

## Projects

### ECE 464 / ECE 520 / DS 510P

#### Spring 2004

Dr. Paul D. Franzon

My intent is the ECE 464 students work in project 1 in pairs, and ECE 520 students work on project 2 in pairs. VBEE and NTU students can either work on project 1 by themselves or on project 2 in pairs. I have also listed an alternative project at the end of this document that can only be done with my permission.

#### **ECE 464 (Pairs) and Individual Off Campus student project (VBEE/NTU)**

You are going to extend an open source design for a micro-controller.

The PIC microcontroller is popular for many embedded applications, especially for mechatronic ones (e.g. robotics). An open core partial PIC18 implementation has been written and is available at [www.opencores.org](http://www.opencores.org) as the AE18 microprocessor core. You are to modify this microprocessor so that it can support added functions of the full implementation. Please go to the PIC18 manufacturers web-site ([www.microchip.com](http://www.microchip.com)) and download the datasheet for the PIC18C242. Using the specification implied by that datasheet, implement the following functions in the AE18 core:

- A multiplier as specified in Section 6 of the datasheet; and
- All interrupt capabilities, as specified in Section 7 of the datasheet; and
- The timers, as specified in Section 10 of the datasheet.

**ECE 464 students are to do this project in self-selected pairs.**

**VBEE and NTU students are to turn in individual projects, or work in pairs on the project below.**

#### **ECE 520 and Pair Off-Campus Projects**

**(You need to find a partner to do these projects)**

##### **Packet Filtering Engine**

You are going to implement a programmable engine for filtering out malicious packets at an internet connection. Such filters are very useful for detecting possible attacks on a network.

You are going to implement a Snort packet filter. Snort is a simple filter language.

Useful references for understanding this project include the following:

1. Martin Roesch, "Snort lightweight intrusion detection for networks", Proceedings of LISA '99: 13th system administration conference, Seattle Washington

[http://www.usenix.org/publications/library/proceedings/lisa99/full\\_papers/roesch/roesch.pdf](http://www.usenix.org/publications/library/proceedings/lisa99/full_papers/roesch/roesch.pdf)

2. Young H. Cho, Shiva Navab, William H. Mangione-Smith, "Specialized Hardware for Deep

Network Packet Filtering", Proceedings FPL'02: 12 International conference of field-programmable logic and applications, Montpellier France.

[http://www.ee.ucla.edu/faculty/papers/billms\\_FPL2002\\_sept02.pdf](http://www.ee.ucla.edu/faculty/papers/billms_FPL2002_sept02.pdf)

This paper shows a sample high level design. **You are not constrained by this design.** It is intended as an example only in order to help you understand the project and give you one starting point.

3. Rules can be found at [www.whitehats.com](http://www.whitehats.com), here is sample rules file

<http://www.whitehats.com/ids/vision18.rules.gz>

4. Larger rule files can be found at <http://www.snort.org/dl/rules/>. A lot of useful information can be found on the [www.snort.org](http://www.snort.org) website.

The I/O for your implementation are via three memories, two for input, and one for output. The two input memories are as follows:

- One memory, organized as a FIFO for storing incoming packets. The FIFO is written to from the test fixture. There is no constraint on the size or the width of this FIFO. You can read multiple packets in parallel if you like.
- A second memory, containing the Snort rule set in binary format. (Format TBD.) Again, this is written to from the test fixture, and you can read it in parallel if you like.
- A third memory containing packets that triggered an "alert". Again this is organized as a FIFO and I don't care about the size or width.
- Passed packets can be output to a port. They don't need to be stored anywhere.

In this project, you are competing with each other on performance (packet throughput for test example provided) per unit area. The memories above do not count towards the area.

## General Instructions

### Project Plan

Your written Project Plan is due in class on **March 18** (off campus students, see the syllabus for due date). Items to be included in this report include the following:

- Block diagrams of your design, clearly identifying any design hierarchy, all registers, all module I/O. Neat block diagrams are expected, not rough hand-drawn sketches. Include a written description of the function of each module. E.g. For a multiplier, which multiplier algorithm are you planning to use? For the microprocessor project, include a block diagram of the existing design and clearly annotate your intended changes.
- For the ECE 520 project, any appropriate high-level (e.g. C) code showing the algorithm has been correctly captured.
- A project execution plan, including the following:
  - Who is responsible for each module

- A verification plan. How do you intend to verify that the design works correctly? Who will be responsible for system verification?
- A risk assessment plan. Where are the greatest risks in this project? E.g. What areas of the design are you unsure of? Where are your greatest concerns in executing this project? Note, this section is intended for your benefit (so you can focus your efforts on the parts you are least sure of). Do not expect written feedback on your risk assessment plan. However, feel free to ask questions of the teaching team.
- A milestone chart, including anticipated dates for completing module design, design integration and design verification.

**Grading** for this report will be as follows (out of 10):

9-10 : All major design elements correctly identified; behavioral code complete with test cases. Neat and clear documentation.

7-8 : At least one major design element missing, or C code incomplete, or confusing and poor documentation.

5-6 : Scrappy, but honest, capturing some elements but conveying no real understanding of the design.

0-4 : Extremely poor attempt.

### **Main Project Report and Demo**

Your main project report is due on **April 15** (VBEE and NTU students see syllabus). There are penalties for late turn-in. This date is set by the desire to complete grading these before you need to prepare for the final exam.

**On-campus students: Turn the report in at your demonstration. Project demos will take place in the week of and after the due date.**

Your project report is to include the following:

- Written description of your approach, including block diagram of your design, description of your verification strategy and a discussion of any part of the design, synthesis or verification that you consider “tricky”, novel or noteworthy.
  - Specifically note and document your *area* and your *throughput*
- Full listings of the following:
  - Verilog files, including test fixture
  - Synthesis scripts
  - Extracts from View\_command.log generated by design\_analyzer, including the results from the read\_command and timing verification. (If you use dc\_shell, you will have to rerun the script in design\_analyzer in order to obtain this).
  - Plots from the final design.
  - Simulation run results (waveforms or equivalent)
  - High level model of the design, if appropriate

You will be graded on the following factors (roughly in order of importance):

- Design correctness in terms of correct functionality
- Design correctness in terms of using good Verilog and Synthesis strategies, structuring, coding approach, etc.
- Correctness of synthesis approach
- Design elegance in terms of Verilog readability, and “coolness” factors
- Design efficiency – performance and area

- Completeness of Verification – do you just simulate a handful of cases, or has some thought and effort gone into it?
- Quality of reporting

The grading scale will be roughly as follows:

100 Excellent job. Several special features.

95 Very good job. At least one special feature.

90 Works properly. Good basic design.

80 - 89 Some significant problems that have a good probability of meaning this design would not be acceptable to an industrial team. e.g. Synthesis problems, features that did not work or poor documentation.

70 - 79 Major problems but made a good attempt, or understood very little of a working design. This includes a terrible report but design that would otherwise get an 80.

0-60 Some fraction of the above. Give something if they made an attempt.

For the ECE 520 project, the points above 90 will primarily be determined by your measured area and performance. I.e. The projects that achieve the highest ratio of performance/area will receive 100, the lowest (working) designs will achieve 90.

For the ECE 464 project, the performance/area ratio must be reported for your multiplier. This will be used for part of your “bonus” grade.

### **Other Instructions**

- You are NOT required to run the verify command in Synopsys. This takes lot of time and adds no educational value to the project (obviously in real life, you would run this command.)
- Make sure to use the 0.25  $\mu\text{m}$  library, so we can compare area and performance accurately.
- An SRAM .v file will be provided. Use this for any SRAMs required. DO NOT SYNTHESIZE ANY SRAMS. These do not count against your area. Please avoid temptations to artificially use this to your advantage (e.g don't implement combinational logic as an SRAM lookup table, or artificially replace a PLA with a micro-control RAM.)

## **Alternative Project**

One of my research projects could benefit from completion of this project. Thus I am looking for one or two ECE 464 groups or ECE 520 individuals to do this project. Please approach me if you are interested in this project. My main criteria for selecting the students who can take this are that you are familiar with concepts in multithreading and pipelined processor design. If you do not have this background knowledge or have are not willing to obtain it on your own, please do not consider this project.

# **EXTENDING THE OpenRISC 1000** **implementation**

## **1. Introduction:**

### **1.1 OpenRISC 1000**

**OpenRISC 1000** (or1k) is an open source architecture definition. It describes the instruction set architecture, the special-purpose registers and other high-level details of the architecture. As an architecture definition only it leaves a lot of the details up to a particular implementation. Being open source there's no licensing issues involved and the definition per se is completely free, no strings attached. The details of the architecture definition are purposely left out of this document and the reader is referred to the Manual [1] and the or1k website [2] for the same.

The key features of this architecture are:

- ◆ Simple Load/Store Architecture
- ◆ 32-bit/64-bit logical address space
- ◆ Instruction Set Definitions:
  - OpenRISC Basic Instruction Set (ORBIS32/64)
  - OpenRISC Vector/DSP eXtension (ORVDX64)
  - OpenRISC Floating Point eXtension (ORFPX32/64)
- ◆ Support for shadowed register-files.
- ◆ Branch delay slot
- ◆ Two memory addressing modes
- ◆ Complete ISA definition and bit-encoding
- ◆ SMT and fast context-switching capability
- ◆ Architectural simulator (albeit limited)
- ◆ GNU tools: compiler, binutils ported
- ◆ ucLinux: embedded Linux ported

### **1.2 OpenRISC 1200**

**OpenRISC 1200** (OR1200) is one such implementation of the or1k processor family. The details regarding this particular implementation and the RTL code can be obtained from the webpage [3]. Key features of this particular implementation are:

- ◆ 32-bit scalar RISC with Harvard microarchitecture.
- ◆ 5 stage integer pipeline
- ◆ Virtual Memory Support (MMU)
- ◆ Basic DSP capabilities
- ◆ 1-way 8KB data cache and 1-way 8KB instruction cache
- ◆ Physically tagged direct-mapped caches
- ◆ 64-entry hash based 1-way direct-mapped Translation Lookaside buffer (TLB) for both instruction and data

The RTL coding of this implementation has been completed and has been tested on several applications. The RTL code is free and available off the OR1200 webpage [3].

## **2. Extending OR1200**

There are two ways in which the OR1200 implementation can be extended:

1. Add support for Fast-Context Switching.
2. Add support for Simultaneous Multi-threading.

There is support for both of these in the architecture definition i.e. both of these can be achieved by a particular implementation. The OR1200 right now does not support Fast-Context Switching or Simultaneous Multi-threading. Section 2.1 and 2.2 below in general explain fast-context switching and simultaneous multi-threading respectively.

### 2.1 Fast-context switching:

When a particular program (**P1**) runs on a processor, the processor fetches and executes instructions from that particular program. The instructions modify the register values and the memory contents as they execute. At any point in time the values in the general-purpose registers, the special-purpose registers and the memory locations define the state of the program. Let's say the processor wants to execute instructions from another program (**P2**) while **P1** has not yet completed (this is called pre-emption). It not only has to start fetching and executing instructions from **P2** but it should also be able to continue execution of **P1** from the point where it was pre-empted by **P2**. In order to be able to do that it should save the state of program **P2** and retrieve it later on for re-execution. This whole process: saving the state of one program and loading the state of another program to continue its execution is called **context-switching**. The state of the program is also referred to as its **context**. So when would a processor want to pre-empt one program and run another program? A simple scenario would be when an interrupt occurs, and it needs to be serviced very quickly. In this scenario the processor pre-empts the current executing task and runs the instruction service routine (ISR).

In doing a context switch as described above a lot of overhead is involved in saving the context related information to memory and loading a new context information for the new program. Fast-context switching can be achieved by being able to maintain multiple-copies of the context (multiple register-files) in the processor/hardware. Only a particular context is active or executing at a time. A simple bit in a global register indicates the current active context. For doing a context-switch the bit needs to be changed to point to the new context that will be executing. The extensions needed here over a basic architecture would be to have support of multiple-contexts in hardware, and book-keeping information for each context (e.g. whether it is active).

### 2.2 Simultaneous Multi-threading (SMT):

Simultaneous Multi-threading is an additional extension over fast-context switching. In fast context switching only one context is active at a time. In SMT multiple-contexts are active and executing simultaneously. So the processor can fetch and execute instructions from multiple-programs at the same time. The processor updates the right-context depending on the context of the instruction (obviously each instruction is tagged with the information of the context to which it belongs). The extensions needed would be to have support for multiple-contexts in the hardware, to be able to fetch from multiple-programs simultaneously, to be able to distinguish between instructions from different contexts so that each instruction only modifies it's own contexts.

### 2.3 Putting it together

The user is encouraged to read the references on or1k and OR1200. Understand how fast-context-switching and simultaneous-multithreading can be done in OpenRISC and what are the implementation requirements. The OpenRISC team has their own mailing-list [5] which would be a good place to ask OpenRISC specific questions.

Reference:

- [1]. OpenRISC1000 website: <http://www.opencores.org/projects/or1k/Home>
- [2]. OpenRISC 1000 Architecture Manual:  
[http://www.opencores.org/tmp/cvsget\\_cache/or1k/docs/openrisc\\_arch3.pdf](http://www.opencores.org/tmp/cvsget_cache/or1k/docs/openrisc_arch3.pdf)
- [3]. OpenRISC 1200 website:  
<http://www.opencores.org/projects/or1k/OpenRISC%201200>
- [4]. OpenRISC 1200 specification:  
[http://www.opencores.org/tmp/cvsget\\_cache/or1k/or1200/doc/or1200\\_spec.pdf](http://www.opencores.org/tmp/cvsget_cache/or1k/or1200/doc/or1200_spec.pdf)
- [5]. OpenRISC mailing list: <http://www.opencores.org/forums/openrisc?current=1>

# **EXTENDING THE OpenRISC 1000 implementation**

## **1. Introduction:**

### **1.1 OpenRISC 1000**

**OpenRISC 1000** (or1k) is an open source architecture definition. It describes the instruction set architecture, the special-purpose registers and other high-level details of the architecture. As an architecture definition only it leaves a lot of the details up to a particular implementation. Being open source there's no licensing issues involved and the definition per se is completely free, no strings attached. The details of the architecture definition are purposely left out of this document and the reader is referred to the Manual [1] and the or1k website [2] for the same.

The key features of this architecture are:

- ◆ Simple Load/Store Architecture
- ◆ 32-bit/64-bit logical address space
- ◆ Instruction Set Definitions:
  - OpenRISC Basic Instruction Set (ORBIS32/64)
  - OpenRISC Vector/DSP eXtension (ORVDX64)
  - OpenRISC Floating Point eXtension (ORFPX32/64)
- ◆ Support for shadowed register-files.
- ◆ Branch delay slot
- ◆ Two memory addressing modes
- ◆ Complete ISA definition and bit-encoding
- ◆ SMT and fast context-switching capability
- ◆ Architectural simulator (albeit limited)
- ◆ GNU tools: compiler, binutils ported
- ◆ ucLinux: embedded Linux ported

### **1.2 OpenRISC 1200**

**OpenRISC 1200** (OR1200) is one such implementation of the or1k processor family. The details regarding this particular implementation and the RTL code can be obtained from the webpage [3]. Key features of this particular implementation are:

- ◆ 32-bit scalar RISC with Harvard microarchitecture.
- ◆ 5 stage integer pipeline
- ◆ Virtual Memory Support (MMU)
- ◆ Basic DSP capabilities
- ◆ 1-way 8KB data cache and 1-way 8KB instruction cache
- ◆ Physically tagged direct-mapped caches
- ◆ 64-entry hash based 1-way direct-mapped Translation Lookaside buffer (TLB) for both instruction and data

The RTL coding of this implementation has been completed and has been tested on several applications. The RTL code is free and available off the OR1200 webpage [3].

## **2. Extending OR1200**

There are two ways in which the OR1200 implementation can be extended:

1. Add support for Fast-Context Switching.
2. Add support for Simultaneous Multi-threading.

There is support for both of these in the architecture definition i.e. both of these can be achieved by a particular implementation. The OR1200 right now does not support Fast-Context Switching or Simultaneous Multi-threading. Section 2.1 and 2.2 below in general explain fast-context switching and simultaneous multi-threading respectively.

### **2.1 Fast-context switching:**

When a particular program (**P1**) runs on a processor, the processor fetches and executes instructions from that particular program. The instructions modify the register values and the memory contents as they execute. At any point in time the values in the general-purpose registers, the special-purpose registers and the memory locations define the state of the program. Let's say the processor wants to execute instructions from another program (**P2**) while **P1** has not yet completed (this is called pre-emption). It not only has to start fetching and executing instructions from **P2** but it should also be able to continue execution of **P1** from the point where it was pre-empted by **P2**. In order to be able to do that it should save the state of program **P2** and retrieve it later on for re-execution. This whole process: saving the state of one program and loading the state of another program to continue its execution is called **context-switching**. The state of the program is also referred to as its **context**. So when would a processor want to pre-empt one program and run another program? A simple scenario would be when an interrupt occurs, and it needs to be serviced very quickly. In this scenario the processor pre-empts the current executing task and runs the instruction service routine (ISR).

In doing a context switch as described above a lot of overhead is involved in saving the context related information to memory and loading a new context information for the new program. Fast-context switching can be achieved by being able to maintain multiple-copies of the context (multiple register-files) in the processor/hardware. Only a particular context is active or executing at a time. A simple bit in a global register indicates the current active context. For doing a context-switch the bit needs to be changed to point to the new context that will be executing. The extensions needed here over a basic architecture would be to have support of multiple-contexts in hardware, and book-keeping information for each context (e.g. whether it is active).

## 2.2 Simultaneous Multi-threading (SMT):

Simultaneous Multi-threading is an additional extension over fast-context switching. In fast context switching only one context is active at a time. In SMT multiple-contexts are active and executing simultaneously. So the processor can fetch and execute instructions from multiple-programs at the same time. The processor updates the right-context depending on the context of the instruction (obviously each instruction is tagged with the information of the context to which it belongs). The extensions needed would be to have support for multiple-contexts in the hardware, to be able to fetch from multiple-programs simultaneously, to be able to distinguish between instructions from different contexts so that each instruction only modifies it's own contexts.

## 2.3 Putting it together

The user is encouraged to read the references on or1k and OR1200. Understand how fast-context-switching and simultaneous-multithreading can be done in OpenRISC and what are the implementation requirements. The OpenRISC team has their own mailing-list [5] which would be a good place to ask OpenRISC specific questions.

Reference:

[1]. OpenRISC1000 website: <http://www.opencores.org/projects/or1k/Home>

[2]. OpenRISC 1000 Architecture Manual:

[http://www.opencores.org/tmp/cvsget\\_cache/or1k/docs/openrisc\\_arch3.pdf](http://www.opencores.org/tmp/cvsget_cache/or1k/docs/openrisc_arch3.pdf)

[3]. OpenRISC 1200 website:

<http://www.opencores.org/projects/or1k/OpenRISC%201200>

[4]. OpenRISC 1200 specification:

[http://www.opencores.org/tmp/cvsget\\_cache/or1k/or1200/doc/or1200\\_spec.pdf](http://www.opencores.org/tmp/cvsget_cache/or1k/or1200/doc/or1200_spec.pdf)

[5]. OpenRISC mailing list: <http://www.opencores.org/forums/openrisc?current=1>

[6] Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor"

Link: <http://citeseer.nj.nec.com/tullsen96exploiting.html>

[7] <http://courses.ncsu.edu:8020/csc591k/lec/004/lectures.html>

[8] Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor by Dean M. Tullsen, Susan J. Eggers, Joel S.

Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm