COMPSYS 701 RECOP AND JOP TANDEM PROCESSOR

Zi Yang Chow and Leighton Hancock

Department of Electrical and Computer Engineering
University of Auckland, Auckland, New Zealand

Abstract

TP-JOP is a unique processor which better suits the requirements of reactive applications than traditional embedded processors. This paper covers the implementation of pipelined TP-JOP processor, starting from a reduced instruction set multicycle ReCOP. It discusses the design decisions and testing done in order to create a functional TP-JOP.

1. Introduction

Embedded systems can be introduced as systems that integrates with the physical environment, processing data collected from the physical world and decides the appropriate response. A reactive embedded system is usually programmed to handle real time data, meaning an output is required within a specific time constraint, ideally instantaneous after an input. Thus, these kinds of system will require a reactive processor in order to deliver proper response to the inputs.

In this project, we study the implementation of the ReCOP (Reactive microprocessor)which is a SoC (system on chip) designed with a simple instructions set , simple datapath in order to work in tandem with JOP. The ReCOP will act as the control processor whilst processing data via Java Virtual Machine (JVM) provided by JOP. Constructing a multi-processor system on chip like such with a tandem network between them increases the functionality of the system, allowing heterogeneous processing elements to utilise memory hierarchies and I/O components for their specific functions. SystemJ is a language comprised of Java, Esterel and CSP, making it a asynchronous/synchronous concurrent language. With the systemJ compiler, it can be split into ReCOP and JVM counterparts which are complimentary to each other.

The goal of the project is to create a pipelined ReCOP and JOP tandem processor,TP-JOP, tandem processing JOP. The new combined processor will be able to compile and execute systemJ programs within a single and multi-clock domain to show concurrency and reactivity of the system.

The rest of the report is organized as follows. Section 2. Overviews the design of the ReCOP processor. This includes the implementation process of the ISAs (pipeline insertion, data hazards, testing) and design decisions made to fulfill project requirements Section 3. Introduces JOP and its connection with ReCOP. Section 4 reports the final design and discusses different elements of the project.

2. Design

2.1. Design Diagram

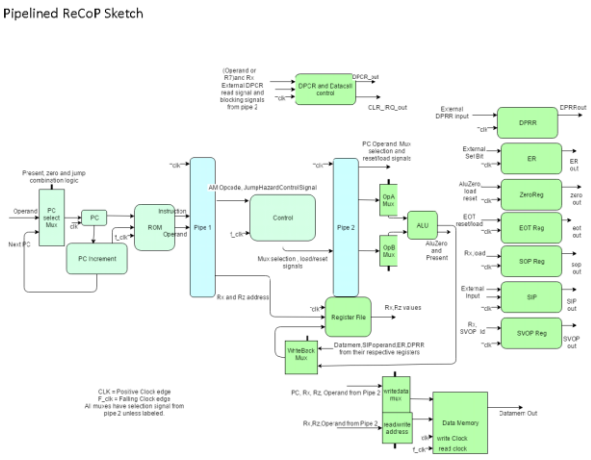

Figure 1: Sketch of Pipeline ReCOP (Larger version available in Appendix A)

In order to execute instructions efficiently delivered from JOP, the ReCOP will need to perform with low critical path and able to execute instructions continuously without wasting clock cycles. The design decision to make ReCOP a 2 pipe implementation is inspired by making the control unit easy to design. The tradeoff of this decision is the Fmax of the system as it may decrease depending on the critical path as the removal of the write back pipe (third pipe) poses a

longer datapath for one of the components. The design can be spilt into 3 distinct sections: the instructions fetch stage, decoding/control signals distribution and execution/write back.

The fetching stage is where the processor fetches the 16 bit instruction and operand if needed for execution. The notable components are the pc incrementer and dual port ROM. Besides pointing to the next instruction available, the pc incrementer also detects whether an operand is present due to the 16 bit word nature of ReCOP. This means when reading a certain instruction, the PC will be incremented by 2 if 16 bit operand is present according to the AM portion of the current instruction, else it will increment by 1 as usual to the next 16 bit instruction. There are 2 outputs of the incrementer, one that feeds back to the PC register and the other towards the second port of the ROM.

Dual port ROM is used in the design so the entire fetch phase can be done within a clock cycle. Accessing the ROM will produce a 1 clock cycle delay due to the RTL design of the component provided by MegaFunction wizard in Quartus, thus by placing the read on a falling edge we effectively utilized that lost cycle to output the instruction and operand to the first pipe.

The first pipe of the processor act as an instruction register to decode the instruction and feed it into the control. The control register readies the signals to be sent to registers and muxes for the next clock edge. Although still a separate entity itself, the register file is regarded as a part of the second pipe in the system. Both register file and pipe are accessed at the same clock edge so the operands can be delivered for the execution stage. This design ensures all the correct resources are sent out accordingly for execution in the next clock cycle.

The last part of the pipeline consists of the ALU for arithmetic operations, registers to store specific signals and data memory. Storage of signals and arithmetic output can all be completed within the clock cycle however, in the case of reading from the data memory to write into target register, the read function need to perform earlier than the next rising edge where the data fetched has to be written into the register. Writing into data memory can be done on the following rising edge of second pipe writing as it is still within the clock cycle.

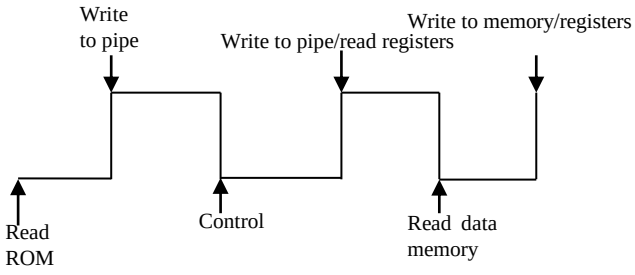

Figure 2: Showing operations at each clock edge

The ReCOP Instruction Set (ISA) is a mixture of simple instructions meant to work in tandem with the Java processor to compensate the short falls of Java e.g. Java does not have a jump or go to type instruction. The ISA includes arithmetic instructions ADDing, SUBtracting and ANDing for specific operations; read and write instructions to manipulate the memory elements in the processor, in this case register file and data memory, either for storage or access; implementation of the jump instructions allows the introduction of both conditional and unconditional branching; there are many instructions in the ISA which handles external signals delivered by JOP or user input, storing or resetting them to accommodate the data calls from JOP.

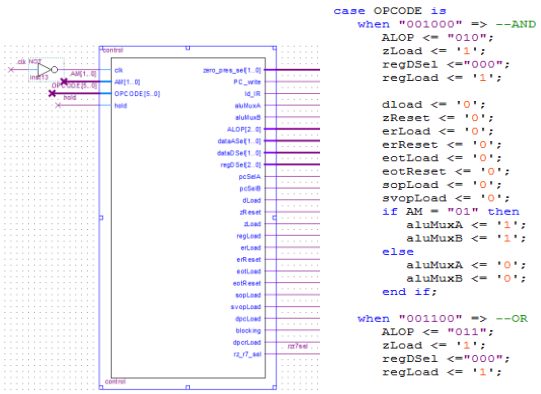

Figure 3: Diagram showing control unit design and logic

According to the design in the figure above, the control unit distributes the signals for the execution stage on a case by case basis by decoding instructions fed by the 1st pipe, splicing the 16 bit word into readable format. The addressing mode bits in the front determines whether an operand exist in the next word, the following 6 bit opcode determines which operation to execute, the following 8 bits provide register or memory address.

AM (2 bits)	OPCODE (6 Bits)	Rz (4 bits)	Rx (4 bits)

Figure 4: Diagram showing splicing of 16 bit word instruction

2.2. Data and Control Hazards Detection

There were not many hazards to account for with the ReCOP ISA. A single write after read (WAR) data hazard could occur when an instruction in the execute stage will write on the clock edge while the next instruction in the pipeline will read from the register file on the same clock edge. This results in old data being used for the next instruction. A simple check in the

register file compares the read and write addresses and forwards the new data if needed.

Control hazards were also present due to jump type instructions. The method used to avoid these hazards was very simple. A small unit was added that checks the inputs to the PC multiplexers. Whenever the multiplexer signals were in a configuration that caused the next count to not be sequential, the unit would output a signal clearing the pipe stages. Therefore at any point where the ReCOP jumps, the pipes are cleared resulting in two unused or missed clock cycles. This method was chosen due to its simplicity and of the opinion that two missed cycles is a worthwhile tradeoff.

2.3. ReCOP Implementation outcome and testing

The implementation was a success after a total of 13 hours spent on the process of routing, programming and testing. A working multicycle edition of the ReCOP is used as the foundation for the pipeline design. The pipelines and registers were then realigned and revamped with several components removed such as the instruction register as it performs the same function a pipeline, optimizing the data flow and reducing the number of logic elements as much as possible. External registers and ALU were implemented last so debugging can test the execution stage by testing the data flow from stage to stage as each section is implemented. Both data memory and read only memory are created with the Megafunction wizard in Quartus, both contain dual port option and dual clock for the data memory. Any external inputs to the ReCOP including signals receiving from the JOP at the later stages were implemented as input ports.

Stage 1	Pipeline multicycle design of ReCOP.
Stage 2	Extend instructions adding registers and fifo to accommodate other functions.
Stage 3	Connect JOP to ReCOP, ensuring both blocking and non-blocking on single and multi-clock domain.

Figure 5: Table of Implementation stages

Initially the control is designed as a combinational unit, out of the pipeline flow, however that leads to synchronization problems within several instructions where required operands for the ALU isn't present at the time of execution. After re-shuffling of the registers, placing the control as per final design ensures no lag between executions and decode for all instructions. Originally the data memory settings are the same as the ROM besides the presence of a write port, the second clock is only implemented later when the write back from memory to register executes on a unwanted fourth clock cycle.

Testing is all done on modelsim, by manually checking the signals or data travelling in the datapath. Each component in the processor is assessed individually to ensure functionality requirements met for each instruction and any changes made on the components does not affect the execution of other instructions. Most of the debugging is done by checking the output of the control unit whilst referencing the signals going into each component and the clock. Data memory and register writing are checked via the memory view option within modelsim. Corresponding signals with tested instruction were isolated to check the input and output (read and write) signals in a register is correct.

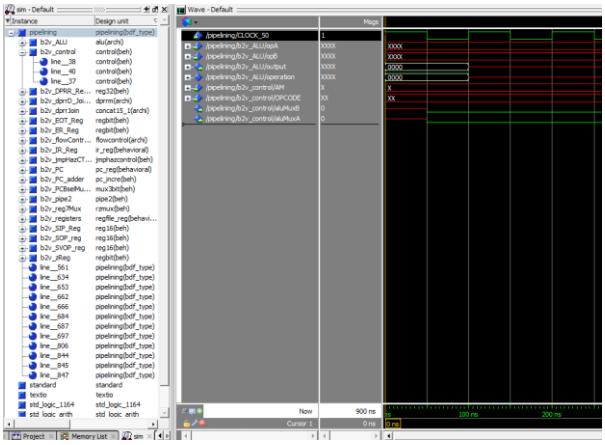

Figure 6: Snapshot of debugging ADD instruction on Modelsim by isolating appropriate signals

During a blocking data call or new result returning, the ReCOP completely pauses all processing. Allowing returned data to be written to the registers or data memory based off the type of data call that the system is running.

2.4. JOP and SystemJ

The Java Optimized Processor or JOP is a programmable processor designed to execute Java bytecode. The only configuration that the JOP needed was connection to the ReCOP which is explained in the next section.

SystemJ is a language for designing and programming concurrent and distributed systems. In order to use SystemJ with the TPJOP, it must be compiled into Java code and ReCOP assembly.

2.5. Connection of JOP and ReCOP

Despite the fact that the ReCOP 'controls' the JOP, the ReCOP is connected to the JOP as a peripheral using the SimpCon (SCIO) interface. The JOP polls for a data call by reading from the DPCR and executes calls whenever one is present in the ReCOP. The JOP writes to the ReCOP once it has completed the processing causing the ReCOP to temporarily pause to insert this data into the processor.

Initial connection of the ReCOP with the JOP did not work. It was thought that the processors were not connected correctly. After changing the channel that the ReCOP was connected to also did not solve the problem. Through testing signals in the system using LEDs we found that the IRQ was connected to the wrong DPRR bit. Once this was found all issues regarding blocking data calls were resolved.

In order to run non-blocking data calls, a FIFO buffer was needed. This was implemented in the ReCOP. Using the FIFO that is available with the SCIO, neither blocking nor non-blocking data calls to work correctly. ModelSim was used to try and understand how the SCIO FIFO worked, however the system would not work correctly in simulation or practice. A custom FIFO was written which worked correctly. The FIFO was made as a register file with head and tail pointers that increment based off data being written to the DPCR and results being returned by the JOP respectively.

2.6. Final Outcome

A TPJOP that works with blocking data calls in single and multi-clock domains and non-blocking data calls in a single clock domain.

I why the TPJOP does not work with non-blocking multi-clock domain. Further investigation is needed in order identify and resolve the cause of the issue.

	FMAX (MHZ)	LEs
ReCOP	66.83	1957
TPJOP	69	6883

3. Discussion

There were several issues with the tools that were provided. The systemJ compiler when compiling for non-blocking code, the resulting java code would be missing characters when in the main loop when trying to invoke the clock domain method(s).

The asm compiler (mrasm) would generate the mif file with a depth variable of 1024. This would need to be changed to fit the depth of the ReCOP implementation (32768).

These were small problems but were minor nuances when trying to compile and run code.

4. Conclusions

Starting with a reduced instruction set multicycle implementation, we developed the pipelined implementation of ReCOP, adding the remaining instructions. Extensive testing was carried out in order to validate the robustness of the design. Combining the ReCOP and JOP proved to be a challenge due to a minor error in the initial pipelined design. Once this was resolved, the TPJOP worked flawlessly with non-blocking data calls. We found out that the provided SCIO FIFO did not suit our requirements and implemented our own FIFO. This allowed non-blocking data calls to run correctly. However with one caveat, the non-blocking multi-clock domain programs did not work as intended. Due to time constraints the cause of this problem was unidentified and will need to be investigated further.

5. References

UOA ECE Department, Embedded System Research Group, "ReCOP - Reactive Coprocessor for Tandem Processor Embedded Execution Platform for SystemJ Language (edited)," 2014.

Appendix A

Pipelined ReCoP Sketch