Implementation of Multi-Threaded MSP430 CPU Revision 1

Abstract

This realization of a multi-threaded MSP430 processor is based on a latency tolerant process model. Generally, process network models are mainly used to represent large distributed heterogeneous systems. However, the capability to allow function isolation and structured sequential relationships benefits even a tiny embedded system. In this report, the design of a multi-threaded MSP430 is described. This design is traced from planning through optimization of a single-threaded core with 2, 3, 4 multi-threaded capability. An overview of the design planning, architecture, behavioral specification and optimization are presented with an eye towards a simple multi-threaded extension of the basic design. The final results show that the multi-threaded MSP430 achieves higher throughput than TI's MSP430.

1 Introduction

Embedded systems are dominated by tiny microcontrollers typically characterized by simple serial architectures. A common theme in such systems is the requirement for real-time constraints on the system execution. This must be accomplished at minimal cost and for many practical designs obviates multi-core techniques. Indeed, the usual case for embedded systems is the notion of "adequate" performance – i.e. one that meets the real-time requirements in all cases. There is little if any benefit to higher than adequate performance.

One way to meet such goals is to build a hardware multi-threaded architecture. The basic idea is to provide a few copies of the processor context which are dynamically used to enable instruction-level switching of activity from one context to another. A common scheme would be to have a fixed number of copies of the processor context, one for each co-executing process (or thread). Instructions for each of the contexts are executed on a simple pipeline with instructions from each context interleaved dynamically. The point of this effort is to greatly increase the flexibility of the processor as to which context is currently executing. Since there are several stored execution contexts, context switching overhead is virtually eliminated. A second benefit is that the pipeline can often be substantially simplified since the next instruction to be executed is likely not to be from the current context, and thus not a possible source of pipeline hazard.

In the following, we shall describe the basic design ideas and organization for this version of MSP430 which is extended to support up to 4 simultaneous processes executing in parallel. First, the overall design is presented, then a simple single-thread version is described followed by a multi-threaded version.

2 Major Concerns of Multi-threaded designs

Usually, “multi-threaded” means dispatching and/or executing multiple instructions concurrently. However, in this particular multi-theading MSP430 design, only single instruction dispatch is supported. This reduces complex modifications to the existing design while allowing much higher performance – without the cost of multiple in-flight instructions per thread which requires complex analysis of data hazards resulting from memory-based operands. The multi-threaded msp430 is an interleaved pipeline and uses single-instruction-issue. Instructions are selected from a possibly different thread every cycle. This strategy provides implicit data path sharing and eliminates data dependencies between consecutive instructions, alleviating many costly stall conditions and providing better utilization of processor resources. A memory-mapped infrastructure is used throughout the multi-threaded MSP430, reducing design complexity by providing a uniform interface to system peripherals and memory. Figure 1 gives an example of how this multi-threaded MSP430 reduces data hazards.

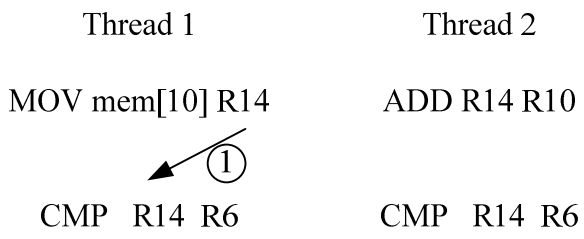

Figure 1 multi-threaded design reducing data hazards created by in flight instructions in one thread

Thread 1 first writes to Register R14 and the next instruction reads R14. The second compare instruction CMP which reads R14 cannot proceed until previous instruction gets completed. This is called read after write data hazards. However, if multiple threads are used, following the execution of MOV in thread 1, instead of dispatching CMP, the addition instruction ADD in thread 2 is issued. By the time the CMP instruction of thread is issued, its previous move instruction already proceeds one clock cycle and reduces stalling cycles of the CMP instruction of thread 1. If more threads are used and they are interleaved, then more data hazards can be avoided. The main considerations of a multi-threaded design are listed below.

1. **Architecture**

   A memory-mapped infrastructure is used throughout the design, reducing design complexity by providing a uniform interface to system peripherals and memory. By supporting interleaved multi-threading, this new design provides implicit sharing of data path resources for increased throughput and low-latency response to dynamic events. Instead of using the dispatch unit in a single-thread MSP430 design, a custom thread scheduler is used and dynamic interrupt handling is implemented inside that scheduler.

2. **Pipeline**

   The event-driven nature of embedded systems means that control flow will change often and unexpectedly; a successful system should minimize the overhead of these transitions. One approach is to use a dynamic pipeline which executes instructions in anywhere from one to several cycles. PyTDL, explained in detail in the reference guide, allows a straightforward path for realizing interlocked pipelines that are readily extensible and lead to high-performance implementations.

3. **Schedule**

   Instead of a conventional dispatch unit, a thread scheduler is used to support multiple concurrent streams of execution. This multi-threaded scheduler is a best-effort, round-robin scheduler targeting balanced execution among threads (no starvation). A round-robin scheduling policy is utilized based on the last dispatched thread, the pool of active threads, and the state of interrupts.

   One challenge of adding multi-threading to a single-threaded system is thread initialization, as each processor context has no direct access to any other. To facilitate this process, the scheduler provides initialization registers which are used when a thread has yet to execute. Modifications to the execution state, history flags and the target instruction address allow any thread to modify the execution flow of another context. The ability to begin execution at a specified function enables thread-based activation of arbitrary code segments.

   System execution always begins in thread zero which enters program code at the beginning of memory. This thread is effectively used to bootstrap the initialization of other threads. Shared memory provides an easily-used communication channel for information passing between threads. This is of particular importance during thread initialization, where an appropriate address for the stack pointer must be established to avoid memory corruption. On acceptance of an interrupt, the scheduler overrides the round-robin policy, scheduling the interrupt at the soonest possible time.

4. **Memory**

   To reduce the programming complexity, a uniform interface to peripherals and memory is used. This allows our controller to handle a large number of different on-chip structures without requiring specialized design. Memory-mapping serves to reduce complexity stemming from otherwise necessary custom instructions and component interconnect. From an implementation standpoint, this methodology reduces the overhead of adding custom peripherals and control elements since interfacing requires no additional hardware. The use of a common memory for process context information and data will have performance consequences later in this design.

5. **Instruction Fetch**

   Many available microcontroller devices provide embedded memories for both program and data. Some devices, such as ARM, use a boot-strapping technique to move program code from non-volatile memory to volatile memory before beginning execution of the main program. With multi-threading, one of the

concerns is the increased pressure on instruction fetch. Many of the embedded applications are very small, on the order of hundred of bytes, and can be pulled completely on-chip. The ability to do single cycle access to all of memory greatly improves processor throughput when compared to the penalties incurred from instruction cache implementations and external memory access. In this multi-threaded MSP430 design, all the code is stored in ROM.

6. **Thread Context**

   This multi-threaded msp430 design implements unique contexts for all of its 4 hardware threads. Each context consists of a 16-bit wide register file, status register, program counter, and stack pointer. Similar to TI's msp430, this new design provides 11 16-bit general purpose registers. Rather than implement independent register files for each thread, a uniform memory can be optimized for speed, area, and locality with data path components. Utilizing the thread identifier as part of the register file address provides a straightforward method for accessing and updating data with very little added complexity.

3 Design Planning and Behavioral Specification

The design flow begins by identifying major design components. A primary goal of this design is that the core be self-contained so as to be amenable to multi-core implementations. Creating interface boundaries between memories and the functional core aids subsequent modification to the design. For example, multiple cores could be implemented with shared memory by simply adding arbitration on the interfaces – a change that should not require modification to the core design.

Given the behavior of multi-core and multi-threading, it is natural for memory interfaces to become execution bottlenecks of this MSP430 design. Careful observation shows that arbitration can be handled better at a lower level where memory activity timing can be more precisely accounted for. Consider the case where two threads are executing concurrently. The first thread has issued a read request to the memory and is stalled waiting for completion and the second thread is sending a write request to the same memory. If implemented as a shared memory, the second thread cannot proceed before the completion of the first thread. However, if the memory unit is constructed as a pair of read and write interfaces, the memory can allow the write to proceed in parallel with the read operation. Figure 2 gives the diagram.

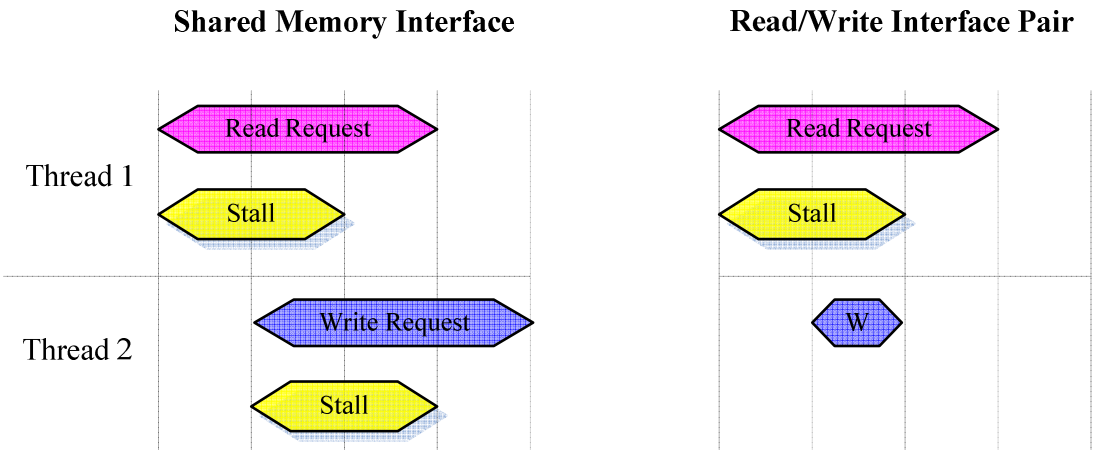

Figure 2. Comparisons of concurrent memory requests between a single shared memory interface and a read/write pair interface.

To facilitate the above memory interfaces, both memory and register file arbitration are constructed in Verilog HDL – a limitation of the current PyTDL implementation is that control observation occurs only in the previous cycle of activity. A Verilog RTL arbiter design however, allows multiple interfaces and memory state to be considered when making arbitration decisions. Furthermore, deterministic timing of these low-level interfaces is necessary to ensure design efficiency. To facilitate the accommodation of multi-threading and multi-core, the dispatch unit is separated from the core design.

The system is divided into four major compoents: the dispatch unit, the processor core, the register file and the memory. Figure 3 depicts the relationships between these four major components where the processor core interfaces to the register file and memory through read/write interface pairs. Though all interfaces to the core unit must respect the valid/stall control communication scheme, only the core is fully latency tolerant.. In factor, the organization allows the core to be considered as a variable latency function with old program counter (PC), stack pointer (SP), status register (SR) as inputs and new PC, SP, SR as outputs, shown as {PC', SP', SR'} = core (PC, SP, SR).

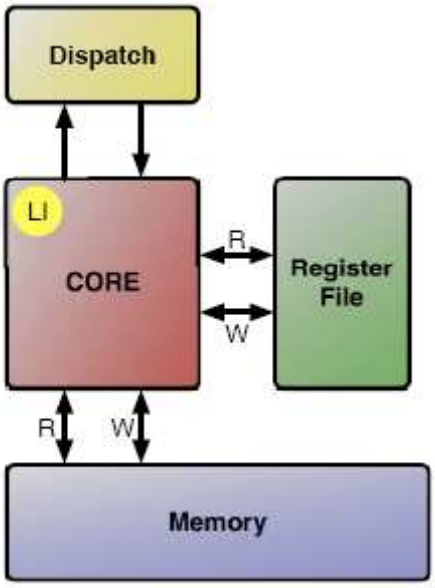

Figure 3 relationships between major components

3.1 PyTDL implementation of the processor core

System behavior in PyTDL is best described through a set of atomic behaviors and their causal connections represented using token-based control flow. Thus the starting point of PyTDL design is to decompose the core behavior into a subset of behavioral modules based on their causal relationships. Before we go into the actual design, let's first take a look at the architecture of the processor core.

3.1.1 Processor core Architecture

The MSP430 supports 27 core instructions and 24 emulated instructions. The core instructions are instructions that have unique op-codes decoded by the CPU. The emulated instructions are instructions that make code easier to write and read, but do not have op-codes themselves. Instead they are replaced automatically by the assembler with an equivalent core instruction. The actual instruction execution flow is shown in Figure 4.

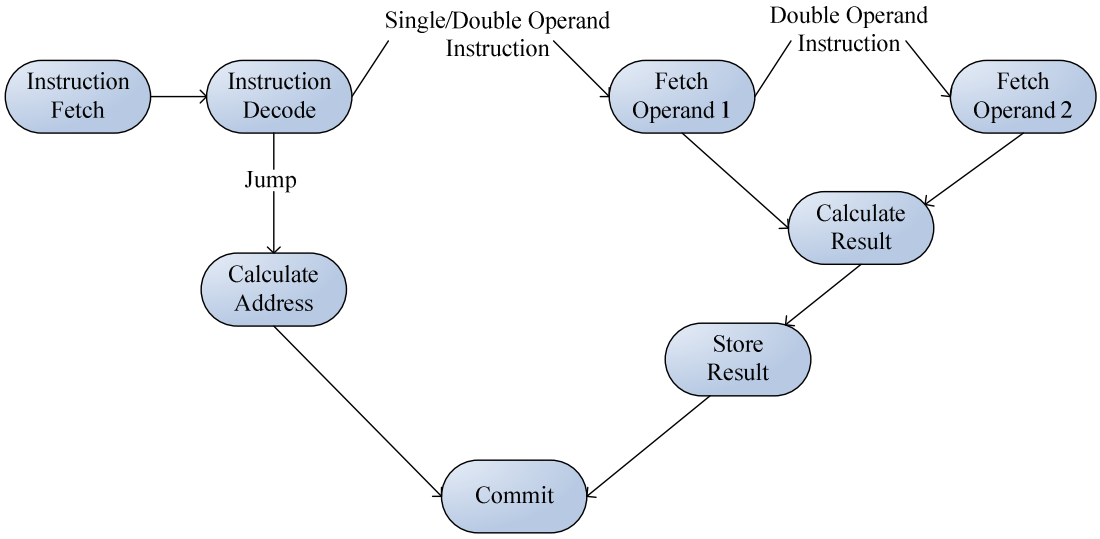

Figure 4 instruction execution flow of the MSP430

Although the control seems to be simple based on high level description of the execution flow, it is deceptive. In fact, there are seven different addressing modes for the source operand (having different timings) and four different addressing modes for the destination operand. Handling these modes in a traditional pipeline would require execution division such that common functions could be extracted from instructions and assigned to specific stages. For instance, memory and register file access typically occur in specific stages, preventing architectural hazards that would impede pipelining. Moreover, a fixed-length pipeline would require many stages to accommodate the available addressing modes, because variant addressing modes result in variant instruction execution cycles. In PyTDL however, pipeline architecture is synthesized and execution path is dynamic. Designs of this style are quite different from those of conventional fixed-length pipelines, requiring different methods to determine architectural latencies.

3.1.2 Processor core specification

A full description of the core specification can be found in TI's website. Here we give a description PyTDL particular token flow for a single operand instruction. Figure 5 shows the actual token flow. Following the diagram in Figure 4, the core specification is further decomposed such that each sub module, represented as nodes in Figure 5, corresponds to a PyTDL atomic behavior. At node the behavior is then elaborated, defining requesite stages for the sequential token flow.

The core execution starts at the instruction fetch node where a memory read request is sent out with address specified by the program counter (PC). The instruction fetch node is replaced by the cycle module, triggered by the external world, for example, an interrupt. The cycle module generates a read request to the ROM through tRomRead token to fetch the instruction. Inside the cycle module, tFetch token is created to establish the pipeline flow and pass values of PC, SP and SR down the pipeline. tRomReadDone signals the completion of ROM read and is sequentially

analyzed by romReadDone behavior which generates tInstructionReady token. The tInstructionReady token along with tFetch token triggers the decode behavior where instructions get decoded. The decode behavior processes the instruction by inspecting their instruction formats. If it is a jump, it flies to the end of the execution through the creation of tCommit token and increases the PC by 2 or adjusts PC to the new destination depending on whether the condition is satisfied. Otherwise it checks the source operand no matter it is a single operand or double operand, because MSP430 microcontroller is designed to calculate the source operand first for a double operand instruction. Calculation of the destination operand may be based the value of source operand.

If the source operand is PC, SP or constant generator, which can be obtained directly from the corresponding register, control flow is routed to the rsGetRegDone behavior through the creation of token tRsIsInternal. Otherwise the source operand is indexed addressed and a register read request is generated through the creation of token tRegRead and token rRsRegWait is created to establish the internal control path.

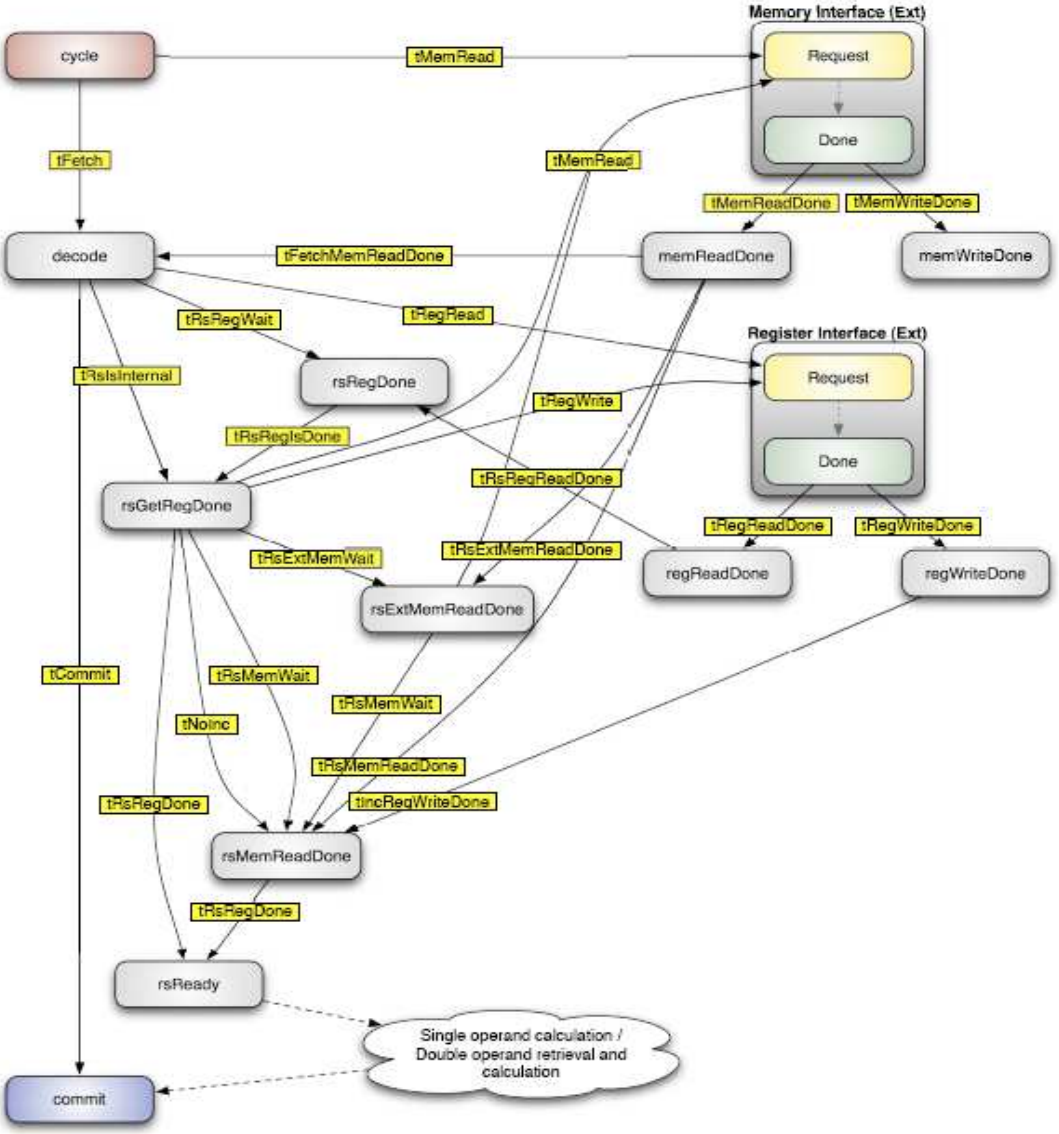

Figure 5 Actual token flow of the PyTDL design of the MSP430

In contrast to ROM Read, a register read request token is tagged to disambiguate the control path when the register read is completed. This is because multiple behaviors make register read requests via token tRegRead. This tagging method is the application of the context-aware concept, which allows local indeterminism but guarantees determinism of the whole system. Once register read is completed, tRegReadDone token with an associated tag is created and processed by the regReadDone behavior which routes the control to rsRegDone behavior through token tRsRegReadDone. The arriving of tRsRegReadDone and rRsRegWait indicates register operations for the source operand are comleted and register values are passed down as pipelined data members.

Regardless of how the source operand is obtained, the control converges to rsGetRegDone behavior. If the source is directly addressed or is a constant value or is in the PC auto-increment addressing mode, token tRsRegDone is created. Otherwise the source is indexed addressed and a memory read request is generated through token tMemRead. Similar to token tRegRead, token tMemRead/tMemWrite has an associated tag value to disambiguate the control path, because memory is read or written by multiple atomic behaviors. The last thing that rsGetRegDone needs to take care is the write-back to register file if the source operand is auto-increment addressed.

Source operand retrieval completes when control reaches the rsReady module. If the instruction is a single operand instruction, then the instruction result is calculated and memory/register writeback is initialized. Otherwise fetching of destination operand begins. The process of fetching the second operand is similar to the fetching of the source operand and is a little bit easier, because there are only four addressing modes for the destination operand. After both operands have been fetched, the result is calculated and memory/register writebacks are initiated. Both single and double operand instructions signal completion via toke tCommit. The commit behavior calculates new values of PC, SP and SR for next round execution.

4 Optimization

After seeing the token-based flow of this PyTDL design of the MSP430, one might argue that this design must have really long execution clock cycles. Indeed, some instructions where both source and destination operands are indexed addressed require more than 20 clock cycles. The token-heavy PyTDL design provides great design flexibility by sacrificing system performance. However, the number of tokens used in the design is always decided by the designer. One could write an alternative specification for the core unit where tokens were only used at interface boundaries to facilitate communication with external components. Specifying the core that way can decrease the number of clock cycles to execute some instructions, but the readability decreases too.

4.1 Optimization through functional composition.

Instead of using fewer tokens in specification description, one can “short” tokens

during optimization step through a constraint file without changing PyTDL specification, thus maintaining the same readability. *Shorting* tokens means selectively removing pipeline buffers by composing functional behaviors. A shorted token becomes a pure combinational function, but still obeys SELF protocol to provide valid/stall interface. For comparison, we use a 16-bit CRC algorithm along with a mix I/O for reporting results and progress. The results without and with optimization are list in table 1.

Table 1 CPI characteristics of the core unit with and without token shorting optimizations for Flip-flop

Description	Shorted Tokens	Throughput	Latency per instruction
Baseline	None	0.071	12.1
Memory/register file routine	tRsMemReadDone, tRdMemReadDone, tRetiSrMemReadDone, tRetiPcMemReadDone, tRsMemWriteDone, tRdMemWriteDone, tIntPcMemWriteDone, tIntSRMemWriteDone, tRsRegReadDone, RdRegReadDone, tRsRegWriteDone, dRegWriteDone, tIncRegWriteDone	0.081	10.4
Memory/register file and internal routing	tRsMemReadDone, RdMemReadDone, tRetiSrMemReadDone, tRetiPcMemReadDone, tRsMemWriteDone, RdMemWriteDone, tIntPcMemWriteDone, tIntSRMemWriteDone, tRsRegReadDone, RdRegReadDone, tRsRegWriteDone, RdRegWriteDone, tIncRegWriteDone, tInstructionReady, tIntMemReadDone, tRsIsInternal, tRsRegIsDone, tRdRegIsDone, tRdRegDone, tRsRegDone,tCommit, tDone	0.118	6.3
48-bit instruction fetching	the same as above	0.123	6.1

Without optimization, the throughput is very low and instruction latency is very long.

The first optimization is carried out by composing post operation routing of memory and register file results. Token-shorting in memReadDone and regReadDone behaviors is the example. By shorting tokens created in these behaviors results in the routing function inlined with subsequent behaviors. This optimization improves throughput by 0.01 and decrease latency by 1.7 clock cycles on average.
The second optimization adds shorting of internal tokens and further improves throughput by almost 0.04 and decreases latency by almost 4.1. Thus the total throughput improvement is 0.05 and latency improvement is almost 6 clock cycles.
In addition to affecting the sequential depth on instruction execution, token shorting may affect both maximum clock frequency and design area. When a token is shorted, the corresponding buffer is removed from the circuit, contributing to a buffer area reduction. When a token is shorted, the resulting combinational function is added to its sequential behavior, resulting in larger and deeper combinational circuits, which may potentially increase the critical path and thus increase minimum clock period. Design vision from Synopsys will be used to synthesize the design and tradeoff among throughput, latency per instruction, maximum clock frequency and design area will be shown. These results will be updated later.

4.2 Optimization through instruction Fetching

In the MSP430, instruction length varies from one 16-bit word to three 16-bit words. A double operand instruction may need as many as 3 16-bit words if both source and destination operands are indexed addressed. Instead of accessing these additional words on a by-need basis, the core unit could always fetch 48-bit data to accommodate all potential instruction lengths. However, 48-bit data does not align well, so instead, a 64-bit interface is used to facilitate the alignment of 48-bit data. This is implemented through two 32-bit wide ROMs. To accommodate this in RTL would need lots of changes to state machines and functional interfaces. In PyTDL, it is easy to implement by extending the Instruction object to allow named access to these new fields along with a ROM interface which is around 50 lines. Figure 6 shows the organization of the two interleaved ROMs, where data is represented in hexadecimal mode.

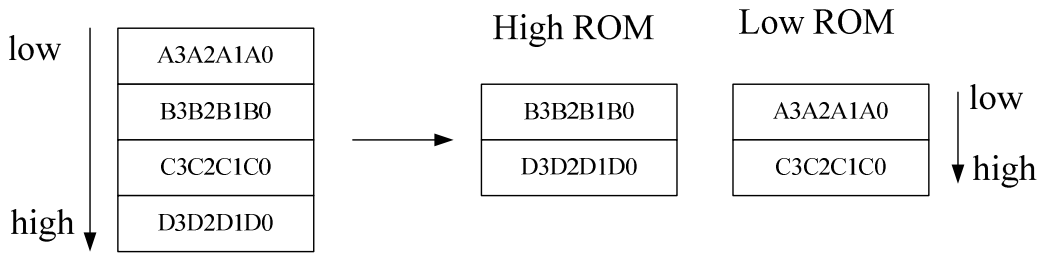

Figure 6 division of a 32-bit ROM into two 32-bit ROMs each with half space

The whole ROM space can be seen as the interleaving of the low ROM and the high ROM. The trick to guarantee correct execution and fetch length-variant instruction in one cycle lies on the novel use of the three least significant bits of the address. Depending on the address, the arbiter can access 2 different 64-bit chunks by

addressing the high 32 bits different than the low 32 bits. For instance, the arbiter can access either [B3B2B1B0 A3A2A1A0] or [C3C2C1C0 B3B2B1B0]. Then the lower 2 bits are used to index into the 64-bit chunk to return a 48-bit value. The possible results for the 48-bit value can be B1B0A3A2A1A0, B2B1B0A3A2A1, B2B3B1B0A3A2, 0B3B2B1B0A3, 00B3B2B1B0.

Due to the infrequency of indexed addressing in this CRC algorithm, this optimization strategy only shows a very slight throughput improvement and latency decrease. The reason is because multi-word instructions are not very often in CRC algorithm implementation.

5 Adding multi-threading

Inspired by the design and implementation of JackKnife architecture, which shows that multi-threaded has quantifiable hardware and software benefits, we incorporate multi-threaded to single-threaded MSP430 design.

This implementation of multi-threaded MSP430 requires only 10 extra lines to the core description in PyTDL to specify thread identifier of the inputs and outputs for register read and write requests, due to the face that each thread has its own register space. Another modification is the replacement of the dispatch unit with a threading unit which performs thread scheduling and maintains PC, SP, SR context for each thread. The threading unit also provides a memory mapped interface to these register values as well as a register designating state: idle or active. Depending on design needs, thread registers can be mapped to different peripheral range of the MSP430.

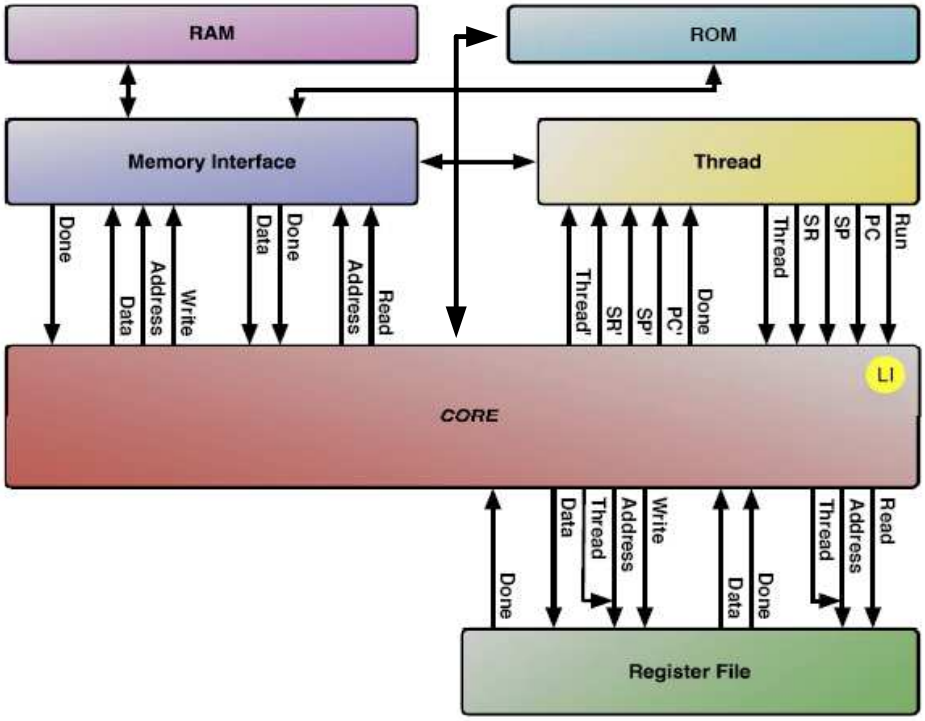

Figure 7 Architecture of Multi-threaded MSP430 design

One test case for this multi-threaded msp430 is based on the CRC algorithm. Each

thread is running on a CRC algorithm independently and instructions are interleaved from the beginning of thread activation. Table 2 shows the results of the design which is optimized with 48-bit instruction fetch and maximum token shorting.

Table 2 Results of multi-threaded MSP430 running on the CRC algorithm

Num of thread	Throughput	Latency per instruction
1	0.123	6.1
2	0.243	5.9
3	0.347	6.1
4	0.427	6.3

It is expected that the addition of multi-threading should provide significant improvement to processor utilization by filling the substantial waste created by single-instruction-issue execution. Moving to 2 threads is shown to increase average throughput by almost 2x while 3 threads provides a 2.8x improvement and 4 threads provides a 3.5x improvement. The fact that throughput improvement is not proportional to the number of threads is because by adding more threads the critical path becomes longer.

By adding more threads to the design, area and clock frequency are subject to change. These will be verified later. However, we estimate little area increase and little decrease of maximum frequency because the code change to add multi-threading to the design is very small, comparing to the size of existing design.

6 Interrupt

Efficient use of system resources often necessitates the use of interrupts. One characteristic of interrupt is the interrupt latency. AVR interrupt routines typically require prologue and epilogue code as long as 17 instructions, totally 4 clock cycles. This long interrupt latency is partly because there is no hardware support for fast interrupt context switching. In contrast, this multi-threaded msp430 design provides zero-cycle context switching through priority scheduling and dedicated interrupt service threads (IST). ISTs allow user code to be executed immediately, without the overhead of prologue and epilogue code. Table 3 summaries the interrupt response time of this multi-threaded msp430 design.

Table 3 Interrupt response time w/o dedicated IST

	With dedicated IST		Without dedicated IST
Scheduling algorithm	Round robin	Priority scheduling	Xxxx
Interrupt response time (# of clock cycles)	n/2	1	17 * (clock cycles per instruction)

Table 3 shows that with a dedicated IST, the interrupt response time is either 1 clock cycle or n/2 clock cycles depending on which scheduling algorithm is used where n is the number of threads. However, without a dedicated IST, the average interrupt

response time is the execution time of 17 instructions, with most of the instructions being push and pop, according msp430-gcc stack point standard. Because of the one instruction response time, the interrupt timing jitter is fundamentally reduced.

7 Peripheral

The embedded domain comprises of many different applications requiring integration with various component types. Because the peripheral sets of commercial devices are fixed, families of devices are typically offered with varying peripheral sets and memory sizes. It is often the case that these predetermined peripheral sets are not well suited to a particular design. Similar to the JackKnife design, this multi-threaded MSP430 provides the ability to customize system peripherals for a given application. Central to this capability is an extensible bus architecture that links the processor core to both data and I/O memory. This decision to fully support a memory-mapped infrastructure allows integration complexity to be masked, and provides a uniform interface to all system components. All peripheral modules adhere to a common bus policy in which write and read requests are specified as single-cycle operations. One extension to this design is to support the use of memory wait signals for slow memory devices. However this extension has the potential to create more stall cycles.

One specific aspect of TI's MSP430 design is that the multiplier is implemented as a peripheral. In this multi-threaded MSP430 design, we do the same thing. Table 4 shows the test results for Fast Fourier Transform (FFT) algorithm.

Table 4 Results of FFT implemented in multi-threaded MSP430 design

Num of threads	Throughput	Latency per instruction in clock cycles
1	0.124	6.0
2	0.246	5.9
3	0.349	6.0
4	0.428	6.1

Results in table 2 and table 4 shows that operation overhead does not increase with the multiplier included. This is because in the design, peripherals are made to share the common bus policy where read and write requests are single-cycle operations -- the same as memory read/write requests.

8 Conclusion & Future Work

In this paper, we talked about the implementation of a multi-threaded MSP430 design using PyTDL and showed the results. Through token-shorting and instruction fetch optimization strategies, the throughput of the design goes up to 0.42 instruction per clock cycle. It is expected that subsequent revisions will achieve better performance through greater optimization of both control and functional partitioning. Many such optimizations are possible given the flexibility of PyTDL's

design paradigm.

To make this multi-threaded MSP430 of industrial value, more thorough tests are needed. One approach is to use some MSP430 simulators to randomly test the design. An extension of the current multi-threaded MSP430 is to make the entire processor latency insensitive. RTSC created by Bob Frankle from TI, provides the software platform for multi-threading tests. Eventually we are going to synthesize this multi-threaded MSP430 design onto a FPGA board to emulate it, or even build a full customized design.