;

996   VIRPC =  ({32{state_r[1]}} & virpc_tr0)
997   | ({32{state_r[2]}} & virpc_tr0)
998   | ({32{state_r[3]}} & virpc_tr1)
999   | ({32{state_r[4]}} & virpc_tr1)
1000 | ({32{state_r[5]}} & virpc_tr1)
1001 | ({32{state_r[6]}} & virpc_tr2)
1002 | ({32{state_r[7]}} & virpc_tr2);
1003 end

1004 // transaction model control for the extension
1005 always @ (posedge CLK) begin
1006 if (!RESET) begin
1007 tran_state_done <= 1;
1008 end else begin
1009 if (EXCEXT & EN) begin
1010 if (transaction_end_this_state) begin
1011 tran_state_done <= 0;
1012 end
1013 end else begin
1014 if (~EN) begin
1015 tran_state_done <= 1;
1016 end
1017 end
1018 end
1019 end


1020 // registers that contain state about this cycle
1021 always @ (posedge CLK) begin
1022 if (~RESET) begin
1023 branch_state_r[1] <= 1'b0;
1024 write_state_r[1]  <= 1'b0;
1025 read_state_r[1]   <= 1'b1;
1026 mem_write_state_r[1]  <= 1'b0;
1027 mem_read_state_r[1]   <= 1'b0;
1028 tran_end_state_r[1]   <= 1'b1;

1029 branch_state_r[2] <= 1'b0;
1030 write_state_r[2]  <= 1'b0;
1031 read_state_r[2]   <= 1'b1;
1032 mem_write_state_r[2]  <= 1'b0;
1033 mem_read_state_r[2]   <= 1'b0;
1034 tran_end_state_r[2]   <= 1'b1;

1035 branch_state_r[3] <= 1'b1;
1036 write_state_r[3]  <= 1'b1;
1037 read_state_r[3]   <= 1'b1;
1038 mem_write_state_r[3]  <= 1'b0;
1039 mem_read_state_r[3]   <= 1'b0;
1040 tran_end_state_r[3]   <= 1'b0;

1041 branch_state_r[4] <= 1'b0;
1042 write_state_r[4]  <= 1'b1;
1043 read_state_r[4]   <= 1'b0;
1044 mem_write_state_r[4]  <= 1'b0;
1045 mem_read_state_r[4]   <= 1'b0;
1046 tran_end_state_r[4]   <= 1'b0;

1047 branch_state_r[5] <= 1'b0;
1048 write_state_r[5]  <= 1'b1;
1049 read_state_r[5]   <= 1'b0;
1050 mem_write_state_r[5]  <= 1'b0;
1051 mem_read_state_r[5]   <= 1'b0;
1052 tran_end_state_r[5]   <= 1'b1;

1053 branch_state_r[6] <= 1'b0;
1054 write_state_r[6]  <= 1'b1;
```

```
1055 read_state_r[6]   <= 1'b0;
1056 mem_write_state_r[6]  <= 1'b0;
1057 mem_read_state_r[6]   <= 1'b0;
1058 tran_end_state_r[6]   <= 1'b0;

1059 branch_state_r[7] <= 1'b0;
1060 write_state_r[7]  <= 1'b1;
1061 read_state_r[7]   <= 1'b0;
1062 mem_write_state_r[7]  <= 1'b0;
1063 mem_read_state_r[7]   <= 1'b0;
1064 tran_end_state_r[7]   <= 1'b1;

1065 end else begin
1066 branch_state_r <= branch_state_r;
1067 write_state_r  <= write_state_r;
1068 read_state_r   <= read_state_r;
1069 mem_write_state_r  <= mem_write_state_r;
1070 mem_read_state_r   <= mem_read_state_r;
1071 tran_end_state_r   <= tran_end_state_r;
1072 end
1073 end


1074 // combinatorial logic to/from the register file
1075 always @ (*) begin
1076 // combinatorial logic for register reads
1077 // use read ports 3 & 4 to prevent write conflicts
1078 RDREG1 = 0;
1079 RDREG2 = 0;
1080 r9_1 =  take_branch_r? r9_8_r : RDREG3DATA;
1081 r8_4 =  take_branch_r? r8_15_r : RDREG2DATA_ID;
1082 r4_11 = take_branch_r? r4_22_r : RDREG1DATA_ID;
1083 r5_18 = take_branch_r? r5_29_r : RDREG4DATA;
1084 r6_23 =  RDREG3DATA;
1085 RDREG3 =  ({5{state_r[2]}} & (RT + 1))
1086 | ({5{state_r[1]}} & RT)
1087 | ({5{state_r[3]}} & (RS + 2));
1088 RDREG4 =  ({5{state_r[1]}} & RS)
1089 | ({5{state_r[2]}} & (RS + 1));

1090 // combinatorial logic for register writes
1091 WRREG1 =  ({5{state_r[3]}} & 11)
1092 | ({5{state_r[4]}} & RT)
1093 | ({5{state_r[5]}} & (RT + 1))
1094 | ({5{state_r[6]}} & RS)
1095 | ({5{state_r[7]}} & 11);
1096 WRDATA1 = ({32{state_r[3]}} & r11_13_r)
1097 | ({32{state_r[4]}} & r8_15_r)
1098 | ({32{state_r[5]}} & r9_8_r)
1099 | ({32{state_r[6]}} & r4_22_r)
1100 | ({32{state_r[7]}} & r11_25_r);
1101 REGWRITE1 = wsm_pulse_r & (state_r[3]
1102 | state_r[4]
1103 | state_r[5]
1104 | state_r[6]
1105 | state_r[7]);
1106 WRREG2 =  ({5{state_r[4]}} & RS)
1107 | ({5{state_r[6]}} & (RS + 1));
1108 WRDATA2 = ({32{state_r[4]}} & r4_17_r)
1109 | ({32{state_r[6]}} & r5_29_r);
1110 REGWRITE2 = wsm_pulse_r & (state_r[4]
1111 | state_r[6]);
1112 end

1113 // internal pipeline logic
1114 always @ (posedge CLK) begin
1115 if (~RESET) begin
1116 r8_4_r <= 32'h0;
1117 r4_11_r <= 32'h0;
```

```
1118 r9_8_r <= 32'h0;
1119 r11_13_r <= 32'h0;
1120 r8_15_r <= 32'h0;
1121 r4_17_r <= 32'h0;
1122 r4_22_r <= 32'h0;
1123 r11_25_r <= 32'h0;
1124 r5_29_r <= 32'h0;
1125 end else begin
1126 r8_4_r <=  state_r[1] ? r8_4 : r8_4_r;
1127 r4_11_r <=  state_r[1] ? r4_11 : r4_11_r;
1128 r9_8_r <=  state_r[2] ? r9_8 : r9_8_r;
1129 r11_13_r <=  state_r[2] ? r11_13 : r11_13_r;
1130 r8_15_r <=  state_r[2] ? r8_15 : r8_15_r;
1131 r4_17_r <=  state_r[2] ? r4_17 : r4_17_r;
1132 r4_22_r <=  state_r[2] ? r4_22 : r4_22_r;
1133 r11_25_r <=  state_r[3] ? r11_25 : r11_25_r;
1134 r5_29_r <=  state_r[2] ? r5_29 : r5_29_r;
1135 end
1136 end


1137 // logic for the Memory address and data out
1138 always @ (posedge CLK) begin
1139 MADDR_OUT <=  (32'h0);
1140 MDATA_OUT <= (32'h0);
1141 end

1142 // combinatorial logic for the instruction nodes
1143 always @ (*) begin
1144 // [0x0]0x10840 sll r9, r9, 1
1145 r9_3 = r9_1 << 1;

1146 // [0x4]0x21fc2  srl r11, r8, 31
1147 r11_6 = r8_4_r >> 31;

1148 // [0x8]0x230825    or  r9, r9, r11
1149 r9_8 = r9_3 | r11_6;

1150 // [0xc]0x21040 sll r8, r8, 1
1151 r8_10 = r8_4_r << 1;

1152 // [0x10]   0x41fc2  srl r11, r4, 31
1153 r11_13 = r4_11_r >> 31;

1154 // [0x14]   0x431025    or  r8, r8, r11
1155 r8_15 = r8_10 | r11_13;

1156 // [0x18]   0x42040 sll r4, r4, 1
1157 r4_17 = r4_11_r << 1;

1158 // [0x1c]   0x51fc2  srl r11, r5, 31
1159 r11_20 = r5_18 >> 31;

1160 // [0x20]   0x832025    or  r4, r4, r11
1161 r4_22 = r4_17 | r11_20;

1162 // [0x24]   0x26182b    sltu    r11, r9, r6
1163 r11_25 = ({1'b0, r9_8_r} < {1'b0, r6_23}) ? 1 : 0;

1164 // [0x28]   0x10030005     beq     r0, r11, 20
1165 take_branch = (32'h0 == r11_25);
1166 CJMPADD = take_branch ? (PC + 4 + {{16{DIMM[15]}},DIMM}) : PC;
1167 PCNEXT = state_r[3] & bsm_waitpr & ~take_branch;

1168 // [0x2c]   0x52840 sll r5, r5, 1
1169 r5_29 = r5_18 << 1;

1170 end
```

```
1171 // primary extension state machine
1172 always @ (posedge CLK) begin
1173 if (~RESET) begin
1174 state_r <= 1;
1175 mdne <= 0;
1176 end else begin
1177 mdne <= mdne_c;
1178 if (en_r) begin
1179 if (done_state) begin
1180 if ((state_r == 8'b1000_0000) & take_branch_r) begin
1181 state_r <= 8'b0000_0001;
1182 end else begin
1183 state_r <= {state_r[MAX_STATE-1:1], 1'b0};
1184 mdne <= 0;
1185 end
1186 end
1187 end
1188 else begin
1189 state_r <= 1;
1190 end
1191 end
1192 end

1193 endmodule
```

## Appendix II – Verilog Output For The Branch Case Of Multiple Blocks

```
1   // block78and79and80.v
2   // auto-generated by m2v revision 1 on Sun Jul 26 02:21:09 2009
3   // MIPS to Verilog (m2v) module (_mod) boilerplate (_bp)

4   // Copyright (c) Microsoft Corporation. All rights reserved.

5   `timescale 1ns / 1ps

6   module mmlite_div64(
7   //
8   // INFO: reading from m2v_mod_bp_ports.v
9   //
10  /*****Ports*************************************************************/
11  /* INPUT PORTS */
12  input             CLK,              /* System Clock 50 - 100 MHZ */
13  input             EN,               /* Enable */
14  input             EXCEXT,             /* Exception Flush */
15  input             EXTNOP_MA,            /* Extension Bubble in Memory Access Phase */
16  input             GR,               /* Grant Pipeline Resources */
17  input [31:0]  INSTR,            /* Current Instruction */
18  input [31:0]  PC,               /* Current PC External */
19  input             PCLK,               /* Pipeline Clock */
20  input [31:0]  RDREG1DATA,         /* Register Read Port 1 Register Data */
21  input [31:0]  RDREG2DATA,         /* Register Read Port 2 Register Data */
22  input [31:0]  RDREG3DATA,         /* Register Read Port 3 Register Data */
23  input [31:0]  RDREG4DATA,         /* Register Read Port 4 Register Data */
24  input             REGEMPTY,         /* Register Write Buffer Empty */
25  input             REGFULL,          /* Register Write Buffer Full */
26  input             REGRDY,           /* Register Write Buffer Ready */
27  input             RESET,            /* System Reset */
28  input [31:0]  MDATA_IN,       /* Memory Data In */
                                  1.  /* Multiplexed: */
                                  2.  /*    Memory Data In */
                                  3.  /*    Peripheral Memory Data In */
                                  4.  /*    Memory Data Monitor */
29  input         MDATA_VLD_IN,   /* Memory Data Valid */

30  /* OUTPUT PORTS */
31  output        ACK,              /* Enable Acknowledged */
32  output [31:0] EXTADD,             /* Extension Address */
                                  1.  /* Multiplexed: */
                                  2.  /*    Next PC */
                                  3.  /*    Exception Address */
                                  4.  /*    PC Memory Access Phase */
33  output        PCNEXT,             /* Conditional PC Update */
34  output [4:0]  RDREG1,             /* Register Read Port 1 Register Number */
                                  1.  /* Multiplexed: */
                                  2.  /*    Register Read Port 1 Register Number */
                                  3.  /*    Register Write Port 1 Register Number */
                                  4.  /*    Write Register Memory Access Phase */
35  output [4:0]  RDREG2,             /* Register Read Port 2 Register Number */
                                  1.  /* Multiplexed: */
                                  2.  /*    Register Read Port 2 Register Number */
                                  3.  /*    Register Write Port 2 Register Number */
                                  4.  /*        <0> Register Write Enable Memory Access Phase */
                                  5.  /*    <1> Memory to Register Memory Acess Phase */
36  output [4:0]  RDREG3,             /* Register Read Port 3 Register Number */
                                  1.  /* Multiplexed: */
                                  2.  /*    Register Read Port 3 Register Number */
37  output [4:0]  RDREG4,             /* Register Read Port 4 Register Number Internal */
                                  1.  /* Multiplexed: */
                                  2.  /*    Register Read Port 4 Register Number */
                                  3.  /*    <1:0> Data Address [1:0] Memory Access Phase */
                                  4.  /*        <2> Right/Left Unaligned Load/Store Memory Access Phase */
                                  5.  /*    <3> Byte/Halfword Load/Store Memory Acess Phase */
38  output        REGWRITE1,          /* Register Write Port 1 Write Enable */
```

```
39  output          REGWRITE2,          /* Register Write Port 2 Write Enable */
40  output          REWB,               /* Re-enter at Writeback */
41  output          RI,                 /* Reserved/Recognized Instruction */
42  output [31:0] WRDATA1,              /* Register Write Port 1 Data Internal */
                                    1.  /* Multiplexed: */
                                    2.  /*    Register Write Port 1 Data */
                                    3.  /*    ALU Result Memory Access Phase */
43  output [31:0] WRDATA2,              /* Register Write Port 2 Data Internal */
                                    1.  /* Multiplexed: */
                                    2.  /*    Register Write Port 2 Data */
                                    3.  /*    Memory Data Out Memory Access Phase */
44  output          BLS_OUT,            /* Byte Load/Store */
45  output      HLS_OUT,            /* Halfword Load/Store */
46  output          RNL_OUT,            /* Memory Right/Left Unaligned Load/Store */
47  output [31:0] MADDR_OUT,          /* Memory Address */
48  output [31:0] MDATA_OUT,          /* Memory Data Out */
                                    1.  /* Multiplexed: */
                                    2.  /*    Memory Data Out */
                                    3.  /*    Peripheral Memory Data Out */
49  output          MOE_OUT,            /* Memory Output Enable */
50  output          MWE_OUT         /* Memory Write Enable */
51  );

52  /*****Signals*********************************************************************/

53  wire [31:0]   ALURESULT_WB;     /* ALU Result to Writeback Phase */
54  wire          BHLS_WB;          /* Byte/Halfword Load/Store to Writeback Phase */
55  wire [31:0]   CJMPADD;          /* Conditional Jump address to offset from Current PC */
56  wire [15:0]   DIMM_EX;          /* Data Immediate Execute Phase */
57  wire [15:0]   DIMM_ID;          /* Data Immediate Instruction Decode Phase */
58  wire [1:0] DMADD_WB;        /* Least Significant Bits of Data Address to Writeback Phase */
59  wire [31:0]   DMDATAOUT_WB;       /* Memory Data Out to Writeback Phase */
60  wire          DNE;              /* Execution Done */
61  wire          EN_EX;            /* Enable Execute Phase */
62  wire [31:0]   JMPADD;           /* Jump address to end of basic block */
63  wire          MEMTOREG_WB;      /* Memory to Register to Writeback Phase */
64  wire [31:0]   PC_EX;            /* PC Execute Phase */
65  wire [31:0]   PC_WB;            /* PC to Writeback Phase */
66  wire [4:0] RD_EX;           /* Destination Register Execution Phase */
67  wire [4:0] RDREG1_EX;       /* Register Read Port 1 Register Number Execute Phase */
68  wire [31:0]   RDREG1DATA_EX;      /* Register Read Port 1 Register Data Execute Phase */
69  wire [4:0] RDREG2_EX;       /* Register Read Port 2 Register Number Execute Phase */
70  wire [31:0]   RDREG2DATA_EX;      /* Register Read Port 2 Register Data Execute Phase */
71  wire [4:0] RDREG3_EX;       /* Register Read Port 3 Register Number Execute Phase */
72  wire [4:0] RDREG4_EX;       /* Register Read Port 4 Register Number Execute Phase */
73  wire          REGWRITE_EX;      /* Register Write Execute Phase */
74  wire          REGWRITE_ID;      /* Register Write Instruction Decode Phase */
75  wire          REGWRITE_WB;      /* Register Write to Writeback Phase */
76  wire          RESET_EX;         /* Reset Execute Phase */
77  wire [31:0]   RESULT_EX;        /* Result Execution Phase */
78  wire          RNL_WB;           /* Right/Left Unaligned Load/Store to Writeback Phase */
79  wire [4:0] RS_EX;           /* Operand Register 1 Execute Phase */
80  wire [4:0] RS_ID;           /* Operand Register 1 Instruction Decode Phase */
81  wire [4:0] RT_EX;           /* Operand Register 2 Execute Phase */
82  wire [4:0] RT_ID;           /* Operand Register 2 Instruction Decode Phase */
83  wire          SLL128_EX;        /* Shift Left Logical 128 bits Execute Phase */
84  wire          SLL128_ID;        /* Shift Left Logical 128 bits Instruction Decode Phase */
85  wire [31:0]   WRDATA1_EX;       /* Register Write Port 1 Data Execute Phase */
86  wire [31:0]   WRDATA2_EX;       /* Register Write Port 2 Data Execute Phase */
87  wire [4:0] WRREG_WB;        /* Write Register Number to Writeback Phase */
88  wire [4:0] WRREG1_EX;       /* Register Write Port 1 Register Number Execute Phase */
89  wire [4:0] WRREG2_EX;       /* Register Write Port 2 Register Number Execute Phase */
90  wire [31:0]   VIRPC;

91  /*****Registers*******************************************************************/

92  reg en_reg;   /* Enable */
93  reg gr_reg;   /* Grant Pipeline Resources */
```

```
94  /*****Initialization*****************************************************************/
95  /*
96  initial
97  begin
98  en_reg = 1'b0;
99  gr_reg = 1'b0;
100     end
101     */

102     /************************************************************************/

103     assign EXTADD  =  (EXCEXT)?                 VIRPC:
                    i.  (en_reg)?               JMPADD:
                   ii.  (PCNEXT)?                 CJMPADD:
                  iii.  (REWB)?                   PC_WB:
                                          a.    32'hffffffff;
104     /*
105     The rest cannot be zero'ed out as the extension state machine
106     might still have to be completed till a particular transaction ends
107     */
108     assign RDREG1  =  (gr_reg & REGWRITE1)?  WRREG1_EX:
                    i.  //(EXCEXT)?                5'b0:
                   ii.  (REWB & gr_reg)?          WRREG_WB:
                  iii.  (gr_reg)?                 RDREG1_EX:
                                          a.    5'b11111;
109     assign RDREG2  =  (gr_reg & REGWRITE2)?  WRREG2_EX:
                    i.  //(EXCEXT)?                5'b0:
                   ii.  (REWB & gr_reg)?          {3'b0,MEMTOREG_WB,REGWRITE_WB}:
                  iii.  (gr_reg)?                 RDREG2_EX:
                                          a.    5'b11111;
110     assign RDREG3  =  (REWB & gr_reg)?          5'b0:
                    i.  //(EXCEXT)?                5'b0:
                   ii.  (gr_reg)?                 RDREG3_EX:
                                          a.    5'b11111;
111     assign RDREG4  =  (REWB & gr_reg)?          {1'b0,BHLS_WB,RNL_WB,DMADD_WB}:
                    i.  //(EXCEXT)?                5'b0:
                   ii.  (gr_reg)?                 RDREG4_EX:
                                          a.    5'b11111;
112     assign WRDATA1 = (gr_reg & REGWRITE1)?  WRDATA1_EX:
                    i.  //(EXCEXT)?                32'b0:
                   ii.  (REWB)?                   ALURESULT_WB:
                                                    i.  32'hffffffff;
113     assign WRDATA2 = (gr_reg & REGWRITE2)?  WRDATA2_EX:
                    i.  //(EXCEXT)?                32'b0:
                   ii.  (REWB)?                   DMDATAOUT_WB:
                                                    i.  32'hffffffff;

114     //
115     // INFO: finished reading from m2v_mod_bp_ports.v
116     //

117     //
118     // instantiate the instruction decode module for the extension instruction
119     //   - the instruction decode module is auto generated and appended to the
120     //     end of the verilog file (a.v unless redefined)
121     //
122     block78and79and80_ext_id id (
123     //
124     // INFO: reading from instantiate_ext_id.v
125     //
126     .CLK(CLK),
127     .DIMM(DIMM_ID),
128     .EN(EN),
129     .JMPADD(JMPADD),
130     .INSTR(INSTR),
131     .PC(PC),
132     .REGWRITE(REGWRITE_ID),
133     .RESET(RESET),
134     .RI(RI),
```

```
135     .RS(RS_ID),
136     .RT(RT_ID),
137     .SLL128(SLL128_ID)
138     );
139     //
140     // INFO: finished reading from instantiate_ext_id.v
141     //

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

143     block78and79and80_toex to_ex (
144     //
145     // INFO: reading from instantiate_to_ex.v
146     //
147     .ACK(ACK),
148     .CLK(CLK),
149     .DIMM_EX(DIMM_EX),
150     .DIMM_ID(DIMM_ID),
151     .EN_EX(EN_EX),
152     .EN_ID(EN),
153     .EXCEXT(EXCEXT),
154     .PC_EX(PC_EX),
155     .PC_ID(PC),
156     .PCLK(PCLK),
157     .RDREG1DATA_EX(RDREG1DATA_EX),
158     .RDREG1DATA_ID(RDREG1DATA),
159     .RDREG2DATA_EX(RDREG2DATA_EX),
160     .RDREG2DATA_ID(RDREG2DATA),
161     .REGWRITE_EX(REGWRITE_EX),
162     .REGWRITE_ID(REGWRITE_ID),
163     .RESET(RESET),
164     .RESET_EX(RESET_EX),
165     .RS_EX(RS_EX),
166     .RS_ID(RS_ID),
167     .RT_EX(RT_EX),
168     .RT_ID(RT_ID),
169     .SLL128_ID(SLL128_ID),
170     .SLL128_EX(SLL128_EX)
171     );

172     //
173     // INFO: finished reading from instantiate_to_ex.v
174     //


175     //
176     // instantiate the execution module for the extension instruction
177     //   - the execution module is auto generated and appended to the
178     //     end of the verilog file (a.v unless redefined)
179     //
180     block78and79and80_ext_ex ex(
181     //
182     // INFO: reading from instantiate_ex.v
183     //
184     .ACK(ACK),
185     .DIMM(DIMM_EX),
186     .DNE(DNE),
187     .CLK(CLK),
188     .CJMPADD(CJMPADD),
189     .EN(EN_EX),
190     .EXTNOP_MA(EXTNOP_MA),
191     .GR(GR),
192     .PC(PC_EX),
193     .PCLK(PCLK),
194     .PCNEXT(PCNEXT),
195     .RD(RD_EX),
196     .RDREG1(RDREG1_EX),
197     .RDREG1DATA(RDREG1DATA),
198     .RDREG1DATA_ID(RDREG1DATA_EX),
```

```
199     .RDREG2(RDREG2_EX),
200     .RDREG2DATA(RDREG2DATA),
201     .RDREG2DATA_ID(RDREG2DATA_EX),
202     .RDREG3(RDREG3_EX),
203     .RDREG3DATA(RDREG3DATA),
204     .RDREG4(RDREG4_EX),
205     .RDREG4DATA(RDREG4DATA),
206     .REGEMPTY(REGEMPTY),
207     .REGFULL(REGFULL),
208     .REGRDY(REGRDY),
209     .REGWRITE1(REGWRITE1),
210     .REGWRITE2(REGWRITE2),
211     .RESET(RESET_EX),
212     .RESULT(RESULT_EX),
213     .RS(RS_EX),
214     .RT(RT_EX),
215     .SLL128(SLL128_EX),
216     .WRDATA1(WRDATA1_EX),
217     .WRDATA2(WRDATA2_EX),
218     .WRREG1(WRREG1_EX),
219     .WRREG2(WRREG2_EX),
220     .MDATA_IN(MDATA_IN),
221     .MDATA_VLD_IN(MDATA_VLD_IN),
222     .BLS_OUT(BLS_OUT),
223     .HLS_OUT(HLS_OUT),
224     .RNL_OUT(RNL_OUT),
225     .MOE_OUT(MOE_OUT),
226     .MWE_OUT(MWE_OUT),
227     .MADDR_OUT(MADDR_OUT),
228     .MDATA_OUT(MDATA_OUT),
229     .EXCEXT(EXCEXT),
230     .VIRPC(VIRPC)
231     );
232     //
233     // INFO: finished reading from instantiate_ex.v
234     //


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

236     block78and79and80_topipe_wb to_wb(
237     //
238     // INFO: reading from instantiate_wb.v
239     //
240     .ACK(ACK),
241     .ALURESULT_WB(ALURESULT_WB),
242     .BHLS_WB(BHLS_WB),
243     .CLK(CLK),
244     .DMADD_WB(DMADD_WB),
245     .DMDATAOUT_WB(DMDATAOUT_WB),
246     .DNE(DNE),
247     .EN_EX(EN_EX),
248     .EXCEXT(EXCEXT),
249     .EXTNOP_MA(EXTNOP_MA),
250     .PC_EX(PC_EX),
251     .PC_WB(PC_WB),
252     .PCLK(PCLK),
253     .MEMTOREG_WB(MEMTOREG_WB),
254     .RD_EX(RD_EX),
255     .REGWRITE_EX(REGWRITE_EX),
256     .REGWRITE_WB(REGWRITE_WB),
257     .RESET(RESET),
258     .RESULT_EX(RESULT_EX),
259     .REWB(REWB),
260     .RNL_WB(RNL_WB),
261     .WRREG_WB(WRREG_WB)
262     );
263     //
264     // INFO: finished reading from instantiate_wb.v
```

```
265     //


266     /*********************************************************************/
267     always@(posedge CLK)
268     begin
269     if (RESET == 1'b0)
270     begin
          a.    en_reg <= 1'b0;
          b.    gr_reg <= 1'b0;
271     end
          a.    else
272     begin
          a.    en_reg <= EN;
          b.    gr_reg <= GR;
273     end
274     end

275     endmodule
276     /*****Execute -> to Writeback*****************************************************/

277     module block78and79and80_topipe_wb(
278     //
279     // INFO: reading from module_topipe_wb.v
280     //
281     /*****Ports*************************************************************/
282     /* INPUT PORTS */
283     input         ACK,            /* Enable Acknowledged */
284     input         CLK,            /* System Clock 50 - 100 MHZ */
285     input         DNE,            /* Execution Done */
286     input         EN_EX,          /* Enable Execute Phase */
287     input         EXCEXT,         /* Exception Flush */
288     input         EXTNOP_MA,      /* Extension Bubble in Memory Access Phase */
289     input [31:0] PC_EX,           /* Current PC Execute Phase */
290     input         PCLK,           /* Pipeline Clock */
291     input [4:0]      RD_EX,       /* Destination Register Execution Phase */
292     input            REGWRITE_EX, /* Register Write Execute Phase */
293     input         RESET,          /* System Reset */
294     input [31:0] RESULT_EX,       /* Result Execution Phase */
295     /* OUTPUT PORTS */
296     output [31:0]   ALURESULT_WB,  /* ALU Result to Writeback Phase */
297     output          BHLS_WB,       /* Byte/Halfword Load/Store to Writeback Phase */
298     output [1:0] DMADD_WB,     /* Least Significant Bits of Data Address to Writeback Phase */
299     output [31:0]   DMDATAOUT_WB,    /* Memory Data Out to Writeback Phase */
300     output          MEMTOREG_WB,   /* Memory to Register to Writeback Phase */
301     output [31:0]   PC_WB,         /* Current PC to Writeback Phase */
302     output          REGWRITE_WB,    /* Register Write to Writeback Phase */
303     output          REWB,          /* Re-enter at Writeback */
304     output          RNL_WB,        /* Right/Left Unaligned Load/Store to Writeback Phase */
305     output [4:0] WRREG_WB          /* Write Register Number to Writeback Phase */
306     );

307     /*****Signals***********************************************************/

308     wire EN_WB;      /* Enable to Writeback Phase */
309     wire RESET_WB;/* Reset to Writeback Phase */

310     /*****Registers*********************************************************/

311     reg [70:0] ex_wb;      /* Execute -> to Writeback Pipeline Register */
312     reg [1:0] pclkcnt;  /* Pipeline Clock edge detection */
313     reg reset_reg;          /* Reset to Writeback Phase */
314     reg rewb_reg;           /* Re-enter at Writeback */

315     /*****Initialization****************************************************/
316     /*
317     initial
318     begin
319     ex_wb = 71'b0;
```

```
320     pclkcnt = 2'b0;
321     rewb_reg = 1'b0;
322     reset_reg = 1'b0;
323     end
324     */
325     /*****************************************************************************/

326     assign RESET_WB       =  reset_reg;
327     assign REWB           =  rewb_reg & EN_WB;
328     assign EN_WB          =  ex_wb[70];     //EN_EX;
329     assign REGWRITE_WB    =  ex_wb[69];     //REGWRITE_EX;
330     assign MEMTOREG_WB    =  1'b0;
331     assign RNL_WB         =  1'b0;
332     assign BHLS_WB        =  1'b0;
333     assign DMADD_WB       =  2'b0;
334     assign WRREG_WB       =  ex_wb[68:64]; //RD_EX;
335     assign ALURESULT_WB   =  ex_wb[63:32]; //RESULT_EX;
336     assign DMDATAOUT_WB   =  32'b0;
337     assign PC_WB          =  ex_wb[31:0];   //PC_EX;

338     /*****************************************************************************/

339     always@(posedge CLK)
340     begin
341     /* Pipeline Clock edge detection */
342     pclkcnt <= {pclkcnt[0],PCLK};   // here Neil suggest to change from = to <=

343     end

344     always@(posedge CLK)
345     begin
346     case(pclkcnt)
347     2'b01     :   begin
                        i.   /* Synchronize Reset to Pipeline Clock */
                       ii.   reset_reg <= RESET;
                      iii.   end
348     default   :   begin
                        i.   end
349     endcase
350     end

351     always@(posedge CLK)
352     begin
353     /* Execute -> to Memory Access Pipeline Register */
354     casex({pclkcnt,RESET_WB,EXTNOP_MA,rewb_reg,ACK,DNE,EXCEXT})
355     8'bxx0xxxxx   :   begin
                        i.   /* Reset */
                       ii.   rewb_reg <= 1'b0;
                      iii.   ex_wb <= 71'b0;
                      iv.    end
356     8'b011xxxx1   :   begin
                        i.   /* Exception in Pipeline, Flush */
                       ii.   rewb_reg <= 1'b0;
                      iii.   ex_wb <= 71'b0;
                      iv.    end
357     8'bxx1x0110   :   begin
                        i.   /* Latch Data and Control after Execution Finishes */
                       ii.   ex_wb <= {EN_EX,REGWRITE_EX,RD_EX,RESULT_EX,PC_EX};
                      iii.   end
358     8'b101100x0   :   begin
                        i.   /* Raise REWB at next Negedge of PCLK after ACK Lowers */
                       ii.   rewb_reg <= 1'b1;
                      iii.   end
359     8'b011x1xx0   :   begin
                        i.   /* Lower REWB at next Posedge and reset register */
                       ii.   rewb_reg <= 1'b0;
                      iii.   ex_wb <= 71'b0;
                      iv.    end
360     default   :       begin
```

```
                    i.  /* NOP */
                   ii.  end
361     endcase
362     end
363     endmodule
364     //
365     // INFO: finished reading from module_topipe_wb.v
366     //


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

368     module block78and79and80_toex (
369     //
370     // INFO: reading from module_toex.v
371     //
372     /*****Ports*************************************************************************/
373     /* INPUT PORTS */
374     input         ACK,            /* Enable Acknowledged */
375     input         CLK,            /* System Clock 50 - 100 MHZ */
376     input [15:0] DIMM_ID,         /* Data Immediate Instruction Decode Phase */
377     input         EN_ID,          /* Enable Instruction Decode Phase */
378     input         EXCEXT,         /* Exception Flush */
379     input [31:0] PC_ID,           /* Current PC Decode Phase */
380     input         PCLK,           /* Pipeline Clock */
381     input [31:0] RDREG1DATA_ID, /* Register Read Port 1 Register Data Instruction Decode Phase */
382     input [31:0] RDREG2DATA_ID, /* Register Read Port 2 Register Data Instruction Decode Phase */
383     input         REGWRITE_ID,  /* Register Write Instruction Decode Phase*/
384     input         RESET,          /* System Reset */
385     input [4:0]      RS_ID,          /* Operand Register 1 Instruction Decode Phase */
386     input [4:0]      RT_ID,          /* Operand Register 2 Instruction Decode Phase */
387     input         SLL128_ID,      /* Shift Left Logical 128 bits Instruction Decode Phase */
388     /* OUTPUT PORTS */
389     output [15:0]   DIMM_EX,        /* Data Immediate Execute Phase */
390     output            EN_EX,        /* Enable Execute Phase */
391     output [31:0]   PC_EX,          /* Current PC Instruction Decode Phase */
392     output [31:0]   RDREG1DATA_EX,/* Register Read Port 1 Register Data Execute Phase */
393     output [31:0]   RDREG2DATA_EX,/* Register Read Port 2 Register Data Execute Phase */
394     output            REGWRITE_EX, /* Register Write Execute Phase*/
395     output            RESET_EX,       /* Reset Execute Phase */
396     output [4:0] RS_EX,           /* Operand Register 1 Execute Phase */
397     output [4:0] RT_EX,           /* Operand Register 2 Execute Phase */
398     output            SLL128_EX       /* Shift Left Logical 128 bits Execute Phase */
399     );

400     /*****Registers*********************************************************************/

401     reg [124:0]  id_ex;     /* Instruction Decode -> Execute Pipeline Register */
402     reg [1:0] pclkcnt;      /* Pipeline Clock edge detection */
403     reg            reset_reg;    /* Reset Execute Phase */

404     /*****Initialization****************************************************************/

405     /*
406     initial
407     begin
408     id_ex = 125'b0;
409     pclkcnt = 2'b0;
410     reset_reg = 1'b0;
411     end
412     */

413     /**********************************************************************************/

414     assign RESET_EX       =  reset_reg;
415     assign EN_EX          =  id_ex[124];          //EN_ID;
416     assign SLL128_EX      =  id_ex[123];          //SLL128_ID;
417     assign REGWRITE_EX =  id_ex[122];          //REGWRITE_ID;
418     assign RS_EX          =  id_ex[121:117];    //RS_ID;
```

```
419     assign RT_EX          =  id_ex[116:112];    //RT_ID;
420     assign DIMM_EX        =  id_ex[111:96];     //DIMM_ID;
421     assign PC_EX          =  id_ex[95:64];      //PC_ID;
422     assign RDREG1DATA_EX=  id_ex[63:32];      //RDREG1DATA_ID;
423     assign RDREG2DATA_EX=  id_ex[31:0];     //RDREG2DATA_ID

424     /*********************************************************************/

425     always@(posedge CLK)
426     begin
427     /* Pipeline Clock edge detection */
428     pclkcnt = {pclkcnt[0],PCLK};   // karl, 9/19, change to non-blocking to
                  i.   // match Neil
429     end

430     always@(posedge CLK)
431     begin
432     case(pclkcnt)
433     2'b01     :  begin
                     i.   /* Synchronize Reset to Pipeline Clock */
                     ii.  reset_reg <= RESET;
                     iii. end
434     default   :  begin
                     i.   end
435     endcase
436     end

437     always@(posedge CLK)
438     begin
439     /* Instruction Decode -> Execute Pipeline Register */
440     casex({pclkcnt,RESET_EX,ACK,EXCEXT})
441     5'bxx0xx:  begin
                     i.   /* Reset */
                     ii.  id_ex <= 109'b0;
                     iii. end
442     5'b011x1:  begin
                     i.   /* Exception in Pipeline, Flush */
                     ii.  id_ex <= 109'b0;
                     iii. end
443     5'bxx110:  begin
                     i.   /* Hold state during Execute Phase */
                     ii.  end
444     5'b01100:  begin
                     i.   /* Clocking the Pipeline */
                     ii.  id_ex <=
                          {EN_ID,SLL128_ID,REGWRITE_ID,RS_ID,RT_ID,DIMM_ID,PC_ID,RDREG1DATA_ID,RDREG2DATA_ID};
                     iii. end
445     default   :  begin
                     i.   /* NOP */
                     ii.  end
446     endcase
447     end
448     endmodule
449     //
450     // INFO: finished reading from module_toex.v
451     //


452     //
453     // extension instruction decode
454     //
455     module block78and79and80_ext_id(
456     input       CLK,
457     input       EN,
458     input [31:0]  INSTR,
459     input [31:0]  PC,
460     input       RESET,

461     output reg [15:0] DIMM,
```

```
462     output reg [31:0] JMPADD,
463     output reg       REGWRITE,
464     output reg       RI,
465     output reg [4:0]  RS,
466     output reg [4:0]  RT,
467     output reg       SLL128
468     );

469     reg [31:0] jmpadd_c;
470     reg en_r;
471     reg [5:0] op_r;
472     reg [31:0] pc_r;
473     reg opcode_match;

474     // combinatorial logic for instruction decode
475     always @ (*) begin
476     jmpadd_c = pc_r + 36 + 4;
477     opcode_match = (op_r == 30);
478     end

479     // sequential logic for instruction decode
480     always @ (posedge CLK) begin
481     if (!RESET) begin
482     DIMM    <= 16'h0;
483     op_r    <= 6'h0;
484     RS      <= 5'h0;
485     RT      <= 5'h0;
486     en_r    <= 1'h0;
487     pc_r    <= 32'h0;
488     JMPADD  <= 32'h0;
489     RI      <= 1'h1;
490     SLL128  <= 1'h0;
491     REGWRITE <= 1'h0;
492     end else begin
493     DIMM    <= INSTR[15:0];
494     op_r    <= INSTR[31:26];
495     RS      <= INSTR[25:21];
496     RT      <= INSTR[20:16];
497     en_r    <= EN;
498     pc_r    <= PC;
499     JMPADD  <= jmpadd_c;
500     RI      <= ~opcode_match;
501     SLL128  <= en_r & opcode_match;
502     REGWRITE <= en_r & opcode_match;
503     end
504     end
505     endmodule

506     module block78and79and80_ext_ex (
507     //
508     // INFO: reading from m2v_ex_bp_ports.v
509     //
510     /*****Ports*******************************************************************/
511     /* INPUT PORTS */
512     input         CLK,              /* System Clock 50 - 100 MHZ */
513     input [15:0] DIMM,                  /* Data Immediate */
514     input         EN,               /* Enable  */
515     input         EXTNOP_MA,              /* Extension Bubble in Memory Access Phase */
516     input         GR,               /* Grant Pipeline Resources */
517     input [31:0] PC,                /* Current PC */
518     input         PCLK,                 /* Pipeline Clock */
519     input [31:0] RDREG1DATA,          /* Register Read Port 1 Register Data */
520     input [31:0] RDREG1DATA_ID,       /* Register Read Port 1 Register Data Instruction Decode Phase */
521     input [31:0] RDREG2DATA,          /* Register Read Port 2 Register Data */
522     input [31:0] RDREG2DATA_ID,       /* Register Read Port 2 Register Data Instruction Decode Phase */
523     input [31:0] RDREG3DATA,          /* Register Read Port 3 Register Data */
524     input [31:0] RDREG4DATA,          /* Register Read Port 4 Register Data */
525     input         REGEMPTY,         /* Register Write Buffer Empty */
526     input         REGFULL,          /* Register Write Buffer Full */
```

```
527     input       REGRDY,             /* Register Write Buffer Ready */
528     input       RESET,          /* System Reset */
529     input [4:0] RS,             /* Operand Register 1 */
530     input [4:0] RT,             /* Operand Register 2 */
531     input       SLL128,         /* Shift Left Logical 128 bits */
532     input [31:0] MDATA_IN,      /* Memory Data In */
                                1.  /* Multiplexed: */
                                2.  /*    Memory Data In */
                                3.  /*    Peripheral Memory Data In */
                                4.  /*    Memory Data Monitor */
533     input       MDATA_VLD_IN,   /* Memory Data Valid */
534     input   EXCEXT,     /* Exception Signal */

535     /* OUTPUT PORTS */
536     output reg          ACK,            /* Enable Acknowledged */
537     output reg [31:0] CJMPADD,          /* Conditional Jump address to offset from Current PC */
538     output reg          DNE,            /* Execution Done */
539     output reg          PCNEXT,         /* Conditional PC Update */
540     output reg [4:0]    RD,             /* Destination Register */
541     output reg          REGWRITE1,      /* Register Write Port 1 Write Enable */
542     output reg          REGWRITE2,      /* Register Write Port 2 Write Enable */
543     output reg [4:0]    RDREG1,         /* Register Read Port 1 Register Number */
544     output reg [4:0]    RDREG2,         /* Register Read Port 2 Register Number */
545     output reg [4:0]    RDREG3,         /* Register Read Port 3 Register Number */
546     output reg [4:0]    RDREG4,         /* Register Read Port 4 Register Number */
547     output reg [31:0] RESULT,       /* Result */
548     output reg [31:0] WRDATA1,          /* Register Write Port 1 Data */
549     output reg [31:0] WRDATA2,          /* Register Write Port 2 Data */
550     output reg [4:0]    WRREG1,         /* Register Write Port 1 Register Number */
551     output reg [4:0]    WRREG2,         /* Register Write Port 2 Register Number */
552     output   reg    BLS_OUT,        /* Byte Load/Store */
553     output reg   HLS_OUT,       /* Halfword Load/Store */
554     output   reg    RNL_OUT,        /* Memory Right/Left Unaligned Load/Store */
555     output reg [31:0]   MADDR_OUT,      /* Memory Address */
556     output reg [31:0]   MDATA_OUT,      /* Memory Data Out */
                                1.  /* Multiplexed: */
                                2.  /*    Memory Data Out */
                                3.  /*    Peripheral Memory Data Out */
557     output   reg    MOE_OUT,            /* Memory Output Enable */
558     output   reg    MWE_OUT,            /* Memory Write Enable */
559     output reg [31:0] VIRPC   /* Virtual PC for interrupt support */
560     );

561     // tie off outputs that are not used in the automated accelerator
562     always @ (posedge CLK) begin
563     RD <= 0;
564     RESULT <= 0;
565     end

566     /*****************************************************************/

567     //
568     // INFO: finished reading from m2v_ex_bp_ports.v
569     //

570     // parameters for extension execution block
571     parameter MAX_STATE = 6;
572     parameter MAX_STATE_AB = 3; // ruirui1
573     parameter MAX_STATE_AC = 6; // ruirui1
574     parameter REG_READ_WAIT_STATES = 5;

575     // declarations for extension state machine
576     reg[MAX_STATE:1] state_r;
577     reg[5:1] branch_state_r;
578     reg[5:1] write_state_r;
579     reg[5:1] read_state_r;
580     reg[MAX_STATE:1] ab_state_r; // ruirui1
581     reg[MAX_STATE:1] ac_state_r; // ruirui1
```

```
582     // declarations for the extension memory state machine
583     reg[5:1] mem_write_state_r;
584     reg[5:1] mem_read_state_r;

585     // declarations for memory data in variable
586     reg[31:0] mdata_in;

587     // declarations for register read variables
588     reg[31:0] r6_1;
589     reg[31:0] r9_2;
590     reg[31:0] r8_4, r8_4_r;
591     reg[31:0] r7_5, r7_5_r;
592     reg[31:0] r5_10;
593     // declarations for register temp variables
594     reg[31:0] r11_7, r11_7_r;
595     reg[31:0] r5_12, r5_12_r;
596     reg[31:0] r9_14;
597     reg[31:0] r11_16, r11_16_r;
598     reg[31:0] r9_18, r9_18_r;
599     reg[31:0] r8_20, r8_20_r;
600     // declarations for memory address temp variables

601     // declarations for transaction model support
602     reg tran_state_done;
603     reg [31:0] virpc_tr0, virpc_tr1, virpc_tr2;
604     reg [5:1] tran_end_state_r;
605     reg transaction_end_this_state;

606     //
607     // INFO: reading from m2v_state_mc.v
608     //
609     // m2v_state_mc.v
610     //
611     // Karl Meier
612     // 8/15/07
613     //
614     // invariant state machine logic for the read, write, and branch state
615     // machines
616     //

617     reg [1:0] pclk_del_r;
618     reg pclk_rise, pclk_fall;
619     reg en_r, sll128_r, gr_r, regrdy_r, regfull_r, regempty_r, extnop_ma_r;
620     reg clr_dne, DNE_c, ACK_c;
621     reg done_state, done_state_r;
622     reg wsm_idle, wsm_idle_r, wsm_pulse, wsm_pulse_r, wsm_wait, wsm_wait_r;
623     reg write_this_state, wsm_done;
624     reg rsm_idle, rsm_idle_r, rsm_latch, rsm_latch_r;
625     reg rsm_wait, rsm_wait_r, rsm_wait2, rsm_wait2_r;
626     reg [3:0] rsm_count, rsm_count_r;
627     reg read_this_state, rsm_done;
628     reg bsm_idle, bsm_idle_r, bsm_calc, bsm_calc_r;
629     reg bsm_wait, bsm_wait_r, bsm_waitpf, bsm_waitpf_r;
630     reg bsm_waitpr, bsm_waitpr_r;
631     reg branch_this_state, bsm_done;
632     reg fsm_idle, fsm_idle_r, fsm_wait2, fsm_wait2_r, fsm_wait, fsm_wait_r;
633     reg final_state, fsm_done;
634     reg take_branch, take_branch_r;
635     reg after_branch_state_r; // ruirui1

636     reg [1:0] mdata_vld_r;
637     reg mdata_vld_rise, mdata_vld_fall;
638     reg mem_read_this_state, mrs_done;
639     reg mem_write_this_state, mws_done;
640     reg [1:0] mem_this_state, mdne, mdne_c;

641     // State machine logic for MMU access instructions
642     // memory access control for the extension
643     always @ (*) begin
```

```
644     mdata_vld_rise = (mdata_vld_r == 2'b01);
645     mdata_vld_fall = (mdata_vld_r == 2'b10);

646     // Place memory read/write request on rising edge of PCLK
647     casex ({RESET, mem_this_state})
648     3'b0xx : begin
          a.    // Reset stage
                  i.    RNL_OUT <= 1'b0;
                 ii.    BLS_OUT <= 1'b0;
                iii.    HLS_OUT <= 1'b0;
                 iv.    MOE_OUT <= 1'b0;
                  v.    MWE_OUT <= 1'b0;

                 vi.    mdata_in <= 32'b0;
                vii.    mem_this_state <= 2'b00;
               viii.    mdne_c <= 0;
          b.    end
649     3'b100 : begin
          a.    casex ({mem_read_this_state, mem_write_this_state})
          b.    2'b10 : begin
                  i.    if (pclk_del_r == 2'b01) begin
                 ii.    /* Memory read state */
                iii.    RNL_OUT <= 1'b0;
                 iv.    BLS_OUT <= 1'b0;
                  v.    HLS_OUT <= 1'b0;
                 vi.    MOE_OUT <= 1'b1;
                vii.    MWE_OUT <= 1'b0;

               viii.    mem_this_state <= 2'b01;   // next state for read operation
                 ix.    end
                  x.    end

                 xi.    2'b01 : begin
                xii.    if (pclk_del_r == 2'b01) begin
               xiii.    /* Memory write state */
                xiv.    RNL_OUT <= 1'b0;
                 xv.    BLS_OUT <= 1'b0;
                xvi.    HLS_OUT <= 1'b0;
               xvii.    MOE_OUT <= 1'b0;
              xviii.    MWE_OUT <= 1'b1;

                xix.    mem_this_state <= 2'b10;
                 xx.    end
                xxi.    end

               xxii.    default : begin
              xxiii.    /* No memory access this state */
               xxiv.    RNL_OUT <= 1'b0;
                xxv.    BLS_OUT <= 1'b0;
               xxvi.    HLS_OUT <= 1'b0;
              xxvii.    MOE_OUT <= 1'b0;
             xxviii.    MWE_OUT <= 1'b0;

               xxix.    mdata_in <= 32'b0;
                xxx.    mem_this_state <= 2'b00;
               xxxi.    end
          c.    endcase
          d.    end
650     3'b101 : begin
          a.    // Remove memory read request after the falling edge of MDATA_VLD_IN
          b.    if (mdata_vld_fall) begin
                  i.    MOE_OUT <= 1'b0;
                 ii.    end
                iii.    if(~MOE_OUT & mdata_vld_rise) begin     // Look for MDATA_VLD_IN signal only after initiating the request
                                                        i.    // and after the falling edge of PCLK
                 iv.    mdata_in <= MDATA_IN;  // Assign the input data to the result register

                  v.    // Latch the memory done signal (MDATA_VLD_IN)
                 vi.    mdne_c <= 1;
```

```
                                    1.  mem_this_state <= 2'b00;
                            vii.    end
                    c.      end
651         3'b110 : begin
                    a.      // Remove memory write request after the falling edge of MDATA_VLD_IN
                    b.      if (mdata_vld_fall) begin
                                i.  MWE_OUT <= 1'b0;
                                ii. end
                                iii.    if(~MWE_OUT & mdata_vld_rise) begin    // Look for MDATA_VLD_IN signal only after initiating the request
                                                                        i.  // and after the falling edge of PCLK

                                iv. // Latch the memory done signal (MDATA_VLD_IN)
                                v.  mdne_c <= 1;
                                        1.  mem_this_state <= 2'b00;
                                vi. end
                    c.      end
652         default : begin
                    a.      end
653         endcase
654         end

655         // state machine logic for compiled extension
656         always @ (*) begin
657         pclk_rise = (pclk_del_r == 2'b01);
658         pclk_fall = (pclk_del_r == 2'b10);

659         // start the extension instruction
660         clr_dne = state_r[1] & en_r & sll128_r;

661         // state machine for read logic
662         read_this_state = (| (state_r & read_state_r));
663         rsm_wait = read_this_state &
664         (rsm_idle_r & gr_r) |
665         (rsm_wait_r & (rsm_count_r != REG_READ_WAIT_STATES));
666         rsm_latch = rsm_wait_r & (rsm_count_r == REG_READ_WAIT_STATES);
667         rsm_wait2 = (rsm_wait2_r | rsm_latch_r) & ~done_state;
668         rsm_idle = ~rsm_wait & ~rsm_wait2 & ~rsm_latch;
669         rsm_count = rsm_idle_r ? 4'h0 : (rsm_count_r + 1);
670         rsm_done = ~read_this_state |
671         (read_this_state & (rsm_latch_r | rsm_wait2_r));

672         // state machine for write logic
673         write_this_state = (| (state_r & write_state_r));
674         wsm_pulse = wsm_idle_r & write_this_state & gr_r & regrdy_r & ~regfull_r;
675         wsm_wait = (wsm_pulse_r & ~done_state) |
676         (wsm_wait_r  & ~done_state);
677         wsm_idle = ~wsm_pulse & ~wsm_wait;
678         wsm_done = ~write_this_state |
679         (write_this_state & (wsm_pulse_r | wsm_wait_r));

680         // state machine for Memory read logic
681         mem_read_this_state = (| (state_r & mem_read_state_r)) & ~mdne;
682         mrs_done = ~mem_read_this_state |
                    a.      (mem_read_this_state & mdne);

683         // state machine for Memory write logic
684         mem_write_this_state = (| (state_r & mem_write_state_r)) & ~mdne;
685         mws_done = ~mem_write_this_state |
                    a.      (mem_write_this_state & mdne);

686         // state machine for branch logic
687         branch_this_state = (| (state_r & branch_state_r));
688         bsm_calc = bsm_idle_r & branch_this_state;
689         bsm_waitpf = (bsm_calc_r & take_branch_r) |
                    a.      (bsm_waitpf_r & ~pclk_fall);
690         bsm_waitpr = (bsm_waitpf_r & pclk_fall) |
                    a.      (bsm_waitpr_r & ~pclk_rise);
691         bsm_wait = (bsm_calc_r & ~take_branch_r & ~done_state) |
692         (bsm_waitpr_r & pclk_rise & ~done_state) |
```

```
693     (bsm_wait_r & ~done_state);
694     bsm_idle = ~bsm_calc & ~bsm_wait & ~bsm_waitpr & ~bsm_waitpf;
695     bsm_done = ~branch_this_state |
696     (branch_this_state &
          a.    ((bsm_calc_r & ~take_branch_r) |
          b.    (bsm_waitpr_r & pclk_rise) |
          c.    bsm_wait_r));

697     // state machine to finish up the extension instruction
698     //final_state = state_r[MAX_STATE];
699     final_state = take_branch_r ? state_r[MAX_STATE_AB] : state_r[MAX_STATE_AC]; // ruirui1

700     fsm_wait = final_state & rsm_idle_r |
701     (fsm_wait_r & ~(gr_r & regempty_r & extnop_ma_r));
702     fsm_wait2 = (fsm_wait_r & gr_r & regempty_r & extnop_ma_r) |
          a.    (fsm_wait2_r & ~en_r);
703     fsm_idle = ~fsm_wait & ~fsm_wait2;
704     fsm_done = final_state & fsm_wait2_r & ~en_r;

705     // clear DNE as the extension instruction is entered
706     // set DNE as the extension instruction is exited
707     DNE_c = (DNE | (fsm_wait_r & gr_r & regempty_r & extnop_ma_r)) & ~clr_dne;
708     ACK_c = (ACK | (~DNE & ~ACK)) & ~(ACK & DNE & pclk_rise) & EN;
709     // & EN -> to bring down ACK when it loses control/resources in case of an interrupt
710     end

711     always @ (*) begin
712     // true when all conditions for a state have been satisfied
713     done_state = clr_dne |
          a.    (~state_r[1] & bsm_done & rsm_done & wsm_done & mrs_done & mws_done & tran_state_done);
714     end


715     // state to determine rising and falling edges of pclk
716     always @ (posedge CLK) begin
717     pclk_del_r <= {pclk_del_r[0], PCLK};
718     // rise and falling edge of MDATA_VLD_IN
719     mdata_vld_r <= {mdata_vld_r[0], MDATA_VLD_IN};
720     end

721     // buffer signals that may be heavily loaded or come from a distance
722     //   - is this needed?  this is present to maintain compatibility with Neil
723     always @ (posedge CLK) begin
724     if (!RESET) begin
725     en_r <= 1'h0;
726     sll128_r <= 1'h0;
727     gr_r <= 1'h0;
728     regrdy_r <= 1'h0;
729     regfull_r <= 1'h0;
730     regempty_r <= 1'h0;
731     extnop_ma_r <= 1'h0;
732     end else begin
733     en_r <= EN;
734     sll128_r <= SLL128;
735     gr_r <= GR;
736     regrdy_r <= REGRDY;
737     regfull_r <= REGFULL;
738     regempty_r <= REGEMPTY;
739     extnop_ma_r <= EXTNOP_MA;
740     end
741     end

742     // misc control for the extension
743     always @ (posedge CLK) begin
744     if (!RESET) begin
745     ACK <= 1'h0;
746     DNE <= 1'h1;
747     done_state_r <= 1'b0;
```

```
748     wsm_idle_r <= 1'b1;
749     wsm_pulse_r <= 1'b0;
750     wsm_wait_r <= 1'b0;

751     rsm_idle_r <= 1'b1;
752     rsm_latch_r <= 1'b0;
753     rsm_wait_r <= 1'b0;
754     rsm_wait2_r <= 1'b0;
755     rsm_count_r <= 4'b0;

756     bsm_idle_r <= 1'b1;
757     bsm_calc_r <= 1'b0;
758     bsm_wait_r <= 1'b0;
759     bsm_waitpr_r <= 1'b0;
760     bsm_waitpf_r <= 1'b0;
761     take_branch_r <= 1'h0;

762     fsm_idle_r <= 1'b1;
763     fsm_wait_r <= 1'b0;
764     fsm_wait2_r <= 1'b0;

765     end else begin
766     /*
767     // clear ack
768     if (ACK & DNE & pclk_rise)
769     ACK <= 1'h0;
770     // set ack
771     else if (~DNE & ~ACK)
772     ACK <= 1'h1;
773     */

774     ACK <= ACK_c;
775     DNE <= DNE_c;
776     done_state_r <= done_state;

777     wsm_idle_r <= wsm_idle;
778     wsm_pulse_r <= wsm_pulse;
779     wsm_wait_r <= wsm_wait;

780     rsm_idle_r <= rsm_idle;
781     rsm_latch_r <= rsm_latch;
782     rsm_wait_r <= rsm_wait;
783     rsm_wait2_r <= rsm_wait2;
784     rsm_count_r <= rsm_count;

785     bsm_idle_r <= bsm_idle;
786     bsm_calc_r <= bsm_calc;
787     bsm_wait_r <= bsm_wait;
788     bsm_waitpr_r <= bsm_waitpr;
789     bsm_waitpf_r <= bsm_waitpf;
790     // if take_branch_r is ever used outside of the branch state machine,
791     // it may need to be cleared at the end of the branch operation
792     take_branch_r <= bsm_calc ? take_branch : take_branch_r;

793     fsm_idle_r <= fsm_idle;
794     fsm_wait_r <= fsm_wait;
795     fsm_wait2_r <= fsm_wait2;
796     end
797     end

798     //
799     // INFO: finished reading from m2v_state_mc.v
800     //

801     // state machine for transaction model
802     always @ (*) begin
803     transaction_end_this_state = (| (state_r & tran_end_state_r));

804     virpc_tr0 = PC;
```

```
805     virpc_tr1 = PC + 32;
806     virpc_tr2 = CJMPADD;

807     VIRPC =  ({32{state_r[1]}} & virpc_tr0)
808     | ({32{state_r[2]}} & virpc_tr0)
809     | ({32{state_r[3]}} & virpc_tr1)
810     | ({32{state_r[4]}} & virpc_tr1)
811     | ({32{state_r[5]}} & virpc_tr2);
812     end

813     // transaction model control for the extension
814     always @ (posedge CLK) begin
815     if (!RESET) begin
816     tran_state_done <= 1;
817     end else begin
818     if (EXCEXT & EN) begin
819     if (transaction_end_this_state) begin
820     tran_state_done <= 0;
821     end
822     end else begin
823     if (~EN) begin
824     tran_state_done <= 1;
825     end
826     end
827     end
828     end


829     // registers that contain state about this cycle
830     always @ (posedge CLK) begin
831     if (~RESET) begin
832     branch_state_r[1] <= 1'b0;
833     write_state_r[1]  <= 1'b0;
834     read_state_r[1]   <= 1'b1;
835     mem_write_state_r[1]  <= 1'b0;
836     mem_read_state_r[1]   <= 1'b0;
837     tran_end_state_r[1]   <= 1'b1;
838     ab_state_r[1] <= 1'b1; // ruirui1
839     ac_state_r[1] <= 1'b1; // ruirui1

840     branch_state_r[2] <= 1'b1;
841     write_state_r[2]  <= 1'b0;
842     read_state_r[2]   <= 1'b1;
843     mem_write_state_r[2]  <= 1'b0;
844     mem_read_state_r[2]   <= 1'b0;
845     tran_end_state_r[2]   <= 1'b1;
846     ab_state_r[2] <= 1'b1; // ruirui1
847     ac_state_r[2] <= 1'b1; // ruirui1

848     branch_state_r[3] <= 1'b1;
849     write_state_r[3]  <= 1'b1;
850     read_state_r[3]   <= 1'b1;
851     mem_write_state_r[3]  <= 1'b0;
852     mem_read_state_r[3]   <= 1'b0;
853     tran_end_state_r[3]   <= 1'b0;
854     ab_state_r[3] <= 1'b0; // ruirui1
855     ac_state_r[3] <= 1'b1; // ruirui1

856     branch_state_r[4] <= 1'b0;
857     write_state_r[4]  <= 1'b1;
858     read_state_r[4]   <= 1'b0;
859     mem_write_state_r[4]  <= 1'b0;
860     mem_read_state_r[4]   <= 1'b0;
861     tran_end_state_r[4]   <= 1'b1;
862     ab_state_r[4] <= 1'b0; // ruirui1
863     ac_state_r[4] <= 1'b1; // ruirui1

864     branch_state_r[5] <= 1'b0;
865     write_state_r[5]  <= 1'b1;
```

```
866     read_state_r[5]   <= 1'b0;
867     mem_write_state_r[5]  <= 1'b0;
868     mem_read_state_r[5]   <= 1'b0;
869     tran_end_state_r[5]   <= 1'b1;
870     ab_state_r[5] <= 1'b0; // ruirui1
871     ac_state_r[5] <= 1'b1; // ruirui1

872     // final state is neccessary for both AB and AC
873     ab_state_r[6] <= 1'b1; // ruirui1
874     ac_state_r[6] <= 1'b1; // ruirui1
875     end else begin
876     branch_state_r <= branch_state_r;
877     write_state_r  <= write_state_r;
878     read_state_r   <= read_state_r;
879     mem_write_state_r  <= mem_write_state_r;
880     mem_read_state_r   <= mem_read_state_r;
881     tran_end_state_r   <= tran_end_state_r;
882     ab_state_r   <= ab_state_r; // ruirui1
883     ac_state_r   <= ac_state_r; // ruirui1
884     end
885     end


886     // combinatorial logic to/from the register file
887     always @ (*) begin
888     // combinatorial logic for register reads
889     // use read ports 3 & 4 to prevent write conflicts
890     RDREG1 = 0;
891     RDREG2 = 0;
892     r6_1 =  RDREG3DATA;
893     r9_2 =  RDREG4DATA;
894     r8_4 =  RDREG1DATA_ID;
895     r7_5 =  RDREG2DATA_ID;
896     r5_10 =  RDREG3DATA;
897     RDREG3 =  ({5{state_r[2]}} & 6)
898     | ({5{state_r[1]}} & RS)
899     | ({5{state_r[3]}} & 0);
900     RDREG4 =  ({5{state_r[2]}} & (RS + 1))
901     | ({5{state_r[1]}} & RT);

902     // combinatorial logic for register writes
903     WRREG1 =  ({5{state_r[3]}} & (RS + 3))
904     | ({5{state_r[5]}} & RS)
905     | ({5{state_r[4]}} & (RS + 1));
906     WRDATA1 = ({32{state_r[3]}} & r11_16_r)
907     | ({32{state_r[5]}} & r8_20_r)
908     | ({32{state_r[4]}} & r9_18_r);
909     REGWRITE1 = wsm_pulse_r & (state_r[3]
910     | state_r[5]
911     | state_r[4]);
912     WRREG2 =  ({5{state_r[4]}} & 0);
913     WRDATA2 = ({32{state_r[4]}} & r5_12_r);
914     REGWRITE2 = wsm_pulse_r & (state_r[4]);
915     end

916     // internal pipeline logic
917     always @ (posedge CLK) begin
918     if (~RESET) begin
919     r8_4_r <= 32'h0;
920     r7_5_r <= 32'h0;
921     r11_7_r <= 32'h0;
922     r5_12_r <= 32'h0;
923     r11_16_r <= 32'h0;
924     r9_18_r <= 32'h0;
925     r8_20_r <= 32'h0;
926     end else begin
927     r8_4_r <=  state_r[1] ? r8_4 : r8_4_r;
928     r7_5_r <=  state_r[1] ? r7_5 : r7_5_r;
929     r11_7_r <=  state_r[2] ? r11_7 : r11_7_r;
```

```
930     r5_12_r <=  state_r[3] ? r5_12 : r5_12_r;
931     r11_16_r <=  state_r[2] ? r11_16 : r11_16_r;
932     r9_18_r <=  state_r[2] ? r9_18 : r9_18_r;
933     r8_20_r <=  state_r[2] ? r8_20 : r8_20_r;
934     end
935     end


936     // logic for the Memory address and data out
937     always @ (posedge CLK) begin
938     MADDR_OUT <=  (32'h0);
939     MDATA_OUT <= (32'h0);
940     end

941     // combinatorial logic for the instruction nodes
942     always @ (*) begin
943     // [0x0]   0x14220003     bne    r6, r9, 12
944     take_branch = (r6_1 != r9_2);
945     //CJMPADD = take_branch ? (PC + 4 + {{16{DIMM[15]}},DIMM}) : PC;
946     //PCNEXT = state_r[2] & bsm_waitpr & take_branch;

947     // [0x4]   0x64282b    sltu    r11, r8, r7
948     r11_7 = ({1'b0, r8_4_r} < {1'b0, r7_5_r}) ? 1 : 0;

949     // [0x8]   0x14010006     bne    r0, r11, 24
950     take_branch = (32'h0 != r11_7_r);
951     CJMPADD = take_branch ? (PC + 4 + 24) : PC;
952     PCNEXT = state_r[3] & bsm_waitpr & take_branch;

953     // [0x10] 0x34210001     ori     r5, r5, 1
954     r5_12 = r5_10 | 1;

955     // [0x14] 0x431023    subu   r9, r9, r6
956     r9_14 = {1'b0, r9_2} - {1'b0, r6_1};

957     // [0x18] 0x85302b    sltu    r11, r8, r7
958     r11_16 = ({1'b0, r8_4_r} < {1'b0, r7_5_r}) ? 1 : 0;

959     // [0x1c] 0x461023    subu   r9, r9, r11
960     r9_18 = {1'b0, r9_14} - {1'b0, r11_16};

961     // [0x20] 0x852023    subu   r8, r8, r7
962     r8_20 = {1'b0, r8_4_r} - {1'b0, r7_5_r};

963     end

964     // primary extension state machine
965     always @ (posedge CLK) begin
966     if (~RESET) begin
967     state_r <= 1;
968     mdne <= 0;
969     after_branch_state_r <= 1'b0;
970     end else begin
971     mdne <= mdne_c;
972     if (en_r) begin
973     if (done_state) begin
974     // state_r <= {state_r[MAX_STATE-1:1], 1'b0};

975     if(state_r[2]) // ruirui1
          a.      after_branch_state_r <= 1; // ruirui1
976     //else // ruirui1

977     // jump over uneccessary state after branch decision is made
978     // after_branch_state will definitely be 1, take_branch_r will depend on the condition
979     state_r <= x_next_state(after_branch_state_r, take_branch_r, state_r, ab_state_r, ac_state_r); // ruirui1

980     end
981     end
982     else begin
```

```
983     state_r <= 1;
984     after_branch_state_r <= 0; // ruirui1, after the execution, clear to 0
985     end
986     end
987     end

988     // ruirui1
989     // max of 3 shifts below should be decided by the states left after beq
990     function [MAX_STATE:1]x_next_state;

991     input x_after_branch_state_r;
992     input x_take_branch_r;
993     input [MAX_STATE:1] x_state_r;
994     input [MAX_STATE:1] x_ab_state_r;
995     input [MAX_STATE:1] x_ac_state_r;

996     reg [MAX_STATE:1] ab_next_state_r;
997     reg [MAX_STATE:1] ac_next_state_r;

998     begin

999     if(|(x_state_r<<1 & x_ab_state_r))
1000    ab_next_state_r = (x_state_r << 1);
1001    else if(|(x_state_r<<2 & x_ab_state_r))
1002    ab_next_state_r = (x_state_r << 2);
1003    else if(|(x_state_r<<3 & x_ab_state_r))
1004    ab_next_state_r = (x_state_r << 3);
1005    else if(|(x_state_r<<4 & x_ab_state_r))
1006    ab_next_state_r = (x_state_r << 4);

1007    if(|(x_state_r<<1 & x_ac_state_r))
1008    ac_next_state_r = (x_state_r << 1);
1009    else if(|(x_state_r<<2 & x_ac_state_r))
1010    ac_next_state_r = (x_state_r << 2);
1011    else if(|(x_state_r<<3 & x_ac_state_r))
1012    ac_next_state_r = (x_state_r << 3);
1013    else if(|(x_state_r<<4 & x_ac_state_r))
1014    ac_next_state_r = (x_state_r << 4);

1015    x_next_state = x_after_branch_state_r ? (x_take_branch_r ? ab_next_state_r : ac_next_state_r)
                  i.    : {x_state_r[MAX_STATE-1:1], 1'b0};

1016    end

1017    endfunction



1018    endmodule
```