Implementation of a Carry Lookahead Fast Adder

Healy, Matthew mhealy@mst.edu
Johnston, Jaxson jnjt37@mst.edu
Grbeša, Lukas lgqq3@mst.edu

Abstract—Carry lookahead adders are one of the most common fast-adder architectures used in high performance computing to date. They allow for arbitrarily large summations with only $O(log(n))$ complexity, rather than the $O(n)$ complexity of a traditional ripple-carry adder.

The purpose of this project is to implement a simulation of the execution of a carry lookahead adder using the Intel 8051 architecture. Although this simulation cannot achieve the performance of a hardware implementation, it serves as a proof of concept for a carry lookahead adder.

I. Introduction

The carry lookahead adder was originally patented by IBM[1] in 1972. As time passed and technology improved, this method of addition was shown to be even more useful than originally thought, as it allows for the quick computation of numbers that would take a non-insignificant amount of time using the traditional ripple-carry method.

Our goal in this project is to implement a simulation of a carry lookahead adder on the 8051 microcontroller as a proof of concept. Since execution is serial, this simulation will not be faster than the built-in adding operations, as we are unable to calculate multiple things at once, as we could were we building a hardware model of a carry lookahead adder.

II. Approach

The carry value of a certain bit addition can be determined by looking at the previous pair of bits, finding their generate and propagate values, and comparing the propagate value to the carry of the previous bit operation. After a string of carries is created then all that must be done is two XOR operations because XORing the first input with the second input and XORing the result of that operation with the carry string gives the logical equivalent of adding the two numbers directly.

There are only two registers in the 8051($R0$ and $R1$) that can be used with indirect addressing. This means that we could only have two pointers to our data at a time. Since we planned on having up to five strings of stored data($A, B, P, G,$ and Carry), we know that we had to reuse memory. To solve this, we had P replace input A and G replace input B in memory, and then had the carry string overwrite each bit of the generate string one by one. We knew this was a workable solution, since only the carry depends on the G string, and the final sum depends only on the P string and the carry string.

Since bitwise operations could only be performed on the carry bit and a single bit of either the A register or the B register, and only the A register can be rotated(which was necessary for our generation of the carry string, which can be seen in the included source code), we were forced to constantly move data from the A register into memory as temporary storage, while the B register was moved to A for rotation. In hindsight, this could have been accomplished much more easily with a simple SWAP command, but we did not feel that the performance increase such a change would provide was worth the additional effort of a refactor.

Since we needed to include the total number of cycles that our operation took to complete, we needed to start and stop a timer before and after the operation, respectively. Timer mode 1 was chosen, as it was 16 bit, and so could run for the longest amount of time without overflowing. This choice was not arbitrary, as any timer that would overflow would cause us to need to check for timer overflow after every instruction, and increment some form of counter to account for the wraparound. This would have also added instructions(and so cycles) to our program that were completely useless for the actual addition. We ensured that our timer never overflowed during our tests, which included the largest input size we could account for, by checking the value of the $TF1$ flag at the end of execution. Since it was never set, we assumed it safe to just let the timer run unmolested until the end of execution, and then read the $TH1$ and $TL1$ bytes to see the total time elapsed.

We chose a fixed-baudrate serial mode, since doing any form of variable baudrate would require the use of another timer, and unnecessarily complicate the program further.

III. Results and Discussion

By first generating our Propogate(P) and Generate(G) strings from the input, and then stepping through our carry string and deriving each bit from the P and G strings and the previous carry bit, we emulate the stages of operation of a carry lookahead adder.

The P string is created by XORing each pair of respective input bits, and the G by ANDing together each pair of input bits. This would be done in hardware with blocks of two logic gates all operating in parallel, and would take approximately $2\Delta t$ to complete.

The next stage, wherein the P and carry bit are XORed together to create the sum, is emulated by a loop that XORs bytes of the carry string with bytes of the P string. While this adds another n time in the software solution, it would implement only another $2\Delta t$ in a hardware solution, since it would bo only two more logic gates for each bit, all operating in parallel.

Algorithm 1 shows the basic implementation of this simulation:

Algorithm 1 Pseudocode for CLA Adder

```
R3 → carry string
R6 → Input A
R7 → Input B
for all bytes $P_i \in P, G_i \in G, RX_i \in RX$ do
    $P \leftarrow R6 \oplus R7$
    $G \leftarrow R6 \wedge R7$
end for
for all bits $C_i \in C, P_i \in P, G_i \in G$ do
    $C \leftarrow G \vee (P \wedge C)$
end for
for all bytes $SUM_i \in SUM, C_i \in C, P_i \in P$ do
    $SUM_i \leftarrow C_i \oplus P_i$
end for
```

TABLE I
Inputs, Sum, and Computation time

Operand 1	Operand 2	Sum	Time(clock pulses)
$0x8A$	$0x25$	$0xAF$	$0xA0$
$0x1FFF$	$0xA222$	$0xC111$	$0x12E$
$0x3EFF$	$0x8AAA$	$0xC9A9$	$0x12E$
$0x254621$	$0xAAAAAA$	$0xCFF0CB$	$0x01D4$
$0x8532AA$	$0x243699$	$0xA96943$	$0x01D5$
$0x23456789$	$0x98765432$	$0xBBBBBBBB$	$0x026A$
$0xABABABAB$	$0x11111111$	$0xBCBCBCBC$	$0x04C2$
$0xC1C1C1C1C1$	$0x1C1C1C1C1C$	$0xDDDDDDDDDD$	$0x0300$
$0x123456789ABC$	$0x123456789ABC$	$0x2468ACF13578$	$0x0396$
$0x1234567C1C1C1C$	$0x11AAAAAA8532AA$	$0x23DF0126A14EC6$	$0x042E$

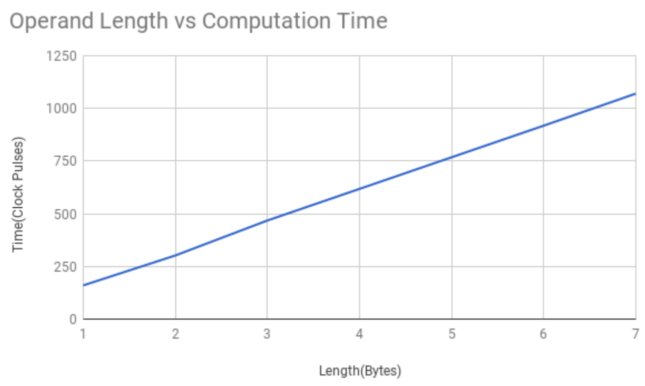

Fig. 1. Inputs vs Computation Time

As can be seen in Fig. 1, this software simulation falls far short of the speed of its hardware implementation. However, it does match very closely the speed of a ripple-carry add on the 8051 microcontroller, changing only the constant in the time-complexity.

IV. Summary and Conclusion

Upon testing we found that because the carry lookahead adder must be executed in serial, the simulation doesn't end up being any faster than simply adding the two values together. With that said, we were still successful in creating and implementing the carry lookahead adder, and, if given access to different hardware, we would be able to lower the execution time for the addition.

V. Source Code

```
         MOV 40H, #00H   ; Input A
         MOV 41H, #00H
         MOV 42H, #00H
         MOV 43H, #00H
         MOV 44H, #00H
         MOV 45H, #00H
         MOV 46H, #00H
         MOV 47H, #00H

         MOV 48H, #00H  ; Input B
         MOV 49H, #00H
         MOV 4AH, #00H
         MOV 4BH, #00H
         MOV 4CH, #00H
         MOV 4DH, #00H
         MOV 4EH, #00H
         MOV 4FH, #00H

         MOV R2,  #8H   ; Length of longest operand in bytes

         MOV R0, #40H
         MOV R1, #48H
         MOV A, R2
         MOV R5, A      ; length of operands stored in R2

SETUP:   MOV SCON, #10000010B
         MOV TMOD, #00010000B
         MOV TL1, #00H  ; start timer at 0
         MOV TH1, #00H  ; start timer at 0

INPUTA:  JNB TI, $      ; wait until ready to transmit
         CLR TI
         MOV A, @R0
         MOV C, P
         MOV TB8, C     ; set odd parity bit
         MOV SBUF, A    ; output byte of A
         INC R0         ; move to next byte
         DJNZ R5, INPUTA
         MOV R0, #40H   ; reset to beginning of A

         MOV A, R2      ; reset counter
         MOV R5, A
INPUTB:  JNB TI, $      ; wait until ready to transmit
         CLR TI
         MOV A, @R1
         MOV C, P
         MOV TB8, C     ; set odd parity bit
         MOV SBUF, A    ; output byte of B
         INC R1
         DJNZ R5, INPUTB
         MOV R1, #48H   ; reset to beginning of B

         SETB TR1       ; start timer
         MOV A, R2      ; init counter
         MOV R5, A
LOAD:    MOV A, @R0     ; temp hold for byte of R6 data
         MOV R4, A
         MOV A, @R0
         XRL A, @R1     ; propagate
         MOV @R0, A

         MOV A, @R1
         ANL A, R4      ; generate
         MOV @R1, A
         INC R0         ; move to next bit of P
         INC R1         ; move to next bit of G
         DJNZ R5, LOAD

         MOV A, R2      ; reset R5
         MOV R5, A
         CLR C          ; C will be used as Ci in boolean equation
         MOV A, R2
         DEC A
         ADD A, #40H
         MOV R0, A      ; point at least significant byte
         MOV A, R2
         DEC A
         ADD A, #48H
         MOV R1, A      ; point at least significant byte


CARRY:   MOV B, @R1
         MOV A, @R0
         MOV R4, #8H    ; set counter for 8 rotations
         MOV 0D6H, C    ; store carry—in
BITE:    ANL C, 0E0H    ; intermediate = Ci AND P(i+1)
         ORL C, 0F0H
         MOV 0F0H, C    ; save Ci into C as well as R3.0
         RR A           ; rotate P byte
         MOV @R1, A     ; store P byte in mem
         MOV A, B       ; move G/C to A for rotation
         RR A           ; rotate G/C byte
         MOV B, A       ; replace byte in R3
         MOV A, @R1     ; reload P from mem
         DJNZ R4, BITE  ; repeat for whole byte
         MOV A, B
         MOV 0D5H, C    ; store carry—out
         CLR C
         RLC A          ; rotate C/G string to align carrys over correct bits
         MOV C, 0D5H    ; restore carry—out
         JNB 0D6H, NOINC
         INC A          ; increment if there was a carry—in
NOINC    MOV @R1, A     ; replace C/G in memory

         DEC R0
         DEC R1
         DJNZ R5, CARRY

         MOV R0, #40H   ; return to beginning of P
         MOV R1, #48H   ; return to beginning of C/G
         MOV A, R2
         MOV R5, A
```

```
SUM:    MOV A, @R0
        XRL A, @R1
        MOV @R0, A      ; compute final sum
        INC R0          ; move to next bit of P
        INC R1          ; move to next bit of Carry string
        DJNZ R5, SUM
        CLR TR1         ; stop timer

TIME:   JNB TI, $       ; wait until ready to transmit
        CLR TI
        MOV A, TH1
        MOV C, P
        MOV TB8, C      ; set odd parity bit
        MOV SBUF, A     ; output high byte of time
        JNB TI, $       ; wait until ready to transmit
        CLR TI
        MOV A, TL1
        MOV C, P        ; set odd parity bit
        MOV TB8, C
        MOV SBUF, A     ; output low byte of time

        MOV A, R2
        MOV R5, A
        MOV R0, #40H    ; reset R0 to beginning of result string
OUT:    JNB TI, $       ; wait until ready to transmit
        CLR TI
        MOV A, @R0
        MOV C, P
        MOV TB8, C      ; set odd parity bit
        MOV SBUF, A     ; output byte of result
        INC R0
        DJNZ R5, OUT

FLUSH:  MOV A, #0FFH
        MOV C, P
        MOV TB8, C
        MOV SBUF, A     ; output dummy byte to flush output
        END
```

References

[1] Franz, S. and Dieter, S. Parallel binary carry look-ahead adder system. [US Patent 3,700,875]. https://www.google.com/patents/US3700875 1972.