A Dual-Issue Embedded Processor For Low Power Devices

Hanni Lozano [1], Mabo Ito [1]

[1] Department of Electrical & Computer Engineering The University of British Columbia , Vancouver, Canada

Abstract—**While energy efficiency is essential to extend the battery life of embedded devices, performance cannot be ignored. High performance superscalar embedded processors are more energy efficient than low performance scalar processors, however, they consume more power which is very limited in battery operated or self powered embedded industrial devices. In this paper we propose an energy efficient dual-issue embedded processor that can deliver up to 60% improvement in IPC (instruction-per-cycle) performance with less than 20% increase in power consumption compared to a single-issue scalar processor. In contrast to traditional multi-issue embedded processors that use power intensive superscalar techniques to extract instruction-level parallelism from applications, the proposed processor uses simple hardware techniques to resolve instruction scheduling conflicts. The processor is optimized for implementation on low cost FPGA which makes it a suitable candidate for cost sensitive embedded industrial applications.**

Keywords-embedded processors; low power

I. INTRODUCTION

The challenge in designing energy efficient embedded devices is how to increase the compute performance while using the least amount of power. Most embedded communication devices like remote sensors in industrial applications operate on batteries and in some cases these batteries cannot be replaced so frequently or at all. In these cases, alternative energy sources like energy-harvesting are used. However, a major drawback of energy-harvesting technology is the very limited supply of power that can be generated from them. For example, a 36.5 x 64 mm 3Volts solar panel from PowerFilm delivers 66 mWatts in 100% sun and less than 20 mWatts in 25% sun which is a typical threshold intensity used by portable devices [1].

Despite the reduction in power budgets, demand for higher performance is increasing as more functions, for example different types of sensors, are incorporated into modern embedded devices. In order to design an energy efficient solution, power and performance need to be properly balanced. Superscalar processors can issue multiple instructions per cycle and, therefore, are more energy efficient than scalar processors [6]. However, superscalar processors use complex hardware techniques to extract instruction level parallelism from applications which increases power consumption. The solution is to design a multi-issue processor that consumes comparable power to a single-issue processor.

We propose a dual-issue embedded processor that delivers up to 60% higher IPC than a single-issue scalar processor while using only 20% more power. The average increase in IPC for all the embedded benchmarks that were simulated is around 40%. The proposed processor has two instructions wide pipeline that can fetch, decode and issue two instructions simultaneously every cycle. Increasing the number of instructions that are issued every cycle reduces applications run-time and allows the processor to be turned off for longer periods to save energy similar to a superscalar processor. In contrast to superscalar processors, we use simple hardware techniques to extract available instruction level parallelism from programs without adding too many additional hardware resources, thus keeping power consumption down. For example, the dual-issue processor uses no more than 30% additional logic gates than the single issue scalar processor when implemented on a low cost FPGA.

The main characteristics of the processor are: a short four-stage pipeline that executes a rich 32-bit RISC instruction set based on the MIPS32 instruction set [2], tightly coupled program and data memories (TCM), a static branch predictor that performs within a 5% range of an ideal branch predictor, a result forwarding scheme that uses FPGA built-in features without any additional external logic, and an optimized multi-port register file that uses a novel reduced gate count memory redundancy technique. A custom cycle-accurate C simulator is used to evaluate architectural design decisions in the early phases of the design process. The final single-issue and dual-issue processors are implemented in a low cost FPGA to measure performance, power consumption and hardware resource usage for each configuration.

Although, there are several academic ([9], [10]) as well as commercial soft core processors ([2], [3], [4]) available, we are not aware of any dual-issue soft core processor that can deliver comparable performance and power consumption as our proposed solution. The processor combination of high performance and low power consumption makes it an ideal target for low cost FPGA based embedded devices.

The paper is organized according to the following outline. The simulation environment including the benchmarks is described in section II. Section III introduces the single-issue scalar processor architecture and discusses its operation. The dual-issue processor extension and simulation results are included in section IV. Section V discusses related work and the conclusion is presented in section VI.

INISCom 2015, March 02-04, Tokyo, Japan
Copyright © 2015 ICST
DOI 10.4108/icst.iniscom.2015.258414

II. SIMULATION ENVIRONMENT

Programs extracted from the MiBench benchmark suite [7] are used as a workload to simulate a wide range of embedded applications grouped into six categories: automotive, consumer, network, office, security and telecom. All benchmarks are cross compiled and statically linked using a GNU MIPS32 cross-compiler version 4.4.3 and a GNU lib version 2.4. Benchmarks are run to completion using the large data set and each output is compared to a reference output generated by executing the same benchmark natively on a host machine. Table I list the benchmarks programs and the total number of instructions executed for each benchmark. The program and data memory image sizes shown in the last two columns of Table I represent program and data memory minimum sizes per benchmark targeting a bare metal implementation without OS support.

TABLE I. MIBENCH BENCHMARK LIST WITH THE CORRESPONDING TOTAL NUMBER OF INSTRUCTIONS EXECUTED PER RUN AND PROG./DATA MEMORY IMAGE SIZE USING THE LARGE DATA SET.

Benchmark Name	Category	Total Number of Instructions Executed	Prog. Size (bytes)	Data Size (bytes)
basicmath	automotive	3,211,569,629	788,464	21,692
bitcount	automotive	595,183,708	608,512	21,808
qsort	automotive	616,386,556	621,904	21,492
susan.smoothing	automotive	392,905,660	684,480	21,748
susan.edges	automotive	69,545,376	684,480	21,748
susan.corner	automotive	23,397,130	684,480	21,748
jpeg.encode	consumer	115,060,309	694,112	22,484
jpeg.decode	consumer	25,411,302	706,560	22,540
stringsearch	office	6,227,880	608,848	32,368
dijkstra	network	289,665,966	605,840	21,600
patricia	network	918,545,085	607,680	21,544
blowfish.encode	security	1,949,847,190	614,128	21,548
blowfish.decode	security	1,946,194,228	614,128	21,548
rijndael.encode	security	451,426,609	644,384	21,624
rijndael.decode	security	439,654,189	644,384	21,624
sha	security	130,156,790	605,792	21,460
ADPCM.encode	telecom	611,853,238	603,136	21,384
ADPCM.decode	telecom	524,099,205	603,136	21,384
CRC32	telecom	6,014,143,443	604,320	21,464
FFT	telecom	501,452,631	663,584	21,788
IFFT	telecom	316,676,257	663,584	21,788

Fig. 1 shows the average distribution per operation type for each MiBench benchmark. The IALU category contains the following combination of integer arithmetic and logical operations: integer addition, integer subtraction, bitwise logical operations and operands comparison. The SHIFT category contains all bitwise logical and arithmetic shift operations which in some benchmarks represent 10% of the total number of operations. There are several observations to be drawn from the results in Fig. 1:

- There are almost no floating point operations (floating point arithmetic and floating point logic) in the majority of the benchmarks. Only the FFT and IFFT benchmarks have a measurable percentage of floating point instructions equal to 4% and 7% respectively. The number of floating point instructions in the remaining benchmarks does not exceed 0.5%.
- For all benchmarks, with the exception of *susan.smoothing*, 95% of total number of operations is distributed between only three types of operations: ALU which includes the shift operation category, control and memory operations.
- ALU and memory operations represent the largest percentage each ranging from 35% to 50% of total number of instructions executed.
- Control operations which includes conditional as well as unconditional branches represent the third largest percentage with conditional branches (not shown) representing less than 15% of total number of branches on average for all benchmark programs.

A custom cycle accurate simulator written in the C language is used to conduct a thorough evaluation of different architectural design decisions as well as to collect run-time statistics (e.g. total number of cycles, number of mispredicted branches, etc.). The base simulator models a 32-bit RISC processor described in details in section III. The number of pipeline stages as well as a number of other options, e.g. branch prediction technique, can be configured by the user dynamically at runtime.

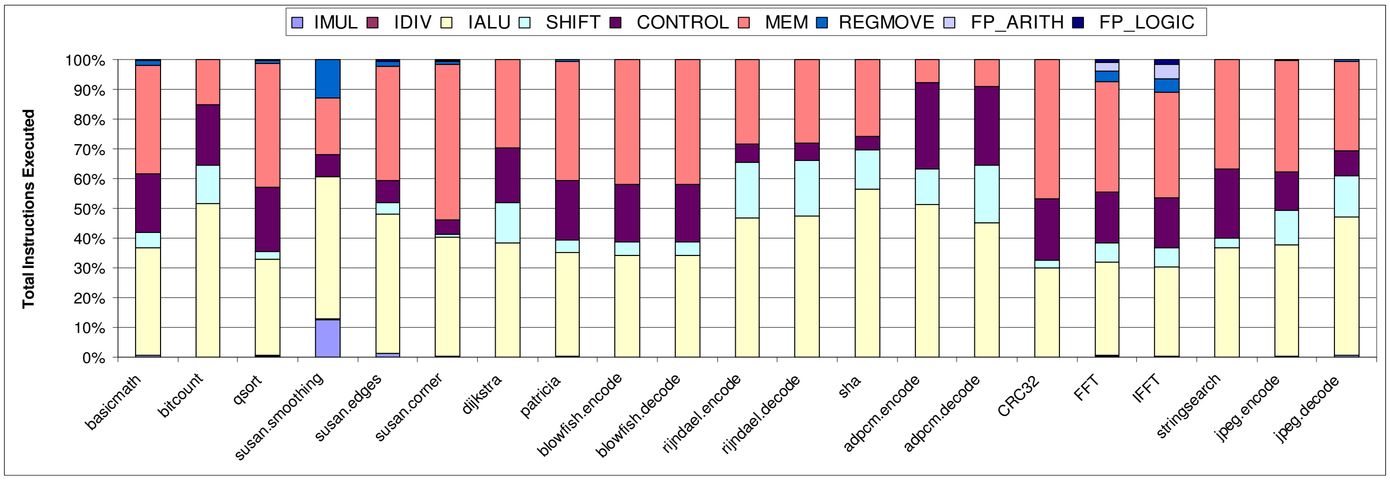


Fig. 1. Instruction type distribution for the large data set per MiBench benchmark (IMUL=integer multiply, IDIV=integer division, IALU=integer ALU as listed in the text, SHIFT=logical and arithmetic shifts, CONTROL=conditional & unconditional branch, MEM=memory load/store, REGMOVE=register move, FP_ARITH=floating point arithmetic operations and FP_LOGIC=floating point logic operations).

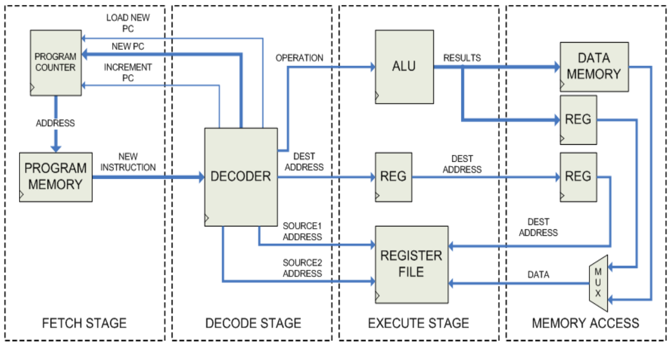


Fig. 2. Base processor top level block diagram

III. BASE SCALAR PROCESSOR

The base processor is a scalar single-issue RISC processor with a Harvard memory configuration. The processor has four pipeline stages: fetch, decode, execute and memory access. We experimented with different pipeline stages ranging from three up to six and found out that a four stage pipeline is less complex to implement in hardware than a five or six stage pipelines and produced better IPC performance than a three stage pipeline which was severely impacted by frequent pipeline stalls created by multi-cycle memory access operations. The processor uses the 32-bit MIPS32 instruction set [2] excluding all floating point and co-processor instructions as previous results (Fig. 1) showed that there are hardly any floating point instructions or co-processor instructions in the embedded benchmarks being simulated. Fig. 2 shows the top level diagram for the base scalar processor.

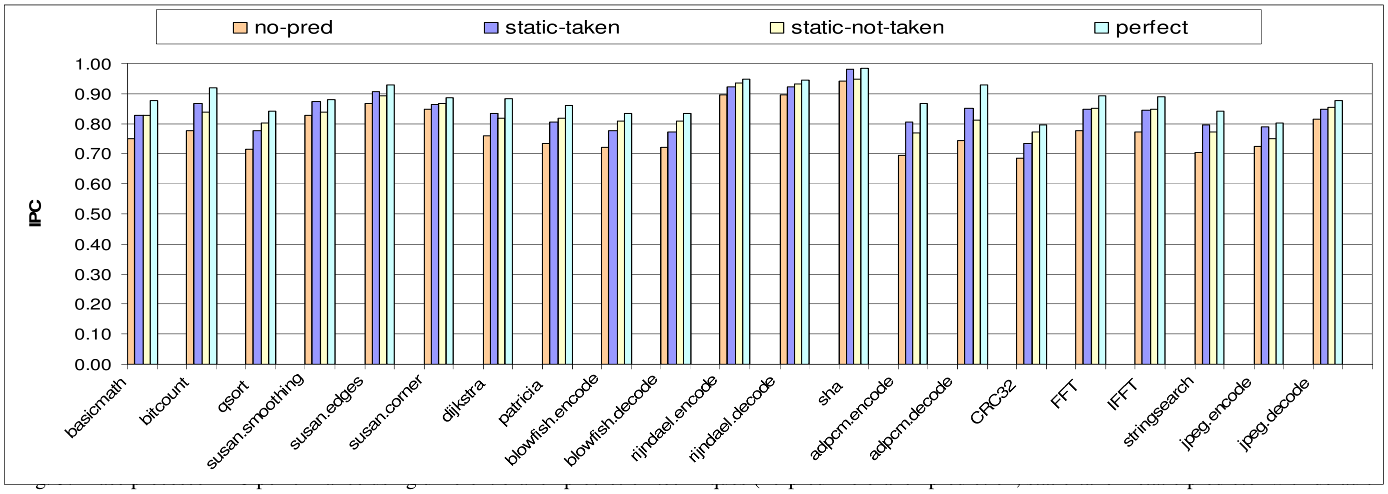
A. *Arithmetic and Logic Unit (ALU)*

The ALU is the largest block in terms of resource usage as it consumes almost 60% of the scalar processor total logic. The ALU includes a 32-bit fixed point adder with overflow detection, a fixed-point 32-bit multiplier with a 64-bit accumulator, a 32-bit logic unit that performs bitwise operations (OR, AND, XOR) and a 32-bit shift unit for bitwise logical and arithmetic shifts. All ALU operations including additions and multiplications excluding division, which is emulated in software, execute in a single cycle.

In the FPGA, the 32-bit multiplier is implemented using the FPGA internal embedded 9-bit multipliers. Using the hardwired FPGA multipliers instead of coding our own reduces the total logic elements count by 20%. The drawback is that several 9-bit embedded multipliers need to be cascaded together to perform 32-bit wide multiplications which increases latency and reduces the processor top clock speed thus affecting performance. For example, in the current implementation, increasing the multiplication width from 16-bits to 32-bits reduced the processor speed by almost half from 100 MHz to little bit over 50 MHz. Considering that the average percentage of integer multiply operations in the embedded benchmarks is less than one percent excluding the *susan.smoothing* benchmark which exhibit an exceptionally high percentage of integer multiplication, the 32-bit multiplier can be replaced with a smaller pipelined version that performs a 32-bit multiplication over two or more cycles, provided that the introduced delay does not decrease overall program latency. Using a smaller pipelined multiplier can also be advantageous when implemented in an ASIC, where a reduction in the total number of gate count will help reduce power consumption as well as die size.

B. *Instruction Decoder*

The second largest block is the instruction decoder and its main function is to extract relevant fields, e.g. addresses of the source and destination registers, from newly fetched instructions. The decoder is also responsible for generating the control signals needed to read from and write to the register file and the data memory. The register file as well as memory addresses and write control signals are buffered for two cycles to account for the delay between the decode stage, when the values are generated, and the memory-access stage, when the results are written to the register file or data memory. The decoder is also responsible for detecting control instructions, calculating the branch target address using a dedicated 32-bit adder and forwarding it to the instruction fetch unit to update the program counter for the next fetch cycle. For each conditional branch instruction, a branch recovery address is also calculated and temporarily stored in a register in case the processor needs to recover from a mispredicted branch.



taken prediction, static-not-taken=static predictor with default no-taken prediction, perfect = perfect branch prediction)

C. Branch Prediction

All branch instructions, except for the indirect jump instructions which store the target address in a general purpose register, are executed during the decode stage. A branch predictor allows conditional branches to be dispatched before they are executed to prevent stalling the pipeline. Fig. 3 compares IPC results for a single-issue processor using a static branch predictor with two prediction heuristics, always taken and always no-taken, a processor without a branch predictor (no-pred) and a processor with a perfect predictor (perfect). The results in Fig. 3 reveal that not using any branch predictors degrades IPC by as much as 10%. On the other hand, a simple static predictor can boost average IPC to within 5% of the IPC of a perfect predictor. Comparing the static predictor two prediction heuristics shows that the always taken heuristic performed better than the always not-taken heuristic by an average of 2%.

Upgrading from a static branch predictor to a 2-bit saturating history counter bimodal dynamic predictor had little impact on performance. A dynamic branch predictor with a 4K fully-associative entry history table improved IPC marginally by as little as 2% to 6%. However, a 4K history table which use more logic and consume more power than the entire processor core. A more realistic and smaller history table size of 64 or 128 entries achieves only a 2% to 4% improvement in IPC, respectively, which we believe is too small of an improvement to justify the added logic and complexity. For example, bimodal predictors require an additional pipeline stage be inserted between the decode stage and execute stage to access the branch history table. Inserting pipeline stages between the decoded stage and execute stage increases the number of speculative instructions that has to be flushed out from the pipeline if a branch outcome is mispredicted which wastes energy.

D. Register File

The register file contains 32 general purpose registers each 32-bit wide with one write port and two read ports. The difficulty in implementing multi-port register files is that FPGA embedded random access memory (RAM) modules have only two ports that can be configured as read ports or write ports [FPGA REF]. Adding three or more ports requires the design of a custom RAM module which cannot be done in an FPGA though might be feasible but very costly in an ASIC implementation. Therefore, register files is the most critical bottleneck to increasing the width of a processor pipeline.

The alternative is to use a technique to emulate multi-read and multi-write memories using standard two-port RAM modules [9]. Most of the techniques listed in [9] can be implemented on an FPGA. However, in our case the replication method is found to use the least amount of FPGA resources to emulate a three-port memory and, therefore, is the method that we used to implement the register file. The drawback of the replication method, as its name implies, is that it requires at least double the number of logical bits to physically implement a three-port memory. On the plus side, the register file uses a small number of logical bits even after replication (2K bits of total RAM organized as 32 registers each 32-bit wide), which can easily fit in a single FPGA embedded RAM block.

E. Result Bypass

When a true dependency, also known as read-after-write dependency, exists between two consecutive instructions, results must be made available to the decode stage as soon as instruction execution is complete. Otherwise, program execution has to be stalled for at least one cycle until the result is written back to the register file or data memory before it can be retrieved by the decoder. The solution is to forward the result directly to the decode stage while, simultaneously, writing it back to the destination register or memory location. This technique is known as result bypass and is usually implemented using a MUX and logic comparators. An alternative solution to the bypass technique is to use the write-through built-in feature that exists in some embedded memories [3]. Using the write-through feature, the new data can be written to a memory location and read from the same memory location at the exact same clock edge which eliminates the external bypass logic entirely. Although, the write-though feature uses some additional logic, the biggest advantage is that the logic itself is transparent to the user and the data can be seamlessly written to and read from the same memory location without any additional effort by the processor. In the target Cyclone FPGA the write-though feature in the register file added less than 10% overhead in logic elements, which is almost equal the number of logic elements that the traditional MUX and comparators solution uses.

F. FPGA Implementation

The base scalar processor is coded in Verilog and implemented on an Altera Cyclone-IV E 22K using Altera QuartusII web edition toolset version 13.0 [3]. Compilation options used are shown in Table II. A subset of the MiBench benchmarks is simulated using Modelsim-Altera before the final implementation in hardware using a commercial development board. Compiled benchmarks are downloaded and stored in the internal program memory during the initialization phase of the FPGA and then executed until completion. The built-in JTAG interface in the FPGA serves as a programming and communication link between the host system and the FPGA through the Quartus-II toolset. All logic memories are implemented using the inferred RAM techniques to make the code portable between different FPGA vendors as well as between FPGA and ASIC implementation.

TABLE II. COMPILATION OPTIONS FOR QUARTUS-II WEB EDITION

Option	Value
Target device	EP4CE22F17C7
Core voltage	1.2 V
Logic elements	22320
User IOs	154
Memory bits	608,256
Optimization technique	Balanced
Synthesis	Timing driven
Physical synthesis effort level	Normal
Fitter	Auto Fit

TABLE III. SUMMARY OF SINGLE-ISSUE PROCESSOR FPGA RESOURCE UTILIZATION

Resource	Base Processor
Logic elements (LE) (LUT / registers)	1,673 (1,642 / 347)
Logic RAM (bits)	2K
Physical RAM block (M9K)	2
Embedded 9-bit multipliers	8
Speed MHz	60 MHz
Dynamic power	0.48 mW/MHz
Coremark/MHz	2.51

Table III shows a summary of the FPGA resources used by the base scalar processor. The total number of FPGA logic elements (LE) is around 1.6K which corresponds to 7% of the total logic elements in the Cyclone IV-E 22K FPGA. The total number of RAM bits reported in Table III does not include program and data memory which can be configured by the end user independently from the processor core. The 3-port register file uses only 3% the total number of embedded memories (M9K) in the FPGA. The 32-bit wide fixed point multiplier uses eight 9-bit embedded multipliers cascaded together which limits the processor speed to 60 MHz. If the multiplier width is reduced to 16-bits, the processor speed can be increased to 100 MHz. However, a 16-bit multiplier has to be pipelined to perform 32-bit multiplications which may potentially reduce performance.

The average dynamic power dissipation of the single-issue processor is around 29 milliWatts when operated at 60 MHz which gives a power rating of 0.48 milliWatts/MHz. Most of the processor power, 48% to be exact, is consumed by the FPGA routing resources. The ALU consumes 32% of the total power and the decoder and register file each consume 10%. Using the PowerFilm solar panel as a power source delivering a max of 20 milliWatts, the processor can be operated at a max speed of roughly 40 MHz. Dynamic power estimates were generated using Altera PowerPlay tool which is part of Quartus-II. Signal activity results generated by Modelsim gate level simulation were fed to PowerPlay to give an accurate power estimates.

IV. DUAL-ISSUE SCALAR PROCESSOR

The base single-issue processor average performance is around 0.83 instructions per cycle as shown in Fig. 3. Even under ideal conditions when branches are predicted perfectly, the average performance does not exceed 0.87 instructions per cycle. In order to boost the performance of the single-issue processor beyond the scalar level, the pipeline needs to be widened to increase the flow of instructions coming into the processor. In [7], the authors show that the MiBench benchmarks have the potential to achieve an IPC of at least two instructions per cycle when a "high-end" processor, the 4-issue superscalar Compaq Alpha 21264, is used. However, superscalar processors use power intensive techniques and, therefore, are not well suited to be used in power sensitive embedded applications.

Instead we use simple scalar techniques to resolve instructions scheduling conflicts (both software conflicts like true-dependencies and hardware conflicts like resource allocation), which can be quite challenging especially when the number of parallel instructions is higher than two. Also, the inter-dependency between instructions within the same cycle as well as resource limitation can add additional complexity. For these reasons, we restrict the pipeline width to two instructions. In this section we evaluate some of the implementation challenges of a dual-issue scalar processor and propose a simple solution to each challenge. The order in which the items are listed is irrelevant and does not represent the challenge severity level.

A. *Register file*

Decoding and executing two instructions simultaneously each cycle requires two writes and four reads to the register file, which is double the number of ports used in the single-issue implementation. The replication method can still be used but the implementation becomes more complex [9]. A six ports logical memory implemented with two ports modules requires a total of 4K bits of physical RAM in addition to a small amount of logic for module selection and a multiplexer which combined together use less than 1% of the Cyclone-IV E 22K FPGA total logic elements. Four embedded memory blocks are used to implement the six port register file which represents 6% of the total number of available memory blocks in the target FPGA.

A single 32-bit register is used to track which memory module contains the latest data for each register. The 32-bit decoded value of the second write address is accumulated into the 32-bit register every clock cycle. Simultaneously, the 32-bit decoded value of each read address is compared with the content of the 32-bit register to generate four separate control signals. Each control signal is further buffered for one clock cycle to select between two outputs. This technique uses a minimal amount of logic and has a very short latency compared to previous replication techniques that store register tags in a separate register file and thus require an additional pipeline stage to access data from the register file [12].

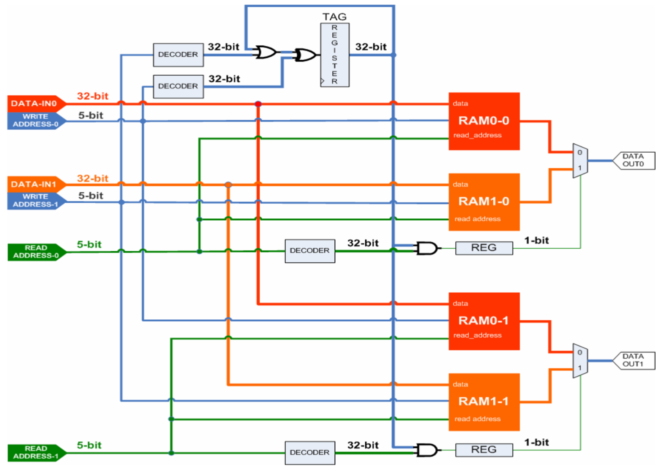


Fig. 4. A six-port register file (2W-4R) showing only two read ports

Previous multi-port memory duplication methods [12] track the location of the register by generating a 2-bit tag every time data is written to the register. The tag indicates the memory module number the register data is located in and is usually stored in a separate register file with 32 entries each 2 bits wide. When data is read from the register the tag is retrieved and the corresponding RAM module is accessed. This method uses one RAM module per read port. However, because tags are stored in a separate register file, an additional pipeline stage is needed to retrieve the tag information before data can be read from the RAM module. Also, this method requires a four-to-one 32-bit multiplexer per read port.

In comparison to the previous duplication method, our method uses double the amount of RAM but does not require an additional pipeline stage for register reads. Also, our method uses just one two-to-one 32-bit multiplexer per read port which eliminates half the propagation latency introduced by the multiplexers. The speed up in our method is a result of reducing the width of the register tag from 2-bit to 1-bit which allows the replacement of the additional 32x2-bit tag register file with a simple 32-bit register. Each bit in the 32-bit tag register represents a single register in the processor. If the processor has more than 32 registers than the tag register need to be increased to accommodate the extra registers. Registers stored in the second set of RAM modules (RAM1-0, RAM1-1, RAM1-2 and RAM1-3) are marked with a logic one (‘1’ bit) in the tag register. Writes to the first set of RAM modules (RAM0-0, RAM0-1, RAM0-2 and RAM0-3) need to clear the corresponding bit in the tag register which is accomplished with a logic ‘XOR” gate. A logic ‘OR’ gate accumulates previous writes tags and updates the tag register with the new values. Reads are executed by checking the value of the corresponding bit in the tag register. If the corresponding bit in the tag register is set then datum is read from the second RAM module and vise versa. Fig. 4 shows only two read ports, however, the logic is identical for each read port.

B. *True-dependency stalls*

In the base scalar processor true dependencies exist only between two consecutive instructions in different pipeline stages. Forwarding the result from the execution stage to the decode stage resolves the problem and avoids stalling the pipeline. In a dual-issue processor, true dependencies can potentially exist between instructions within the same cycle. In this case, the pipeline has to be stalled for at least one clock cycle until the first instruction executes and its result is forwarded to the next instruction. Fig. 5 shows an example of true-dependency between two instructions issued in parallel. Although, instruction fetching is halted for a single cycle, in the next two consecutive cycles only a single instruction is issued which reduces the IPC performance during these cycles by half. A solution is to issue instructions *I2* in parallel with *I1* to fill up the empty fetch slot and maintain the flow of instructions coming to the decoder. This means that in the following cycle, *I3* is re-fetched in parallel with the new instruction *I4*. This whole process is equivalent to sliding instructions by a single slot which can be easily achieved by incrementing the program counter by one instruction instead of the regular two instructions increment.

```
I0: add reg2, reg3, 0x01 (reg2 = reg3+0x01)
I1: add reg4, reg2, 0x02 (reg4 = reg2+0x2)
```

Pipeline Stages	**Cycles (each column represents a single clock cycle)**					
Fetch	I0:I1	I2:I3	bubble	…	…	…
Decode		I0:I1	I1	I2:I3	…	…
Execute			I0	I1	I2:I3	…
Memory Access				I0	I1	I2:I3
Pipeline Stages	**Cycles**					
Fetch	I0:I1	I2:I3	I3:I4	…	…	…
Decode		I0:I1	I1:I2	I3:I4	…	…
Execute			I0	I1:I2	I3:I4	…
Memory Access				I0	I1:I2	I3:I4

Fig. 5. Example of a true dependency between instructions within the same cycle and a proposed solution to reduce the number of empty instruction slots created by the true dependency.

C. *Branch Misprediction*

Recovering from a mispredicted branch decision is a difficult challenge in a dual-issue pipeline because a the instruction slot that a branch instruction is located in affects the calculation of the recovery address as well as the selection of instructions to be flushed out from the pipeline. In order to simplify the logic and limit the number of resources used, a branch instruction is decoded only when it is in the first instruction slot. This guarantees that the instruction in the second slot is always going to be the delay slot instruction. Delay slots are a feature of RISC architectures which requires that the instruction that immediately follows a branch instruction be executed before the branch is taken. Therefore, when a branch instruction is detected in the decoder second instruction slot, an exception is triggered and the issue of the branch instruction is delayed until it is moved to the first slot.

```
I0: nop
I1: beqz reg0, addr
I2: add  reg0, 0x01  (delay slot instruction)
```

Pipeline Stages	**Cycles (each column represents a single clock cycle)**					
Fetch	I0:I1	I2:I3	I2:I3	…	…	…
Decode		I0:I1	I1:I2	…	…	…
Execute			I0	I1:I2	…	…
Memory Access				I0	I1:I2	…
Pipeline Stages	**Cycles**					
Fetch	I0:I1	I8:I9	I2:I3	…	…	…
Decode		I0:I1	I1	I1:I2	…	…
Execute			I0	bubble	I1:I2	…
Memory Access				I0	bubble	I1:I2

Fig. 6. Example of a branch instruction located in the second slot and the corresponding delay slot instruction (*I2*) present in (top) and not present (bottom) in the fetch unit.

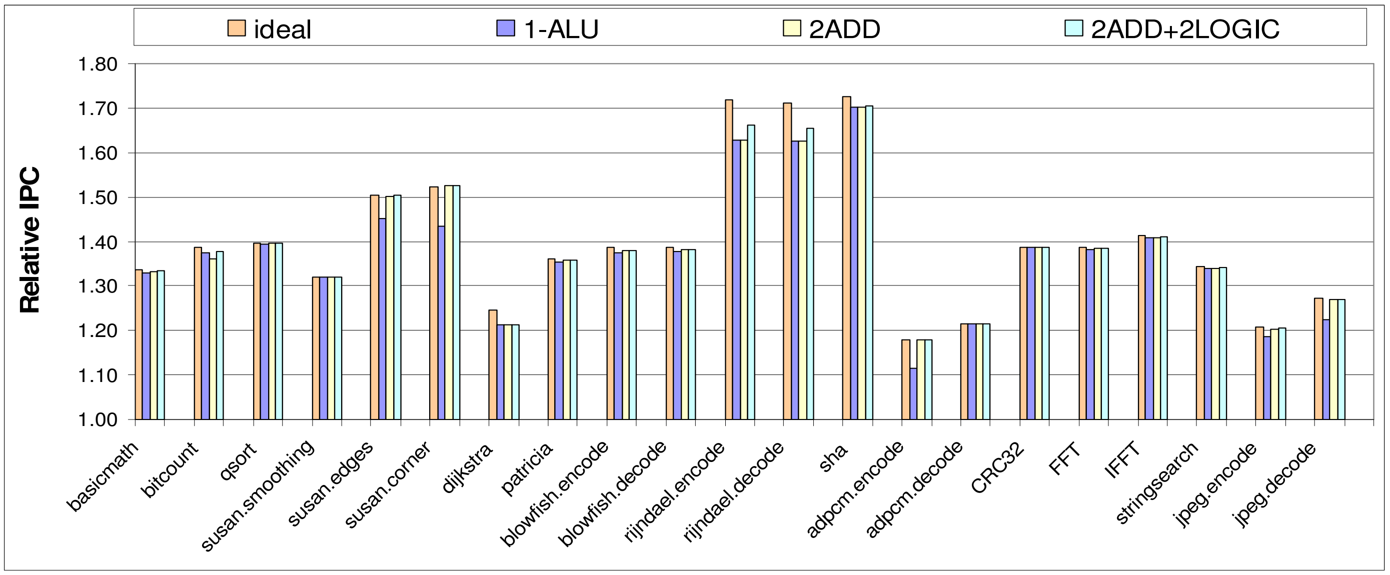


Fig. 7. IPC results for a dual-issue processor relative to the IPC of a single-issue processor using a static branch predictor with an always taken heuristic under different ALU configurations. The ideal case is for a processor with unlimited hardware resources and perfect branch prediction.

Although the branch instruction issue is delayed for one clock cycle, the fetching of the branch target address still gets executed unless the branch delay slot instruction is not present in the fetch unit. Otherwise, the delay slot instruction is fetched and is issued in the following cycle together with the branch instruction. The technique used to align branches to the first instruction slot is identical to the technique used to resolve true-dependency between instructions discussed earlier which consists of sliding instructions over by one slot. As a result, the true-dependency stall resolution logic can be reused as is to align branches to the first decoder slot with some minor additional logic, just a single two input logic gate, for detecting branch instructions when they are present in the decoder second slot.

In the previous discussion it is assumed that the delay slot instruction is already fetched and ready for decoding. In some cases the delay slot instruction might not be present in the fetch unit and has to be explicitly fetched; for example when a branch instruction is decoded in the previous cycle and the target address of that branch instruction is loaded into the fetch unit instead of the current branch delay slot instruction as shown in Fig. 6 example. If this condition occurs then the issuing of the branch instruction as well as the fetching of the branch target address are delayed for an additional cycle until the delay slot instruction is fetched and loaded into the decoder unit (Fig. 6 bottom).

The decision to delay the issue of branch instructions until they are located in the first decoder slot wastes only a single decoder slot and only when the delay slot instruction *I2* is present in the fetch unit (Fig. 6 top example). Otherwise, if the delay slot instruction *I2* is not present in the fetch unit (Fig. 6 bottom example) then both approaches (issue branch instructions from the second decoder slot or delay them until they are moved to the first slot) achieve similar performance because the branch instruction is delayed anyway until the delay slot instruction is fetched regardless of which slot the branch is present in when it is decoded.

D. Hardware Dependencies

In order to process two instructions per cycle most of the processor resources need to be duplicated. However, we know from previous instruction distribution analysis (Fig. 1) that some of the operations like for example multiplication appears very infrequently in the benchmarks. Therefore, adding two multipliers will not affect performance for the majority of the benchmarks, whereas it will increase gate count dramatically as well as the max power budget. From the point of view of energy efficiency, idling resources can be turned off to reduce energy waste. However, if we need to limit the increase in total gate count we must determine which processor resources are the most critical to performance and just duplicate these resources. We limit our investigation to the ALU because the ALU is the biggest block in the processor in terms of logic elements at 60%. We conducted a thorough quantitative analysis to determine which ALU resource must be duplicated by selectively duplicating one resource at a time and comparing the IPC results to the results of a dual-issue processor with a single ALU (*1-ALU* case in Fig. 7) to see if there is any improvement in performance. Results are shown in Fig. 7 and discussed below.

E. Results

The average increase in IPC for a dual-issue processor is around 40%. Some benchmarks like *rijndael* and *sha* experienced more than 70% improvement in IPC. The higher than average performance of these benchmarks is a result of the small number of control instructions contained in them which means that the basic block size is larger and, consequently, the number of instructions that can be issued in parallel is much higher than in the rest of the benchmarks. Because the dual-issue processor lacks advanced superscalar techniques to issue instructions out-of-order it can only extract instruction level parallelism from within basic blocks boundaries. Compiler assisted techniques that increase the size of basic blocks like loop unrolling can increase performance even further.

The worst performing benchmark, *adpcm,* still managed to achieve at least a 20% gain in IPC. The *adpcm* benchmark happens to have the highest percentage of control instructions of all the benchmarks. Benchmarks with a high percentage of control instructions might benefit from using a high performance branch predictor like the bimodal predictor. However, we did not evaluate the performance of the bimodal branch predictor in the dual-issue implementation because using a branch history buffer requires the insertion of an additional pipeline stage between the decode stage and execute stage which increases the number of speculative instructions that needs to be flushed out from the pipeline following a mispredicted branch.

Earlier we mentioned that duplicating resources unnecessarily especially ALU functions will increase the max power budget and total gate count without benefiting performance. The more effective approach is to only duplicate the resources that affect performance the most. First, we notice that the average drop in IPC for a single ALU configuration is around 2%, which is unexpectedly low. For some benchmarks, like *susan.corner*, *rijndael, adpcm* and *jpeg*, the IPC loss is higher than 5%. Increasing the number of adders to two helped these benchmarks, excluding *rijndael*, recover all the lost performance. The *rijndael* benchmark benefited the most from duplicating the logic unit but only managed to recover half the IPC loss. The remaining IPC loss is tied to other ALU functions like the comparator and shifter. Although, duplicating these ALU functions benefited the *rijndael* benchmark they had no impact on the IPC of the remaining benchmarks.

The ideal case shown in Fig. 7 is for a dual-issue processor with unlimited resources and perfect branch prediction. For most benchmarks, the IPC performance of a dual-issue processor with a single ALU unit is within 5% to 8% the IPC performance of an ideal processor. This gap can be reduced to less than 2% if a second 32-bit adder is added. Adding a second logic unit has no impact on performance for the majority of the benchmarks. In the case of the *rijndael* benchmarks the improvement in IPC did not exceed 3% which does not justify the additional resources.

F. *FPGA Implementation*

Table IV shows a summary of the resources used to implement the dual-issue processor on the target FPGA. The first column shows the results for the single-issue processor copied from Table II. The total number of FPGA logical elements increased by almost 30%. The total number of logic bits used by the register file doubled from 2-Kbits to 8-Kbits which requires the use of eight separate embedded memory blocks (M9K). The number of embedded 9-bit multipliers remains the same because only a single ALU functional unit is used. Speed also remains the same which is determined by the multiplier latency similar to the single-issue implementation.

The Coremark result for the dual-issue processor shows a 40% improvement in performance over the single-issue processor. It should be noted that the Coremark results should be considered in combination with other factors such as power consumption in order to determine the real advantage of the new architecture. The power consumption increased by almost 20% from 0.48 milliWatts/MHz to 0.57 milliWatts/MHz. Using the PowerFilm solar panel introduced earlier as a power source, the dual-issue processor can operate at a max speed of 35 MHz which is roughly 12% lower than the max speed of the single-issue processor using the same power source. Even with 12% reduction in speed, the Coremark/MHz performance increased by almost 40% compared to the single-issue processor which is twice the increase in power consumption. Therefore, we expect a comparable sizable increase in energy efficiency.

TABLE IV. SUMMARY OF DUAL-ISSUE PROCESSOR FPGA RESOURCE UTILIZATION

Resource	Single issue	Dual Issue	Change
Total logic elements	1673	2215	+30%
LUT	1642	2169	
Registers	347	419	
Logic RAM (bits)	2K	4K	+100%
Physical RAM blocks (M9K)	2	8	+200%
Embedded 9-bit multipliers	8	8	
Speed (MHz)	60	60	
Dynamic power (mWatt/MHz)	0.48	0.57	+18%
Coremark/MHz	2.5	3.4	+40%

V. RELATED WORK

The majority of academic research in the area of embedded processor architecture focuses on superscalar techniques, with some of the work specifically targeting FPGA implementation [9] and [10]. A number of other works investigated the use of single scalar cores in multi-chip processors targeting standard-cell implementations [11] as well as FPGA [13]. Some research looked into specific aspect of the architecture, for example static branch prediction [11] or the design of the register file [12], etc. Other works investigated the design of embedded processors targeting a specific application like biomedical [14] or smart grid [15]. For example, the authors in [14] studied the performance impact of several static and dynamic branch predictors for biomedical applications. The branch predictors that they studied are similar to the ones we evaluated although targeting different applications. Their conclusion is identical to ours; it shows that the static predictor with the taken heuristic provides the best return compared to other predictors even the more advanced dynamic history based predictors.

On the key components in the implementation of the processor is the register file which uses non-standard memories with multiple read and write ports. A novel implementation of the register file was introduced. There are an extensive amount of prior work on the implementation of the register file or reorder buffer in case of superscalar processors. [12] gives a brief introduction to the different techniques. In [16], the authors combined several of these techniques to optimize the implementation of a 12-read and 6-write port register file. However, their technique similar to most previous techniques requires an additional pipeline stage to read data from the register file. We introduced a novel memory replication technique that requires no additional pipeline stages and uses the least amount of logic compared to all other published techniques though it uses more physical memory.

Commercial processors optimized for low power embedded applications, e.g. ARM Cortex-M series [3] with the exception of the ARM Cortex-M7 which is a 6-stage superscalar processor, are mostly single-issue scalar processors. Higher end embedded processors capable of dual-issue, e.g. Cortex-R or Cortex-A series, consume substantially more power and are unsuitable for battery operated applications. High-end superscalar processors are exclusively used in products which prioritize performance over power. Whereas low-end single issue scalar processors are usually reserved for low-cost and low-power devices such as industrial sensors that tend to be powered by either small embedded batteries that last for several years or energy harvesting power sources such as the PowerFilm solar panel introduced earlier which requires less maintenance than battery operated sensors.

Also, most FPGA vendors provide have their own soft processor cores optimized for their own platforms. Altera NIOS-II is a family of 32-bit soft processor cores that comes in three different types: economy, standard and fast [4]. The economy soft core is a basic sequential microcontroller with a 6 cycles-per-instruction performance. The standard soft core is five-stage processor with static branch prediction and optional support for tightly coupled memories. The fast soft core is Altera highest end processor core and is a six-stage processor with dynamic branch prediction. The power consumption of the NIOS-II fast core is around 1.65 milliWatts/MHz [4], [17] and it's highest Coremark/MHz performance is 1.60 [5]. Our dual-issue soft processor core is roughly three times more power efficient, 0.57 compared to 1.65 milliWatts/MHz, than a NIOS-II core and delivers twice as much performance, 3.4 compared to 1.60 Coremark/MHz.

We are not aware of any multi-issue commercial processor that specifically targets the ultra-low power embedded market segment. In fact all the multi-issue processors surveyed are uniquely geared towards the high end of the embedded market where performance matters more than power consumption. Our research pushes the limit of processor performance without the use of superscalar techniques in order to achieve the highest energy efficiency possible. In this paper we specifically explore the impact of doubling the pipeline width on IPC performance while using simple hardware techniques to resolve conflicts in instruction scheduling

VI. CONCLUSION

A single-issue scalar processor can operate on a very low power budget but its performance is constrained. A multi-issue superscalar processor can deliver multiple fold increase in performance but is power intensive. To address this problem, a dual-issue scalar processor capable of processing up to two instructions simultaneously every cycle is introduced. The processor delivers an average 40% higher IPC than a single-issue processor while consuming less than 20% more power. Applications that contain a large amount of instruction-level parallelism are particularly suitable for the dual-issue processor and experience up to 60% improvement in IPC. The dual-issue processor is implemented on a low cost FPGA using less than 10% of the total logic elements which leaves plenty of resources to implement additional functions. Compared to Altera NIOS-II soft processor core, the proposed dual-issue processor is three times more power efficient and it can deliver twice the performance. The proposed processor is an ideal candidate for embedded industrial devices that are powered by green sources of energy like solar panels. Power supplied by a 36.5 x 64 mm small solar panel is enough to operate the dual-issue processor at a max speed of 35 MHz and deliver a Coremark/MHz performance of 3.4 which is 30% higher than the Coremark/MHz performance of the single-issue scalar processor.

REFERENCES

[1] www.powerfilmsolar.com

[2] www.mips.com

[3] www.altera.com

[4] www.arm.com

[5] www.eembc.com/coremark

[6] P. E. Gronowski, W. J. Bowhill, R. P. Preston, M. K. Gowen, R. L. Allmon, "High-Performance Microprocessor Design,"IEEE Journal of Solid-State Circuits,Vol. 33, No. 5, May 1998.

[7] Guthaus, M.R.; Ringenberg, J.S.; Ernst, D.; Austin, T.M.; Mudge, T.; Brown, R.B., "MiBench: A free, commercially representative embedded benchmark suite," Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on, pp.3,14, 2 Dec. 2001.

[8] Michael K. Gowan, Larry L. Biro, Daniel B. Jackson. Power Considerations in the Design of the Alpha 21264 Microprocessor, DAC 98, June 15-19, 1998.

[9] Dwiel, B.H.; Choudhary, N.K.; Rotenberg, E., "FPGA modeling of diverse superscalar processors," Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on, pp.188,199, 1-3 April 2012.

[10] Rosiere, M.; Desbarbieux, J.-L.; Drach, N.; Wajsburt, F., "Morpheo: A high-performance processor generator for a FPGA implementation," Design and Architectures for Signal and Image Processing (DASIP), 2011 Conference on, pp.1,8, 2-4 Nov. 2011.

[11] Bechara, C.; Berhault, A; Ventroux, N.; Chevobbe, S.; Lhuillier, Y.; David, R.; Etiemble, D., "A small footprint interleaved multithreaded processor for embedded systems," Electronics, Circuits and Systems (ICECS), 2011 18th IEEE International Conference on, pp.685,690, 11-14 Dec. 2011.

[12] Rosiere, M.; Desbarbieux, J.-L.; Drach, N.; Wajsburt, F., "An out-of-order superscalar processor on FPGA: The ReOrder Buffer design," Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012, pp.1549,1554, 12-16 March 2012.

[13] Saldana, M.; Nunes, D.; Ramalho, E.; Chow, P., "Configuration and Programming of Heterogeneous Multiprocessors on a Multi-FPGA System Using TMD-MPI," *Reconfigurable Computing and FPGA's, 2006. ReConFig 2006. IEEE International Conference on*, pp.1,10, Sept. 2006.

[14] Strydis, C.; Gaydadjiev, G.N., "Evaluating Various Branch-Prediction Schemes for Biomedical-Implant Processors," *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*, pp.169,176, 7-9 July 2009.

[15] Sai, R.T.S.; Mukherjee, A.; Cecchi, V.; Kailas, A., "Architecture exploration of a heterogeneous embedded processor for the smart grid," *Southeastcon, 2013 Proceedings of IEEE*, pp.1,6, 4-7 April 2013.

[16] Yantir, H.E.; Bayar, S.; Yurdakul, A., "Efficient Implementations of Multi-pumped Multi-port Register Files in FPGAs," Digital System Design (DSD), 2013 Euromicro Conference on, pp.185,192, 4-6 Sept. 2013.

[17] Senn, L.; Senn, E.; Samoyeau, C., "Modelling the Power and Energy Consumption of NIOS II Softcores on FPGA," Cluster Computing Workshops (CLUSTER WORKSHOPS), 2012 IEEE International Conference on, pp.179,183, 24-28 Sept. 2012.