

Predictable, Low-power Arithmetic Logic Unit for the 8051 Microcontroller using Asynchronous Logic

D. Mc Carthy, N. Zeinolabedini, J. Chen, E. Popovici

Abstract—Modern embedded systems require all their components, including their microcontroller, to be optimised with respect to the power budget. Two properties are desirable. The first is low power usage and the second is predictable power usage, for improved estimation of firmware performance.

In this paper we present all the arithmetic circuits for an ALU of an 8051 microcontroller implemented in Asynchronous Charge Sharing Logic (ACSL). This implementation seeks to give these two desirable properties for a processor for embedded systems. The first, low power usage, obtained through charge sharing. The second, predictable power usage, is sought by ensuring that the power required to complete an operation is independent of its inputs.

The experimental techniques used in designing ACSL were also improved in the execution of this work, allowing the ACSL circuits to be entered using Verilog for fast initial testing and then translated to SPICE for detailed simulation.

Through implementation and simulation, it was determined that the use of ACSL can offer power predictability.

I. INTRODUCTION

Modern embedded systems require all their components, including their microcontroller, to be optimised with respect to the power budget. Two properties are desirable. The first is low power usage as the battery life of embedded devices is a bottleneck in the field. The second is easily predictable power usage [1], [2].

In this paper we present a full set of arithmetic components for the ALU of an 8051 microcontroller [3] implemented in Asynchronous Charge Sharing Logic (ACSL) [4]. This implementation seeks to give these two desirable properties for a processor for embedded systems. The first, low power usage, is sought through asynchronous charge sharing. The second, predictable power usage, is sought by ensuring that the power required to complete an operation does not depend on the inputs. If this property can be provided, average-case power analysis is very straightforward in the design which could be significant in battery-life estimation. Only the types of operation (addition, multiplication, division, logic) would have to be considered during power analysis rather than the input patterns.

All authors are with the Centre for Efficiency Oriented Languages (CEOL) and the Department of Electrical and Electronic Engineering, University College Cork, Ireland. E-mail: davidmc@ue.ucc.ie

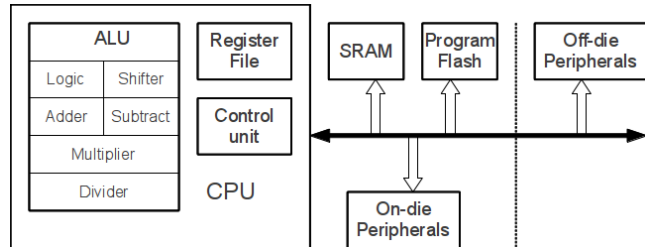


Fig. 1. Block diagram of an 8051 microcontroller.

The 8051 is an 8-bit microcontroller designed by Intel in 1980. A block diagram is shown in Figure 1. This microcontroller was chosen for this work because it is a very common microcontroller. 8051 compatible parts are available from a wide variety of sources. More background of the 8051 will be discussed in section II. In areas where ultra-low power operation is desirable, such as wireless sensor networks, the 8051 is particularly popular since variants are available with an on-die radio.

Asynchronous Charge Sharing Logic (ACSL) [4] is a asynchronous logic [5] family that uses charge sharing techniques to reduce power consumption. ACSL uses adiabatic gate designs but does not generate the power-clock in an adiabatic fashion (usually involving resonant circuits) and is thus not itself an adiabatic logic family. Instead ACSL recovers energy from each stage using charge sharing. The variability due to charge sharing operation is compensated by using asynchronous design principles. This has several advantages, such as not requiring a multi-phase AC supply [6] to function and allowing asynchronous operation. While ACSL does not have the theoretical power saving potential of adiabatic logic, in practice it achieves similar savings [4]. Another advantage of ACSL resides in the fact that the drawbacks of variability associated with charge sharing logic are compensated by the properties of asynchronous logic. Compared with other dual-rail asynchronous logic families, ACSL does not require complex completion detectors which not only improves power efficiency but also decreases the area required.

II. THE MICROCONTROLLER

The 8051 is a Complex Instruction Set Computer (CISC) design, with an instruction set optimised for manually developed assembly code. It contains single instructions for each of the 4 arithmetic operations: add, subtract, multiply, divide. Because of two's complement, the add and subtract instructions are suitable for both signed and unsigned operation. The multiply and divide instructions are unsigned. Either a one or (add, subtract) or two (multiply and divide) byte results is produced. Three flags are also set based on arithmetic functions: carry, auxiliary carry and overflow.

Logic and shift operations are also available. Logical AND, OR, XOR and NOT are provided. Five Shift operations are supported. Four are rotations (cyclical), allowing shifting left or right of either the 8-bit accumulator register by itself or through both the accumulator and the carry-bit. The 5th shift is a nybble-swap, exchanging the top and bottom 4 bits of the accumulator.

Additionally a 'decimal adjust' instruction is provided to allow for BCD operations. This assumes a packed BCD format with one BCD digit in each nybble of a byte. This is achieved by a regular addition followed by the decimal adjust instruction, which selectively increases the accumulator to make the contents valid BCD. Finally 16-bit plus 8-bit additions are needed for address calculations, this is provided for by the ALU in this work.

The arithmetic operations resulting from this work are designed to resemble the ALU of the opencores.org 8051 [7], an open-source soft-core 8051. In this ALU, there are 3 input bytes and 2 output bytes to the ALU, as well as input and output flags.

III. ASYNCHRONOUS CHARGE SHARING LOGIC

Asynchronous Charge Sharing Logic (ACSL) is based on the PFAL [8], [6] adiabatic logic family. It uses the same gate structure, consisting of both true and complementary NMOS pull-up networks and a cross coupled inverter pair. Figure 2(b) shows a PFAL/ ACSL AND gate (output F). When idle, the power-clock is zero. To evaluate the gate, the power-clock is raised, first with shared charge for the previous stage and then with fresh charge from the power supply. The NMOS pull-up tree provides an initial pull to set one output of the gate, and the output levels are fully restored by feedback in the inverter pair. A latch is placed on each gate output. A 4-transistor dynamic latch is used, functioning like a NOR-type SR latch. This serves to provide clear logic levels from one stage to the next even while the power clock charge is being shared between them.

Both adiabatic logic and ACSL save power from capacitive losses through re-use of charge from the power-clock. In adiabatic logic, power saving is obtained

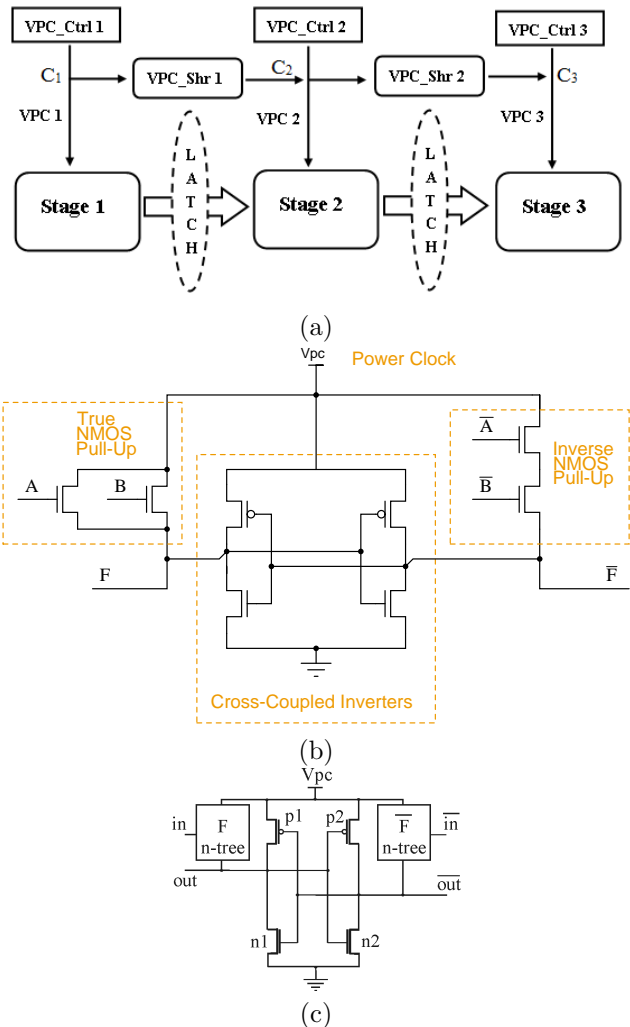


Fig. 2. ACSL architecture (a), transistor-level design of an AND gate (b) and 4-transistor latch (c)

through recovery of the charge into the AC supply. ACSL instead saves power through charge sharing. The charge in the capacitances in the previous stage of stage of gates is allowed to distribute into the current stage before fresh charge is drawn from the power supply.

The operation of ACSL is determined by an asynchronous handshake, which triggers the operation of each stage. The handshake is defined in terms of the stage power-clock $vpc[n]$ and 3 additional signals, $ctrl[n]$, $complete[n]$ and $shared[n]$. The handshake is detailed in algorithm 1. This handshake is implemented using a mix of static and (traditional) dynamic gates, with the dynamic gates being used for storing state. Charging of the power-clock is done with a wide-dimension PMOS pull-up, sharing with a wide NMOS pass transistor and charge-dump using a wide NMOS pull-down.

$ctrl[n - 1]$ causes $shared[n] = 0$;
 $complete[n - 1]$ causes $ctrl[n] = 1$;
 $ctrl[n] \cdot \neg shared[n]$ causes sharing $vpc[n - 1]$ between $vpc[n]$;
 $vpc[n] \geq 0.5vdd$ causes $shared[n] = 1$;
 $ctrl[n] \cdot shared[n]$ causes charging of $vpc[n]$;
 $vpc[n] = vdd$ causes $complete[n] = 1$;
 $ctrl[n + 1]$ causes $ctrl[n] = 0$;
 $complete[n + 1]$ causes $complete[n] = 0$, discharging of $vpc[n]$;

Algorithm 1: Basic ACSL handshake

IV. ARITHMETIC CIRCUIT DESIGN

The arithmetic circuits resulting from this work designed with reference to the ALU of the opencores.org 8051 [7], an open-source soft-core 8051.

Addition and subtraction are implemented using a ripple carry adder, with XOR gates before and after to perform the one's complement and adjust the input and output flags. To facilitate the 16-bit plus 8-bit adds, the top 8-bits are incremented and a multiplexer is used to select between the incremented and the original based on the carry out signal of the full adder in the bottom 8 bits. All additions and subtractions are performed in 10 stages of asynchronous logic, 1 for setup, 8 for the main computation and 1 for producing the flags and high byte of a 16-bit result.

An array multiplier [9] is used for the multiplication operation. A 7-bit by 8 layer array rectangular array is used. The AND gates for partial product generation is interleaved with the array to equalise per-stage load. The final adder is implemented using a static ripple carry adder. This multiplier uses 9 stages of asynchronous logic. One generates initial partial products, and 8 constitute the main multiplication array

Division is achieved using a non-restoring array divider [10]. This divider is similar to the standard repeated-subtraction algorithm, but avoids using multiplexers at each stage for restoration by selecting addition for the following stage instead of subtraction when overflow occurs. Ripple carry adders plus XOR gates are used internally. The main divider array consists of 9 by 8-bit add/subtract units, 8 producing 1 bit each of the 8 quotient bits and 1 to correct the remainder. A stage of ACSL multiplexers is then used to select between the corrected and uncorrected remainder. However the basic non-restoring divider array performs signed division. To give the unsigned division required for an 8051, an additional side logic path is provided to handle divisors with the most significant bit set. A single subtraction and restore is performed. A final multiplexer is used to select between the main result and the big divisor result.

Each adder/subtractor is implemented in 9 stages,

consisting of a diagonal arrangement of XOR gates and full adders. The whole circuit consists of 94 stages of asynchronous logic. The logic for handling large stages consists of 11 stages: 1 stage of buffer, 1 stage of multiplexer and 9 stages for a subtractor. The main division array consists of 72 stages, and the correction adder and multiplexer totals 10 stages. The final multiplexer for selecting between the big divisor and regular result is the 94th stage.

Using Kogge-Stone adders [11] was also considered to decrease the stage count but this was in fact slower, larger and more power-hungry, due to the increased gate count (in spite of the reduced stage count). Also, charge sharing was not as effective between the different types of stages involved in such an adder as between the homogeneous stages of ripple carry divider.

Logic operations are implemented using a single stage of the relevant ACSL gates in parallel. Shift operations are implemented using a single stage of buffers, wired appropriately.

The final operation to be implemented is the decimal adjust. 1 3-bit adder is used to correct the bottom nybble, of which the bottom bit remains unchanged. Having corrected the bottom nybble, 2 4-bit adders are used to generate possible corrections for the top nybble. Multiplexers are used to choose the correct output based on the flags input and carry-out signals of the adders.

V. EXPERIMENTAL METHODS

The circuit capture techniques used in designing ACSL were also improved in the execution of this work. Previous work on ACSL has been done using either analog IC schematic capture tools or worked directly using SPICE netlists. In this work, the Verilog hardware description language was used to design most of the ACSL circuits.

A structural style of Verilog was used, with the circuit described entirely using module instances. For the dynamic gates implementing the ACSL handshake, behavioural models were coded. For the ACSL gates themselves, each type of gate is described in a module. Standard Verilog Boolean expressions are given for the true and false pull-up networks of the gate, and given as inputs to a generic gate behavioural module.

The first advantage of this is that Verilog is a high level hardware description language. It has a much more readable syntax, in particular having buses rather than single wires greatly reduces repetition in the code. Module connections by port name also help greatly. Secondly Verilog can be simulated at a high level very quickly, allowing for easy logical verification of the circuit. Thirdly Verilog has detailed warnings, making mistakes in schematic capture, such as disconnected wires much more obvious.

However, Verilog cannot model the power usage of

TABLE I
SIMULATION RESULTS

Operation	AND	Add,Sub	Multiply	Divide
Power (μW)	13.45	11.71	14.45	111.72
Variation	3.8%	3.4%	9.8%	1.5%
Delay (ns)	0.47	1.38	3.16	21.60
Variation	1.7%	1.0%	33.5%	0.4%
Transistors	204	1308	3041	8954

ACSL. To measure power, the circuitry is translated to a SPICE netlist to be simulated in a conventional manner. This is achieved using a mix of Synopsys V2S and a custom translator for gate descriptions. Synopsys V2S is a generic Verilog-to-Spice translated included in the Synopsys HSPICE package. However V2S will attempt to translate ACSL gate descriptions into a set of static gates plus an instance of generic gate behavioural model. Thus a custom translator was written, which finds the gate definitions and writes SPICE netlists, with a generic gate core (cross-coupled inverters) and NMOS pull-up networks corresponding to the given logic functions. The gate definitions are written into their own Verilog file, the gate translator is used to produce a Spice netlist for the gates. Then the V2S utility is used on the rest of the Verilog sources with the Spice file of gates used as a library. Finally, this SPICE netlist can be simulated using HSPICE.

VI. RESULTS

Power measurements were taken of the ACSL circuits using HSPICE for transistor level simulation. A TSMC 45 nm process was used operating at 1.0 V. Each circuit was evaluated with an operation every 30 ns (33.3 MHz). The first few operations for each circuit were discarded to ignore startup effects, then 10 input patterns were recorded. Inputs were randomly selected. Both delay and power were measured for each input. Given in Table I are the average power and delay, and also how far as a percentage the furthest outlier was away from the average.

For comparison, a divider was implemented in static logic, also at transistor level in the same process on HSPICE. The average power of the divider was found to be 23.00 μW , with deviations of 43.0%. The average delay was 0.377 ns with a maximum of 0.85 ns. This circuit contained 3450 transistors.

VII. CONCLUSIONS

Very good power predictability was observed in 3 of the 4 circuits tested (AND, Add-Subtract, Divide) with slightly higher power variations observed on the multiplier. Delay was also almost constant for the

AND, Add-subtract and Divide circuits but varied more widely for the Multiplier. All of the power variation figures were significantly better than the variation associated with the static divider.

Absolute power savings were not observed in this work. The ACSL divider consumed an average of 111.72 μW compared to 23.00 for the static divider, thus ACSL is 4.86 times higher. Possible explanations for this include the dual-rail versus single rail nature of ACSL, the overhead of the handshake, and reevaluating gates that would not have to change in static operation.

Delay for ACSL is also much worse, being 25.5 times worse in worst case and 57.3 times worse in the average case, almost two orders of magnitude. However, ACSL is still sufficiently fast that it could lead to instruction times in the tens of Megahertz range that typical 8051 microcontrollers are used at.

Future work will focus on different multiplier architectures or a more fully asynchronous version of the current architecture to minimise variation. Other operations will be optimised to reduce power usage.

ACKNOWLEDGEMENTS

This work was funded by the Science Foundation Ireland under Grant number 07/IN.1/I977 and by the Seventh Framework Programme of the European Union, under Grant Agreement number 309129 (i-RISC project).

REFERENCES

- [1] F. Najm, "A survey of power estimation techniques in vlsi circuits," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 2, no. 4, pp. 446–455, 1994.
- [2] S. Sultan and S. Masud, "Rapid software power estimation of embedded pipelined processor through instruction level power model," in *Performance Evaluation of Computer Telecommunication Systems, 2009. SPECTS 2009. International Symposium on*, vol. 41, 2009, pp. 27–34.
- [3] *MCS-51 8-bit Control-Oriented Microcomputers*, Intel Corporation, 1988.
- [4] J. Chen, E. Popovici, D. Vasudevan, and M. Schellekens, "Ultra low power booth multiplier using asynchronous logic," in *Asynchronous Circuits and Systems (ASYNC), 2012 18th IEEE International Symposium on*, 2012, pp. 81–88.
- [5] A. Martin and M. Nystrom, "Asynchronous techniques for system-on-chip design," *Proceedings of the IEEE*, vol. 94, no. 6, pp. 1089–1120, 2006.
- [6] P. Teichmann, *Adiabatic Logic: Future Trend and System Level Perspective*, 1st ed. New York, NY, USA: Springer, 2012.
- [7] J. Simsic and S. Teran. (2013, Sep.) 8051 core. [Online]. Available: <http://opencores.org/project.8051>
- [8] A. Vetuli, S. Pascoli, and L. Reyneri, "Positive feedback in adiabatic logic," *Electronics Letters*, vol. 32, no. 20, pp. 1867–1869, 1996.
- [9] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*. New York: Oxford University Press, 2010.
- [10] J. J. F. Cavanagh, *Digital Computer Arithmetic: Design and Implementation*. McGraw-Hill Book Company, 1984, ch. 4.
- [11] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *Computers, IEEE Transactions on*, vol. C-22, no. 8, pp. 786–793, 1973.