Design of a Low Power, Relative Timing based Asynchronous MSP430 Microprocessor

Dipanjan Bhadra, Kenneth S. Stevens
University of Utah

***Abstract*—Power dissipation is one of the primary design constraints in modern digital circuits. From a magnitude of hand-held portable devices to big data analytics using high-performance computing, low energy dissipation is a key requirement for most modern devices. This paper showcases an elegant low power circuit design methodology based on Relative Timing driven asynchronous techniques. A low power MSP430 microprocessor design based on a novel asynchronous finite state machine implementation is presented. The design showcases the power benefits of the proposed asynchronous implementation over the synchronous counterpart and avoids major architectural modification which would directly influence the performance or power consumption. The implemented asynchronous MSP430 exhibits a minimum of 8× power benefit over the synchronous design for an almost identical pipeline structure and comparable throughput. The paper further elaborates on the novel asynchronous state machine implementation used for the design and presents an efficient method to design communicating asynchronous finite state machines in clock-less systems.**

***Index Terms*—MSP430, Microprocessor, Low-power, Asynchronous Circuits, Relative Timing**

I. Introduction

Power dissipation has became a primary concern for designers targeting circuits for handheld devices and parallel computing systems for big data computations. Asynchronous designs provide an elegant solution to low power circuit design. Handshake signals ensure asynchronous circuits only operate when provided with valid data, and also provide a modular interface that operates independent of frequency domains. Synchronization across clock domains for large SoCs has emerged as a correctness, design time, and power challenge. The global clock distribution in modern designs account for considerable power dissipation and design efforts. Asynchronous approaches mitigate the energy overhead of crossing clock domain boundaries and global clock distribution, save power because circuits are only active when they are assigned a task, and simplify IP integration and validation.

Researchers have documented the challenges and advantages of asynchronous designs. Over the years the Caltech Asynchronous Microprocessor [1], the Caltech R3000 [2], the Lutonium [3], the DLX design [4], the Amulets [5], [6] the asynchronous 80C51 [7], and the Intel Pentium [8] have showcased different design methodologies and techniques for design and optimization of asynchronous circuits.

Some of these designs come at a considerable overhead in terms of added design effort in absence of an automated design flow. Others, like the 80C51 [7], use custom automated design flows such as Tangram [9], [10] and Balsa [11]. Some designs go even further as to use custom gate libraries for computation. Instead, this work leverages relative timing in order to use industry standard synchronous CAD tools and HDL [12].

II. Background

This paper reports on a new design approach for implementing communicating asynchronous state machines to build a power efficient MSP430. The MSP430 is a simple 16-bit microprocessor used for low-cost, low-power embedded systems. The synchronous design is built with emphasis on low power operation. It is provided with multiple operational modes to reduce power consumption at idle states. The synchronous design is based on the openMSP430 from the Opencores repository [13]. To provide a fair comparison, no architectural modifications were made to the clocked data path in the asynchronous design. A custom control network replaces the clock network for the asynchronous design. A fair comparison was achieved by using same EDA flow for both designs.

A previous asynchronous MSP430 implementations uses the Balsa tool flow [14]. That design uses a back-end retargeting method to map the design to specific libraries. However, the design optimization phase does not utilize the library information to perform timing driven optimizations to the circuit. Another design is implemented using the Tangram flow [15]. The TiDE design uses a similar HDL to generate the required design. In this work the entire control network for the AFSM design is built using a graph based method which results in a completely asynchronous state machine. The state machines use standard Verilog implementations of combinational logic and registers.

A major advantage of our design over the past work is in the design methodology. This MSP430 was designed leveraging standard commercial VLSI tool flow. The design is specified in Verilog. Some minor additions to the flow to support relative timing constraints allow the commercial EDA tools to perform internal timing driven circuit optimizations. This has enabled us to create a novel method of implementing the pipelined interaction between the decode and execute state machines that vastly differs from previous design approaches.

III. Design Methodology

The asynchronous design implementation uses a return to zero bundled data design that is shown in Fig. 1. Combinational functions are behaviorally specified between pipeline stages which are controlled by a handshaking network for both sequencing and delay. Relative timing (RT) constraints are used to optimize the circuit for performance and to make hazards in the control circuitry unreachable. One such RT constraint is shown in the figure.

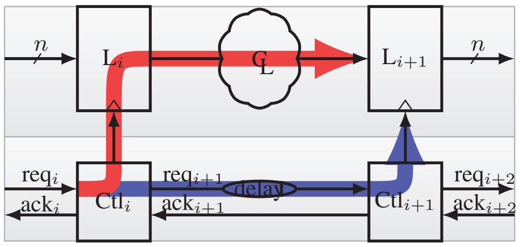


Fig. 1. Timed (bundled data) handshake design with the related relative timing constraint $\text{req}_i\!\uparrow\ \mapsto L_{i+1}/D + margin \prec L_{i+1}/CLK\!\uparrow$ highlighted. The maximum delay from $\text{req}_i\!\uparrow$ to data arriving at the downstream latch data input plus a margin m must be less than the minimum delay from $\text{req}_i\!\uparrow$ to data arriving at the downstream latch clock pin.

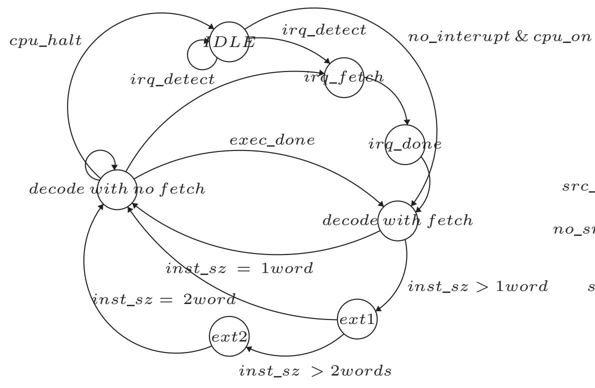


Fig. 2. Decode State Machine for MSP430

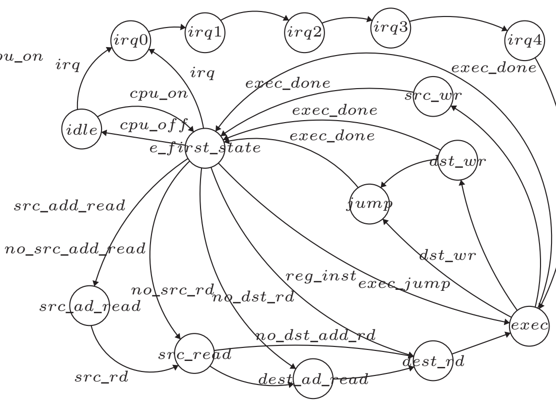


Fig. 3. Execute State Machine for MSP430

Clocked design methodologies assume a single discrete time delay for all operations. This results in worst case performance for the design but reduces the design effort. One of the main goals of this design is to customize the delay for each data path being employed based on the stateful behavior of the controller. In this design the control delays match the data path function delay to obtain a average case delay. A novel approach to implementing multiple control paths is implemented and supported by the relative timing methodology. This has evolved into an approach for state machine communication and interaction that allows reactive updates between communicating state machines.

A. Relative Timing (RT)

A synchronous circuit uses a clock network to order and sequence different events to achieve proper functionality. In this work we have adopted the relative timing (RT) methodology to achieve the desired sequencing in the absence of a clock network [12]. A RT constraint consists of a common timing reference event and a pair of events that are ordered in time for correct circuit operation. The common reference is called a point-of-divergence, or pod, and the ordered events point-of-convergence, or poc. A constraint is represented as $\mathsf{pod} \mapsto \mathsf{poc}_0 + m \prec \mathsf{poc}_1$ where poc_0 must occur in time before poc_1 with margin m. A RT constraints for a bundled data linear pipeline is shown in Fig. 1.

B. Asynchronous CAD Tool Flow

The same commercial EDA tool flow was used to design and optimize the clocked and asynchronous designs. Behavioral Verilog is used as a specification language, Design Compiler used for Synthesis, IC Compiler for physical design, and Primetime for timing and performance verification. The only significant difference comes in how timing is modeled. Rather than specifying a global clock, individual RT pipeline delays are explicitly specified as sdc constraints that drive EDA tool optimization. Some additional custom tools are used to interface the EDA tools and RT constraints [16].

IV. MSP430 Design Specification & Clocked Design

The synchronous openMSP430 provides a cycle accurate implementation for the TI MSP430 microprocessor. The MSP430 is a 16-bit processor with a 16-bit address space. The address space is divided between the peripheral devices, data memory and program memory. The processor has 16 general purpose registers. Registers R0, R1, and R2 are the PC, Stack pointer, and Status registers respectively.

A. Instruction Set and Addressing modes

The MSP430 CPU architecture was designed to operate with a reduced instruction set with very simple formats. The instruction set consists of two types of instructions: core and emulated. The core instructions are directly implemented in the hardware while the emulated instructions require hardware constructions using two constant generators CG1 and CG2 (registers R2 and R3) and efficiently emulate instructions.

There are seven different addressing modes for the MSP430: direct, indexed register, register indirect, indirect auto-increment, PC relative, absolute, and immediate. The source addressing may use any of the seven modes while the destination addressing can only use register, indexed register, PC relative and absolute modes. The MSP430 instructions range from one to three words (2 bytes each) depending on the addressing mode used. The indexed, symbolic, absolute, and immediate modes of addressing need an extra word of instruction which follows the first word of the instruction in the program memory.

B. Decode State Machine

The synchronous implementation consists of two interacting finite state machines (FSMs) - decode and execute. The decode state machine, shown in Fig. 2, fetches instructions from the program memory one word at a time. The instruction is decoded and if it is a multiple word instruction the ext1 and ext2 states fetch additional instruction words from the program memory. Once it has the entire instruction it waits for the execution to be done to fetch the next instruction. The decode state machine polls for a pending interrupt before fetching every new instruction. On encountering an interrupt the decode state machine replaces the PC with the appropriate interrupt vector and jumps to the interrupt routine.

C. Execute State Machine

The execute state machine, shown in Fig. 3, controls the ALU operations and updates the registers and the memory. Execution starts when the first instruction word is fetched and decoded. The execution state machine then runs until the instruction execution completes or until it needs additional instruction words from the decode state machine. The number of communications between the decode state machine and execute state machine depends on the particular instruction.

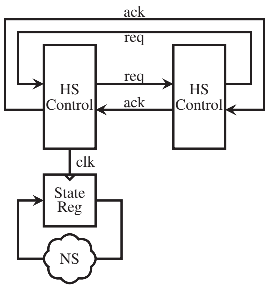


Fig. 4. The design of a bundled data state machine requires two pipeline stages configured in a ring to update the current state with the next state. The second pipeline stage can be modeled as a delay constant added to the ring.

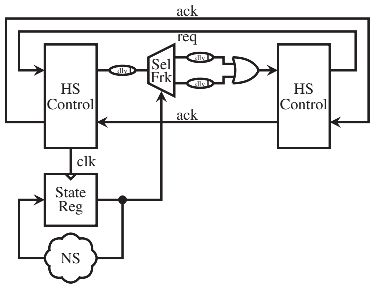


Fig. 5. Multiple delays are generated with a *selective fork* module that interacts with both the data path and control path of the AFSM.

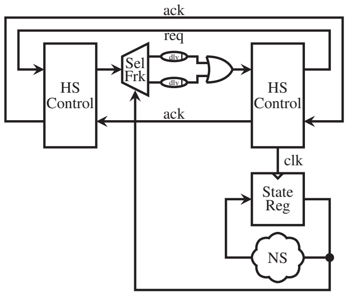


Fig. 6. Delays added to the control path to ensure correct operation can be removed by moving the state register to the second handshake controller which provides sufficient setup time to the selective fork.

D. Memory Interface

The MSP430 address space is partitioned into the program memory, data memory, and the peripherals. The program memory is read only. The decode FSM only reads from the program memory while the execute FSM may access the entire memory. For the clocked system in case of conflicting access request to program memory from both FSM, the execution request is prioritized. Stalled decode memory accesses are processed in the next clock cycle. The processor polls status signals in the memory controller to ensure correct data is read by the decode module when stalled.

V. Asynchronous Implementation

The goal of the asynchronous design is to showcase an improvement in power dissipation through the use of an asynchronous techniques over the synchronous design for a comparable architecture. Hence, design optimizations which may improve the performance over synchronous while changing the basic architecture like adding pipeline stages to increase throughput or improvements to the data path were avoided. The data path for the asynchronous design is nearly identical to the clocked design.

A. Asynchronous Finite State Machines

Finite state machines form an important building block for digital circuits. In clocked FSMs a next state function is sampled and updated from current state values on each rising clock edge. Communication between state machines is performed by allowing current state information to be shared across multiple state machines. Such an approach for communication between asynchronous state machines does not directly work because the state of each asynchronous machine is updated on local independent events.

There are numerous methods that can be used to design asynchronous finite state machines (AFSMs). Fully custom AFSMs can be specified in various asynchronous languages such as a petri net and synthesized with tools such as Petrify. These AFSM must be designed such that no hazards are manifest in the implementation. A single AFSM for a complex specification such as the MSP430 fetch or decode unit can be large with complex timing to insure hazard free operation. This normally results in significant power and slow operation. Another approach is to unroll the state machine onto a one-hot ring with each state implemented in a single pipeline stage [17]. This reduces the complexity of the AFSM and improves performance of the design. However, this generally creates a large circuit area for a design even with a moderate state space. In this approach each state requires its own asynchronous handshake controller and often pipeline data. We deemed this approach to have too large an area penalty for a FSM which handles only a single instruction at any given time. Both of these approaches are difficult to modify and update.

B. Multi-Frequency Bundled Data AFSMs

Instead, we implemented the asynchronous state machines in a manner very similar to clocked design. A register contains the current state, which is updated at the completion of the operation with a next state function. The state function becomes a simple combinational function that is small, energy efficient, and easy to modify and debug. Such a design can not be implemented with direct feedback as is the case in clocked design. A two controller ring is required as shown in Fig. 4 because a single controller will deadlock if its input channel and output channel are tied together. Thus two handshake controllers in a ring generate the sequential control path, while registers are used to maintain states as in synchronous design. This considerably reduces the area overhead as no matter how large the state space, only two handshake controllers, a register bank, and combinational logic are required for the implementation. Burst-mode controllers are used in our AFSM for pipeline control. Our implementation uses pulse clocks generated by the handshake controllers to latch data.

An extension of considerable importance to this simple burst-mode AFSM design is to achieve programmable cycle time for each state in the AFSM. There is considerable delay difference between various operations in the state machines, and using the worst case delays for all cases penalizes the design performance. To create programmable delays, the handshake signal and the data path must interact. We use a *selective fork* module for this interaction, where the current state values of the data path are used to steer requests and create matching delays based on the current state of the AFSM (Fig. 5). Interactions between the data and control path require additional delays on the control path to ensure proper steering of the control signals. This can burden the cycle time of the FSM. However, this can be dealt with by moving the state-holding registers to the other controller in the ring (Fig. 6).

This now allows us to quickly build and modify generic AFSMs with overhead similar to the clocked design. Each finite state machine can be independently designed as in the clocked design. The decode and execute state machines operate with independent frequencies. Relative timing provides the additional advantage that each state can be designed to

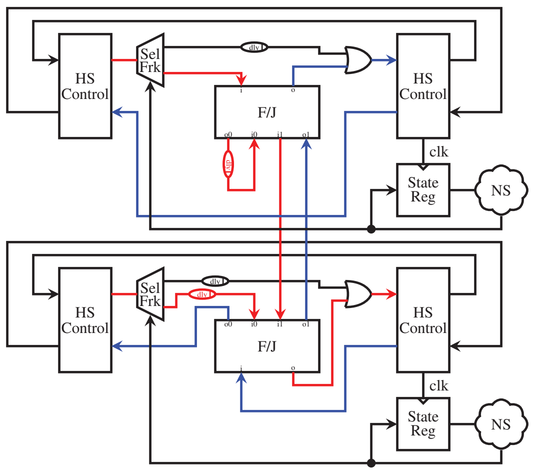
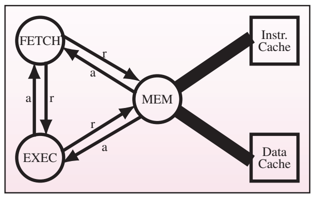


Fig. 7. The conjunctive stateful communication between state machines is demonstrated, which allows safe observation of state in external AFSMs with low overhead reactive communication.



```
agent MSP430 = ( PDECODE | PEXEC
| CHOICE ) \{ sSRC, sDST, sINT,
sINST, gSRC, gDST, gEXEC, gINT,
done };
agent PDECODE = miINST.done.PINST;
agent PINST   = ...
agent CHOICE = 'sSRC.CHOICE +
'sDST.CHOICE + 'sINT.CHOICE +
'sINST.CHOICE;
agent PEXEC   = 'done.PETOP;
agent PETOP   = ...
```

Simulation Trace

```
I-CACHE   D-CACHE
-------   -------
 miINST
 miINST    mdEXE
 miDST
 miINST    mdDRD
           mdEXE
 miSRC
 miDST     mdSRD
 miINST    mdDRD
 miINST    mdEXE
 miINST
 miINST
 miINST    mdEXE
```

Fig. 8. Formal concurrency model of the asynchronous MSP430 design

operate at its optimal frequency. Thus the cycle time of each state machine will vary significantly based on its current state.

C. Interacting Bundled Data AFSMs

One AFSM can not directly observe the state of another AFSM because updates in state values are not synchronized between the machines. Therefore synchronization and interaction between AFSMs require additional invention. Conjunctive stateful handshaking between the AFSMs was developed as a solution that provides a handshaking mechanism to communicate states.

The handshaking is based on traditional bundled data handshake channel protocols that contains control and data signals. When a source AFSM must interact with a destination AFSM, it sends a request and data (typically state information) to the destination controller. In the destination AFSM, the handshake is disabled until the destination AFSM is in a receptive state. When the conjunction of the external request and the local enable signal are asserted in the destination controller, the data can be cleanly observed with a join element. Appropriate AFSM evolution will occur, including completing the handshake with the source AFSM. This interaction mechanism is shown in Fig. 7.

D. Asynchronous MSP430 Control

Synchronization between the fetch and execute AFSMs is done through handshake signals triggered at appropriate states. A formal model of the two state machines and the memory controller were created to identify maximal concurrency and interleaving that can be implemented between these components (Fig. 8). This formal model in CCS was built to verify that system deadlocks could not occur between the communicating AFSMs as well as to maximize concurrency and independence of the fetch and decode AFSMs.

For example, every time the end of instruction state is reached the execution AFSM must wait for a request from the decode unit to signify that the next instruction is fetched and decoded. Once the decode AFSM latches the first word of the instruction and starts processing the instruction it is independent of the state of the fetch decode state machine, unless this is a multi-word instruction or a end of instruction has been reached. The formal model allowed us to maximize the concurrency and interaction between the two AFSMs. In this process, we proved that the clocked FSMs also used the same maximal concurrency.

E. Memory Access

The asynchronous design has a similar memory interface to the synchronous one which distributes the memory access request between the program, data memory and the peripherals based on the address ranges. However, it arbitrates between conflicting requests to program memory from the decode and execute AFSM using a mutex. Whichever request comes first goes through while the other is stalled until the handshake completes for the winning request. Once the first request is serviced and the data is latched, the other request gets memory access.

F. Decode and Execute AFSM Interactions

Conjunctive stateful communication is used between the two AFSMs. Every time the Decode AFSM fetches a new instruction word it sends a request to the execute AFSM and stalls until the execute state machine is ready to use the data. When the execution AFSM acknowledges receiving data, the decode AFSM moves to the next operation. In all other states which are independent of any synchronization, the control simply loops through the two pipelines and can be as fast as the data paths permit. For a more complex state machine this can improve the performance considerably.

VI. Results

Clocked and asynchronous designs are synthesized in the IBM 65nm 10SF node using the same EDA tools and scripts. Math operations and official processor benchmark code is used to verify and compare the designs. This analysis shows a large power improvement for the asynchronous design. Official Texas Instruments test bench data for the MSP430 are used to generate the closest simulation comparison to existing synchronous designs. Benchmark software was used from both the "MSP430 Competitive Benchmarking Application Report" and the provided examples from openMSP430 [13], [18].

Tables I and II show the area, power, runtime, and energy results for the different test benches. The area numbers provided in Table I identify gate area from Design Compiler. The asynchronous design is 5% smaller than the clocked design. The asynchronous design is on average 33% slower than the clocked design across the test bench suite. However, the

TABLE I
POWER NUMBERS FOR DIFFERENT TEST BENCHES ON CLOCKED AND ASYNCHRONOUS DESIGN

Design	Test Bench	Area	Power μW					runtime	
		mm^2	Switching	Internal	Leakage	Total	Improvement	ns	Improvement
Async	8 bit math	13,406	2.0	3.0	77.2	82.3	10.7×	3,986	0.65×
	16 bit math		2.7	4.0	77.2	83.9	10.7×	4,852	0.62×
	32 bit math		5.6	8.4	77.2	91.2	10.6×	9,750	0.57×
	8 bit switch		0.6	0.8	77.2	78.6	10.7×	1,494	0.80×
	16 bit switch		0.7	1.0	77.2	78.9	10.7×	1,679	0.77×
	8 bit matrix		16.8	25.1	77.2	119.1	10.4×	27,513	0.65×
	16 bit matrix		8.9	13.4	77.3	99.6	10.5×	15,075	0.66×
	matrix mult		41.7	62.8	77.2	181.7	9.6×	72,483	0.57×
Ave.			9.9	14.8	77.2	101.9	10.5×	15,417	0.66×
Clocked	8 bit math	14,125	15.8	792.4	72.3	882.3	–	2,592	–
	16 bit math		20.5	819.8	72.3	902.4	–	3,024	–
	32 bit math		41.9	860.5	72.4	974.4	–	5,586	–
	8 bit switch		4.5	765.8	72.5	842.8	–	1,198	–
	16 bit switch		5.3	766.8	72.7	844.9	–	1,290	–
	8 bit matrix		129.0	1,036.0	72.7	1,238.0	–	18,010	–
	16 bit matrix		68.2	911.8	72.6	1,053.0	–	9,887	–
	matrix mult		306.7	1,367.0	72.6	1,737.0	–	41,977	–
Ave.			74.0	915.0	72.5	1,059.4	–	10,446	–

TABLE II
COMPARISON OF ENERGY NUMBERS FOR DIFFERENT TEST BENCHES ON CLOCKED AND ASYNCHRONOUS DESIGN.

Test bench	Energy pJ								Async
	Clocked				Async				Energy
	Switching	Internal	Leakage	Total	Switching	Internal	Leakage	Total	Improvement
8 bit math	40.9	2,053.6	187.3	2,286.6	7.9	12.0	307.8	327.8	6.9×
16 bit math	61.9	2,479.0	218.5	2,728.9	13.0	19.5	374.7	407.2	6.7×
32 bit math	234.2	4,806.7	404.5	5,442.9	54.8	82.2	752.4	889.3	6.1×
8 bit switch	5.4	917.3	86.8	1,009.5	0.8	1.2	115.3	117.4	8.5×
16 bit switch	6.8	989.0	93.7	1,089.7	1.1	1.7	129.6	132.4	8.2×
8 bit matrix	2,324.5	18,667.7	1,309.3	22,308.3	4,613.9	691.1	2,122.9	3,276.8	6.8×
16 bit matrix	674.7	9,015.1	717.6	10,411.2	134.6	201.7	1,165.0	1,501.2	6.9×
matrix mult	12,874.3	57,382.2	3,046.7	72,913.7	3,019.6	4,554.9	5,593.6	13,170.3	5.5×
Average	2,027.8	12,038.8	758.1	14,773.9	980.7	695.5	1,320.2	2,477.8	7.0×

asynchronous design consumes on average less than one-tenth the power of the clocked design, and averages one-seventh the energy per operation.

VII. DESIGN COMPARISON

Identical behavioral data paths are used in the two designs to provide a fair comparison. For most cases the leakage power of the two different implementations is quite comparable due to their similar area. Leakage in the asynchronous design is slightly higher due to more inverters and buffers in the design. The asynchronous design significantly reduces both switching and internal power.

The clocked implementation consists of a clock generation module which the asynchronous design does not have. This module constantly monitors the two FSM states to check if both are idle. This is an enhancement to save power when the FSMs are indeed idle by switching operational modes. This module also synchronizes data from the environment to the system clock using register chains. Since the asynchronous circuit is reactive it is always idle when not requested to operate.

The clocked design operates at the worst case frequency which is determined by the memory access time. The maximum cycle time through the asynchronous control is also dictated by memory access latency. Some performance advantage is achieved in the asynchronous design, which adapts AFSM cycle time to the current state by using a shorter cycle time that more closely matches the function in that state. Using multiple delays require that data and control interact through the selective forks to steer the request from the controller to the different paths depending on the state variable.

The performance penalty of the asynchronous design primarily comes from two sources: additional synchronizations that are not necessary in the clocked design, and the memory access delay modeling for the memory access path in the asynchronous design. Memory access forms the most critical path for the design because both the clocked and asynchronous designs access only one instruction at a time, and access one to five memory words for each instruction depending on its type and addressing mode. Like all modern designs, the memory access times for this design is much higher than the combinational circuit delay on any of the data paths. The asynchronous design has an additional penalty because each memory access from the decode AFSM adds a synchronization handshake with the execute AFSM before it can move forward. This does not exist in the clocked design. Since the memory access time for the clocked system closely mirrors the clock period, adding the handshake overhead slows down the asynchronous design. Additionally, an extra state/cycle was added to all the instructions in order to allow for the two independent FSMs to synchronize in order to service an interrupts. In the clocked design both FSMs will observe the interrupt at the same clock and needs no synchronization. This adds to the per instruction execution time, and deters the overall performance. This can be observed by the system slowing down more for higher memory intensive test benches like the matrix operations. Future work will directly address these penalties.

Memory accesses have additional penalty because some of the addressing modes require ALU operations to generate the correct memory address. This address generation is not pipelined in this design, and thus becomes part of the critical path. Because it is performance critical, the address generation unit (AGU) uses additional power. One of the power advantages of the asynchronous design is to enable the AGU early by latching with the second controller in the execute AFSM ring. This provides an earlier data to control setup similar to the multiple delay elements in the previous paragraph. In this case, the extra latency allows for less energy through the AGU logic.

Distributed asynchronous control with multiple delays provides more freedom in reducing energy. Early data setup can also be leveraged with the two-latch AFSM rings. Even though the clocked system is also retimed by the synthesis tools, the asynchronous design can leverage timing and delays in ways that result in significant additional reduction in power. Slowing down certain paths provides power benefits as faster and larger gates are no longer required for computation.

All the registers in the asynchronous design are level triggered latches rather than flip-flops. This contributes much of the area reduction of the asynchronous design as well as to the power benefit. A quick look at breakdown of the power dissipation for the different test benches elucidates the advantages of the asynchronous implementation.

Stalls are implemented by NOP operations or by polling in the clocked design. When a state is reached which cannot be completed due to lack of data availability, the following cycles require the clocked design to keep trying to attain completion while checking for a certain status bit. This results in significant power increase compared to the asynchronous design. There are multiple states in the clocked architecture where the system must poll for data. These include states in the memory interface when the decode state machine request is delayed in relation to the execute state machine, when entering interrupts, when the decode hasn't fetched an additional instruction word the execution state machine needs, or when the decode stalls for the execution to complete. The asynchronous system gains power advantage during these operations as it reactively waits for the desired signal to assert. Although flip-flops in the clocked design are gated, clock gating can not be applied to stages that poll. Thus the distributed control system for the asynchronous design plays a vital role in providing a low power microprocessor implementation.

VIII. Conclusion

This paper presents a relative timing based timing driven implementations for the MSP430 microprocessor. A distributed control path is used in the asynchronous design to replace the clock network while avoiding architectural modifications or optimizations. The asynchronous design is implemented using a new conjunctive stateful communication methodology between asynchronous finite state machines that is enabled using relative timing technology. This new method for implementing concurrent communicating asynchronous state machines provides an elegant alternative to other methods providing benefits of small area, multiple operational cycle times, and ease of modification.

The asynchronous design is 5% smaller than the clocked design. Leakage energy is slightly higher due to more buffers and inverters in the asynchronous design. The asynchronous design shows a minimum of a $5\times$ energy per instruction improvement over the equivalent clocked design for a variety of test benches. The asynchronous design is slower than the synchronous design, by a benefit factor of 0.6–0.8$\times$, but averages a more that $10\times$ reduction in power.

IX. Acknowledgments

This material is based upon work supported by the Semiconductor Research Corporation under contract No. 2012-TJ-2235 and by the National Science Foundation under Grants No. 1111533 and 1218012.

References

[1] A. J. Martin, S. M. Burns, T. K. Lee, D. Borkovic, and P. J. Hazewindus, "The Design of an Asynchronous Microprocessor," in *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, C. L. Seitz, Ed. MIT Press, March 1989, pp. 251–273.

[2] A. J. Martin, A. Lines, R. Manohar, M. Nystrom, P. Penzes, R. Southworth, U. Cummings, and T. K. Lee, "The Design of an Asynchronous MIPS R3000 Microprocessor," in *Proceedings of the 17th Conference on Advanced Research in VLSI*, Sept 1997, pp. 164–181.

[3] A. J. Martin, M. Nyström, K. Papadantonakis, P. I. Pénzes, P. Prakash, C. G. Wong, J. Chang, K. S. Ko, B. Lee, E. Ou, J. Pugh, E.-V. Talvala, J. T. Tong, and A. Tura, "The Lutonium: A Sub-Nonojoule Asynchronous 8051 Microcontroller," in *9th International Symposium on Asynchronous Circuits and Systems*. IEEE, May 2003, pp. 14–23.

[4] J. Cortadella, A. Kondratyev, L. Lavagno, and C. Sotiriou, "Desynchronization: Synthesis of asynchronous circuits from synchronous specifications," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 25, no. 10, pp. 1904–1921, Oct 2006.

[5] O. A. Petlin and S. B. Furber, "Built-In Self-Testing of Micropipelines," in *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE, Apr 1997, pp. 22–29.

[6] S. Furber, P. Day, J. Garside, N. Paver, and J. Woods, "AMULET1: A Micropipelined ARM," in *Compcon*. IEEE, Mar 1994, pp. 476–485.

[7] H. van Gageldonk, K. van Berkel, A. Peeters, D. Baumann, D. Gloor, and G. Stegmann, "An Asynchronous Low-Power 80C51 Microcontroller," in *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE, Mar 1998, pp. 96–107.

[8] K. S. Stevens, S. Rotem, R. Ginosar, P. Beerel, C. J. Myers, K. Y. Yun, R. Kol, C. Dike, and M. Roncken, "An Asynchronous Instruction Length Decoder," *IEEE Journal of Solid State Circuits*, vol. 36, no. 2, pp. 217–228, Feb. 2001.

[9] K. van Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij, "The VLSI programming language Tangram and its translation into handshake circuits," in *Proceedings of the European Design Automation Conference*, Feb 1991, pp. 384–389.

[10] J. Kessels and A. Peeters, "The Tangram Framework: Asynchronous Circuits for Low Power," in *Proceedings of the 2001 Asia and South Pacific Design Automation Conference (ASPDAC)*. ACM, 2001, pp. 255–260.

[11] A. Bardsley and D. Edwards, *Hardware Description Languages and their Applications: Specification, modelling, verification and synthesis of microelectronic systems*, ser. International Conference on Computer Hardware Description Languages and their Applications. Springer US, April 1997, ch. Compiling the language Balsa to delay insensitive hardware, pp. 89–91.

[12] K. S. Stevens, R. Ginosar, and S. Rotem, "Relative Timing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 1, no. 11, pp. 129–140, Feb. 2003.

[13] O. Girard, *OpenMSP430 Specification*, , 2010, http://opencores.org/project,openmsp430.

[14] S. Kwak, H.-W. Lee, Y. Zafar, M.-H. Oh, and D. Har, "Design of Asynchronous MSP430 Microprocessor Using Balsa Back-end Retargeting," in *5th Southern Conference on Programmable Logic (SPL)*, Apr 2009, pp. 223–228.

[15] M.-H. Oh, C. Shin, and S. Kim, "Design of Low-Power Asynchronous MSP430 Processor Core Using AFSM Based Controllers," in *The 23rd International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC 2008)*, April 2008, pp. 1109–1112.

[16] W. Lee, T. Sharma, and K. S. Stevens, "Path Based Timing Validation for Timed Asynchronous Design," in *The 29th International Conference on VLSI Design (VLSID)*. IEEE, Jan 2016, pp. 511–516.

[17] L. Hollaar, "Direct Implementation of Asynchronous Control Units," *IEEE Transactions on Computers*, vol. C-31, no. 2, February 1982.

[18] Texas Instrument, "Msp430 competitive benchmarking," Texas Instrument, Tech. Rep., 2006, http://www.ti.com/lit/an/slaa205c/slaa205c.pdf.