Embedded and Reconfigurable Systems Research at DemoFest'09

Zhimin Chen, Ken Eguro, Alessandro Forin, Ruirui Gu, Zhanpeng Jin, Paul Larson, Wenchao Li, Weiqin Ma, Rene Müller, Neil Pittman

Microsoft Research

William Bengston, Meg Davis, Drew Fisher, Larry Laugesen, Steve Liu, Grant Marvin, Jon Moeller, Brandon Nance, William Somers, Jillian Weise

Texas A&M University

July 2009

Technical Report
MSR-TR-2009-187

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Embedded and Reconfigurable Systems Research at DemoFest'09

Zhimin Chen, Ken Eguro, Alessandro Forin, Ruirui Gu, Zhanpeng Jin, Paul Larson, Wenchao Li, Weiqin Ma, Rene Mueller, Neil Pittman

Microsoft Research

William Bengston, Meg Davis, Drew Fisher, Larry Laugesen, Steve Liu, Grant Marvin, Jon Moeller, Brandon Nance, William Somers, Jillian Weise

Texas A&M University

Abstract

The Embedded and Reconfigurable Systems Group at Microsoft Research has been engaged in joint academic collaborations spanning both teaching and research activities. The results of these collaborations are showcased annually at the Faculty Summit at Microsoft headquarters in Redmond, WA, during the DemoFest event. This is also a good opportunity to review some of the other research projects that the group is engaged in, with special consideration to the research performed as part of the graduate internships. This report presents the demonstrations that took place during the 2009 DemoFest. We presented two undergraduate student projects from the Real Time Distributed System group at Texas A&M. One is a touch screen prototype that uses light diffraction rather than pressure-sensing to realize a multi-touch 2D input device. The second is a LED-input based dance pad that overcomes the wear and tear problems of traditional dance pads by using light sensing, and connecting LEDs as input rather than output devices.

Members of the ERSG group presented a number of research projects and demos. A set of APIs simplifies the communication between PCs and FPGA boards and when using Gigabit Ethernet achieves full-bandwidth speed. FPGAs are also used to accelerate the processing of networking protocols in a database system, with automatic generation of the circuits directly from the protocol specification. A new CPU model for the Giano full-system simulator supports the x86 ISA, and additionally realizes mixed concrete and symbolic execution of binary codes to detect data races in multi-threaded programs. A novel system for mining specifications deduces timing constraints in timed traces for digital circuits, embedded software, and network protocols. The system accurately pinpoints the source of errors in a faulty eMIPS micro-processor design. IEEE compliant Floating-Point execution units are fully optimized on a per-application basis and dynamically un/loaded in the reconfigurable logic portion of the micro-processor. The M2V compiler can now handle multi-basic blocks of MIPS binary code to automatically generate application accelerator circuits. A dual-core version of the eMIPS system demonstrated near-perfect speedup on the Montgomery modulo multiplication of large integers. The NetBSD Operating System runs in multi-user mode on the eMIPS system on two FPGA platforms. It uses a new, online scheduler to allocate the available accelerators slots to competing software applications.

1 Introduction

The Microsoft Faculty Summit is a workshop that attracts highly qualified participants from universities across the globe. One of the most favorably received events at the Summit is the DemoFest event, when a number of research projects and related activities are presented in a fair-like environment over a very short period of time (just about three hours!). The Embedded and Reconfigurable Systems Group has been present at the Summit since its inception, demonstrating both the results of its own research and the results of joint collaborations with academic partners and other researchers. The goal of this document is to attempt to communicate the vitality and excitement of the DemoFest event for those who could not attend it. While the atmosphere of free-flow communications and discussions is clearly impossible to reproduce, we can at least recapture and recount the artifacts and some of the practical demonstrations that took place in our small section of the event. Each of the document's sections is dedicated to one of the demonstrations that occurred at the event. Each section has a corresponding poster that was displayed at the booth and which is reproduced in Appendix A. Short movies and memorable moments are recorded in Appendix B.

It is likewise impossible to describe the atmosphere of the days that immediately preceded the event, when the demos where finalized sometime very late into the night and very many and very busy people crammed into a couple of offices, interacting, arguing, feeding, competing, laughing, sleeping and generally helping each other reach a common goal of complete and amazing success. Our heartfelt thanks go to the administrative and support people for their help with the ensuing bedlam, and to all our significant others for their huge patience and tolerance.

Broadening the Impact of the Goal Driven, Self-Propelling Process Learning in Embedded Systems

Steve Liu
Texas A&M University

Abstract

As we continue our efforts to enhance the learning experiences of the undergraduate computer engineering program, we have passed the experimental stage of the new curriculum, and made it a main stream approach in teaching embedded systems design. We are confident to say that giving students fun and freedom in exploring technical issues along directions of their own choices provided an excellent platform for students to translate their theoretical knowledge practices. Technologies evolve with time rapidly, yet the old-fashioned method of teaching students how to formulate and attack difficult problems, clearly can stand the test of time. It is the most effective ways for teachers to guide engineering students to gain real-world experiences, after they acquired their basic engineering theoretical foundation. Using data sheets, live codes, and design tools as the primary technical materials brings students to the real world engineering R&D environments. Using wiki pages to report their projects regularly proved to be much more interesting and productive than traditional written assignments and quizzes.

Both the Computer Science and Engineering department and the Electrical computer Engineering department have adopted the curriculum as the standard, pre-capstone design class to better prepare students before they took on the capstone design classes. The department is also planning for a new class of software studio prior to this embedded systems class, to enhance students' learning experiences through hands-on experiences. The author hosted a series of lab tours for attendees of the Summer Honors Invitational Program (SHIP) using class projects. Per the coordinator of event, "The students raved about the projects they saw".

Past few years of experiments helped us to understand the importance of adequate communications prior beginning of semesters and also broad participation of class evaluation from students. The lab-based course is expected to be more expensive than purely lecture based classes, but the value of the outcomes clearly justifies the investment, not to mention its powerful impact to the students' competence in meeting the industrial demands for higher, more advanced problem solving skills to face the ever higher global challenges.

The Curriculum

The design-*centric* curriculum aims to create a goal-driven, self-propelled learning process for the teacher and students work together to reach a common goal:

"At end of a semester, students can realize their new design concepts on a working prototype using basic hardware and software components."

With system design and creativity exploration as the class goals, students have one semester to adjust to the "learning by doing" learning process. The technical aspects of the new curriculum are conceptualized in figure 1. It was implemented in the CPSC 462 Microcomputer System class in the Computer Science Department of Texas A&M University. The changes cover much more than the technical foci. We took a clean slate approach in creating and revising the course contents; literally everything related to the class—teaching and learning objectives, technical goals, technical materials, labs, assignments, technology, even the lab physical layout itself—are new. Instead of covering myriad of technical details, many of which are broadly available on numerous web sites, we focused on the design and development processes of embedded systems. The goal is to bootstrap students' self-learning process using a "learning by doing" approach, so that eventually students can take charge of their own learning responsibility and deliver their final projects.

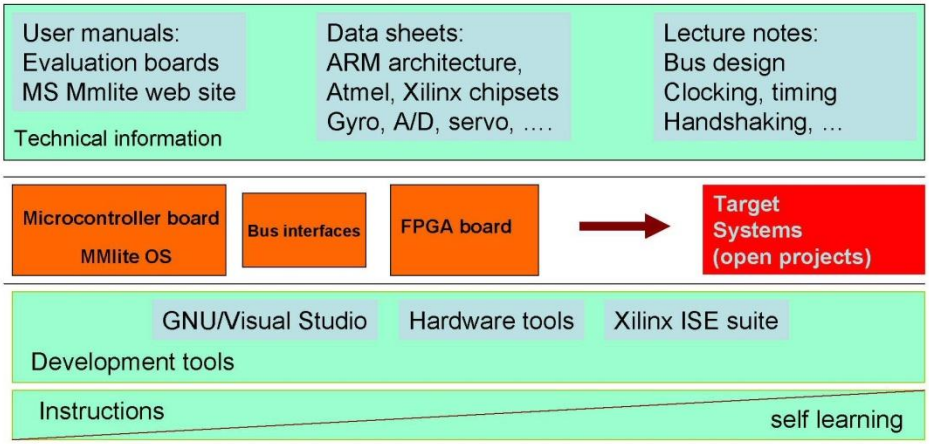


Figure 1 A design-centric curriculum for the embedded systems class.

Open Literature vs. Textbooks

A critical decision of this new curriculum was to eliminate the textbook in favor of microelectronic datasheets (published by their vendors), source codes and lecture notes. "Time to market" is an industry expression that reflects the competitive nature to deliver a product to the consumer in the shortest time possible. Similarly, after students have finished their foundational courses, we might define "time to classroom" as the need to bring contemporary materials to students so that they will be much better prepared for the job market. While textbook will continue to play its critical role in many courses, it is not necessarily the most effective tool to serve our objective in building the bound between basic knowledge and design skills. Knowing that professional engineers must rely on datasheets to carry out their designs, selective use of the datasheets help students make the transition from college textbooks to the professional "textbooks". To make the material more manageable, selection of datasheets was based on their readability, prevalence of the technology, and friendliness of the vendor web site. Using the industry datasheets instead of the textbook can be an intimidating experience for students, especially in preparation of tests. In our approach, we limited the discussion to the major chips on the evaluation boards used in the lab exercises. Only the architectural and functional portions of datasheets were included in the two written tests.

The system platform

We chose both microcontrollers and field programmable gate array (FPGA) as the hardware platforms for the class. The technical discussions of the class started with introductions of their chip-board level architectures, basic programming issues (in C and Verilog), and concluded with the interface and integration of the two types of technology at the bus level. Introduction of applications, operating system, sensors and actuators was synchronized to lab exercises for students to understand the relationship between high level languages, executable binary codes, and hardware. Typically, students were directly guided through their first lab programming assignment, after a similar lecture on the development steps. After two assignments, the discussion was shifted to the FPGA, with similar lab arrangements. Then, students were asked to build a simple bus interface of the two hardware families. A sensor/motor exercise wrapped up the lab exercises. Basic system design issues, i.e., bus architecture, timing, and handshaking protocols dominated the lecture time, with working examples presented by the TA. As of now, we continue to use the Atmel eb63 evaluation board (equipped with the AT91M63200 chip with ARM7TDMI core), and Xilinx Spartan III evaluation board for the learning phase of the class. We also continue to use the *Microsoft Invisible Computing (MIC), (http://research.microsoft.com/invisible/, aka MMLite)* as a reference operating system for this kind of small hardware platforms.

As the new generation of small microcontrollers with on-chip A/D converters begin to emerge on the market, together with highly compact wireless radios, we begin to adopt those microcontrollers for the student projects. We also began to incorporate soft-core based FPGA technology, i.e., eMIPS, for students to gain an understanding of the FPGA based systems.

Hands-on and Fun

Fun and curiosity are the key factors that stimulate the students' willingness to tackle advanced ideas and transform them into working systems. The value of building an artifact is lost on the students when it cannot be shown to friends, or if they do not find it interesting. This is a particularly important issue for embedded computing systems, because of the tight integration of hardware, software, algorithms and even mechanical design into any complete, working system. It makes the difficult process of building small computing systems under the constraints of power, sizes, and mechanical structures, etc., much more interesting. The author took a challenge-response process to mentor students through this process. Students are challenged at every stage about the scope and progress of their projects, but there is no definite correlation between their adoption of the instructors' inputs and the final project outcomes. In fact, some of the best projects were realized in total defiance of the instructors' suggestions. The process clearly stimulates the students to think out of the box and to take on more interesting projects.

Students were granted a lot of freedom in their projects, but they are also held accountable for their decisions. They can use any information, including existing designs in the projects, provided that they can add new ideas to the project. They can regroup, both scale up or scale down, as their projects progresses with time. They can even redirect the project goals anytime. That said, students are expected to justify their decisions and choices throughout the process to minimize waste of time and resources. Safe or "canned" projects are categorically discouraged, and students took this route usually received the lowest and possibly failing grades.

From the teacher's perspective, the objective of hands-on and fun is not the artifact itself, but rather the stimulation of the students' creativity, knowledge usage and teamwork. The result of the final project is only a portion of the final grade; even a failed project can receive a good grade if the team can demonstrate high quality in their project development process. The level of challenges, creativity, and teamwork vs. accountability are all important decision factors. Students are encouraged to follow and to expand upon the successful projects from previous semesters. Admittedly, some projects did resemble each other but every semester has produced at least one very interesting project that is worth preserving and that will be cited as example in the next semester. Outstanding projects are submitted to the DemoFest fair to share our experiences with participating faculty. The history of exceptional projects is maintained in a website, and together with the exposure at DemoFest it creates further motivation for the new students.

Industry Collaboration

Experience tells us that there is a strong correlation between the degree of interaction between industry and university and these educational outcomes. Ever since the inception of our collaboration, our MSR partners have played a critical role in the progress that we have been able to make. Our experiences consistently pointed out the effectiveness of industry participation of hands-on design classes. In addition to the on-going, strong collaboration with Microsoft researchers, local industries also attended the class project review activities. The feedback from them is loud and clear: we need to train students for problem solving skills, so that they can handle the ever changing technical problems on a daily basis. The unique opportunity of being able to attend the DemoFest is particularly invaluable to enhance the class outcomes as a whole. Students realized that their learning outcomes matter, and they become highly motivated to prove their competence.

Summary

In summary, we believe we have identified a sustainable, productive educational model to boost students' interests in the computer engineering program, and likely the whole engineering discipline as a whole. Students are able to complete far more advanced projects in their capstone design courses, and they demonstrate greater confidence in their job interviews. We realized that all the artifacts created by students serve as an excellent live teaching material for students to emerge in an approachable, yet challenging environment to advance their professional preparedness. It is a great investment that cannot go wrong.

Multi-touch Screen

Meg Davis, Larry Laugesen, Grant Marvin, Jon Moeller
Brandon Nance, Jillian Weise
Department of Computer Science and Engineering
Texas A&M University

Our group decided to make a multi-touch screen using infrared diodes and sensors. This method, to the best of our knowledge has never been used before in practice but in a very simple test we have determined that theoretically our design could work. Assuming this design works, we would like to build some kind of application on top of it that allows for a multi-touch interface. The following paragraphs outline the design proposal and plan of action that we thus far think should be necessary to complete the assignment to the best of our ability.

Scope & Expected Outcomes:

Depending on the success or failure of different components and steps in the system design process, the scope of our project is not as concrete as it would be if the project were to follow a strict set of implementation details. At the highest level however, our scope encompasses two main areas: screen design and application design. The screen design phase must be done first since no applications can be written until the data coming out of the sensor array is better defined. After the screen design is at a prototype stage, work may begin to progress on the application design, but development of both of these will most likely occur in tandem as each relies on the other in very crucial ways.

Action Plan:

Because of the short duration of this final project, we plan to work through the agile development methodology. This iterative methodology calls for short sprints of work and allows the project development to be flexible, quickly accounting for any changes in design. We plan to have sprints of one week in which we will strive to create complete and working modules of design for the final product. The agile development methodology also places high emphasis on frequent communication and teamwork. Not only we will have frequent meetings during the week, but we have also established a Google Group in order to effectively communicate ideas to the whole group.

We will first focus on prototyping solutions by experimenting with different sensors and different materials for the interface. Once we have decided on a direction in which to design, we will first start experimenting with the microcontrollers and the algorithms to read the sensors. Our brainstorming for how to do this involves polling by seeing how one touch on the board is read by each sensor. Then we will shift our focus to signal processing and filtering out the noise from the signals. Application development will also begin at this point. If at any point in the design, we need to change our approach, the agile methodology will give us the flexibility to do so.

Our action plan relies heavily on communication in order to work together effectively to create a product over this short period of time.

Design:

The system is based on a simple optical phenomenon called Total Internal Reflectance, which relies on light being shone into glass at such a large angle that it cannot escape. This effectively

traps the light inside the glass. When the glass is touched by something (such as a finger) the total internal reflectance becomes "frustrated", and some of that light will escape the glass.

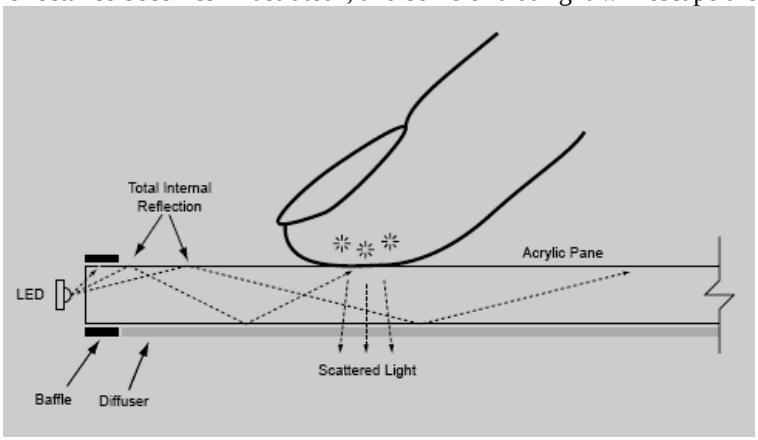
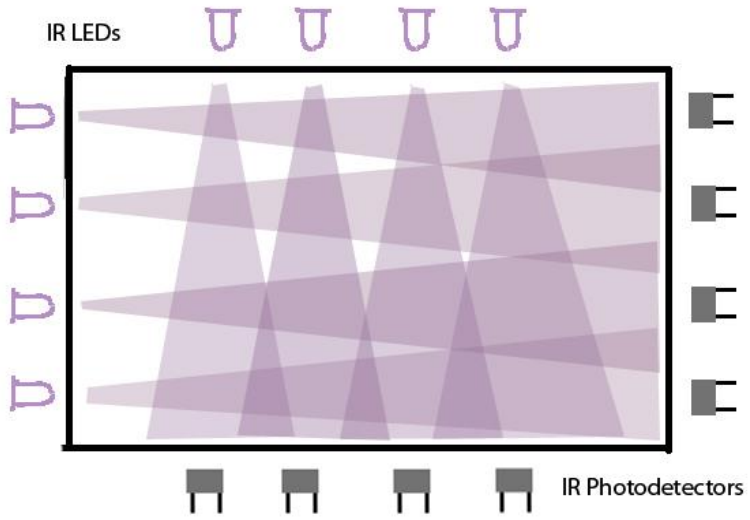


Image by Jeff Han ©2006 http://cs.nyu.edu/~jhan/ftirsense/

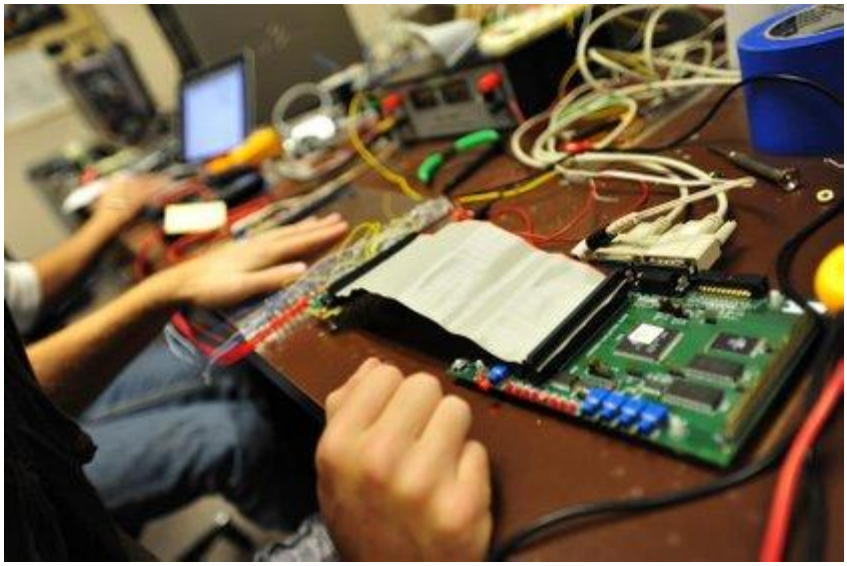
In our setup, we have photo detectors along the edges opposite the LED light sources. The goal is to detect the decrease in the amount of light hitting the sensor caused by a finger touching the surface. Touching the screen should cause an intensity dip in both the x and y axes, allowing us to locate the point of the touch.



There are a few things to note about this diagram. First, the actual design would use more than 4 photo detectors per side. It would be possible to fit as many as 40 sensors along 1 edge of a typical flat screen monitor, though it is unlikely we would need that many. The other thing of note is that the width of each beam spreads, so that a single LED might hit multiple sensors. Because of this, if the user touches a spot not directly across from a sensor, several sensors will be affected anyway. An algorithm can then use this information to determine the actual point of contact. In this way, we can have a resolution higher than the number of sensors along the edge.

Since such an approach has not been taken before (to our knowledge) the nature of the algorithm is not clear yet. However, by showing a graph of each sensors output under different types of touches would provide an excellent starting point for understanding the exact nature of the interface.

Eb63 Controls



The eb63 board is utilized in our design to act as a mediator between the output from the A/D converter and the processing software. The following files contain the primary control mechanisms used to carry out this function.

common.h :

Contains functions used to effectively manage the communication between the A/D converter and processing. OpenCom2 establishes a connection between the eb63 and the com2 serial port, allowing for data transmission from the A/D converter to the eb63. The ReadOneInteger and WriteOneInteger functions act as the data pipeline. ReadOneInteger reads the A/D converter output at a specific pint and returns the value to the eb63, while the WriteOneInteger function passes the value to the processing software.

ADCDriver.c:

Sets up many of the background utilities that allow for the actions of common.h to execute properly. StartSystemClock instantiates a 4 MHz clock to run the system, accomplished by dividing the internal clock of the eb63 to down to the desired rate. ConfigureSPI prepares the SPI protocol used to communicate with the A/D converter, allowing the eb63 to read from two separate 8-bit A/D converters. NextRead sets up which of the pins (and consequently the photo-transistor) is currently being read from, allowing the ReadOneInteger function to extract samples from the proper source.

VoltTest.cpp:

Runs the processes that manage how and when each pin is checked. The ScanVoltages function cycles through each of the photo-transistor pins and uses NextRead to prepare the pins for data extraction. After reading from the A/D converter, the function passes on the sensor number, the value received, and a 255 value to signal that the data set is complete for that transmission.

The SelfTest function was used to check that the A/D converter was functioning correctly, as this hardware was paramount to the operation of ScanVoltages. All these processes are ran In the main function at the end of the file.

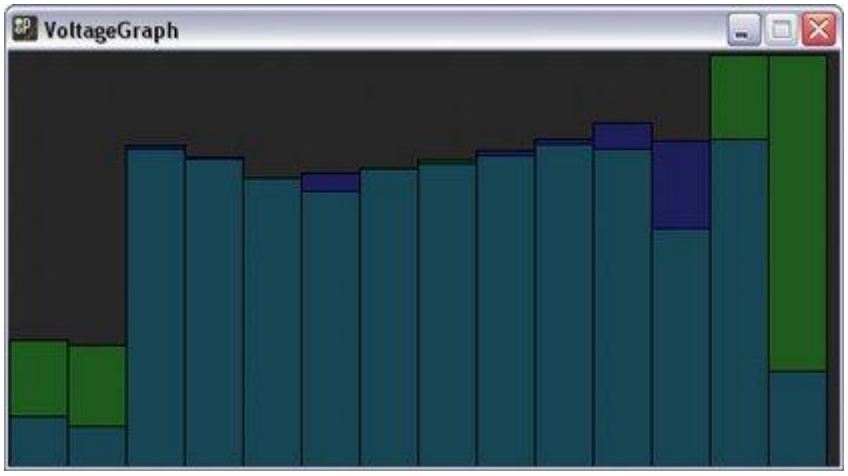
Processing

Here is a list of explanations of our processing applications.

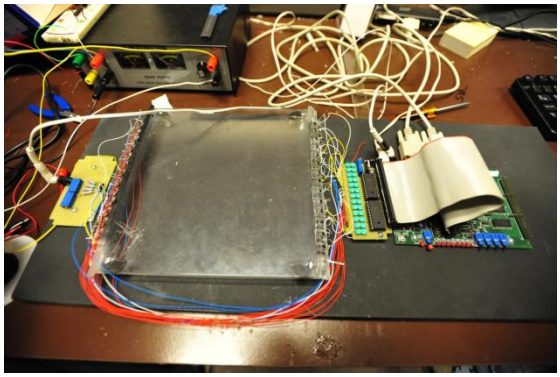
VoltageGraph

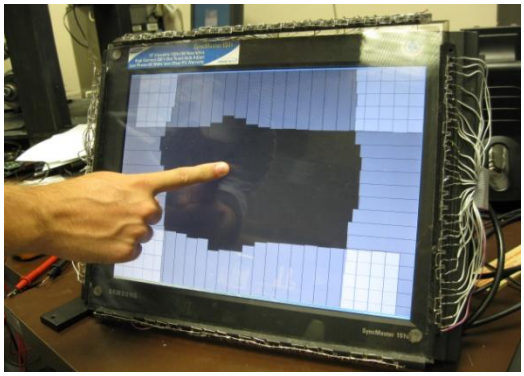
This is our flagship application that we have used nearly exclusively in the testing and refining of our multi-touch screen. In short, this application takes values given to it from the EB63 microcontroller over the serial port and maps those values in a simple bar graph. Each bar corresponds to a photo-transistor on our prototype. The graph that you see is showing the current reading of each PT, with a very short bar being a PT with a very small reading whereas a tall bar corresponds to a PT with a large reading. When you press down on the acrylic, which is termed as a "touch," the PT's that are affected by your touch will show a dip in height. To understand the physics behind why it reads less, refer to our project proposal for a detailed explanation. Our program has quite a few modes of operation, which are detailed below:

1. Normal Mode: In this mode, you see two bar graphs: one green and one blue. The green bars correspond to the right side PT's, and the blue bars correspond to the left PT's. This mode is the main mode we use to calibrate the individual potentiometers for each PT since we can try and line up each bar with all the other ones on that side. As expected, the PT's on each end are significantly less than the middle ones because less LED's are affecting those PT's. If you touch the screen in the middle of the acrylic, you should see mostly teal (green and blue combined). If you touch to either side, you will begin to see the opposite color since the dip is greater when you are closer to one side or the other. The first graph is uncalibrated (what we use to calibrate the hardware pots) and the second graph is calibrated and shows a touch on the right side.

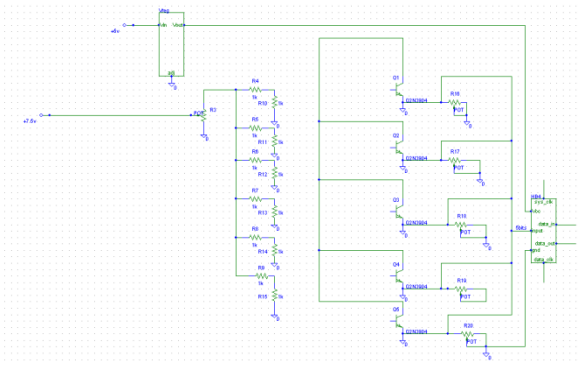


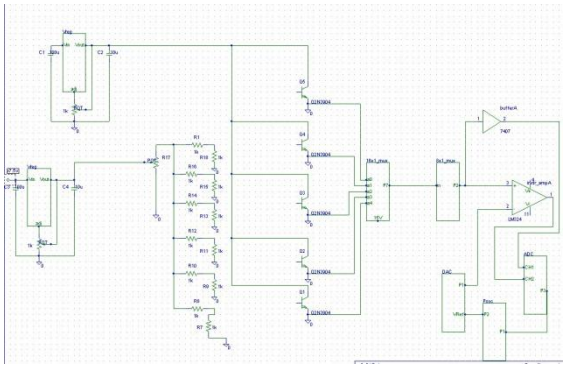
Results





Hardware Schematics





LED DDR Pad

William Somers, Drew Fisher, and William Bengston
Department of Computer Science and Engineering
Texas A&M University

Project overview

This project was created for Dr. Steve Liu's CPSC462 (Microcomputer Systems) class at Texas A&M University.

Initially, we saw a neat paper by Mitsubishi Electric Research Laboratories on using LEDs as light sensors (Dietz, P.H.; Yerazunis, W.S.; Leigh, D.L., "Very Low Cost Sensing and Communication Using Bidirectional LEDS", International Conference on Ubiquitous Computing (UbiComp), October 2003). We decided we wanted to use this concept to implement something cool. In this project, we designed, implemented, and constructed a DDR pad. A problem with most industrial designs is that over time, moving parts wear out and the pad no longer recognizes steps on a particular arrow. We designed our pad to have no moving parts. Instead, we chose to measure light reflectance off the user's feet to determine if one of the arrows was being stepped on or not. Since a design with no moving parts offers little to no tactile feedback, we wanted to make sure the pad offered some other form of feedback, so the user would know when they had successfully stepped on the arrow. To this end, we chose to embed LEDs in the arrows themselves, and to have them illuminate when the arrow was pressed. We also decided we'd like to have these LEDs flash in time with the music.

Design overview

The design consists of one master controller and four slave controllers. Each slave controls a set of 15 LEDs. Thirteen LEDs are placed around the border of the arrow and flash in rhythm with the music. The remaining two LEDs are used as an illumination and sensing pair - the LED pointing straight up acts as a permanent flashlight, and the LED at an angle serves to sense reflection of the previous LED off the foot of the player.

Readings are taken with the technique described in the MERL paper, summarized here. We place the interior sensing LED in reverse bias mode to allow the LED to charge up a capacitance in the LED junction itself. When a reading is to be taken, we set the cathode to input mode, disable the internal pull-up resistor, and time how long it takes the capacitance to discharge. Greater light input results in a shorter discharge time. As light from the illumination LED reflects off the foot of the player, the time the LED takes to discharge through the microcontroller decreases. By comparing this time to a threshold, the slaves determine if the arrow is pressed, and will raise or lower a line to the master accordingly.

The slaves can also read the voltage of a line from the master. When this bit is high, the slaves illuminate the arrow LEDs.

The master runs the main DDR pad controller program. It determines when the arrow LEDs should be on by and receives constant readings from the slaves. The master passes the readings from the slaves on to the Xbox 360 controller. The master can receive lighting commands from Stepmania running on a computer through a USB serial port, or it can just leech power from an Xbox 360.

Hardware design

The frame of the DDR pad was built using plywood and 2x4 pieces of wood to divide the pad into 9 equal squares in a 3x3 fashion. The bottom of the pad is a piece of plywood cut to 33"x33". The left and right edges were constructed using 33" pieces of 2x4 laid on their narrow side. The upper and bottom edges and the two horizontal dividing pieces were constructed with 30" pieces of 2x4 on their thin sides, so that they fit within the left and right edges. The vertical dividing pieces were constructed as 6 10" pieces of 2x4 on their thin sides to fit within the horizontal dividers. The 2x4 pieces of wood are screwed together at every meeting point and screwed into the plywood from the bottom. The four pieces that create the middle square had holes drilled in them before being screwed in, so that wires could pass underneath the 2x4. Similar treatment was given to the two horizontal dividing pieces of the top left and top right squares. 1" circular holes were drilled in the upper edge in the center of the top left and top right

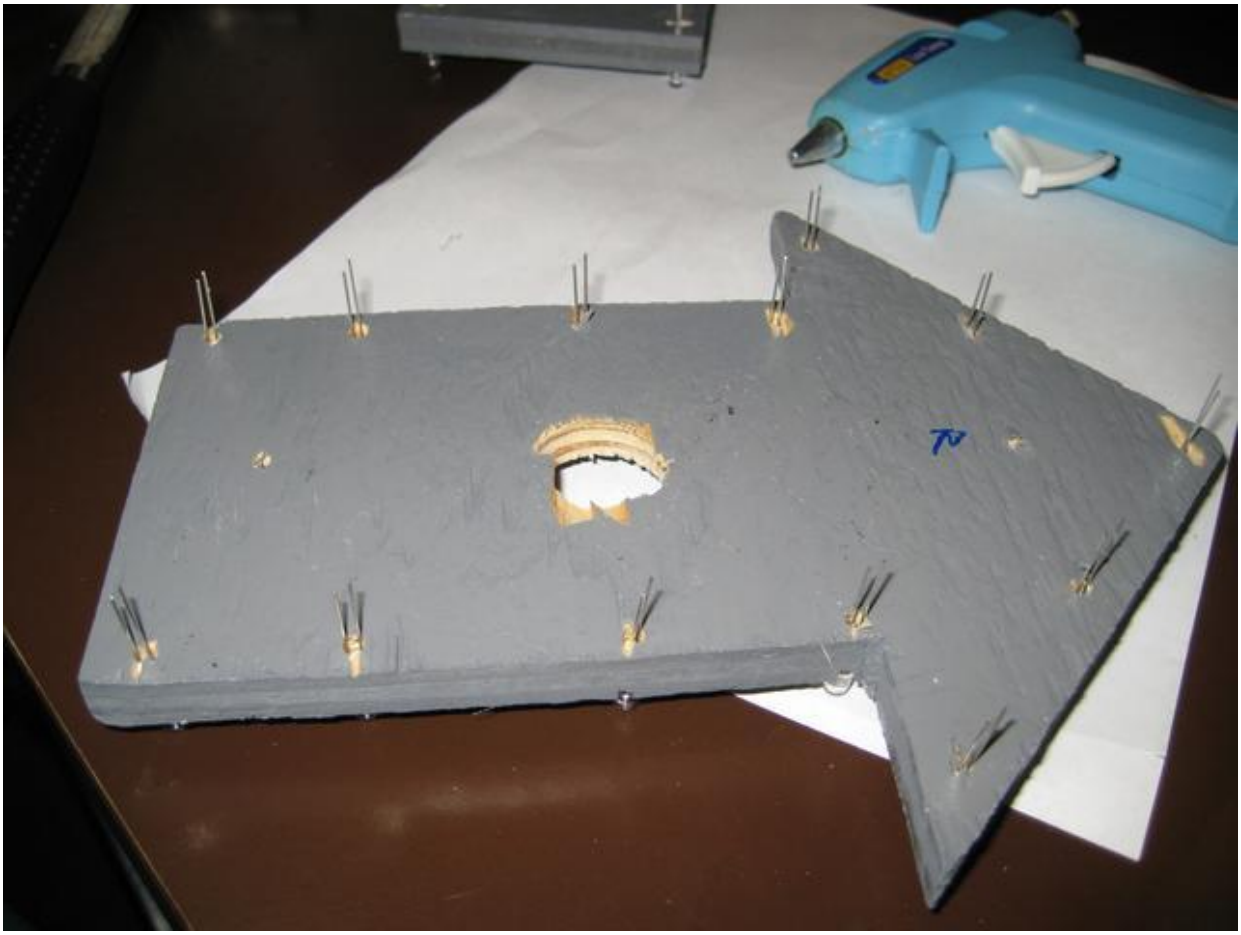
squares so communication and power cables could be run to the pad. After all holes were drilled and all the pieces were screwed together, the frame as a whole was spray painted with one coat of primer and two coats of black paint.

Frame of the wood base.

The top of the DDR pad was constructed using a 1/4" thick 33"x33" sheet of Lexan screwed to the four edges of the frame with twelve screws. A large heavy duty black handle was attached to center of the top edge.

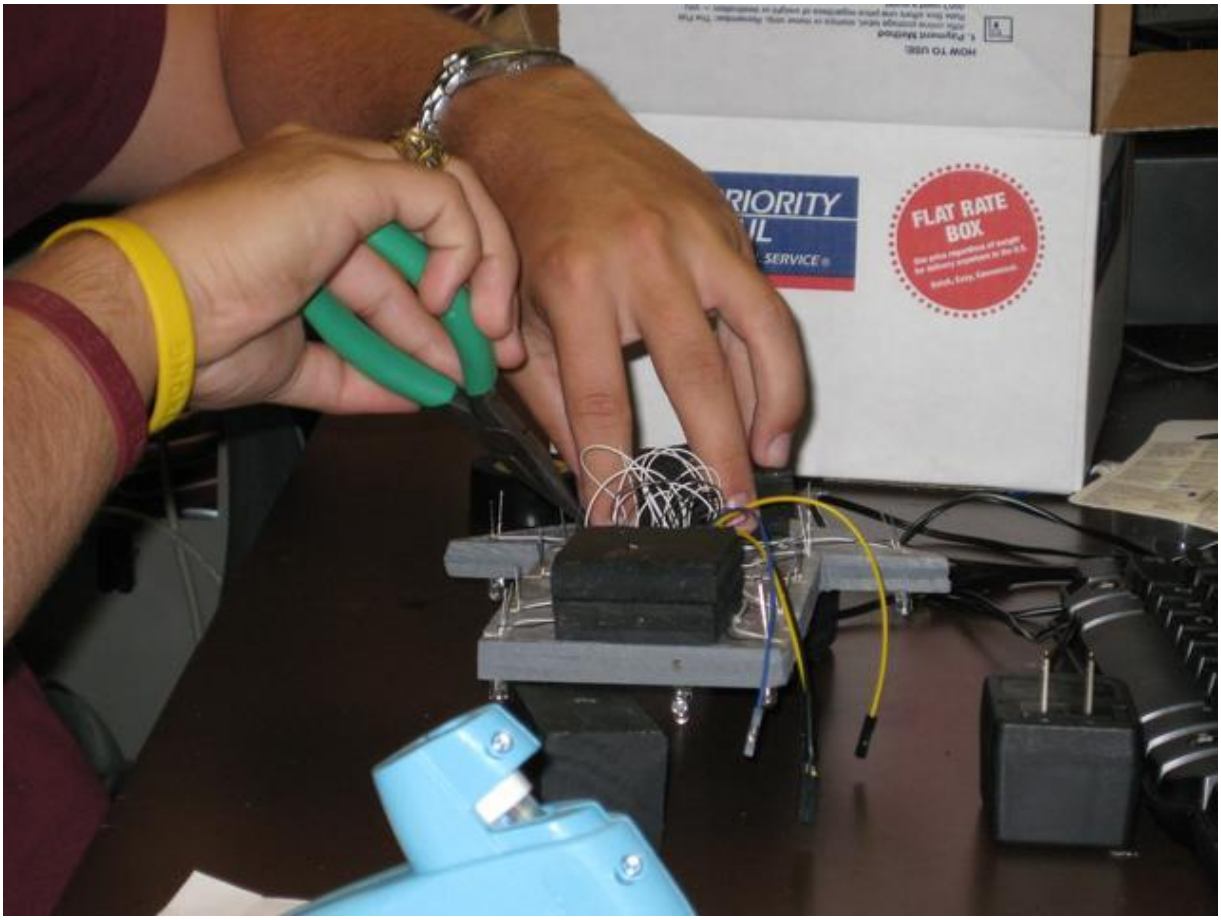
Adding the handle.

The four arrows of the DDR pad were constructed out of the same plywood as the base of the pad. They were cut into the shape of arrows about 8" long and 4" wide at the base. 13 holes were drilled around the edge of the arrow for the LED leads to fit through, and a 1" circular hole was drilled in the center for all wires of the slave board to pass through.



The underside of an arrow, before connecting leads.

The supporting pieces of the arrows were constructed out of four 2"x2" plywood squares; two stacked together each for the point and for the base of the arrow. The arrows were attached to the supporting posts with one screw from the top of the arrow and the posts were attached to the bottom of the frame with one screw from the bottom of the frame (through the plywood base). Before the pieces were assembled, they were painted. We applied one coat of primer to all the pieces, and then painted the arrow with two coats of granite colored paint and the supporting square posts with two coats of black paint.



Adjusting wires. Note that each arrow has four connectors: power, ground, data from master, and data to master.

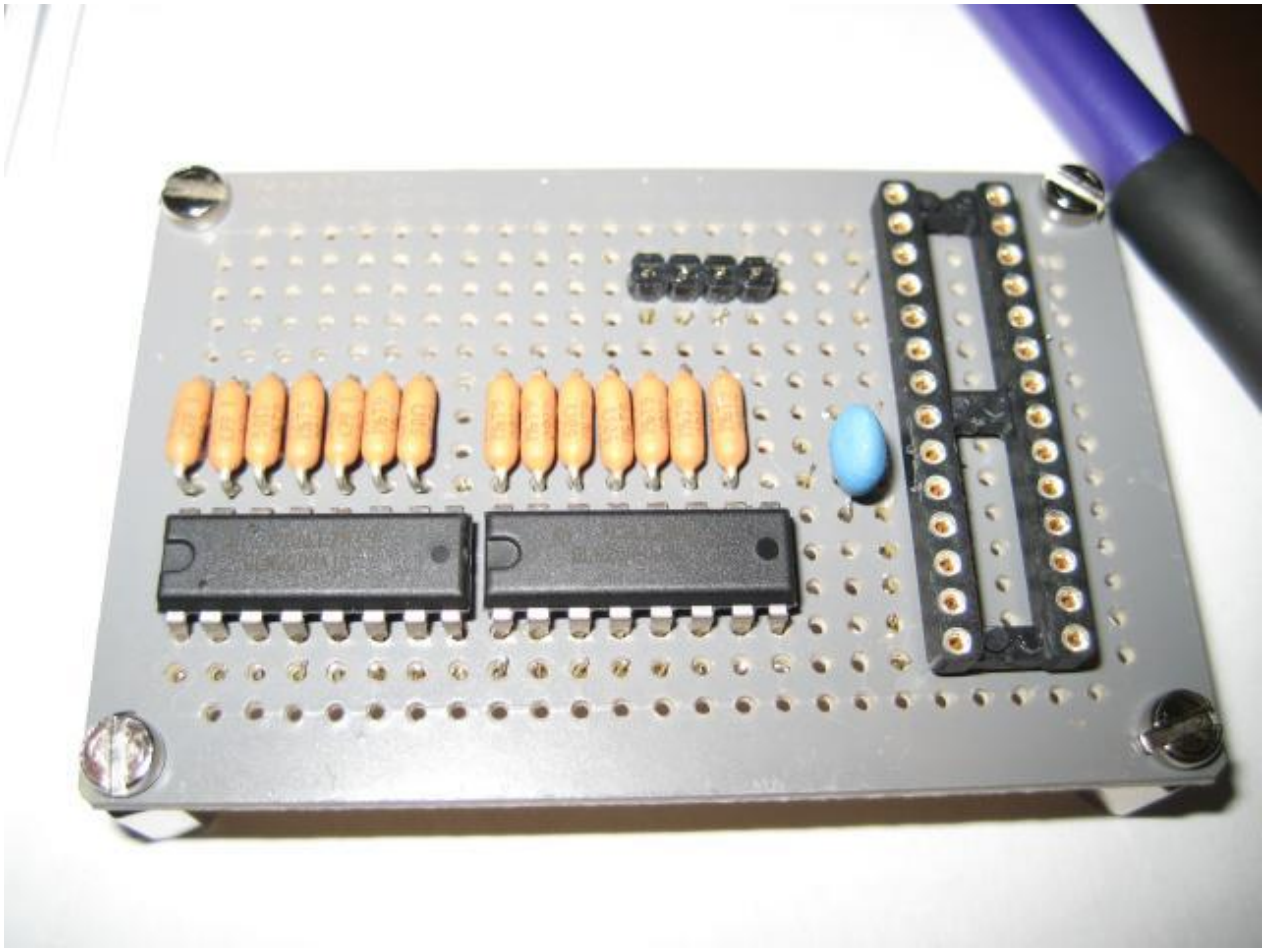
The Xbox 360 controller board was mounted in the top left square, along with the master chip (in the Arduino development board), and the master board (with hardware used to interface the master chip and Xbox 360 controller board). The Xbox 360 controller was mounted on two 2"x2" pieces of 2x4 spray painted black and screwed into the bottom of the frame. The master chip was screwed onto 1" metal risers and hot glued to the bottom of the DDR frame. The master breakout was screwed onto 1/2" risers and hot glued to the bottom of the frame. A Dell desktop computer power supply was mounted in the top right square by screwing it to the right edge of the frame. All four slave boards were screwed onto 1/2" risers and hot glued to the top of the arrow above the 1" circular hole. All wires throughout the pad were either hot glued out of sight or hot glued to an edge and spray painted black.

The board with power supply, master board, Xbox 360 controller, and cables run inside.

Electronics design

Our electronics design consists of 4 slave boards located on each arrow and 1 master board that serves as the interface between the slave boards, the computer, and the Xbox360 controller. These slave boards each held an ATmega328P microcontroller, one 20 MHz resonator with internal capacitors, two ULN2004A LED drivers and 15 LEDs. A main design was used for all the slave boards. The only difference between the two slave board designs (one for blue LEDs, the other for red LEDs) are the resistors used in series with the LEDs to prevent them from passing too much current.

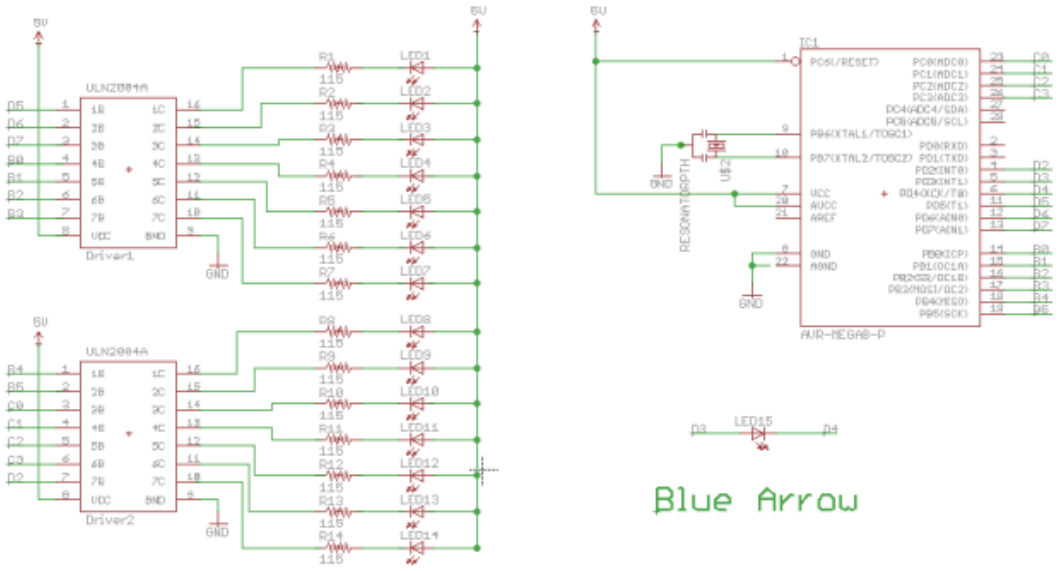
We calculated that, to keep current through the LEDs below 20mA (well within their maximum rating), we would need 50 Ohm resistors for the blue LEDs and 115 Ohm resistors for the red LEDs. Each slave board used two blue LEDs for sensing: one LED is kept constantly lit while the other is polled to detect light reflection.



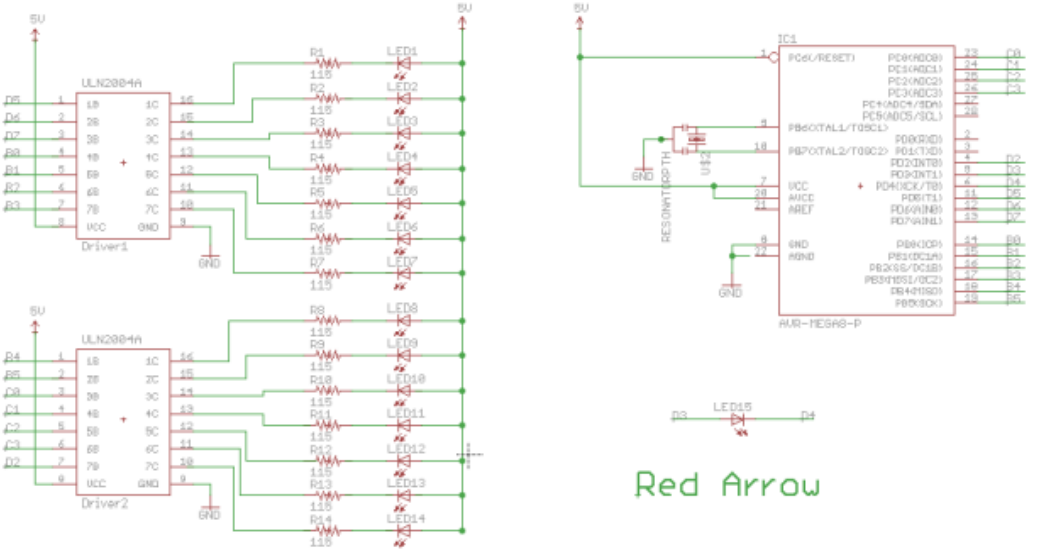
Close-up of the slave board.

Each driver is connected to VCC and GND. The drivers act as an open circuit when the pin from the ATmega328 is low. When said pin goes high, the circuit closes which allows the LED to turn on. Each LED, excluding the sensing LED, has the cathode tied to VCC and the anode tied to the LED driver.

The blue slave board schematic shows the pin configurations on the ATmega328P microcontroller, their connections to the LED drivers and the use of the 50 Ohm resistors.

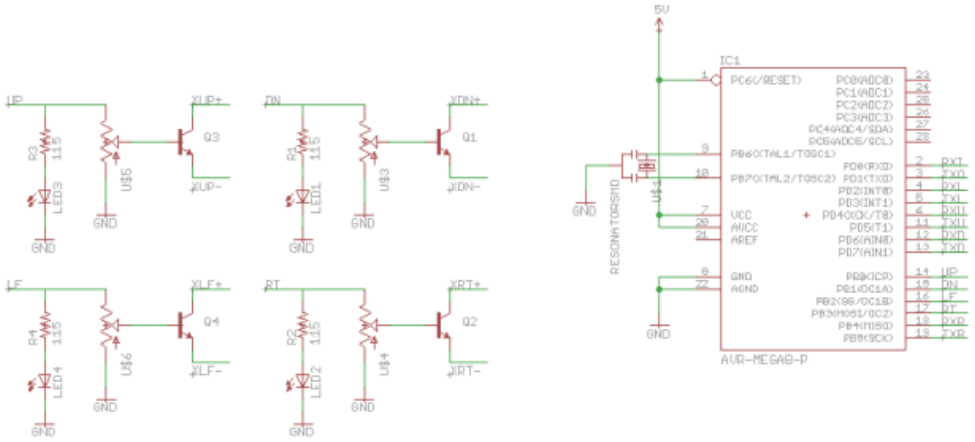


The red slave board schematic shows the pin configurations on the ATmega328P microcontroller, their connections to the LED drivers and the use of the 115 Ohm resistors.



The master board schematic shows the pin configurations on the ATmega328P microcontroller, their communications to the slaves, and the communication to the Xbox 360 controller. Ordinarily, when a user presses a button on the Xbox 360 controller, it completes a circuit through a resistive pad. To emulate this, we have a transistor tied to the two sides of the resistive pads and connect the base to a pin from the master board. We use potentiometers as voltage dividers to adjust the voltage applied at the base to prevent over-volting the Xbox 360 controller. We raise a pin on the master board, the transistor activates, current flows, and the Xbox 360 controller thinks the button is pressed.

In parallel with this circuit we have an LED and current-limiting resistor to indicate to the user that their foot step is being registered. The whole Xbox 360 button control circuit is seen four times on our master board to accommodate the four arrows: up, down, right, and left.



Software design

The software portion of this project consists of three important parts:

1. "Master" microcontroller code
2. "Slave" microcontroller code
3. Computer game code

Master microcontroller code

Part 1 is implemented as a simple infinite loop that samples the four input lines from the four slaves and writes matching values to the output lines to the Xbox360 controller. It also listens for serial commands - when it receives a '1' character, it will raise the four output lines to the four slaves to a digital HIGH, and it will drop them upon receipt of a '0'. The slaves, in turn, will see the line go high and illuminate the LEDs in the arrow.

Slave microcontroller code

Part 2 is a bit more complex. Each slave arrow has fourteen separately controllable LEDs on 14 different pins, and one LED that acts as a sensor taking up two pins.

Sensing involves applying a reverse bias to the sensing LED to charge the LED's internal capacitance, then swapping the anode to input mode and disables the internal pull-up resistor. It then counts the number of times it goes through a busy loop before the anode reads a logical low again. The more light the LED receives, the lower the number of loops.

To ensure that the arrow responds promptly to changes in the line from the master indicating light data, each iteration through the busy loop checks the status of that pin, and updates the LEDs accordingly.

We experimented with different loop counts to determine a threshold below which our code assumes that a foot is present and above which our code assumes the foot is absent. We thus go through the busy loop this threshold number of times - if the anode input hasn't gone low yet, we assume no foot is present and begin a new sample. This allows us to keep a fast sample rate, which is important for a rhythm game like DDR.

Our current code will illuminate all the LEDs in the arrow when:

- The last sample of the sensing LED went low before threshold iterations (the button is "pressed"), or
- The master has indicated that the lights should be illuminated by raising that pin to logical high.

Since each LED can be individually controlled, it is possible to reprogram the atmega328 chips to display various patterns. Getting consistent timings may be difficult, due to the nature of the variable speed at which the slaves sample (since reading a low value takes less time than reading a high one). Such code has not been implemented yet, but would require no hardware changes to effect.

Computer game code

Part 3 consists of modifications to an existing open-source program, Stepmania to send light data to our master microcontroller. Stepmania already has support for the use of standard joysticks as input devices, so by using a USB HID device (like the Xbox360 controller), we did not have to modify any code to make Stepmania recognize our pad as an input device.

Our software development was done on Linux machines, but we also wanted to ensure that the DDR pad could be used on Windows machines as well. To this end, we implemented the serial-port lighting control for both platforms.

Since we were unsure what COM port the master board would appear as to the Windows systems, we also added a command line switch to allow the user to specify which serial port Stepmania should use to send lighting data. Invoking Stepmania in the following manner will tell the program to send light data to COM10:

```
Stepmania.exe --serialport=COM10
```

If Stepmania is unable to open the appropriate serial port, it will simply continue running without sending any lighting data to the pad, and will continue to function with the pad as an input device.

As Stepmania is open-source and distributed under the GNU GPL, we have provided our patches which apply against SVN r28063 at the time of writing (2009-05-10).

Challenges faced

While the project has come together into a successful product, we experienced many trials along the way. This document serves to exhibit some of our failures, how we discovered them, and how we resolved them.

Challenge 1

We discovered that the use of the LEDs as sensors was sensitive to wire length. Specifically, we found that six inches of 22-gauge wire had substantially more capacitance to charge and discharge than that of the LEDs that we were trying to measure. The signal-to-noise ratio, as measured with a multimeter, was about 1 to 20. This meant that we would have to use minimal wire to connect the sensing LEDs, which meant we'd need a microcontroller next to each sensing LED. As such, it became infeasible to have as many sensing LEDs as we had originally hoped for.

While we toyed with the thought of calibrating the wire lengths to serve as antennae, which would change capacitance depending on proximity of the player, the difficulty in implementing such an idea put it out of our consideration.

Challenge 2

We were trying to create a breakout board on which to test the ATmega328 chips, as we would need them for the slave boards (which we needed because any significant length of wire would ruin our light readings). We failed to realize that the RESET pin (Pin 1) needed to be tied to 5V for the chip to operate properly. As we left that pin floating, the chip remained in a constant state of RESET, which meant that it wasn't doing much useful.

We also failed to attach a crystal resonator to the ATmega328s, which, compounded with the aforementioned problem, rendered the breakout boards fully nonresponsive. After rereading the spec sheets and reviewing the Arduino board schematics, we discovered our errors, and purchased the appropriate crystals and connected Pin 1 to power. Now our breakout boards would execute code and blink an LED.

Challenge 3

We tried communicating with the slave boards via a serial port with a RS232 shifter to convert TTL voltage levels to those of RS232. Unfortunately, the data came through all wrong - an ASCII 01000001 came out as an ASCII 11000001. It seemed that the timings were close, but slightly off. We realized that this was because the software was written to have delays for a 16MHz crystal, rather than the 20MHz crystals we were using. A one-line change in the Arduino dev environment configuration, and we had correctly-functioning serial communications again.

Challenge 4

We purchased a sheet of 1/8th inch polycarbonate. It proved too flimsy, so we wound up having to purchase a 1/4th inch sheet, to the tune of 125 dollars. Now it's pretty, although polycarbonate scratches rather easily. We decided that players will be required to wear socks while using our pad, to ensure its longevity.

Challenge 5

After we finished constructing all of the slave boards, we connected them all to the master, and tried to power them up. Some of the boards wouldn't power on. It turns out that we were trying to pull too much current from the power supply on the same pins, and the PSU couldn't handle that kind of sudden draw. By rewiring our connections to use different PSU pins, we got all the slaves to power on properly together.

Challenge 6

Once we had all the arrows connected, and the slaves communicating successfully with the master, we discovered that our slaves sampled somewhat slowly and would not blink in time together. This was because a single sample of

the sensing LEDs could take a quarter of a second if said LED had little illumination. Since the software only updated the arrow LEDs in between samples, if the state of the master TX pin changed during a sample, that slave would not update the lights until the end of that sample. This was resolved by having the slaves update the lights constantly during the sample-collection period, and adjusting the threshold constants accordingly. This resolved our last major problem.

Future Improvements

While this project did successfully implement our original goals, there is still room for further development.

One improvement would be to fabricate Printed Circuit Boards (PCBs) in the shape of arrows to allow for more uniform design and additional sensing LEDs within each square. This would provide greater reliability and a larger sensing surface.

A second improvement we might make would be to improve the light sensing routine to take more uniform time, or perhaps even drop the LED-as-sensor idea and just use a normal photo resistor or phototransistor for greater sensitivity. We could also make improvements to the algorithm to make the sensor more robust to different light conditions and footwear.

Another improvement would be to have more complicated blinking patterns of the arrow LEDS. They could show a circular pattern, a pattern that turned the LEDs on in order from the base to the head of the arrow (traffic diversion), and other random patterns.

High-Speed Communication API for FPGAs

Ken Eguro
Microsoft Research

Abstract

FPGAs can be used to speed up many applications by several orders of magnitude. Most of these computations require both a software and hardware component. Unfortunately, setting up the communication between software running on a host PC and a FPGA-based accelerator can often present a problem. Not only does this interface generally require laborious, custom low-level software and hardware development, it is often a critical performance bottleneck for the system as a whole. The lack of high-level support for fast and reliable communication discourages programmers from using FPGAs for their applications. This project simplifies the process of building FPGA-based hardware accelerators by providing a simple and high-performance software/hardware API infrastructure.

1 Introduction

FPGAs can exploit massive parallelism to accelerate a wide range of different applications. However, despite many successful academic and industrial research projects, FPGAs have not really gained widespread popularity. Part of the reason for this is that FPGAs are notoriously difficult to use. This project focuses on solving one aspect of this accessibility problem: simplifying the communication between software running on a host PC and accelerator hardware mapped to an FPGA.

The communication between software and hardware is an important consideration for most potential FPGA applications. This is because real-world computations are generally built with multiple phases of execution, each with their own characteristics. For example, while the central loop of an application may be computationally-intensive and naturally parallel, the data setup/teardown before and after the central loop may be control-heavy and highly sequential. In this case, it makes sense to map the computational core of the application to an FPGA in order to make best use of the available resources. The rest of the application is likely better suited to run on a conventional desktop processor. This intrinsic division of labor makes the communication scheme between the host PC and FPGA extremely important to the fundamental operation of the system.

Although it is a necessary part of most FPGA-accelerated applications, the existing level of support for building a software/hardware communication interface is relatively poor. Today's systems present two major issues for application developers. First, they require programming and debugging at a very low level of abstraction. This is problematic because relatively few developers have sufficient knowledge and experience to implement fast and reliable communication. Second, each application that a user would like to map to an FPGA requires device and/or protocol-specific code. This makes development time-consuming and prevents user code from being portable across different FPGAs or communication technologies.

The work presented here provides programmers with a simple and reusable communication API. Since the interface contains only a small set of easy-to-understand communication commands, it allows programmers to focus their development effort on their own software and hardware kernels. Furthermore, since it completely abstracts away any device or communication protocol-specific details, all of the user's code is completely portable to any system that supports the API. The standard is completely open, allowing the community at large to support new devices and communication protocols looking into the future.

2 Software and Hardware APIs

As seen in Figures 1 and 2, the software API to the user's C++ code consists of an object class that provides a small set of control and communication functions for the FPGA. These functions allow the user to configure the FPGA, send/receive data to or from the device, and control/monitor the execution of their hardware logic circuit. The software API translates these requests into protocol-specific commands, taking care of any necessary negotiation, packetization, or error checking needed to use the desired data transport medium. As seen in Figures 1 and 3, the hardware API to the user's Verilog-based circuit consists of a set of I/O memories and control signals. The user is able to send and receive bulk data through separate input and output buffers while smaller control or

status values are exchanged through a set of parameter registers. The API negotiates execution of the user's logic through a simple "start" and "done" signaling system. Similar to what is implemented internally within the software API, a controller instantiated within the hardware API's logic handles all of the protocol-specific transaction details.

Looking at Figure 2 in greater detail, the user's C++ code has access to the FPGA through a set of nine functions. The first function is simply the API class constructor. When a user would like to map some part of their computation to an FPGA, they simply create an instance of the API class object for a given communication protocol. A simplified example of using the API for spam filtering is shown in Figure 4. Our system currently supports communication over gigabit Ethernet and will soon be extended to include PCI-Express. The user can then configure the FPGA by calling the object's configure function. Our system currently pulls FPGA configuration bitstreams from a CompactFlash card attached to the supported FPGA development board (the Digilent XUP-V5). The Xilinx SystemACE chip that controls this configuration is capable of sending one of eight bitstreams to the FPGA. In the future, we plan to overload the configure function to allow the user to directly send the bitstream binary to the system. After the FPGA has been configured, the user can then send input data for their circuit using the sendWrite and sendParamRegWrite functions. The user notifies their accelerator logic that it can execute with the sendRun function. The existing system is build around the concept of batched processing, so when the user subsequently calls the waitDone function, it will spin until the user's circuit has indicated that it has completed execution. At this point, the results can be retrieved from the FPGA with the sendRead and sendParamRegRead functions. The throughput of the system for some kinds of computations may be improved by using stream-based transactions, so this will be included in future work. The final function of the API class is abort. As will be described later, after the user executes the sendRun function but before the user's circuit has indicated that is it done computing, control of the I/O buffers and parameter registers is transferred to the user's circuit.

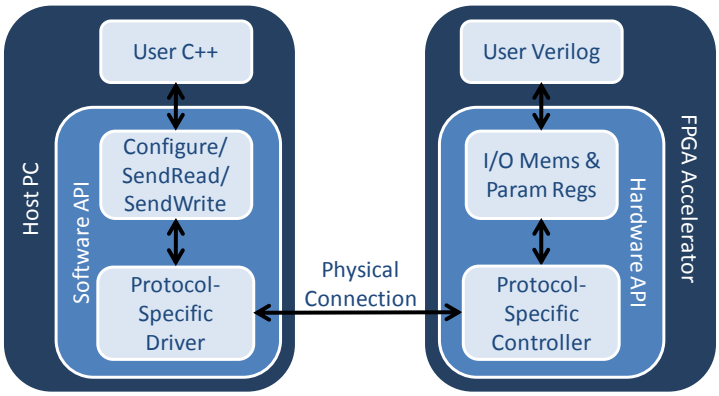


Figure 2: Software and hardware API architecture

```
class fpgaAPI{
public:
    fpgaAPI(protocolType type);

    //Command to configure FPGA from SystemACE
    bool configure(int configNum);

    //Memory I/O commands
    bool sendRead(int startAddress, int length, byte* outputBuffer);
    bool sendWrite(int startAddress, int length, byte* inputBuffer);
    //Parameter Register I/O commands
    bool sendParamRegRead(int regNumber, int *value);
    bool sendParamRegWrite(int regNumber, int value);

    //Execution commands
    bool sendRun();
    bool waitDone():
    bool abort();
}
```

Figure 3: User-accessible software API functions

```
module fpgaAPI(
    input userClk;              //Clock for user I/O
    output reset;               //Reset for user logic

    input inputMemReadAdd;      //Input memory
    output inputMemReadData;
    input outputMemWriteAdd;    //Output Memory
    input outputMemWriteData;
    input outputMemWriteEn;
    input regAddress;           //Parameter Registers
    output regReadData;
    input regWriteData;
    input regWriteEn

    output runSignal;           //Should user logic run?
    input resetRunSignal;       //Computation complete?
);
```

Figure 4: Hardware API module interface

```
processMessage(message *input, buffer *results){
    fpgaAPI *apiP = new (fpgaAPI(gigaEth));                 //Create API object of type gigabit ethernet
    apiP->configure(0);                                     //Configure FPGA with config #0 from SystemACE

    //Setup computation
    apiP->sendWrite(0, input->length, input->data);         //Send input data to FPGA
    apiP->sendParamRegWrite(0, input->length);              //Set register #0 to indicate message length

    //Process message
    apiP->sendRun();                                        //Activate user circuit
    apiP->waitDone();                                       //Wait until user circuit resets run signal

    //Retrieve results
    apiP->sendParamRegRead(1, &(results->length));          //Read register #1 that indicates result length
    apiP->sendRead(0, results->length, results->data);      //Read back output data
}
```

Figure 5: Pseudo-code for host PC process of e-mail spam filtering

```
module processMessage(input clock);
    wire reset, [7:0] inputByte, [31:0] regRData, runSignal;
    reg [31:0] inputAdd, [31:0] outputAdd, [7:0] outputByte, outputWE;
    reg [7:0] regAdd, [31:0] regWData, regWE, computationDone;

    fpgaAPI api(clock, reset, inputAdd, inputByte, outputAdd,      //Create instance of API I/O buffers, parameter registers and controller
                outputByte, outputWE, regAdd, regRData,
                regWData, regWE, runSignal, computationDone);
    initial begin
        currState = IDLE;
    end
    always @(posedge clock) begin
        if(reset) begin
            currState <= IDLE;                                    //Begin so that the API controller has control of the I/O buffers
        end
        case(currState)
            IDLE: begin
                if(runSignal)                                     //Wait until the run signal goes high
                    currState <= RUNNING;                         //Control over the I/O buffers has been given to the user circuit, start execution
            end
            RUNNING: begin
                if(computationDone)                               //Wait until the user logic is done
                    currState <=IDLE;                             //The API controller regains control of the I/O buffers
                else if(!runSignal)                               //Stop execution if the host PC aborts
                    currState <=IDLE;
                else begin
                    user logic reads message length from param reg #0, reads N bytes of message data from the input buffer, processes the message,
                        writes M bytes of result data into output buffer, puts M into param reg #1 and raises the computationDone flag
                end
            end
        endcase
    end
endmodule
```

Figure 6: Pseudo-code for user logic for e-mail spam filtering

If something goes wrong, or if the user simply wants to cancel execution, they can call the abort function to regain control of the I/O buffers and registers.

Looking at Figure 3 in more detail, the user's accelerator Verilog code connects to the hardware API I/O buffers, parameter registers and control mechanism via 13 signals. The first signal is a user domain clock. The user presents a clock to the API controller that synchronizes communication between the user's logic and the API I/O buffers and parameter registers. This is independent of the clock used internally within the controller for the actual physical communication interface to the host PC. The second signal is a system reset. When the system is initially powered on (and potentially other times during operation), the entire system is reset before beginning normal operation. If the user's circuit requires this kind of reset signal, it can pull it from the hardware API. The next two signals connect the user's circuit to the input memory buffer. The API supports up to 32-bit byte-wise addressing (4GB), although the input buffer in the current implementation of only contains 256KB (18-bit byte-wise addressing). The next three signals are used to connect the user's circuit to the output memory buffer. Similar to the

input buffer, the API supports up to 4GB of byte-addressable output memory, although the current implementation only contains 8KB (13-bit byte-wise addressing). The next four signals can be used to read and write 255 32-bit parameter registers. The last two signals for the API are used to negotiate execution of the user logic and write control over the I/O buffers and parameter registers.

Figure 5 shows simplified Verilog pseudo-code of how a user might integrate the communication API with their own logic. Essentially, the user's circuit should wait until runSignal goes high. While runSignal is low, the input buffer and parameter registers will only accept write commands from the host PC. During this time, any writes attempted by the user logic to the output buffer or parameter registers will be ignored. When the user's software application calls sendRun, the API controller will raise runSignal. After runSignal goes high, the user's logic can process the input data and write to the output buffer and parameter registers. While runSignal is high, any attempts by the user's software to write to the input buffer or parameter registers will fail. When the user's circuit has completed computation, it will raise resetRunSignal. In response, the API controller will lower runSignal and return write control of the input buffer and parameter registers to the host PC. The user's circuit should also monitor runSignal while it is running in case the user's software application aborts execution.

3 Lessons Learned

Taking a step back for a moment, the overall goal of this project is to make FPGAs more accessible. We believe that this will encourage more developers to incorporate hardware accelerators into their applications. As part of this, we hope that this API will be extended to run on more FPGA platforms and with more communication protocols. Towards this end, we would like to share some of the lessons learned during the development of our initial prototype communicating over gigabit Ethernet. Perhaps most importantly, while it is relatively straightforward for the hardware on the FPGA to send or receive data at full bandwidth, the same cannot be said for the host PC. 1 gigabit per second only corresponds to the FPGA producing or receiving one byte per clock cycle at 125 MHz. Designs mapped to modern FPGAs can run at four times that clock rate, so it is relatively easy to build circuits to meet this timing requirement. On the other hand, even when transferring maximum sized Ethernet frames, 1 gigabit per second requires the host PC to produce or receive over 80,000 packets per second.

We took four main steps to minimize the CPU load and maximize the performance. First, it is essential to use I/O completion ports so that the OS can handle the transfers asynchronously. Without I/O completion ports, the API simply spends too much time spinning. Second, interrupts should be moderated to reduce the number of system disruptions. Rather than raising an interrupt each time a packet is received, the system can bundle multiple requests together. On the other hand, the high throughput that asynchronous I/O and moderated interrupts provide makes buffering very important. Through empirical testing, we found that it was necessary to provide space for at least 500 incoming packets to avoid dropping packets when running near 1 Gbps. This additional space is necessary to handle messages that arrive while the API code is context switched out and not actively running. Lastly, it is essential to find drivers that are very efficient. Our current implementation for gigabit Ethernet piggybacks on the Virtual Machine Network Services driver built into Virtual PC.

4 Conclusions

FPGAs are capable of exploiting such massive parallelism that they can be a disruptive technology. One FPGA board may be capable of replacing racks and racks of conventional processor-based machines. Not only might an FPGA-based implementation be faster than hundreds of processors, such a system would be easier to maintain and only require a small fraction of the power. However, for FPGAs to be used in real-world deployable systems, they need to be more accessible to a wider range of programmers. Our hope is that with better interfacing and circuit development support, FPGA-based systems will open a new world of possibilities.

FPGA-Accelerated Processing of Network Data

Rene Mueller
ETH Zurich

Ken Eguro, Paul Larson
Microsoft Research

Abstract

Emerging large scale multicore architectures provide abundant resources for parallel computation. In practice, however, the speedup gained by parallelization is limited by the fraction of code that inherently needs to be executed sequentially (Amdahl's Law). Commonly encountered examples are I/O operations such as network or disk access and object serialization, e.g., marshalling of arguments in a remote procedure call. In this work, we study acceleration by offloading sequential processing to a custom hardware circuit in an FPGA. The FPGA is placed in the data path, i.e., between the network interface and the CPU.

As a use case we investigate acceleration of the processing of network data that is exchanged between the Microsoft SQL Server and its clients. Frequently occurring requests are offloaded to a hardware accelerator whereas all other requests bypass the accelerator and are handled in the existing software stack. The object structures resulting from parsing and unmarshalling of the requests are copied into the memory space of the host system where they can be directly accessed by the DB engine.

The research problem is the automatic generation of hardware logic for a given network or serialization protocol. This involves defining a specification language for the protocol itself and a language for the semantic actions. The latter is used to define how the memory objects have to be created.

1 Introduction

Indubitably, future system architectures will consist of multiple cores. The big problem that needs to be addressed by the software community is how to efficiently make use of the additional resources. Most research is focused on parallel algorithms and cache-conscious implementations. However, obtaining large performance improvements by just using multicore architectures alone is difficult [1]. In particular, the speedup that can be obtained is limited by the inherently sequential fraction of a program. This is known as Amdahl's Law [2]. Its statement is the following: If by optimization (multicore, etc.) the parallel fraction f of a program experiences a speedup of S the speedup of the overall program is

$$\text{Speedup} = \frac{1}{(1-f) + {}^{f}/_{S}}.$$

Clearly, for $S \to \infty$ the speedup is bound to ${}^{1}/_{1-f}$ by the sequential fraction $1-f$.

In practice, the sequential fraction of a program involves I/O operations such as disk and network access, i.e., serialization of data. In this work, one particular type of serialization and deserialization is considered; object marshalling and unmarshalling in *Remote Procedure Calls (RPCs)*. RPCs play an important role in modern distributed and networked systems. To that extend, minimizing overhead and communication cost is crucial. Our approach uses an FPGA that is placed in the data path between the network interface and the host interface as illustrated in Figure 1. The sequential protocol handling is offloaded to an FPGA, hence, reducing the work to be performed by the CPU cores.

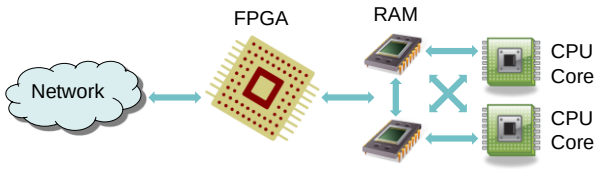


Figure 1: FPGA in data path between network and CPU

Multiple protocol handling engines can be instantiated on the FPGA allowing concurrent processing of multiple user requests. The unmarshalled

objects are written back into main memory via DMA transfers. The data structures then can be directly used by the parallelized application program.

In the next sections we illustrate the use case our work is based upon. In Section 3 We describe our approach to automatically generate hardware circuits out of protocol specifications before we conclude in Section **Error! Reference source not found.**

2 Use Case: TDS Processing in Microsoft SQL Server

Like other database management systems Microsoft SQL Server uses a complex network protocol for communicating with the database clients. In SQL Server the *Tabular Data Stream (TDS)* protocol is used. One particularly important subset of the protocol is messages for RPCs. These RPC occur when a database client invokes a *Stored Procedure* on the server through an ODBC connection.

For example, consider the following invocation of a Stored Procedure with an integer and a string argument:

```
EXEC Broker_Volume
   @topk = 2,
   @sector_name = 'Zurich'
```

This call is translated in a binary TDS message by the client-side driver. The message (Figure 2) is then sent to the server. The message contains next to the arguments also the name of the invoked procedure as a variable length string.

2.1 Traditional Software Approach

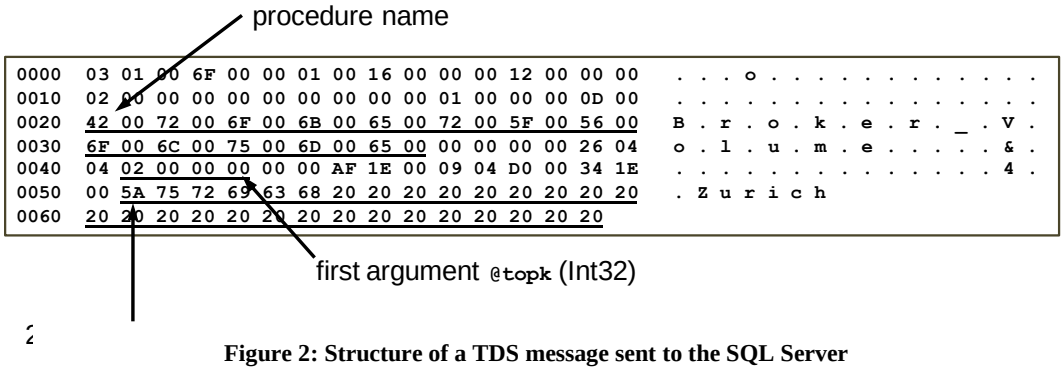


Figure 2: Structure of a TDS message sent to the SQL Server

Parsing this message is not difficult but, nevertheless, consumes CPU resources that could otherwise be spent for the actual query processing.

2.2 Hardware Acceleration using FPGAs

By inserting an FPGA into the data path the parsing of the data can be offloaded. A possible system architecture is depicted in Figure 3.

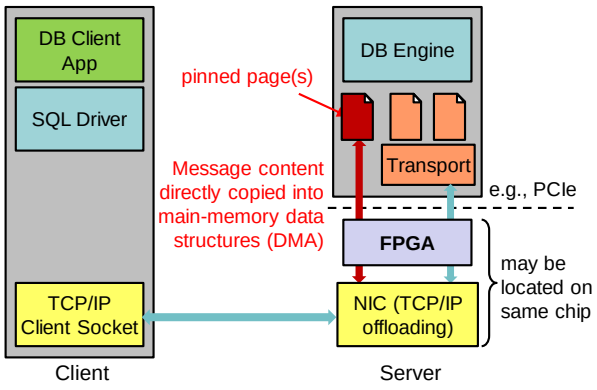


Figure 3: Architecture with protocol offloading to an FPGA

The low-level Ethernet protocol and the TCP/IP stack are also handled on the FPGA. The Ethernet MAC hard-IP cores found on modern FPGAs such as the Virtex-5 can be directly used. For the TCP/IP stack, existing soft-IP cores can be added to the design. The protocol engine is implemented as a custom circuit on the FPGA. In that circuit the serial data stream is parsed and the corresponding data structures are created in the on-chip memory (BRAM). The FPGA itself is placed on a PCI Express board that is inserted into a traditional database server system. The created data structures are then copied using DMA transfers to the main memory of the host system and the database application notified that new data has arrived. The communication through PCI Express also requires a driver stack on the server side, in particular, memory pages must be pinned in order to allow DMA transfers. This results in changes

of the memory system of the database server application.

2.3 Implementation of the Protocol Engine in FPGA Hardware

The protocol hardware is essentially a pattern matching engine of known RPC message types. For example, the FPGA can implement a simple finite state automaton for each RPC request it can handle. Because the message format for the underlying RPC data is small these state automatons are very compact. For efficiency and space reasons it makes sense not to handle every possible RPC request type on the FPGA. Instead, only the most frequent or most time-consuming requests (for parsing), i.e., the *heavy hitters*, are offloaded to the FPGA. The handled RPCs depends on the given query load. All remaining RPC requests bypass the offloading engine and are forwarded to the server where they are processed in the conventional software stack of SQL Server. This allows us to trade complexity/chip area vs. functionality.

The protocol handling engine that is able to process a certain set of calls can further be replicated on the chip multiple times. This allows processing multiple requests sent by several users concurrently.

3 Automatic Generation of Circuits out of Protocol Specifications

The research goal is to automate the generation of hardware circuits from given protocol specifications. Similar to existing generator tools for parsers (Yacc) or lexical scanners (Lex) a tool is currently being developed that takes a protocol specification that is annotated with semantic actions as input and produces either VHDL or Verilog code. The key problems to be addressed are the expressiveness of the protocol language and the language of the semantic actions.

3.1 Overview

The generator tool generates a hardware circuit that accelerates the protocol handling while at the same time can be easily integrated into the existing software stack. In this work, we study the integration into C/C++ software, i.e., data structures must be aligned in the FPGA such that they remain compatible with the layout of C types and C++ classes. The tool workflow is illustrated below:

The user specifies the network protocol and the corresponding semantic actions that are necessary to create the object structures that are later used in software. Thus, in order to generate the memory layout of the objects the tool needs to know about the high-level language type information, i.e., the C++ classes, which are also have to provided to the tool (Figure 4).

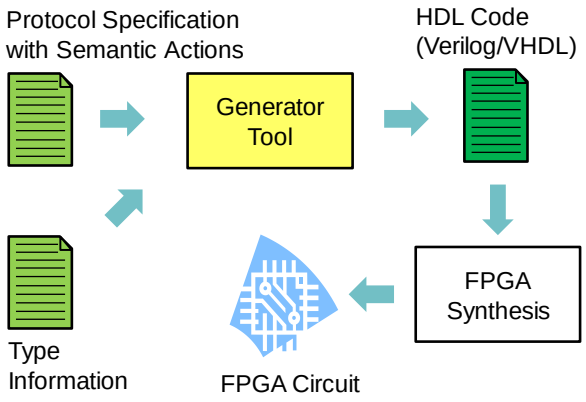


Figure 4: FPGA Circuit created out of a protocol specification and type information.

3.2 Protocol Specification Language

In simple scenarios, protocol specifications can correspond to regular languages. Hence, the specification then represents a regular expression. The implementation on the FPGA therefore is a finite state automaton. More complex protocols that use some form of nesting are no longer regular; they are context-free languages. In this case, the language can be expressed in Backus-Naur Form (BNF) and can be recognized by a parser on the FPGA.

The expression language for the semantic actions has to be expressive enough to create and manipulate data structures in memory. For space reasons on the chip and efficient implementation it should not be over expressive. For example, there is no need to support recursion or iteration.

3.3 Example

Although the work on both the specification and action languages is still ongoing we provide an example of a protocol specification for illustration purposes below. The example shows the pattern expression and the corresponding action code in {: :} for the `Broker_Value` procedure shown in Section **Error! Reference source not found.**. The language is similar on the Yacc grammar. `RPCCall` and `Int32Argument` and `StringArgument` correspond to C++ types. The Yacc language is extended, for example, to capture variable length arrays such as strings (the second argument).

From this specification and the C++ type information the generator tool produces a hardware circuit in VHDL/Verilog that generates the data structures from a

data stream received over the network. This memory block is then copied to the host memory via DMA where it can be accessed by the CPU.

```
BrokerVolumeRPC ::= BVHeader
          ArgTopK:a1 BVBlock ArgName:a2
{: %% = new RPCCall(2);
   %%.id       = 7;
   %%.arg[0]   = a1;
   %%.arg[1]   = a2;
 :}

ArgTopK ::= INT32:i
{:
   %% = new Int32Argument();
   %%.value = i;
 :}

ArgName ::= UINT16:len STRING(len):str
{:
   %% = new StringArgument();
   %%.length = len;
   %%.string = str;
 :}

BVHeader ::= 0x03 0x01 0x6F ...
BVBlock  ::= 0x00 0x00 0xAF ...

// Terminals
terminal signed (0 to 31) INT32;
terminal unsigned (0 to 31) UINT32;
terminal STRING(len) (0 to len) CHAR;
```

3.4 Hardware-Software Interface

The object layout on the FPGA has to be chosen such that it is consistent once copied to the host memory. For efficiency reasons no pointer relocation performed by the host CPU after that memory block is copied into the host memory. Thus, all object pointers must be setup correctly in the FPGA. Furthermore, for object allocation and deallocation the heap management on the host system and the FPGA need to be kept synchronized. Additional dynamic information is needed on the FPGA for heap management and setup of pointers to vtables (virtual method table). This information is provided by the CPU and updated when necessarily through mapped registers.

3.5 Example continued

Figure 5 shows the object diagram of the resulting object structure. In the following it is assumed that the classes **`Int32Argument`** and **`StringArgument`** have virtual methods and, hence, have pointers to their vtables.

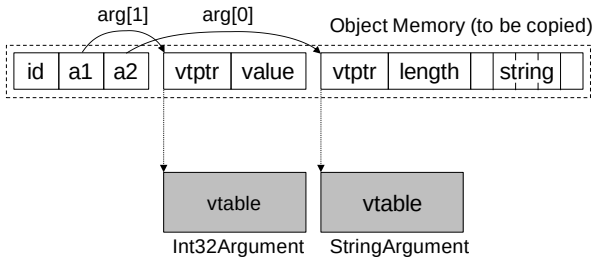


Figure 5: Object structure created on FPGA and copied to host memory

The object structure is aligned using conventional alignment rules. The FPGA circuit uses a set of registers that hold the pointer location of the vtables and the destination address of the top-level object in on the heap in the host memory. This information is used by the FPGA circuit to properly align the object structures.

4 Conclusions

The proposed solution is a non-invasive attempt to offload the processing of I/O for marshalling and object serialization to a custom FPGA circuit. A generator automatically produces the digital logic out of a protocol specification and the corresponding high-level language types.

The generator tool is currently being implemented. Both languages for protocol and semantic actions are currently investigated for the necessary expressiveness that they can be used in real application scenario such as in the Microsoft SQL Server. Next, the resulting circuits have to be evaluated and the resulting speedup compared to a traditional CPU-based implementation measured.

5 Bibliography

1 Laurus, James. Spending Moore's Dividend. *Communications of the ACM 5(52)* (2009), 62-69.

2 Amdahl, Gene. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In *AFIPS Conference Proceedings* (1967), 483-485.

A Full-System Concolic Simulator for Real-time System

Weiqin Ma
Texas A&M University

Alessandro Forin
Microsoft Research

Abstract

In this project, we add an x86 CPU model to the Giano full-system simulator which can run x86 programs in real-time. We also develop a concolic execution engine to Giano, to run programs in a mixed symbolic-concrete (concolic) way. Based on the concolic engine, we perform a case study to detect data races in a multi-threaded program.

1 Introduction

The first goal of this project is the real-time simulation of the x86 instruction set. The second goal is to create a concolic module for a full-system simulator such as Giano, to execute applications in a symbolic way but guided by concrete inputs.

Most existing monitoring techniques are based on direct execution or dynamic binary translation for pursue of performance. Such techniques create inconsistencies in the timing behavior of the program. The Giano simulator provides a facility to monitor and adjust the program execution speed, to maintain real-time consistency. As a result, the executable of the software program does not change its temporal behavior.

Existing monitors also lack a systematic testing approach and tools to verify the correctness of the simulator. We test the correctness of our CPU model by comparing the execution results of a large, comprehensive test set on an "oracle" machine against the results on the simulator. We especially test the boundary values for the instructions. We plan to automatically generate the tests using formal instruction specifications.

The first step is to add the x86 CPU model to Giano, as indicated on the left side of Figure 1. The second step is to build a concolic engine to run the program in symbolic manner. The resulting simulator architecture is shown in Figure 1.

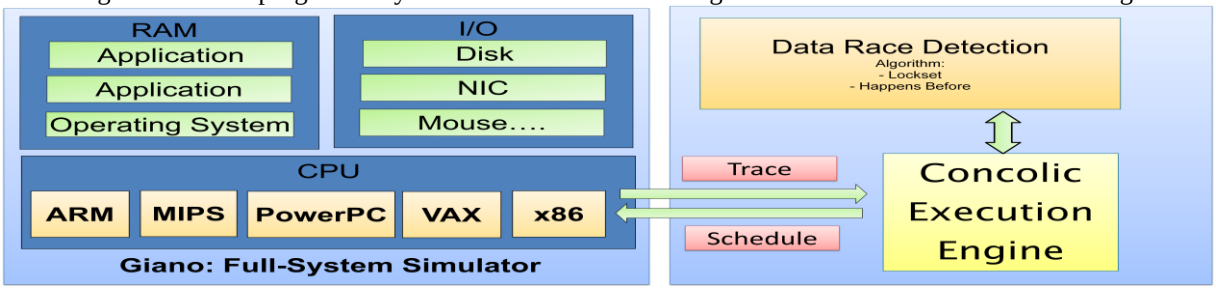


Figure 1: Simulator Architecture

The concolic engine includes symbolic execution using symbolic values and concrete execution using concrete values. The concolic engine needs to generate path conditions and to solve the path conditions using a constraint solver, as shown in Figure 2. We generate the path conditions based on the control flow graph (CFG), and use the data flow graph (DFG) to prune the irrelevant paths and reduce the path explosion.

We use Z3 as the constraint solver to check the satisfiability of the path conditions and to generate the set of input values. The input values are then used to concretely execute the program.

In the case study, we trace and analyze a multi-threaded application in a real-time embedded system. Our motivation for operating at the binary level rather than at the source level is that programs are actually changed by the compiler optimizations and by the out of order instruction execution in modern architectures. Our goal is to trace the binary program at the machine instruction level and find the concurrency errors using concolic execution.

Data races are one type of Heisenbugs which include data race (race condition), live lock, dead lock etc. If two memory accesses conflict concurrently, we have a data race, as show in Figure 3. A data race has three conditions: the instructions must target the same location (a shared variable), the instructions must not be both reads, and the instructions cannot both be synchronization operations.

We use the following scheme for data race detection. First, we can infer the sequential program execution of a concurrent program by getting the [Min,Max] program sequence. We can eliminate equivalence traced statements using existing reduction theories. Finally, we can compare the interleaved concurrent program execution with the sequential program execution to find whether the result is the same.

Even though we operate at the machine level we do not need to lose track of all software abstractions. We can use instruction introspection to recover those abstractions that are still relevant. For instance, on the x86 we use the value of the CR3 register to identify different processes. We use the ESP register to identify different stacks which means different threads. Also we record and trace the shared variables by tracing the memory locations which are accessed by more than one thread.

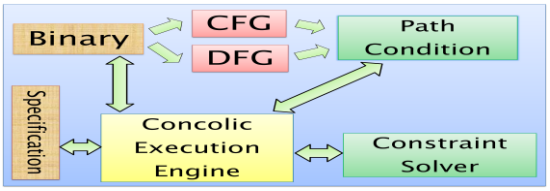


Figure 2: Architecture of Concolic Engine

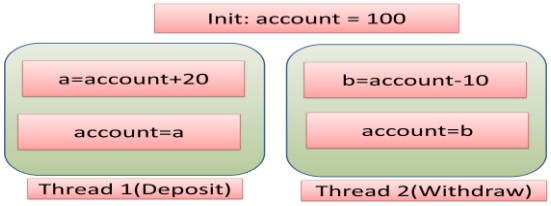


Figure 3: Data Race Example

2 Project Demo

We demonstrate (a) that the Giano simulator is real-time and (b) how we test the instruction set against the oracle machine. We use the Doom video game, with audio, to show the real-time property. This demo currently uses an existing ARM CPU model; we will eventually replace it with our x86 CPU model. We also show the xml-based configuration of the Giano simulator. This makes it easy for users to compose a simulator using existing different modules.

We show the testing of the simulator using tests generated by an oracle. As show in Figure 4, the test engine (on the oracle) sends different instructions with boundary values to the Giano simulator. The simulator runs the instruction and shows any discrepancy between the results on the oracle test machine and the simulator, as shown in Figure 5.

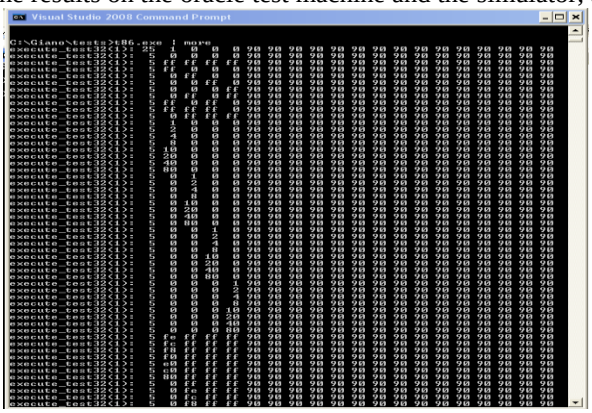


Figure 4: The test engine sends different tests, one per line in the log above

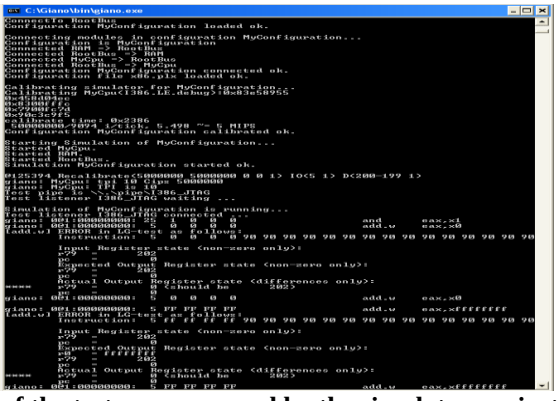


Figure 5: The results of the test are compared by the simulator against the oracle's results

Specification Mining in Real-time Embedded Systems

Wenchao Li
University of California at Berkeley

Alessandro Forin
Microsoft Research

Abstract

Software and hardware systems are often built without detailed documentations. The correctness of these systems can only be verified as well as the specifications are written. The lack of sufficient specifications often leads to misses of critical bugs, design re-spins, and time-to-market slips. In this project, we address this problem by mining specification dynamically from simulation traces. Building on algorithms for pattern mining, we propose a novel technique that mines specifications with timing constraints and we apply it to a number of practical cases. Timing constraints are expressed as either inequalities or distributions. The technique applies to both time-labeled and unlabeled traces. Specifications mined from unlabeled traces can be automatically synthesized using our PSL-to-Verilog compiler to achieve zero-overhead runtime monitoring. Specifications mined from labeled traces can be used to pinpoint sources of error. In this work, we focus on embedded software, digital circuits, and network protocols, but any ordered trace of events is amenable to this analysis.

Introduction

We try to answer two common challenges in verification – "Did I miss any specification in my verification process?" and "Where should I look in my error trace?" The first question is closely related to *assertion coverage*. In assertion coverage, we check whether the verification test suite has exercised some specific functionalities of the design. However, assertions are still supplied manually. As a result, engineers often face the question of when they can stop writing assertions. We address this problem partially by dynamically mining recurring patterns from existing simulation traces. These patterns can then be examined by the engineer to see whether they match the designer's intent and check with further verification. The intuition is that frequent patterns are likely to be true. Hence, in the context of *mining for verification*, our tool takes a *trace* and optionally a user-defined *event definition* as input, and generates a set of *behavioral patterns* that are *almost always true* in the trace as output. A trace is a sequence of events ordered by the time of occurrences. Events in this case here can be the valuation of a set of signals in digital circuits, signatures of function calls, or network packets. Given the trace, we match it to a library of parametric patterns. The matching algorithm will be discussed in more details in the following section. Once the parametric patterns are instantiated, we rank them according to some relevance metrics that we found useful empirically. The second question that we are trying to answer is essentially *trace diagnosis*. Given a normal trace and an error trace, the goal is to first understand what goes wrong in the error trace and then locate the source of error. We use our specification mining algorithm as a subroutine and look for *difference patterns* between two traces. These are patterns that exist in one trace but not in the other, or patterns that exist in both traces but with different timing bounds. After the difference patterns are found, a localization procedure is applied to pinpoint the potential source of error.

Specification Mining

Our main contribution is the inclusion of *time* in specification mining. Timing constraints are ubiquitous in an embedded environment. For example, in an *x-by-wire* automotive system, every task has an associated deadline. A task can be a signal for turning off the air conditioner, or a signal for breaking. Missing the deadlines for some of these tasks can have catastrophic effects. On the other hand, specifications are sometimes written in a way to just ensure logical correctness, but not timing correctness. For example, the Linear Temporal Logic (LTL) formula "*always* (request → *eventually* grant)" says every "request" will be eventually followed by a "grant". However, depending on the environment in which the design is deployed, the latency for when the grant signal is received can vary, and some of

them may be unacceptable. This mirrors the fact that in software, it may be fine as long as every "lock" is followed by an "unlock". But the same *alternating pattern* is not sufficient for mining useful specifications in a *(real-time) embedded* system.

We extend the algorithm for mining alternating patterns to include timing constraints on the events in these patterns. An alternating pattern can be expressed as a parametric regular expression *(ab)**, where *a* and *b* are parameters that can be instantiated with actual events. Without timing constraints, this alternating pattern can be used to express specification such as "every request is followed by a response." With timing constraints (timing bounds for example), we can express richer specifications, such as "every request is followed by a response within 3 cycles, and two requests are separated by at least 5 cycles." As a feasibility study, we implement the algorithm in the naïve way by maintaining a 2D table for all possible instantiations along with counters for each combination and timing bound recorders. Each time a new cycle is parsed, we iterate over the symbols and for each symbol *s*, the corresponding row (*s*,*) and column (*,*s*) are checked to see if these patterns are still true. If they are true, the counts and time bounds are updated. The online aspect of operating on cycles as they come in makes mining this simple pattern appealing. The algorithm has a runtime of O(*nkl*) where *n* is the total number of events, *k* is the maximum number of events at a cycle, and *l* is the length of the trace. We further address the scalability issue by clustering traces according to their modules so that we can reduce both the storage requirement (significantly) and runtime. However, inference rules may be required to compose local patterns to form end-to-end specifications. In addition, we allow various degrees of *imperfectness* in the trace. For example, the pattern is always true except for the last occurrence. Currently, we mine only alternating patterns with timing bounds. We plan to extend both the pattern library to include more complex patterns and the timing constraints to include richer constraints.

Experimental Results

A prototype of the algorithm is written in Perl. We apply the algorithm for learning difference patterns to a full MIPS core that has approximately 18000 signals (wires and registers). We trace only the control signals, which results in approximately 1500 events. In the absence of event definitions, we treat the value of a signal as an event. For example, if a signal *A* changes value from 0 to 1 at time t_1, we record the event *A* with tag t_1. If *A* changes from 1 to 0 at a later time t_2, we record the event ~*A* with tag t_2. We obtain two simulation traces, a correct one from the current design ran for about 6.7 million clock cycles, and an error trace from a previous version with a known design bug and ran for about 2.35 million clock cycles. The bug can be described as the signal TLB_ERR going low too soon for exception handling to finish. The objective of the experiment is to evaluate whether the difference patterns mined are useful to localize the error.

The tool mines about 200 difference patterns with half of them due to differences in timing bounds. We rank the two sets of difference patterns separately. For the set containing patterns that exist in one trace but not the other, we rank them by the number of occurrences. For the set containing patterns due to differences in timing bounds, we rank them first by time of first divergence and then tie break by occurrences in the normal trace. The top candidates in the "untimed" set capture the effects of bug, such as constant flagging of exceptions. The top candidates in the "timed" set localize to signals that are very close to TLB_ERR. The tool does not find TLB_ERR directly because the original bug is triggered by a specific combination of values across a few cycles. This is not considered in the current implementation when event definition is not available. We are currently looking at the possibility of synthesizing such combination by leveraging well-studied techniques in the domain of sequential pattern mining.

Ongoing work

We are currently developing algorithms that can efficiently mine all chain patterns $(a^+b^+c^+\ldots)^*$. Our technique may also be useful in anomaly detection. One possible direction is to combine machine learning techniques such as Principal Component Analysis (PCA) and classification to detect abnormal behaviors. The patterns mined can be treated as features and used as input to the aforementioned techniques. We are also applying our pattern mining tool to embedded software, with the aid of the Giano simulator.

Custom Floating-Point Units

Zhanpeng Jin
University of Pittsburgh

Richard Neil Pittman
Microsoft Research

Abstract

Multimedia and communication algorithms in the embedded system domain often make extensive use of floating-point arithmetic. Due to the complexity and expense of the floating-point hardware, final implementation of these algorithms are usually carried out using floating-point emulation in software, or by conversion of the floating-point operations to fixed point operations. This study presents the design and implementation of custom floating-point units making use of the flexibility and reconfigurability of FPGAs. In the eMIPS architecture, such custom floating-point units can be dynamically configured, loaded, and executed when needed by software applications. We investigate the optimization strategies for area, power, speed, and duty cycle of the custom units. We show how to construct a set of functional modules that are optimized on a per-application basis. According to the analysis of the program characteristics and design specification, the system is able to dynamically configure its own customized "optimal" floating-point units.

1 Introduction

Most of the current available microprocessors are implemented using fixed instruction sets, no matter what instruction set architecture (ISA) they use, either Reduced Instruction Set Computer (RISC) or Complex Instruction Set Computer (CISC). When designing instruction sets, computer architects attempt to capture all the instructions which are necessary to cover the largest domain of potential applications, while such way might bring remarkable overhead in size, cost, and power. Despite all these efforts, the goal of implementing an "optimal fixed instruction set architecture" is theoretically impossible because the design space of applications to which the designers apply general purpose processors evolves constantly [1]. In this case, especially for the emerging embedded market, such an "optimal general purpose" microprocessor is inefficient and underutilized when the majority of applications never use a large subset of the capabilities it provides. Thus, a popular solution is to use custom microprocessors with reduced instruction sets but with customized instructions added specifically for intended application space at some certain execution phases.

Our current work in the eMIPS project [1], addresses such reconfigurable computing challenges and depicts a promising vision for future computation-efficient embedded systems by using "dynamically extensible processors". eMIPS offers the infrastructure to allow for the kind of flexibility and extensibility possible through the use of Field Programmable Gate Arrays (FPGAs). The FPGA is partitioned into sections containing a standard fixed logic processor core with interconnects to reconfigurable regions termed "Extensions" that contain customized instructions and functionality that loads, modifies and enables while the fixed counterpart continues to execute without any interruption. In this way, the dynamically extensible processor, using a set of Extensions from which it can draw, adapts to the changing application needs in the field [1]. The Extensions are able to take the form of any optimized or even new instructions developed to meet certain application needs of the market. For more details about eMIPS architecture, please refer to [2].

Floating-point (F-P) arithmetic, although extremely common in the general purpose computing market, was rarely used in embedded systems world until recently. A number of communication and multimedia algorithms are designed and simulated using floating-point arithmetic, but the implementation platforms for such algorithms often leave out any hardware floating-point unit in favor of software emulation or float to fixed point conversion [3]. A variety of research efforts are considering field-programmable gate arrays (FPGAs) as a means to accelerate floating-point computations using their well-proved flexibility and reconfigurability [4] [5] [6]. Thus, it seems appealing to deploy floating-point arithmetic into the eMIPS architecture, as a mean to advance floating-point intensive applications in the embedded market. For instance, one clear advantage is that the reconfigurable floating-point extensions can be loaded only if and when software applications use them. The architectural diagram of the resulting design is shown in Figure 1. The basic

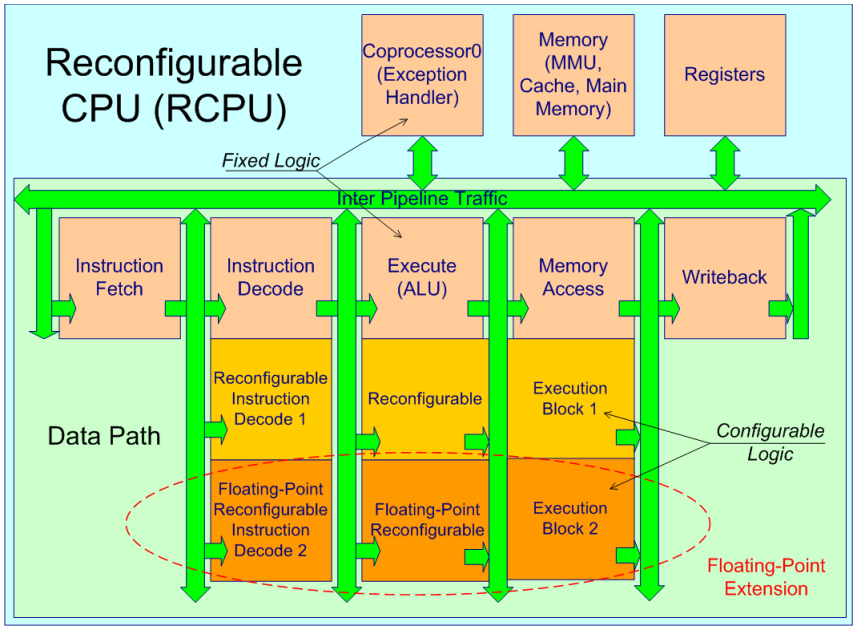


Figure 1: Reconfigurable Processor with Configurable Floating-Point Extension

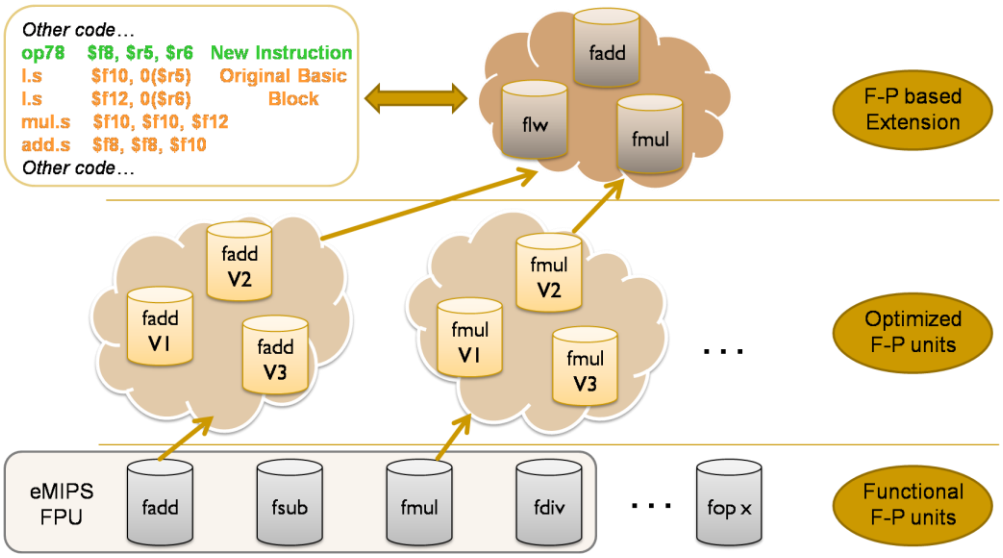


Figure 2: Proposed Design Strategy for Custom Floating-Point Units

idea is simple, but the implementation of an efficient and correct FPU is an extremely difficult, involved and time consuming task. In addition, mapping difficulties occur due to the inherent complexity of floating-point arithmetic [7]. The increasing demand of application-specific floating-point arithmetic data paths presents the challenge of accommodating several floating-point functional modules in the limited resources available [8]. This makes considerations for cost-effectiveness a priority. Many recent studies have explored opportunities to improve the floating-point performance on FPGAs by optimizing the device architecture. Beauchamp et al. [9] present three architectural modifications that make floating-point operations more efficient on FPGAs, including an embedded floating-point multiply-add units and variable length shifters. Chong et al. [10] propose multi-mode embedded FPUs implemented on a single FPGA, configuring each unit to either perform a different task or to collectively build massively parallel circuits.

The main contributions of this project are three-folds, as depicted in Figure 2:

1) Implementation of IEEE-754 compliant, modular floating-point functional units (e.g., fadd, fsub, fmul, fdiv, fsqrt, etc.) using the standard MIPS floating point ISA.
2) Investigation of the floating-point optimization strategies for area, power, speed, and duty cycle, and delivering of a set of optimized implementation solutions.
3) Procedures for analyzing an application characteristics, identifying corresponding performance requirements, and then dynamically constructing and configuring the optimal floating-point functional modules and deploying them as eMIPS extensible instructions.

Table 1: Synthesis Results of Floating-Point Functional Units

	Registers	LUTs	LUT-FF pairs	IOBs	DSP48E
eMIPS-FPU	2000 (2%)	3727 (5%)	1774 (44%)	113 (17%)	18 (28%)
-fadd	844	1341	711	113	0
-fmul	833	1534	1683	113	18
-fdiv	688	1127	1269	113	0
Available	69120	69120	3953	640	64

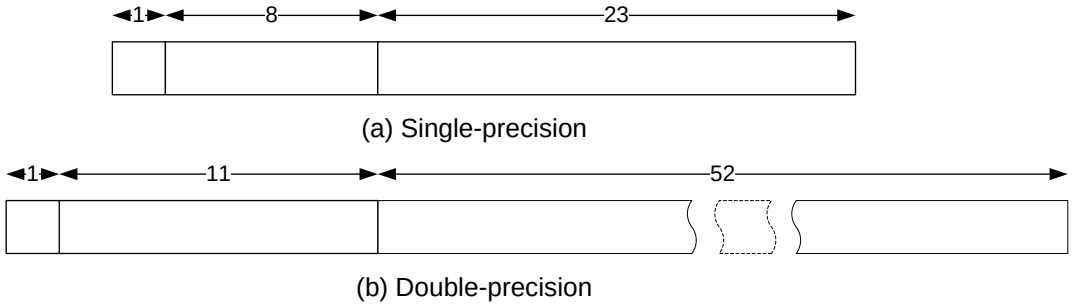


Figure 3: IEEE Floating-Point Numbers

2 Floating-Point Format Representation

Floating-point numbers have the advantage of being able to cover a much larger dynamic range compared to fixed-point numbers. However, they also bring much more complexity for the implementation in hardware.

The IEEE-754 standard [11] [12] specifies a representation for single and double precision floating-point numbers. It is currently the standard that is used for real numbers on most computing platforms. Floating-point numbers consist of three parts: sign bit, mantissa, and exponent. In the IEEE-754 format, the mantissa is stored as a fraction (*f*), which is combined with an implied one to form a mantissa (1.*f*) such that the mantissa is multiplied by the base number (two) to an exponent *e*, as shown in equation (1) and (2), single and double precision, respectively [9] [13]

$$X = (-1)^s \cdot 1 \cdot f \cdot 2^{E-127} \quad (1)$$

$$X = (-1)^s \cdot 1 \cdot f \cdot 2^{E-1023} \quad (2)$$

The IEEE standard specifies a sign bit, an 8-bit exponent, and a 23-bit mantissa for a single precision floating-point number, as shown in Figure 3(a). A double precision floating-point number has a sign bit, an 11-bit exponent and 52-bit mantissa, as shown in Figure 3(b). Since the mantissa is normalized to the range [1, 2) there will always be a leading one in the mantissa. By implying the leading one instead of explicitly specifying it, a single bit of storage could be saved, but it does raise the complexity of floating-point implementations.

Table 2: Floating-Point ADD Extension Definition

```
ENTRY(fadd_test)                    // void fadd_test(UINT32 *a0, UINT32 *a1, UINT32 *a2);
        nop
        l.s     $f0,offset($a0)     // The contents of the word in memory is loaded into F-P register f0
        l.s     $f1,offset($a1)     // The contents of the word in memory is loaded into F-P register f1
        add.s   $f2,$f1,$f0         // The contents in f0 and f1 are arithmetically added
        s.s     $f2,offset($a2)     // The contents of the word in F-P register f0 is stored back into the memory
        jr      $ra
        nop
END(fadd_test)
```

3 Implementation

In the current design, following the state-of-the-art algorithmic design method, we implement four basic floating-point operations using Verilog HDL: floating-point addition, subtraction, multiplication, and division. Figure 4 shows the data path for a floating-point addition, which typically consists of five stages – exponent difference, pre-alignment, addition, normalization and rounding [14]. Figure 5 shows the data path for a floating-point multiplier and the core Radix-4 Modified Booth Encoded (MBE) Wallace multiplier was used in our design (shown in Figure 6). For more details on floating-point arithmetic algorithms, please refer to [15] [16] [17]. The first version of the basic floating-point arithmetic units was implemented and synthesized in Xilinx ISE 10.1 targeting a Vertex-5 FPGA. The preliminary synthesis data is shown in Table 1.

We implement the floating-point unit (FPU) as a eMIPS extension, that is, the FPU can be loaded and executed by the eMIPS system as a reconfigurable module. A semantic definition for one such type of block is shown in Table 2.

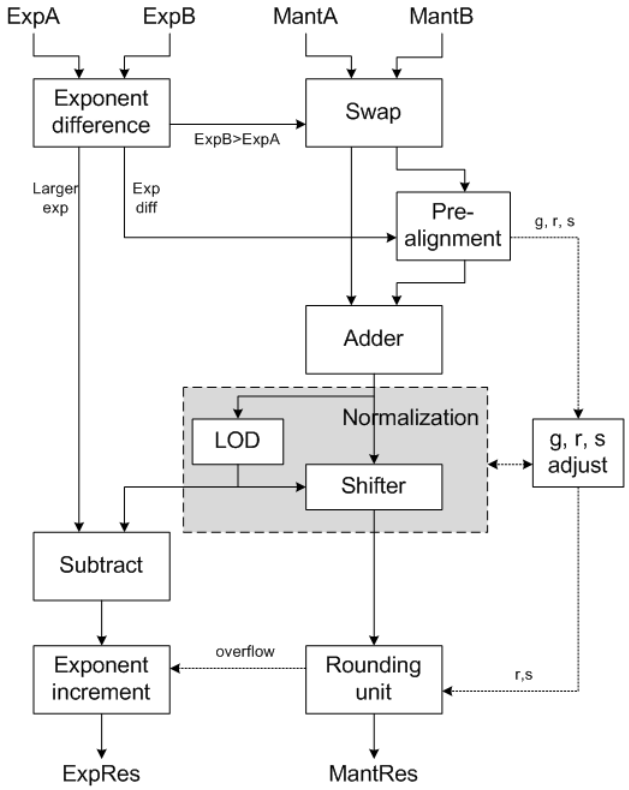


Figure 4: Floating-Point Adder Datapath

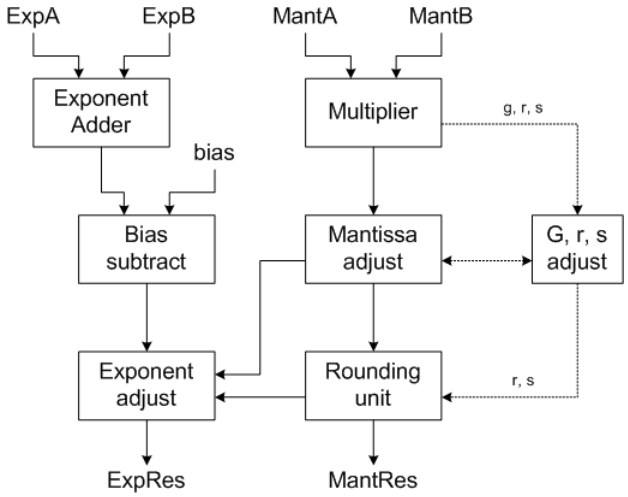


Figure 5: Floating-Point Multiplier Datapath

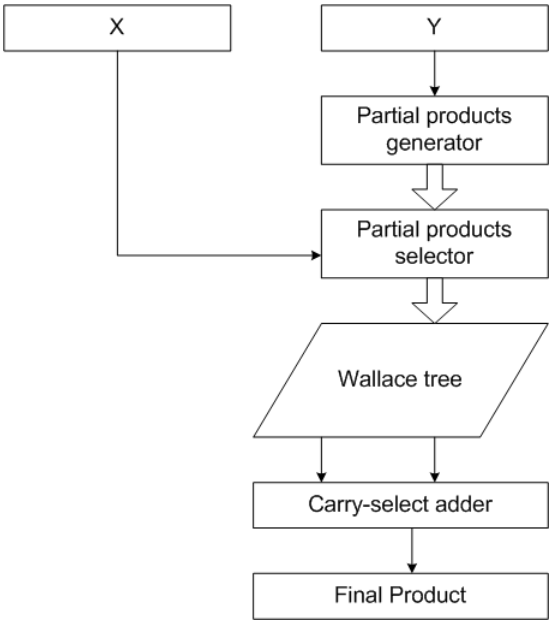


Figure 6: Booth Wallace Multiplier Structure

4 Testing and Verification

Testing and verification of the floating-point unit have long presented a unique challenge in the field of processor verification, due to F-P unit inherent much complicated operations, such as rounding and normalization. The particular complexity of this area also stems from the vast test space, which includes many corner cases that need to be targeted, and from the intricacies of the implement of floating-point operations.

Main stream test generation tools, such as Genesys [18] and AVPGEN [19], offer some control for F-P test generation. However, their lack of focus and internal knowledge of the F-P domain render them inadequate for providing a full solution to the F-P verification problem. Test generation is supplemented by static legacy tests and by large quantities of purely random testing.

In this study, we use the SoftFloat, a free, high-quality software implementation of the IEC/IEEE Standard for Binary Floating-point arithmetic [20]. All functions dictated by the IEEE-754 Standard are supported except for conversions to and from decimal. SoftFloat fully implements single-precision (32 bits) and double-precision (64 bits) floating-point formats as well as the four most common rounding modes: round to nearest even, round up, round down, and round toward zero.

A sample piece of generated test cases is shown in Table 3.

Table 3: Floating-Point Test Cases by SoftFloat

537bffbe	Floating-Point Operand 1
4e6c6b5c	Floating-Point Operand 2
000	Floating-Point Operations – "000": Add – "001": Subtract – "010": Multiply – "011": Divide – "100": Square Root
00	Rounding Modes – "00": Round to nearest even – "01": Round up – "10": Round down – "11": Round toward zero
537c3ad9	Expected Floating-Point Result

The aforementioned floating-point ADD extension (Table 3) including FADD, FLWC1 (load word), FSWC1 (store word) instructions, has been fully tested using such type of test cases generated by SoftFloat. The simulation results using ModelSim and Giano are printed on the console screen, as shown in Figure 7 and Figure 8.

5 Bibliography

1 Pittman, Richard Neil, Lynch, Nathaniel Lee, and Forin, Alessandro. *eMIPS, A Dynamically Extensible Processor*. MSR-TR-2006-143, Microsoft Research, Redmond, WA, 2006.

2 Forin, Alessandro and Pittman, Richard Neil. *eMIPS*. http://research.microsoft.com/en-us/projects/emips/default.aspx.

3 Karuri, Kingshuk, Leupers, Rainer, and Kedia, Monu. Design and Implementation of a Modular and Portable IEEE 754 Compliant Floating-Point Unit. In *Proceedings of Design, Automation and Test in Europe* (Munich, Germany 2006), 1-6.

4 Ho, Chun Hok, Yu, Chi Wai, Leong, Philip, Luk, Wayne, and Wilton, Steven J. E. Floating-Point FPGA: Architecture and Modeling. *to appear in IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2009).

5 Karlstrom, Per, Ehliar, Andreas, and Liu, Dake. High Performance Low Latency FPGA based Floating Point Adder and Multiplier Units in a Virtex 4. In *Proceedings of the 24th Norchip Conference* (Linkoping, Sweden 2006), 31-34.

6 Sahin, Suhap, Kavak, Adnan, Becerikli, Yasar, and Demiray, H. Engin. Implementation of Floating-Point Arithmetics using an FPGA. *Mathematical Methods in Engineering* (2007), 445-453.

7 Shirazi, Nabeel, Walters, Al, and Athanas, Peter. Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines. In *Proceeding of the IEEE Symposium on FPGAs for Custom Computing Machines* (Napa Valley, CA 1995), 155-162.

8 Krueger, Steven D. and Seidel, Peter-Michael. Design of an On-Line IEEE Floating-Point Addition Unit for FPGAs. In *Proceedings of the 12th Annual Symposium on Field-Programmable Custom Computing Machines (FCCM)* (Napa, CA 2004), 239-246.

9 Beauchamp, Michael J., Hauck, Scott, Underwood, Keith D., and Hemmert, Scott. Architectural Modifications to Enhance the Floating-Point Performance of FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16, 2 (2008), 177-187.

10 Chong, Yee Jern and Parameswaran, Sri. Flexible Multi-Mode Embedded Floating-Point Unit for Field Programmable Gate Arrays. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)* (Monterey, CA 2009), 171-180.

11 IEEE STD 754-1985. *IEEE Standard for Binary Floating-Point Arithmetic*. IEEE Computer Society. 1985.

12 IEEE STD 754-2008. *IEEE Standard for Floating-Point Arithmetic*. IEEE Computer Society. 2008.

13 Koren, Israel. *Computer Arithmetic Algorithms*. A. K. Peter, Natick, MA, 2002.

14 Hennessy, John L. and Patterson, David A. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, 2006.

15 Overton, Michael L. *Numerical Computing with IEEE Floating Point Arithmetic*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2001.

16 Flynn, Michael J. and Oberman, Stuart F. *Advanced Computer Arithmetic Design*. Wiley-Interscience, Malden, MA, 2001.

17 Ercegovac, Milos D. and Lang, Tomas. *Digital Arithmetic*. Morgan Kaufmann, San Francisco, CA, 2004.

18 Behm, M., Ludden, J., Lichtenstein, Y., Rimon, M., and Vinov, M. Industrial experience with test generation languages for processor verification. In *Proceedings of the 41st ACM/EDAC/IEEE Design Automation Conference* (San Diego, CA 2004), 36-40.

19 Chandra, A., Geist, D, Wolfsthal, Y. et al. AVPGEN - A test generator for architecture verification. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 3 (1995), 188-200.

20 Hauser, J. *SoftFloat*. http://www.jhauser.us/arithmetic/SoftFloat.html, 2002.

21 Underwood, Keith. FPGAs vs. CPUs: Trends in Peak Floating-Point Performance. In *Proceedings of the 12th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)* (Monterey, CA 2004), 171-180.

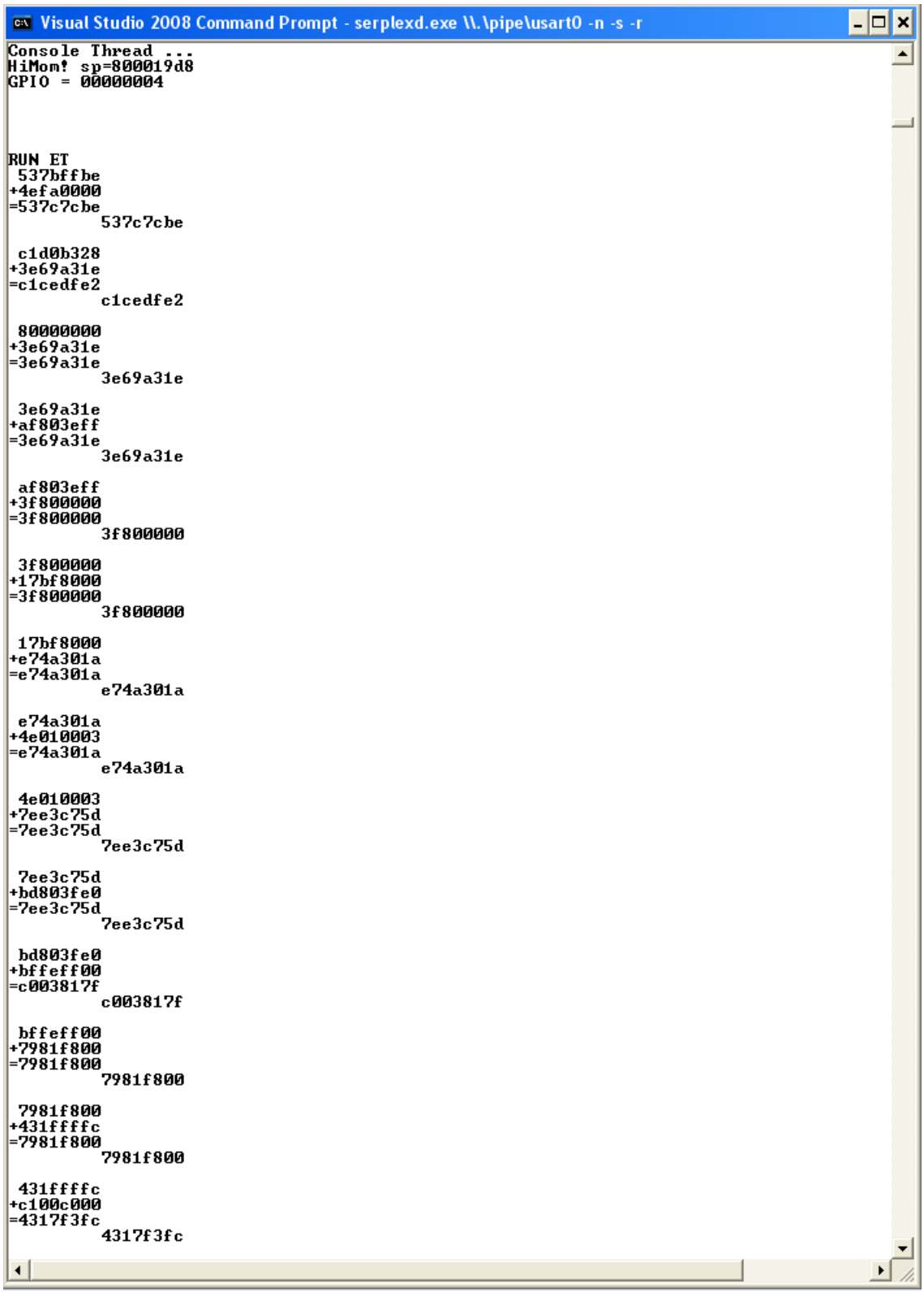


Figure 7: Floating-Point Add Extension Test Start

Figure 8: Floating-Point Add Extension Test End

M2V – Automatic Hardware Generation from Software Binaries

Ruirui Gu
University of Maryland at College Park

Alessandro Forin
Microsoft Research

Abstract

The MIPS-to-Verilog (M2V) compiler translates blocks of MIPS machine code into a hardware design represented in Verilog. The design constitutes an Extension for the eMIPS processor, a dynamically extensible processor realized on the Xilinx Virtex-4 and Virtex-5 FPGAs. The Extension interacts closely with the basic pipeline of the microprocessor and recognizes special extended instructions, instructions that are not part of the basic MIPS ISA. Each instruction is semantically equivalent to one or more blocks of MIPS code. The tool-chain involving M2V automatically executes, profiles, and patches the original binary executable to taek advantage of hardware acceleration platforms.

We are planning the first source-level release of the M2V compiler. The previous M2V version can accelerate single block cases with the supports of load and stores, interrupts, and the automatic encoding of extended instructions. At the summit we demonstrated the further development of the compiler to support self-looped basic blocks, which takes advantage of both dataflow graphs and control flow structures. The released M2V will support multiple basic blocks with the four basic control patterns and their combinations thereof.

1 Introduction

The goal of the project is to automatically generate hardware accelerators from software binaries. **Figure 9** shows the complete tool chain to automate the generation of hardware accelerators, and the role M2V plays in it. Other tools (not shown) synthesize the Verilog file, generate the configuration bitfile, and merge it with the patched binary.

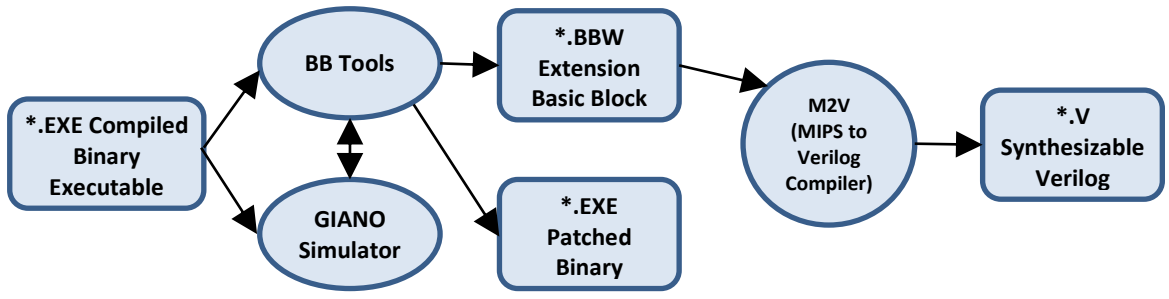


Figure 9: Tool-chain for automatically accelerating executable binary files. GIANO is a full-system simulator that executes the application and extracts basic block profiles. The BB Tools are a series of tools to select the basic blocks to accelerate and patch the binary image with the special instructions for the accelerator.

The input to the tool chain is executable binary files, which are generated by off-the-shelf compilers for the given ISA. This tool-chain restricts the code selection problem to the set of most-frequently executed basic blocks in the application. Each basic block is a directed acyclic graph (DAG), which is 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 of computation and occur at high frequency in the application. These candidates are extracted by executing the application

using the Giano full-system simulator, in concert with the data obtained via static analysis of the application binary. The profile directs the BBTools in selecting the candidate basic blocks, and in patching the binary image with the special instructions for the accelerator. M2V automatically generates the design for the hardware accelerator, which is then synthesized onto programmable logic such as FPGA boards, using the manufacturer's tools (e.g. Xilinx ISE).

This tool-chain applies to any programmable logic attached to a tightly coupled pipeline, since the tight coupling creates 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. 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. These extensions are used to accelerate the execution of an application. The machine code 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. See Figure 7 for a simple example.

The objective 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 when necessary, which ensures the reliability of the system and provides software more flexibility in scheduling the accelerator units.

2 M2V Compiler Architecture

The M2V compiler is a four-pass compiler, as shown in **Figure 10**.

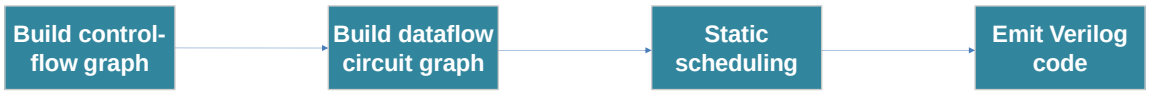


Figure 10: Four-pass MIPS instruction set to Verilog Compiler

The first pass builds the control flow graph based on the relationship between basic blocks. Besides single block cases, we are dealing with four kinds of basic control patterns among basic blocks: Sequential, Self-loop, Branch and Join, as shown in **Figure 11**. It is easy to see that these four patterns cover all possible control graphs.

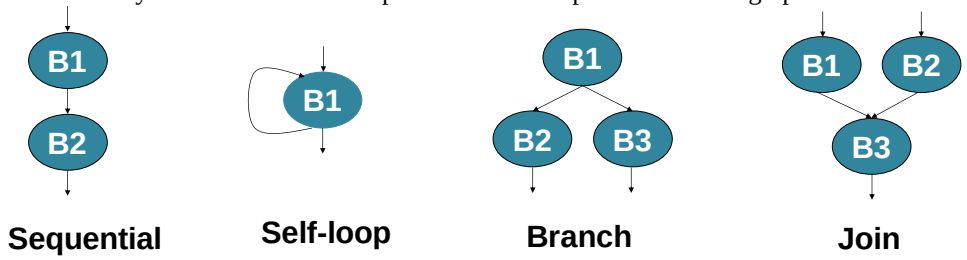


Figure 11: An application's control flow graph is built out of these four kinds of basic control patterns.

The Sequential pattern is the case where 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 set. The Self-loop pattern is the case where at the end of the basic block *B1* there is a conditional branch back to the beginning of the block. A simple "FOR" loop in software will generate this pattern. The Branch pattern is the case where at the end of *B1* there is a conditional branch targeting either *B2* or *B3*. The Join pattern is the case where two entry blocks *B1* and *B2* both unconditionally jump to the same block *B3;* note that only one of either *B1* or *B2* can be active at any given time. After

analyzing the control flow between basic blocks, we conduct different strategies for optimized implementations. At the summit, we demonstrated how M2V recognizes a Self-loop pattern, and how to compile Self-loops efficiently.

The second pass in M2V semantically analyzes the MIPS instructions within one block, and builds and connects nodes in a dataflow graph revealing their 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 locally held, 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 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 12** shows how the instructions proceed through the eMIPS pipeline, where instruction *m* is the extended instruction executed on EU. Here *m* is a single instruction that replaces one or more basic blocks.

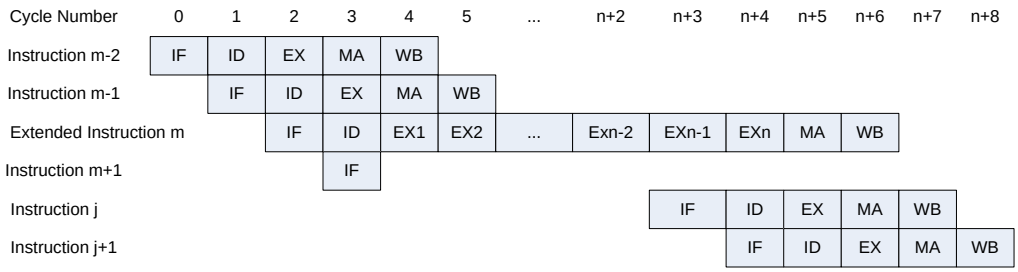


Figure 12: eMIPS pipeline with the extended instruction on EU. Extended instruction *m* represents the basic block executed on EU.

During cycle 3 in **Figure 12**, 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.

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 (e.g. memory loads) is more than one cycle. During static scheduling of the dataflow graph, one important factor is the determination of the two registers, *rs* and *rt*, to be encoded in the extension instruction. These two registers are available directly from the decode stage of the MIPS pipeline, without further access penalty. The selection of 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 EU.

3 Self-loop Basic Blocks

In the previous version of M2V, one basic block is executed only once, and then the EU resource is returned back to the processor. In the case of a self-loop, this results in a pipeline-constrained system, as shown in **Figure 13**. Every time one loop is finished, the resource and program counter (PC) are returned back to the processor's main pipeline. When looping, the processor simply executes the extended instruction again. This is not 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 loops is not efficient for hardware accelerators.

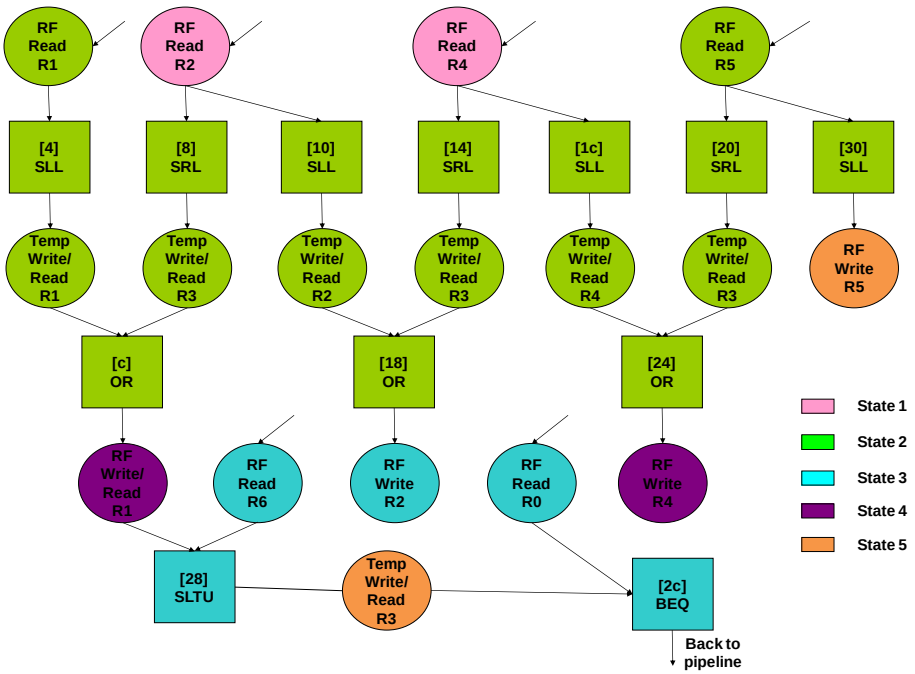


Figure 13: Circuit graph: Pipeline-based implementation of self-loop basic block. RF represents register file. The number in [] is the byte-index of the corresponding MIPS instruction. It starts from 4 because 0 corresponds to the extended instruction. Different colors represent different states, and each state may execute for one or more than one cycle. "Back to pipeline" represents the resource being returned to the processor.

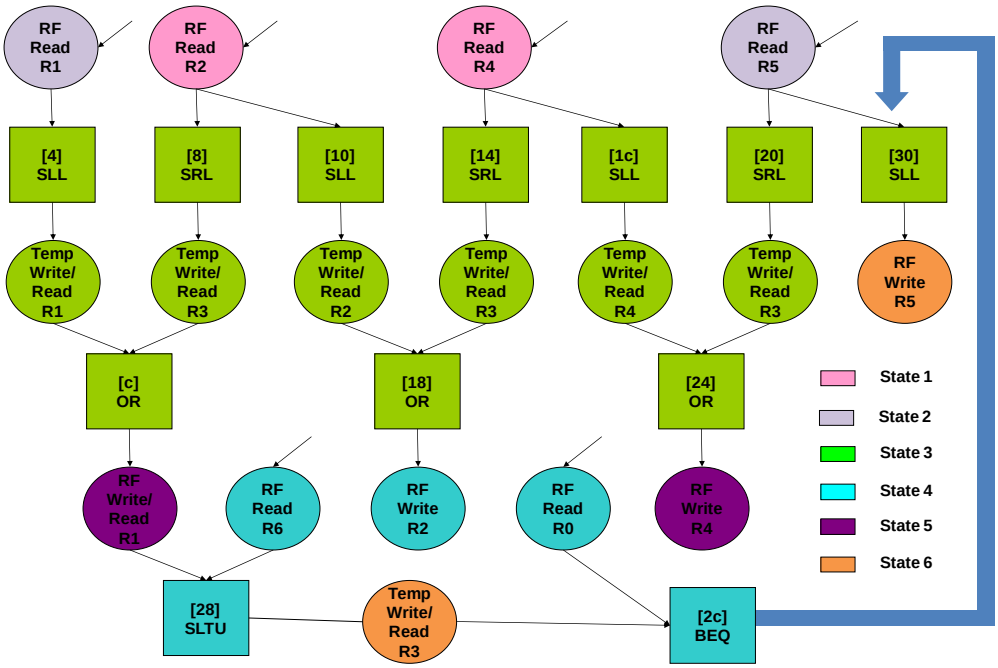


Figure 14: Circuit graph: Control-flow-based implementation of self-loop basic block. The blue arrow represents the transition from state 4 to state 1.

We propose the control-flow based routine to implement the self-loop basic block shown in **Figure 14**. In this implementation, the extension executes all the loops at one time before returning the resource back to the processor. At the end of each 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.

4 Demo and results

At the summit, we showed the demo of M2V, including the following parts:

A. Input .BBW file to M2V
B. Generated circuit graph in M2V
C. Generated Verilog coding from M2V
D. Simulation results in ModelSim

The example basic block implements part of a 64-bit division, which turns out to be frequently used in real-time scheduling and in video games. The disassembly of the block is given in **Figure 15**:

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

Figure 15: Example basic block

The automatically generated circuit graph is shown in
Figure 16. NB: The index of instructions here starts from 0.

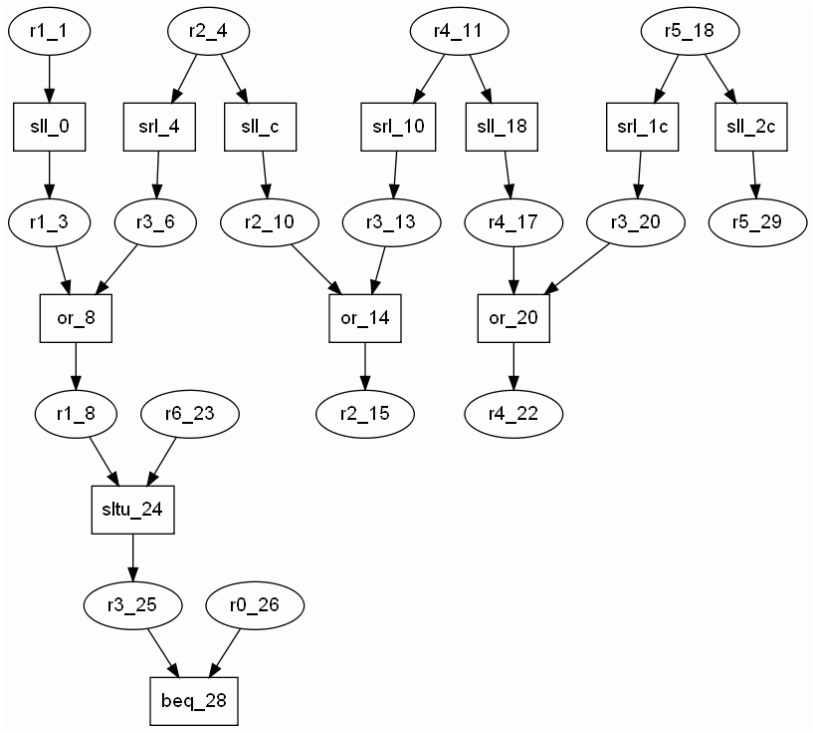


Figure 16: M2V automatically generated this circuit graph for the example basic block in Figure 15.

A portion of the ModelSim-based simulation of eMIPS executing the Self-loop example is shown in **Figure 17**. The highlighted signal of *state_r* corresponds to the state shown in **Figure 14**. The area confined by two yellow lines shows two loops of this block. It is easy to find that the states in each loop are just repetition of some fixed patterns.

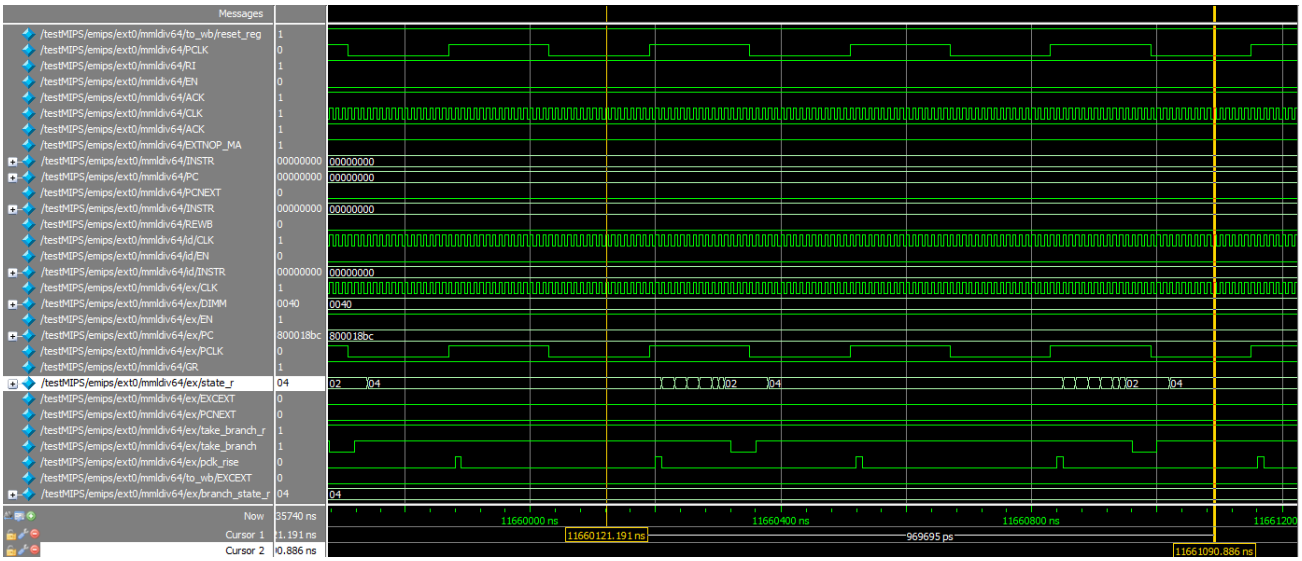


Figure 17: Wave form in the execution of the example Self-looped basic block. The signal of state_r represents the corresponding states in Figure 14.

5 Status of M2V Development

The current version of M2V supports the implementation of single basic block on the extension. It also supports memory load and store instructions, external interrupts, and TLB misses. We have already extended M2V to support the efficient implementation of the self-loop basic block pattern.

We are going to release the first source version of M2V in August 2009. The M2V release will support multiple basic blocks with the four basic control patterns and their combinations. By using the M2V involved tool chain described in **Figure 9**, one can enjoy hardware acceleration from easily obtained executable binary files.

Dual-Core eMIPS

Zhimin Chen
Virginia Tech

Richard Neil Pittman
Microsoft Research

Abstract

The Dual-Core eMIPS research platform shown at the Faculty Summit integrates two eMIPS cores in a Xilinx Virtex-5 C5VLX110T FPGA on the BEE3 board. The platform provides a shared-memory architecture for inter-processor communication, barrier synchronization support, and a number of peripherals. The first demo is a software self-test process for the critical modules in the dual-core system. The second is a parallelized Montgomery modular multiplication of large integers, with a speedup factor of 1.9x over the sequential version.

1 Introduction

As a part of the Multi-Core eMIPS platform, the Dual-Core eMIPS platform contains two eMIPS cores in one Xilinx Virtex-5 C5VLX110T FPGA on the BEE3 board. It implements an on-chip shared-memory architecture with both local and shared memories as well as shared I/O peripherals. Figure 1 is a block diagram of the system.

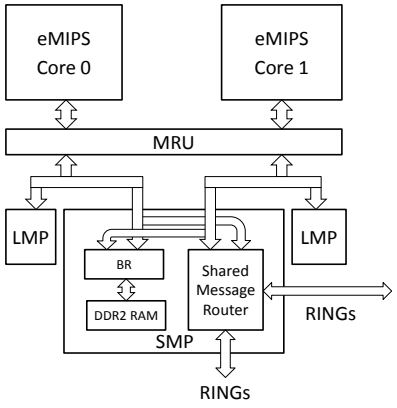


Figure 1: Overview of the Dual-Core eMIPS platform

In Figure 1, MRU represents the Memory Reservation Unit, which supports the LoadLink and StoreConditional standard instructions in the MIPS-2 ISA. LMP is the Local Memory Peripherals, including local BlockRAM and local timer. SMP is the Shared Memory Peripherals which contains the shared DDR2 SDRAM, shared BlockRAM, shared interrupt controller, shared USART module, shared GPIO module and all other (shared) peripherals. Inside SMP, BR is the bridge connecting the local memory bus to the shared peripherals. Every memory module or I/O module is connected to the two eMIPS cores by means of this bridge. Shared Message Router is the module that handles inter-FPGA communication through the RING connections among FPGAs on the BEE3 board.

Besides the MRU, the platform integrates another simple but efficient mechanism to support barrier synchronization. Two shared message boxes (32-bit registers) are connected to both eMIPS cores through a BR. After a system boot-up reset, both boxes contain 0x0. With the following code, operations in two eMIPS cores can be synchronized at every pair of barrier functions.

eMIPS Core 0	eMIPS Core 1
void barrier(void){	void barrier(void){
volatile UINT32 * mb0 = MBADDR0;	volatile UINT32 * mb0 = MBADDR0;
volatile UINT32 * mb1 = MBADDR1;	volatile UINT32 * mb1 = MBADDR1;
*mb0 = 0x5555aaaa;	*mb1 = 0xaaaa5555;
while(*mb1 != 0xaaaa5555);	while(*mb0 != 0x5555aaaa);
*mb1 = 0x0;	*mb0 = 0x0;
}	}

This platform is easy to use to explore parallel programming and/or scheduling. In our demonstrations, programs are cross-compiled on a PC and the user interacts with them through a USART console.

2 Demonstration

At the Faculty Summit, we present two demonstrations. The first is a self-test process of the critical modules in the dual-core system. The second is a Montgomery modular multiplication of large integers, including both a sequential and a parallel version. The parallel version shows a speedup over the sequential one of 1.9x.

2.1 Demonstration 1: test process of critical modules

In the first demonstration, modules under test include the shared BlockRAM, the shared DDR2 SDRAM, the Memory Reservation Unit, and the Processor ID module.

To test the shared BlockRAM, we go through the following steps.

1) eMIPS Core 0 writes 0xFFFFFFFF to every address of the shared BlockRAM, then eMIPS Core 1 reads every address of the shared BlockRAM to check the value;
2) Change the value to 0x00000000, 0xAAAAAAAA, and 0x55555555, and go through step 1 another 3 times;
3) eMIPS Core 1 writes 0xFFFFFFFF to every address of the shared BlockRAM, then eMIPS Core 0 reads every address of the shared BlockRAM to check the value;
4) Change the value to 0x00000000, 0xAAAAAAAA, and 0x55555555, and go through step 3 another 3 times;

We use the same method to test the shared DDR2 SDRAM. Only 1K space of the DDR2 SDRAM is under test. If the test passes, we consider the shared DDR2 memory works correctly.

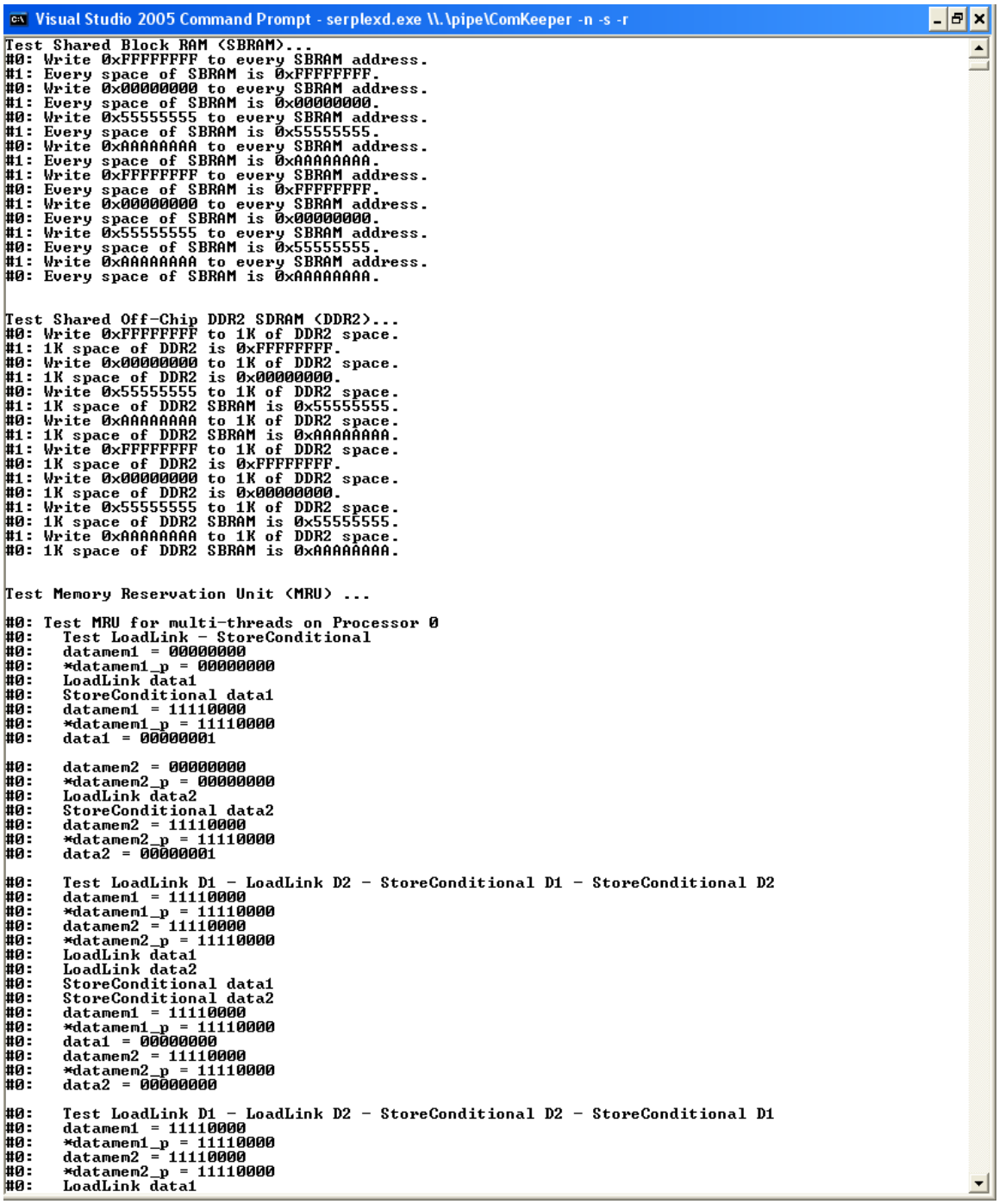
The third test is on the Memory Reservation Unit, which consists of the following steps. For convenience, we use LL0, SC0, LL1, SC1 to represent the LoadLink and StoreConditional operations performed by eMIPS Core 0 and eMIPS Core 1.

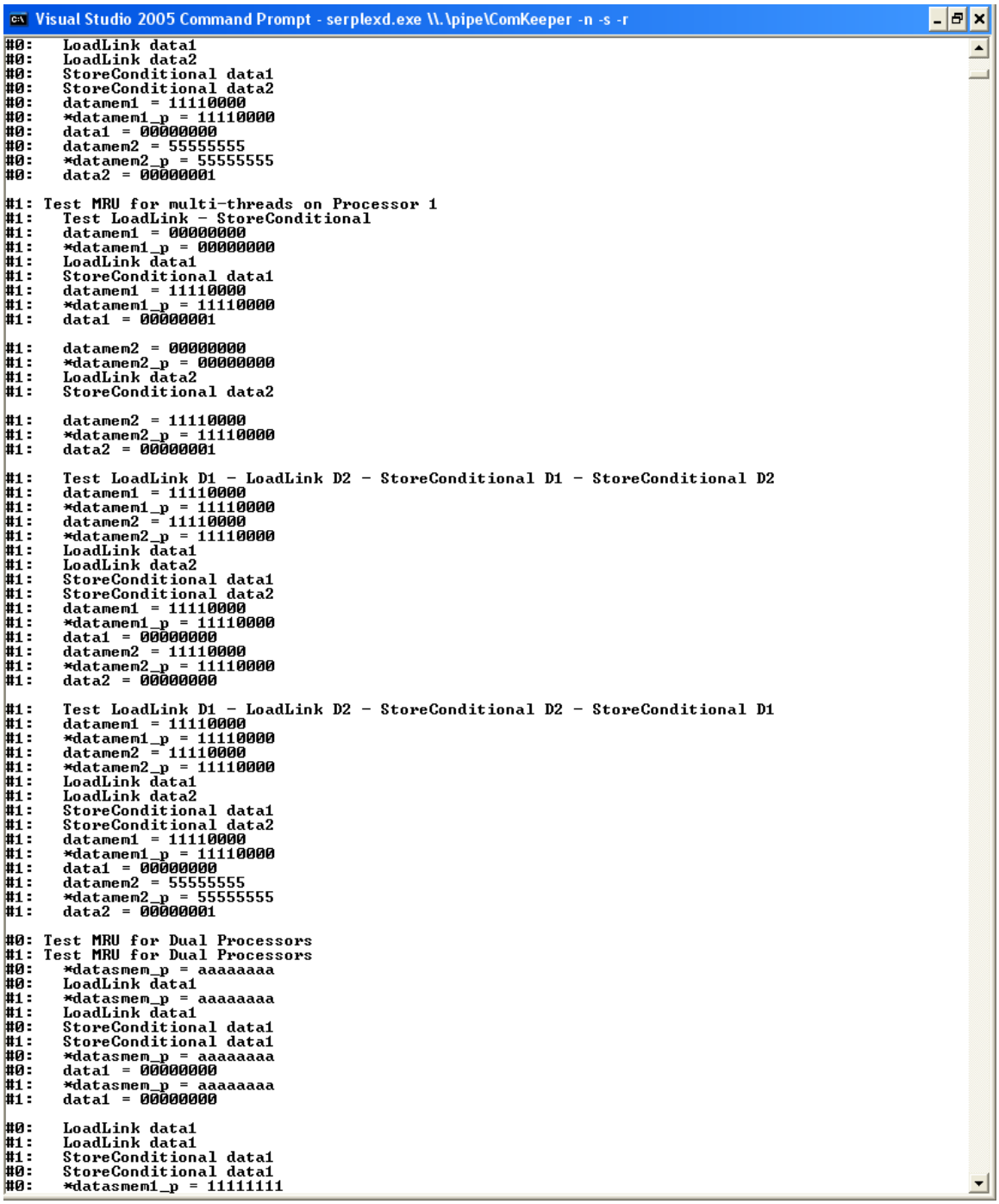
1) In sequence, LL0(datamem1), SC0(datamem1, 0x11110000), LL0(datamem2), SC0(datamem2, 0x11110000) are performed. data1 and data2 are used to indicate whether SC0 operations are successful or not. If the MRU works correctly, after the operations, both datamem1 and datamem2 should be 0x11110000; both data1 and data2 should be 0x1.
2) In sequence, LL0(datamem1), LL0(datamem2), SC0(datamem1, 0x55555555), SC0(datamem2, 0xaaaaaaaa) are performed. data1 and data2 are used to indicate whether SC0 operations are successful or not. If the MRU works correctly, after the operations, both datamem1 and datamem2 should remain 0x11110000; both data1 and data2 should be 0x0.

3) In sequence, LL0(datamem1), LL0(datamem2), SC0(datamem2, 0x55555555), SC0(datamem1, 0xaaaaaaaa) are performed. data1 and data2 are used to indicate whether SC0 operations are successful or not. If the MRU works correctly, after the operations, datamem1 should remain 0x11110000, datamem2 should be 0x55555555, data1 should be 0x0 while data2 should be 0x1.
4) eMIPS Core 1 performs the same operations as shown from step 1 to step 3.
5) In sequence, LL0(datasmem), LL1(datasmem), SC0(datasmem, 0x11111111), SC1(datasmem, 0x55555555) are performed. Both cores use their own data1 to indicate whether the SC operations are successful. If the MRU works correctly, after the operations, datasmem should be unchanged; both cores have data1 as 0x0.
6) In sequence, LL0(datasmem), LL1(datasmem), SC1(datasmem, 0x11111111), SC0(datasmem, 0x55555555) are performed. Both cores use their own data1 to indicate whether the SC operations are successful. If the MRU works correctly, after the operations, datasmem should be 0x11111111; Core 0 has data1 as 0x0 while Core 1 has data1 as 0x1.

Finally, we test the Processor ID module. Only Core 0 always has its PID valid (0x20). Therefore, after reset, Core 0 gets its PID 0x20 while Core 1 gets its PID 0x00. As the primary core, Core 0 can configure the PID of Core 1 to a valid PID (e.g. 0x21). After configuration, Core 1 gets its own PID 0x21. Except for Core 0, all the other cores are not able to access the PID of other processors. So even after configuration, when Core 1 tries to read others' ID, it always gets 0x00.

Figure 2 shows the results of the above 4 tests, which proves that the parallel platform works correctly.


Visual Studio 2005 Command Prompt - serplexd.exe \\.\pipe\ComKeeper -n -s -r
Test Shared Block RAM (SBRAM)...
#0: Write 0xFFFFFFFF to every SBRAM address.
#1: Every space of SBRAM is 0xFFFFFFFF.
#0: Write 0x00000000 to every SBRAM address.
#1: Every space of SBRAM is 0x00000000.
#0: Write 0x55555555 to every SBRAM address.
#1: Every space of SBRAM is 0x55555555.
#0: Write 0xAAAAAAAA to every SBRAM address.
#1: Every space of SBRAM is 0xAAAAAAAA.
#1: Write 0xFFFFFFFF to every SBRAM address.
#0: Every space of SBRAM is 0xFFFFFFFF.
#1: Write 0x00000000 to every SBRAM address.
#0: Every space of SBRAM is 0x00000000.
#1: Write 0x55555555 to every SBRAM address.
#0: Every space of SBRAM is 0x55555555.
#1: Write 0xAAAAAAAA to every SBRAM address.
#0: Every space of SBRAM is 0xAAAAAAAA.
Test Shared Off-Chip DDR2 SDRAM (DDR2)...
#0: Write 0xFFFFFFFF to 1K of DDR2 space.
#1: 1K space of DDR2 is 0xFFFFFFFF.
#0: Write 0x00000000 to 1K of DDR2 space.
#1: 1K space of DDR2 is 0x00000000.
#0: Write 0x55555555 to 1K of DDR2 space.
#1: 1K space of DDR2 SBRAM is 0x55555555.
#0: Write 0xAAAAAAAA to 1K of DDR2 space.
#1: 1K space of DDR2 SBRAM is 0xAAAAAAAA.
#1: Write 0xFFFFFFFF to 1K of DDR2 space.
#0: 1K space of DDR2 is 0xFFFFFFFF.
#1: Write 0x00000000 to 1K of DDR2 space.
#0: 1K space of DDR2 is 0x00000000.
#1: Write 0x55555555 to 1K of DDR2 space.
#0: 1K space of DDR2 SBRAM is 0x55555555.
#1: Write 0xAAAAAAAA to 1K of DDR2 space.
#0: 1K space of DDR2 SBRAM is 0xAAAAAAAA.
Test Memory Reservation Unit (MRU) ...
#0: Test MRU for multi-threads on Processor 0
#0: Test LoadLink - StoreConditional
#0: datamem1 = 00000000
#0: *datamem1_p = 00000000
#0: LoadLink data1
#0: StoreConditional data1
#0: datamem1 = 11110000
#0: *datamem1_p = 11110000
#0: data1 = 00000001
#0: datamem2 = 00000000
#0: *datamem2_p = 00000000
#0: LoadLink data2
#0: StoreConditional data2
#0: datamem2 = 11110000
#0: *datamem2_p = 11110000
#0: data2 = 00000001
#0: Test LoadLink D1 - LoadLink D2 - StoreConditional D1 - StoreConditional D2
#0: datamem1 = 11110000
#0: *datamem1_p = 11110000
#0: datamem2 = 11110000
#0: *datamem2_p = 11110000
#0: LoadLink data1
#0: LoadLink data2
#0: StoreConditional data1
#0: StoreConditional data2
#0: datamem1 = 11110000
#0: *datamem1_p = 11110000
#0: data1 = 00000000
#0: datamem2 = 11110000
#0: *datamem2_p = 11110000
#0: data2 = 00000000
#0: Test LoadLink D1 - LoadLink D2 - StoreConditional D2 - StoreConditional D1
#0: datamem1 = 11110000
#0: *datamem1_p = 11110000
#0: datamem2 = 11110000
#0: *datamem2_p = 11110000
#0: LoadLink data1


Visual Studio 2005 Command Prompt - serplexd.exe \\.\pipe\ComKeeper -n -s -r
#0: LoadLink data1
#0: LoadLink data2
#0: StoreConditional data1
#0: StoreConditional data2
#0: datamem1 = 11110000
#0: *datamem1_p = 11110000
#0: data1 = 00000000
#0: datamem2 = 55555555
#0: *datamem2_p = 55555555
#0: data2 = 00000001
#1: Test MRU for multi-threads on Processor 1
#1: Test LoadLink - StoreConditional
#1: datamem1 = 00000000
#1: *datamem1_p = 00000000
#1: LoadLink data1
#1: StoreConditional data1
#1: datamem1 = 11110000
#1: *datamem1_p = 11110000
#1: data1 = 00000001
#1: datamem2 = 00000000
#1: *datamem2_p = 00000000
#1: LoadLink data2
#1: StoreConditional data2
#1: datamem2 = 11110000
#1: *datamem2_p = 11110000
#1: data2 = 00000001
#1: Test LoadLink D1 - LoadLink D2 - StoreConditional D1 - StoreConditional D2
#1: datamem1 = 11110000
#1: *datamem1_p = 11110000
#1: datamem2 = 11110000
#1: *datamem2_p = 11110000
#1: LoadLink data1
#1: LoadLink data2
#1: StoreConditional data1
#1: StoreConditional data2
#1: datamem1 = 11110000
#1: *datamem1_p = 11110000
#1: data1 = 00000000
#1: datamem2 = 11110000
#1: *datamem2_p = 11110000
#1: data2 = 00000000
#1: Test LoadLink D1 - LoadLink D2 - StoreConditional D2 - StoreConditional D1
#1: datamem1 = 11110000
#1: *datamem1_p = 11110000
#1: datamem2 = 11110000
#1: *datamem2_p = 11110000
#1: LoadLink data1
#1: LoadLink data2
#1: StoreConditional data1
#1: StoreConditional data2
#1: datamem1 = 11110000
#1: *datamem1_p = 11110000
#1: data1 = 00000000
#1: datamem2 = 55555555
#1: *datamem2_p = 55555555
#1: data2 = 00000001
#0: Test MRU for Dual Processors
#1: Test MRU for Dual Processors
#0: *datasmem_p = aaaaaaaa
#0: LoadLink data1
#1: *datasmem_p = aaaaaaaa
#1: LoadLink data1
#0: StoreConditional data1
#1: StoreConditional data1
#0: *datasmem_p = aaaaaaaa
#0: data1 = 00000000
#1: *datasmem_p = aaaaaaaa
#1: data1 = 00000000
#0: LoadLink data1
#1: LoadLink data1
#1: StoreConditional data1
#0: StoreConditional data1
#0: *datasmem1_p = 11111111

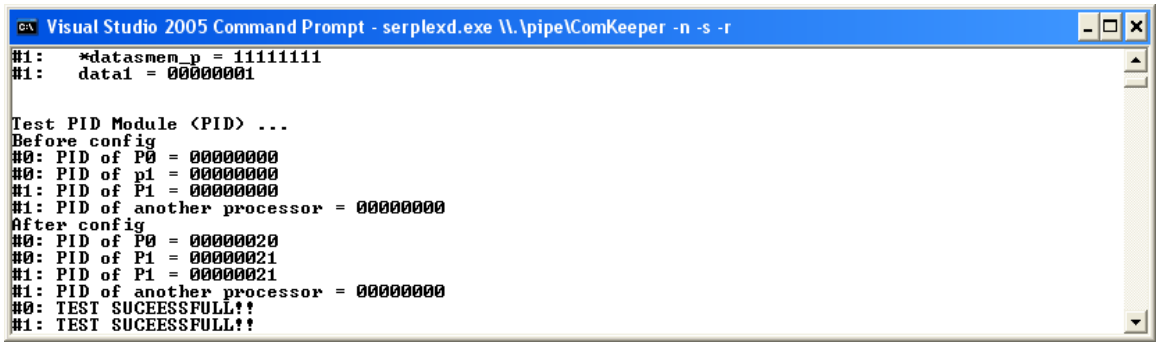


Figure 2: Results of demonstration 1.

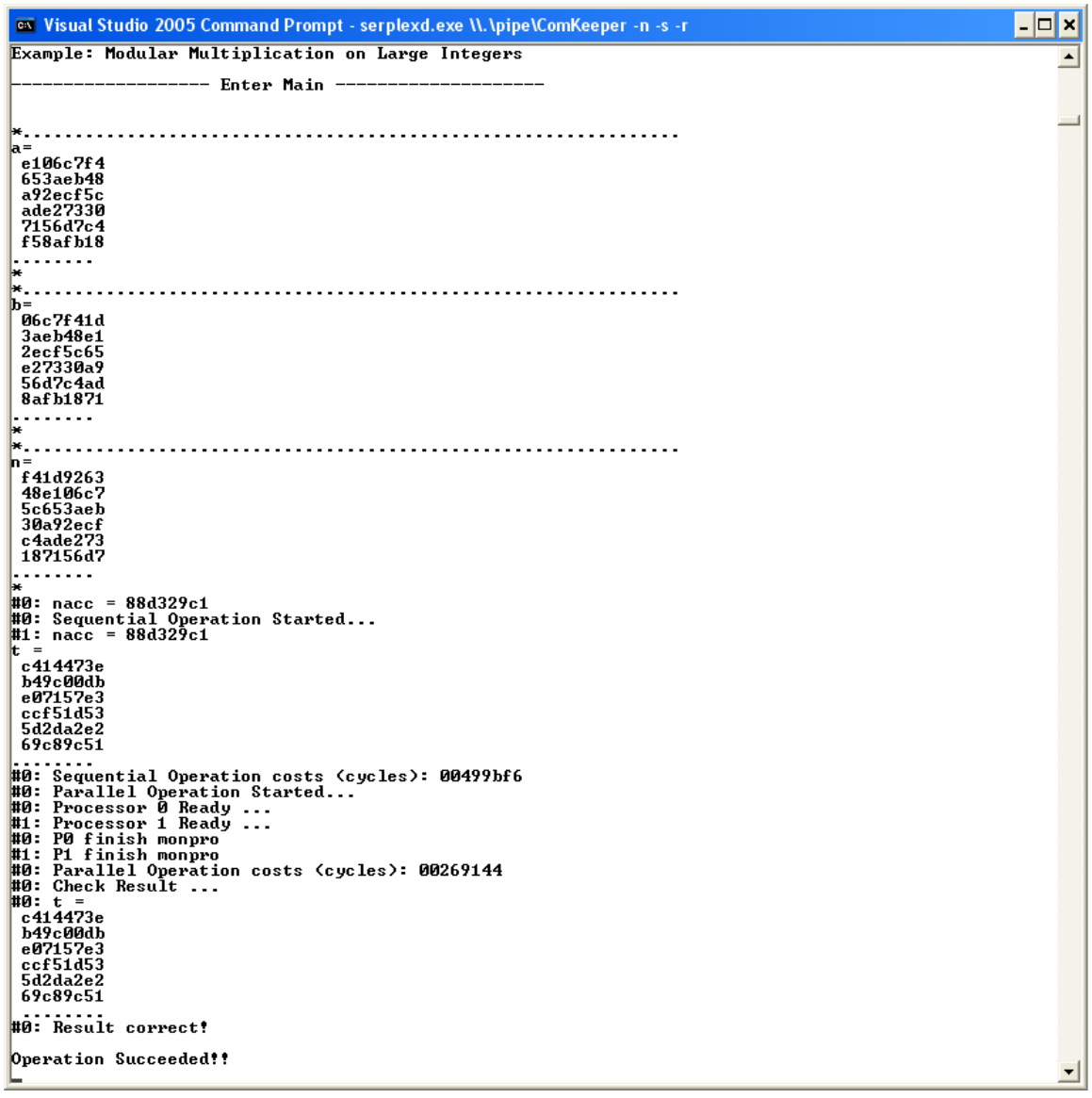


Figure 3: Results of demonstration 2.

2.2 Demonstration 2: parallelized Montgomery modular multiplication on large integers

The second demonstration gives an example of parallel programming based on the dual-core platform. The application is a Montgomery modular multiplication without the final subtraction adjustment. We tested both the sequential implementation as well as the parallel one. The speedup of the parallel one over the sequential one is up to 1.9. The results are presented in Figure 3.

From the results, we found that, through proper parallel programming, high speed up can be achieved based on the current platform.

NetBSD on eMIPS

David Sheldon, Alessandro Forin
Microsoft Research

Abstract

We demonstrate an online scheduling algorithm for hardware accelerators and its implementation on the NetBSD operating system. The scheduler uses the current performance characteristics of the accelerators to select which accelerators to load or unload. The evaluation on a number of workloads shows that the scheduler is typically within 20% of the optimal schedule computed offline. The hardware support consists of simple cost-benefit indicators, usable for any online scheduling algorithm. The NetBSD modifications consist primarily in loadable kernel modules, with minimal changes to the operating system itself. The system was demonstrated running multi-user on an ML402 board and running diskless on a BEE3 board.

1 Introduction

eMIPS is a dynamically extensible processor that includes a standard MIPS trusted ISA tightly connected to reconfigurable hardware. The programmable logic is divided in *extension slots* that plug into the main pipeline stages during the execution of a program, as depicted in **Figure**. At DemoFest, we present a scheduling algorithm for allocating the extension slots to competing applications, under a general-purpose operating system such as NetBSD.

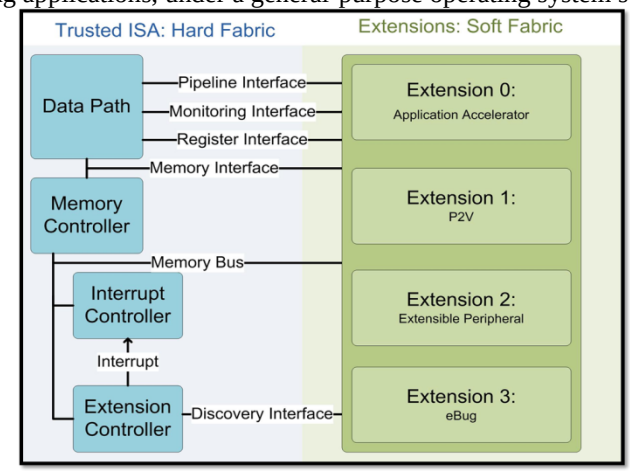


Figure 1: The scheduler supports tightly-coupled micro-processor architecture with a number of hardware *extension slots* usable for accelerating software applications.

2 Hardware and Software Support

Hardware support for accelerator scheduling is based on a pair of performance counters shown in Figure 2, which identify the costs and benefits in using the accelerators. The choice is intentionally general enough that software has ample freedom to schedule the resources as desired. The scheduler we implemented is independent of thread scheduling, which we considered an orthogonal problem. The scheduler is realized as a loadable kernel module, thereby eliminating

all fixed overheads (e.g. in case it is not used) and allowing for easy selection of alternate implementations. Additional software support includes a new image format for accelerators and related utilities.

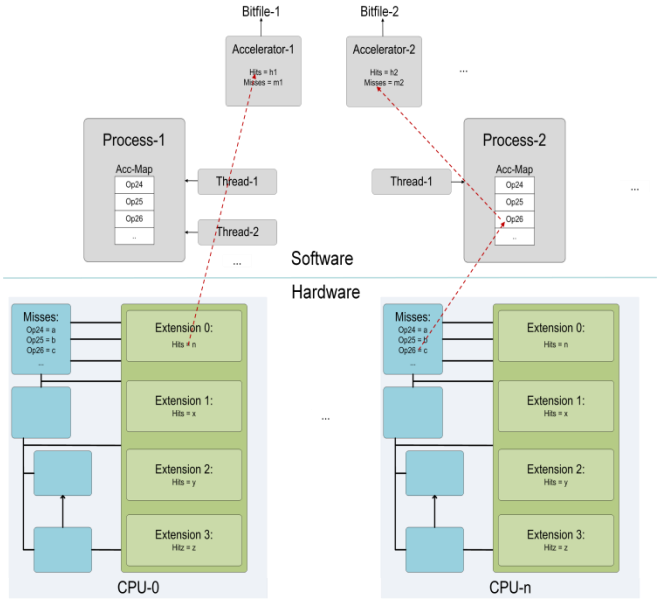


Figure 2: Hardware support for scheduling includes *hit* and *miss* counters. Each time an Extension slot recognizes an extended instruction the corresponding hit counter is incremented. Each time an extended opcode is not recognized by any of the slots the corresponding miss counter is incremented. The mapping from opcodes to accelerators is provided by software.

3 The demos

The first demo shows the system running in multi-user mode on the ML402 board. On this system, we used a 2GB compact flash card as disk storage, taking advantage of the SystemACE component. The system can actually operate in much less disk storage, but about 400MB of disk space are usually recommended for a fully-featured NetBSD system. This system has been operational for some time and is very stable. A fan-sink solves the excessive heat generated at the DDR memory interface and the system can therefore be left running indefinitely. There is no network on this system.

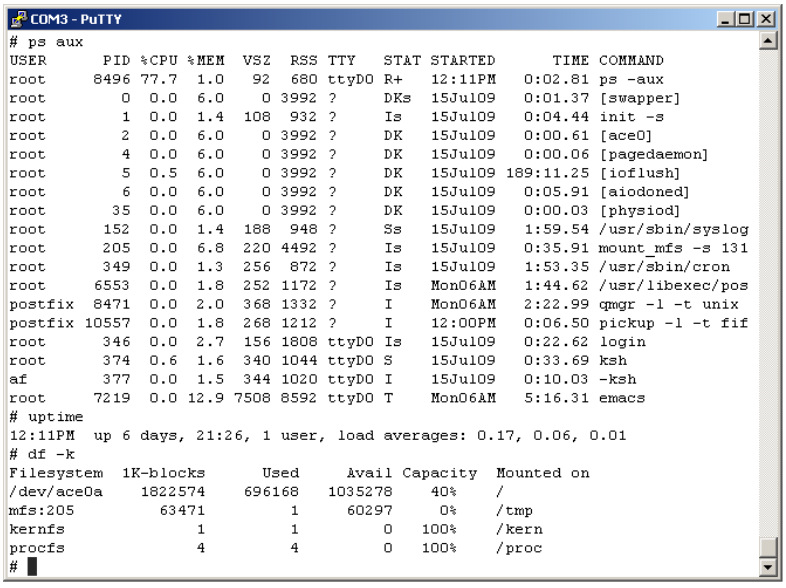


Figure 3: The NetBSD system on the ML402 board is complete and fairly stable.

The second demo shows the system booting diskless on the BEE3 board, using the just-completed eNIC Gigabit Ethernet peripheral. We use a VirtualPC on a portable PC to run the DHCP, TFTP and NFS servers used to assign an IP and configuration data to the BEE3, to provide the kernel image, and to provide the disk storage services, respectively. Figure 4 shows the complete output on the serial line, from system reset to single-user prompt. In this system, we incorporate the boot loader in the BRAM image to work around the very poor performance of the USB serial interface. We ask the loader (typing the "bbbb" command) to extract itself into DDRAM and to fetch the BSD kernel image via the eNIC. The image we select is the default "nfsnetbsd". Once fetched and started, the NetBSD kernel initializes itself, then repeats the DHCP inquiry to find its NFS file server and eventually gives us the single-user shell prompt. The whole process takes 130 seconds, from start to finish.

Figure 18: Boot sequence of NetBSD on the BEE3 board, from reset to single-user prompt.

Appendix A: Posters

The following posters were presented at DemoFest:
Education
Multi-Touch
Dance Pad
Gigabit
SQL
Concolic
Mining specs
FPU
eMIPS overview
Vikram
M2V
Multicore
NetBSD

Appendix B: Movies and Pictures

The following movie demonstrates something. If you are looking at the Microsoft Word version of this document you can double-click on the icons to watch the videos.

Here are a few pictures from the DemoFest booth.