

Lecture 7

Architecture & Components

What is a computer?

- A device that processes data.
- Data is processed in a predetermined manner controlled by a 'computer program'.
- A computer can be general purpose so that it can perform different tasks by running different programs.
- Embedded computers do not need to be general purpose (in most cases) but, with 'design reuse', it is economical to use a general purpose computer and application specific programs or software.

How do you make a computer?

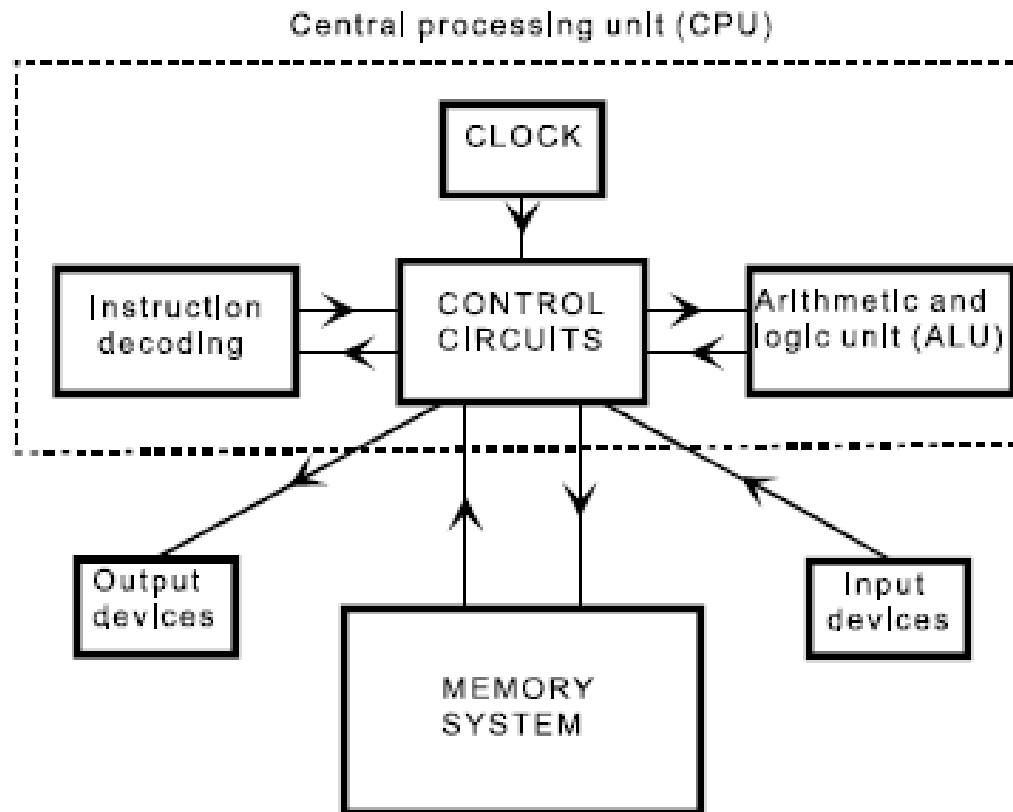
There are a number of things all computers possess:

- Input and output devices
- Memory (for both data and instructions)
- A 'central processing unit' CPU that can be further broken down into
 - a) control circuits
 - b) instruction decoder
 - c) arithmetic and logic circuits (ALU).

The way these are connected together is known as the 'computer architecture'. (Further differentiates between computer architecture and computer organization. Page 2)

Computer architecture describes the user's view of the computer.
The instruction set, visible registers, memory management table structures and exception handling model are all part of the architecture.

Computer organization describes the user-invisible implementation of the architecture.
The pipeline structure, transparent cache, table-walking hardware and translation look-aside buffer are all aspects of the organization.



Central Processing Unit (CPU)

The core component is the CPU which contains

- control circuits
- registers
- instruction decoding circuits
- an arithmetic and logic unit (ALU)
- an oscillator or clock to drive the system

The CPU is a sequential logic circuit. It implements the control sequence necessary to cause the system to follow the instructions of a computer program.

CPU operation – basic principle

Instructions are stored in memory.

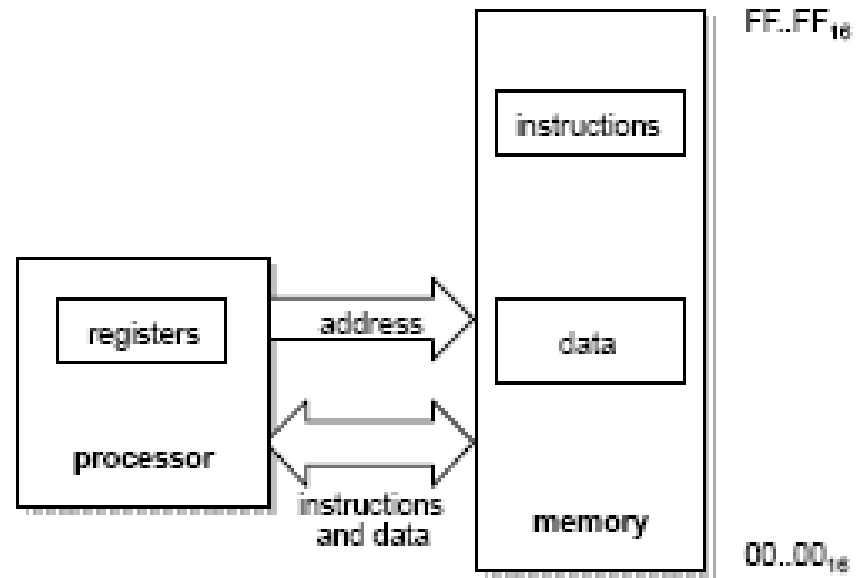
Once started, or reset, the control circuits automatically fetch the instruction codes of a program in the correct sequence from memory.

After each fetch, the control system determines which action is required for the instruction then performs the required action.

CPU + memory = total system

- Input / output can be integrated within the memory system.
- Memory is addressable, so that each byte in memory has a unique memory address.
- For a 32 bit address bus, each byte has an address from 0x00000000 to 0xFFFFFFFF.

- ✓ Instructions are fetched from memory on the data bus.
- ✓ The data bus is also used to pass data to or from the CPU.



Von Neumann

1946 Mathematician John von Neumann produced a report for the US army & the Princetown Institute for Advanced Study which describes the structure and operation of a computer system.

Preliminary description of the logic design of an electronic computing instrument.

A.W. Burks, H.H. Goldstein and T. von Neumann

Von Neumann systems

von Neumann systems are:

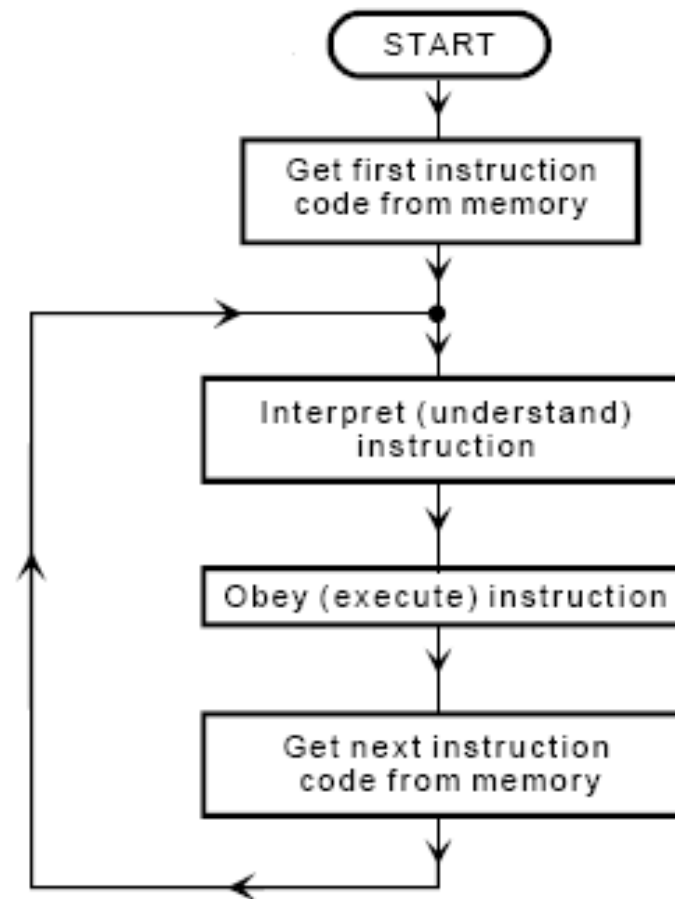
- said to have a von Neumann architecture/structure,
- said to follow the von Neumann operating sequence.

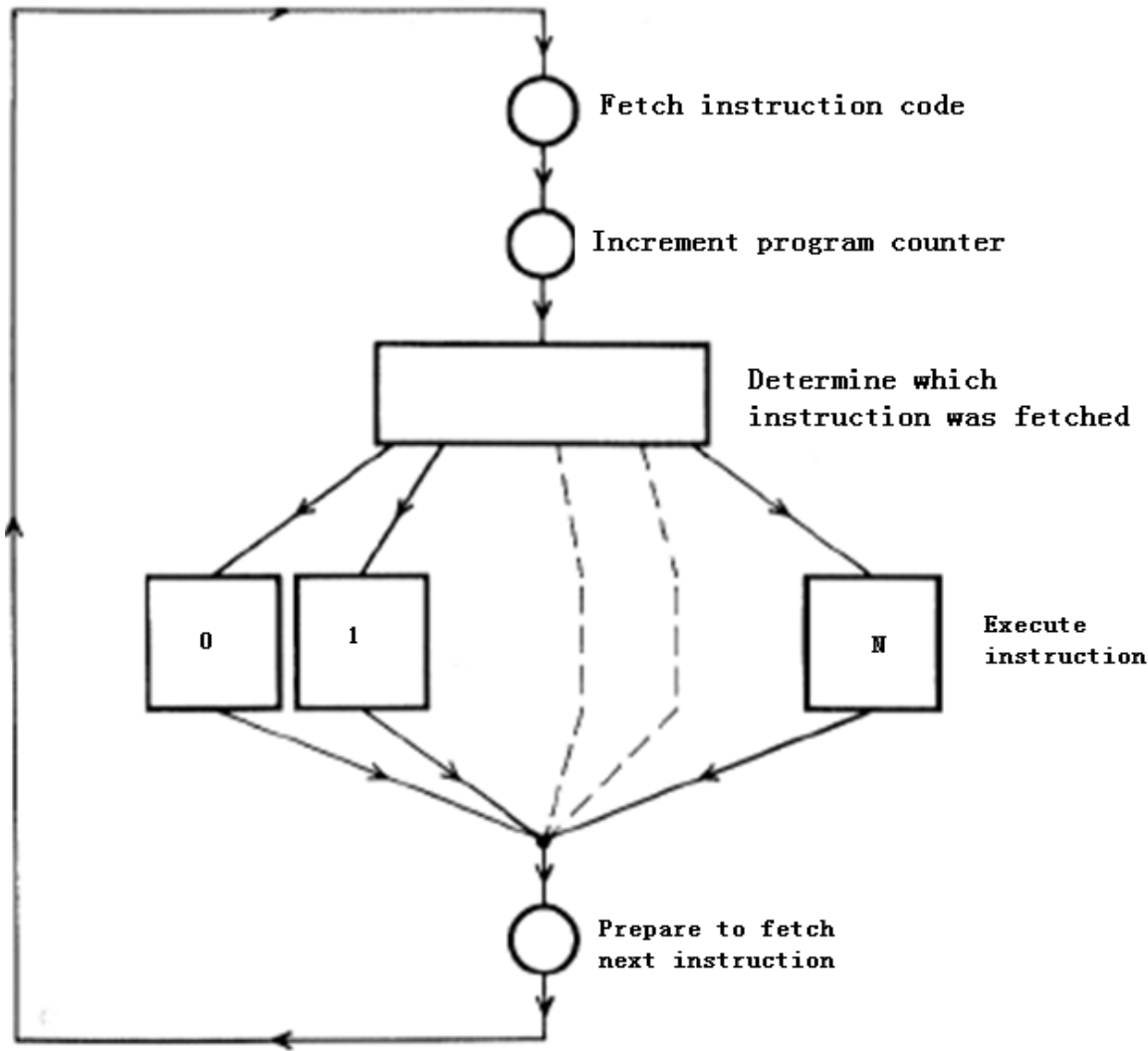
The basic von Neumann architecture/structure has

- Memory for storing both data and instructions
- Control and decode
- Arithmetic unit (now arithmetic and logic unit, ALU)
- Input and output, IO, mechanisms

The von Neumann cycle

- Computer programs are a sequence of instructions.
- Each instruction has a unique 'machine' code.
- Machine code has to be decoded before the instruction can be executed.
- The processor system operates a simple fetch, decode & execute loop - often called the von Neumann cycle.





A von Neuman system can be implemented as a finite state machine, in which each instruction has a different state.

Much of the hardware complexity is that for performing the execution stage - it is a multiway branch.

Control sequence

The control sequence is very simple

- A small sequential circuit is driven by the system clock and a different action takes place when it is in each state. In the most simple form the circuit has only three states
 - FETCH - DECODE - EXECUTE
- A single special purpose register in the control unit (may be called 'program counter' PC, 'instruction pointer' IP, etc) holds the address of the next instruction to be fetched.

What's in a CPU?

The CPU hardware includes

- The ALU circuits
- Registers/buffers (not all known to the user), including some for handling the bus system
- Several 'datapaths' / internal buses along which data is moved from one register to another
- Multiplexers and demultiplexers to select source and destination for transfers along the internal datapaths
- Sequential circuits to cause specific sequences of transfers
- Various clock, control and related circuits

How to design a CPU?

Different CPU architectures vary greatly

- ALU circuit offer different functionality (e.g. multiplication)
- Different number of internal registers/buffers
- Different datapaths / internal buses

What determines the architecture of a particular CPU e.g. the ARM7 or Intel 8080?

MU0 – A simple microprocessor

MU0 processor (see Furber, section 1.3).

3 internal registers;

- accumulator (ACC),
- instruction register (IR)
- and program counter (PC).

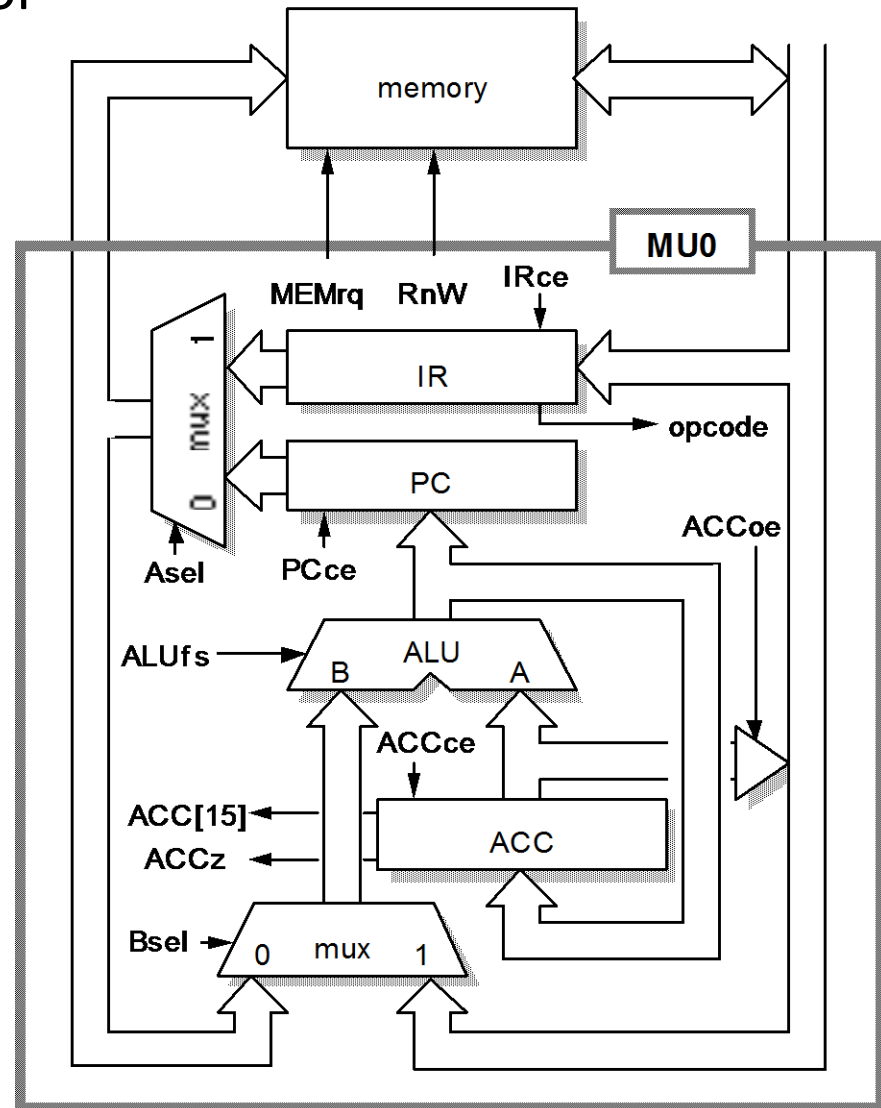
MU0 is a 16 bit machine with a 12 bit address space

Instructions are 16 bits long with a 4 bit opcode and a 12 bit address word

The Control Logic: Everything else such as decode and control use **FSM approach**.

Too simple for any 'real world' application.

A design to illustrate the principles of processor design.



Instruction Execution Sequence

Like any CPU, MU0 goes through the three phases of execution: These are repeated indefinitely. In more detail ...

a) Fetch Instruction from Memory [PC]

b) $PC = PC + 1$

c) Decode Instruction

d) Get Operand(s) from:

Memory {LDA, ADD, SUB}
IR (S) {JMP, JGE, JNE}
Acc {STO, ADD, SUB}

e) Perform Operation

f) Write Result to:

Acc {LDA, ADD, SUB}
PC {JMP, JGE, JNE}
Memory {STO}

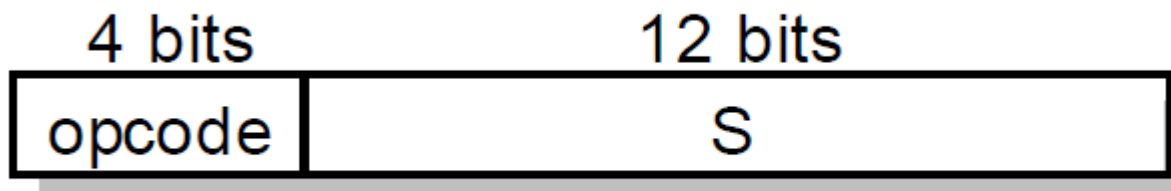
The MU0 instruction format

12 bit address space => 4k memory with 16 bit each location

4096 individually addressable memory locations

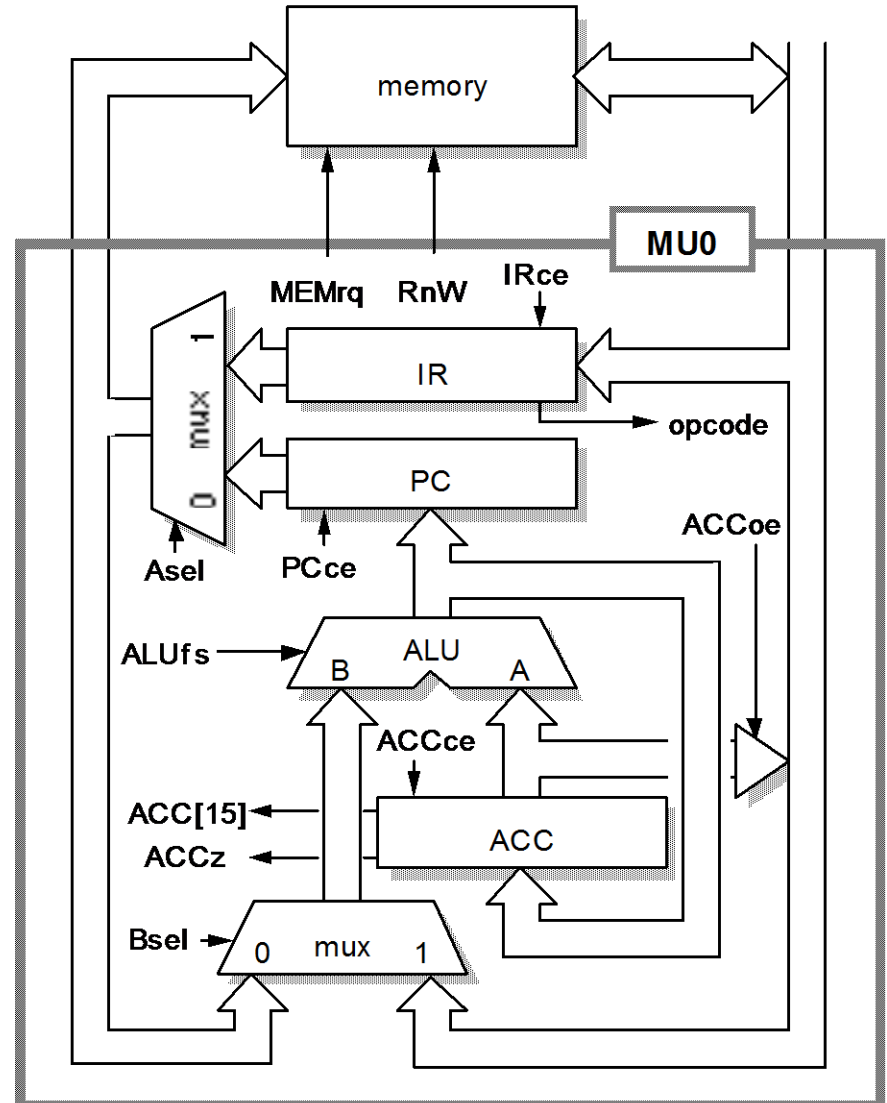
4 bit opcode => 16 possible assembly language

Commands only use 8! - good practice

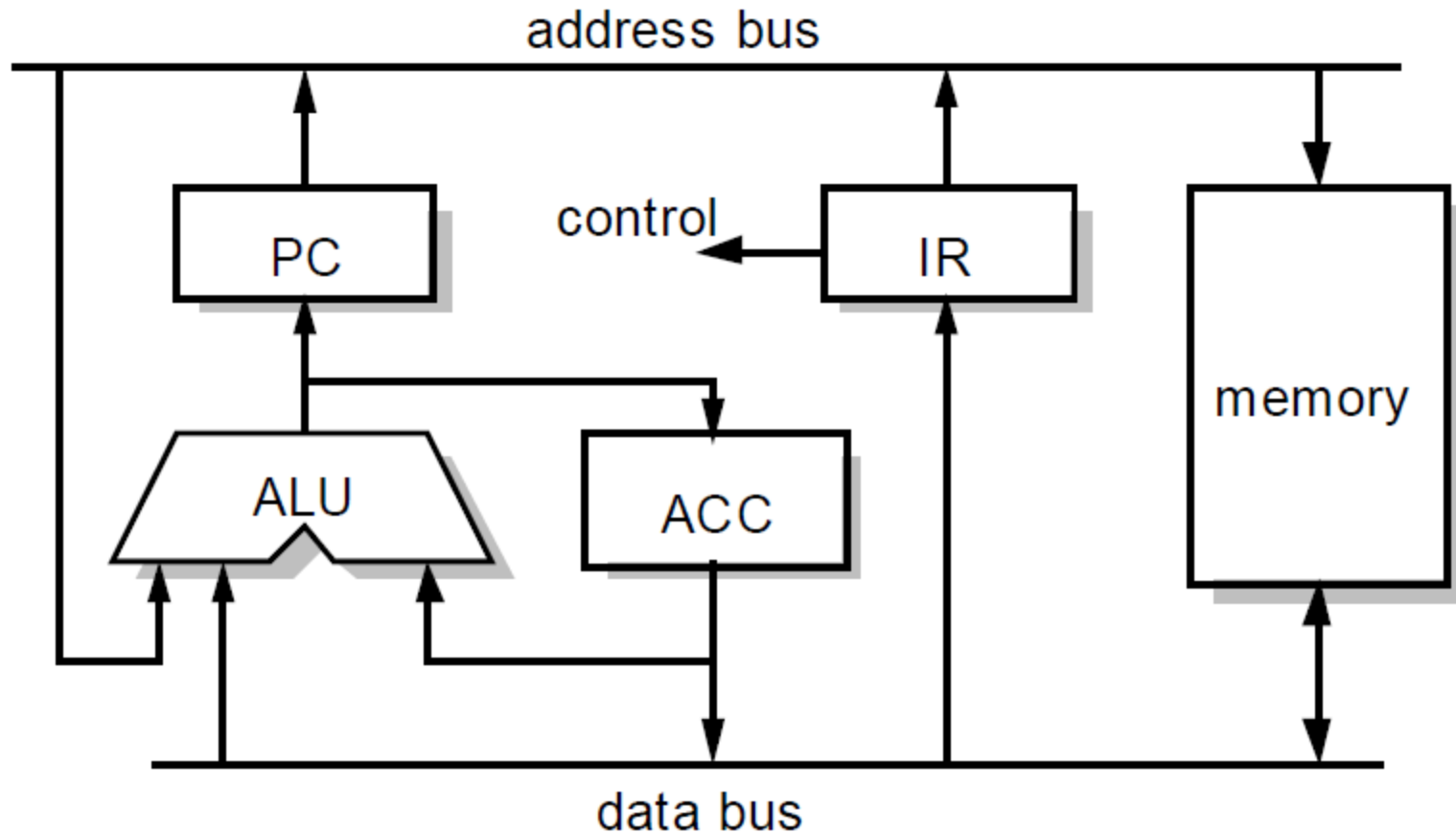


The MU0 instruction set

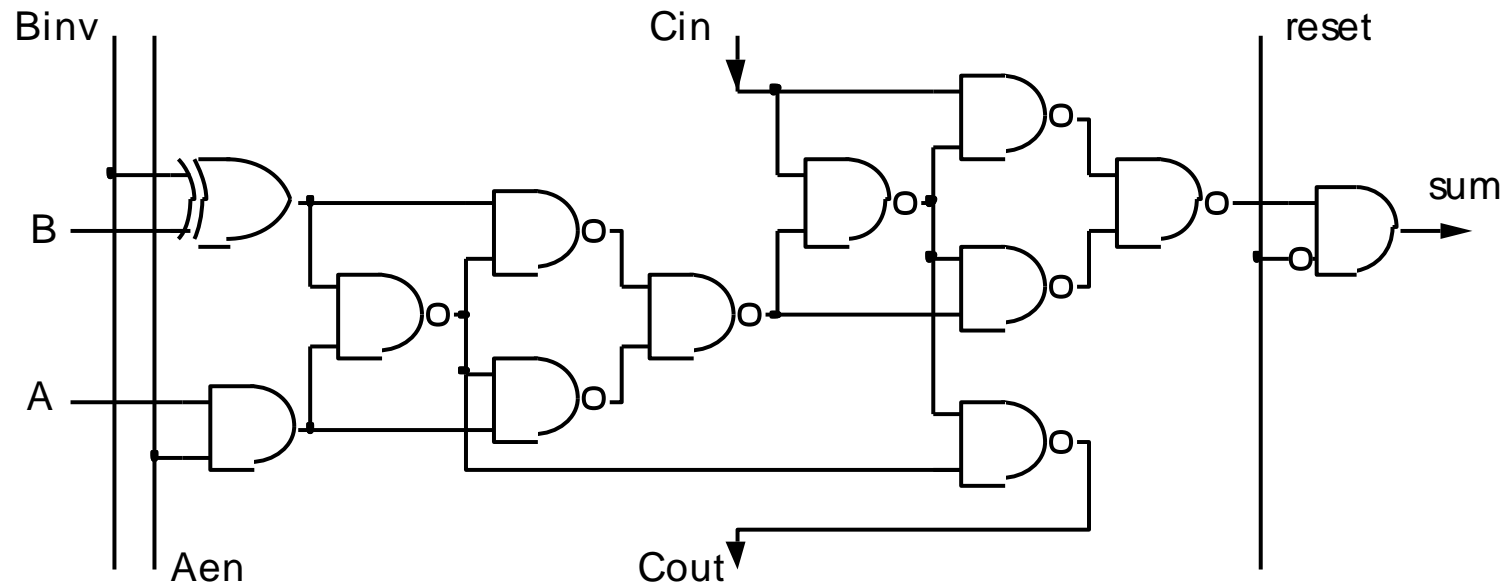
Instruction	Opcode	Effect
LDA S	0000	$ACC := mem_{16}[S]$
STO S	0001	$mem_{16}[S] := ACC$
ADD S	0010	$ACC := ACC + mem_{16}[S]$
SUB S	0011	$ACC := ACC - mem_{16}[S]$
JMP S	0100	$PC := S$
JGE S	0101	if $ACC \geq 0$ $PC := S$
JNE S	0110	if $ACC \neq 0$ $PC := S$
STP	0111	stop



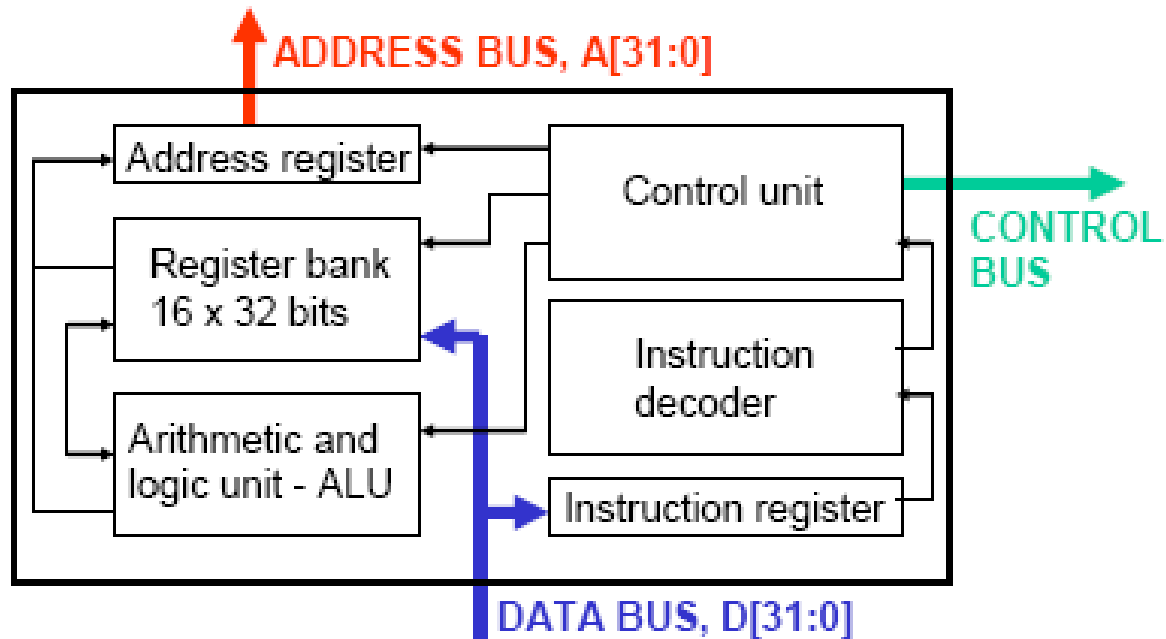
MU0 datapath example



MU0 ALU logic for one bit



ARM7 CPU architecture

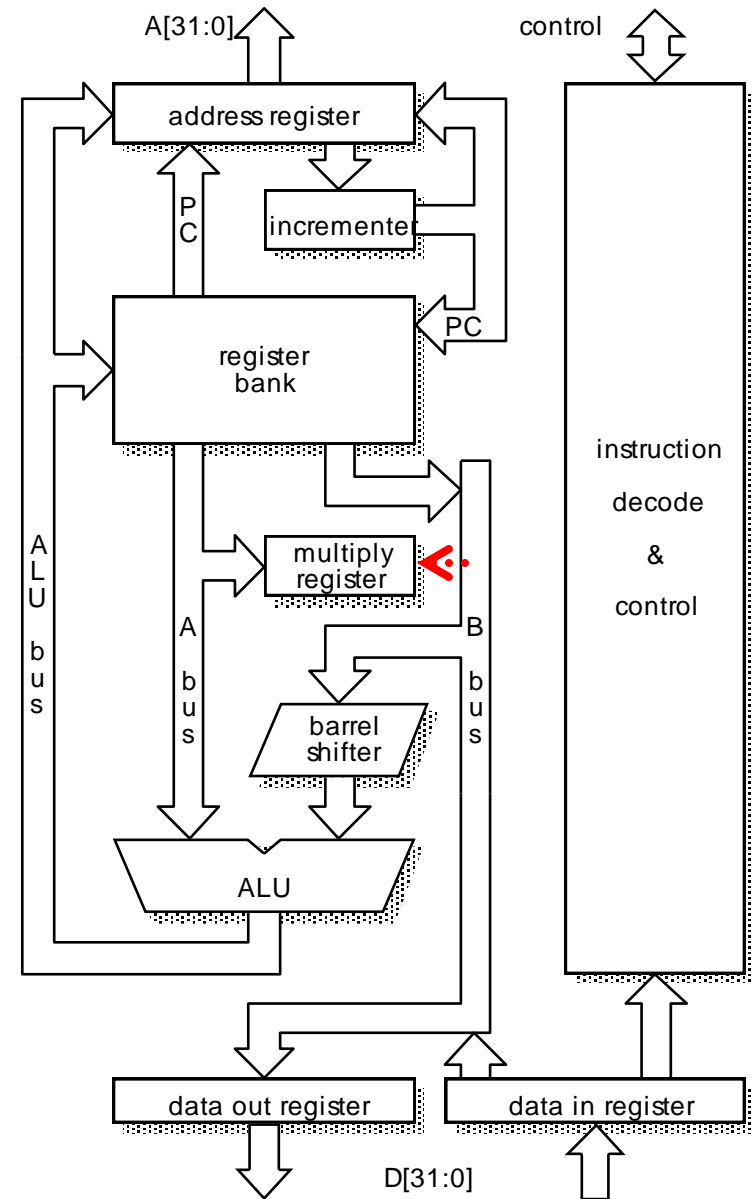


ARM7 CPU:

many internal registers;
address register
and register bank.

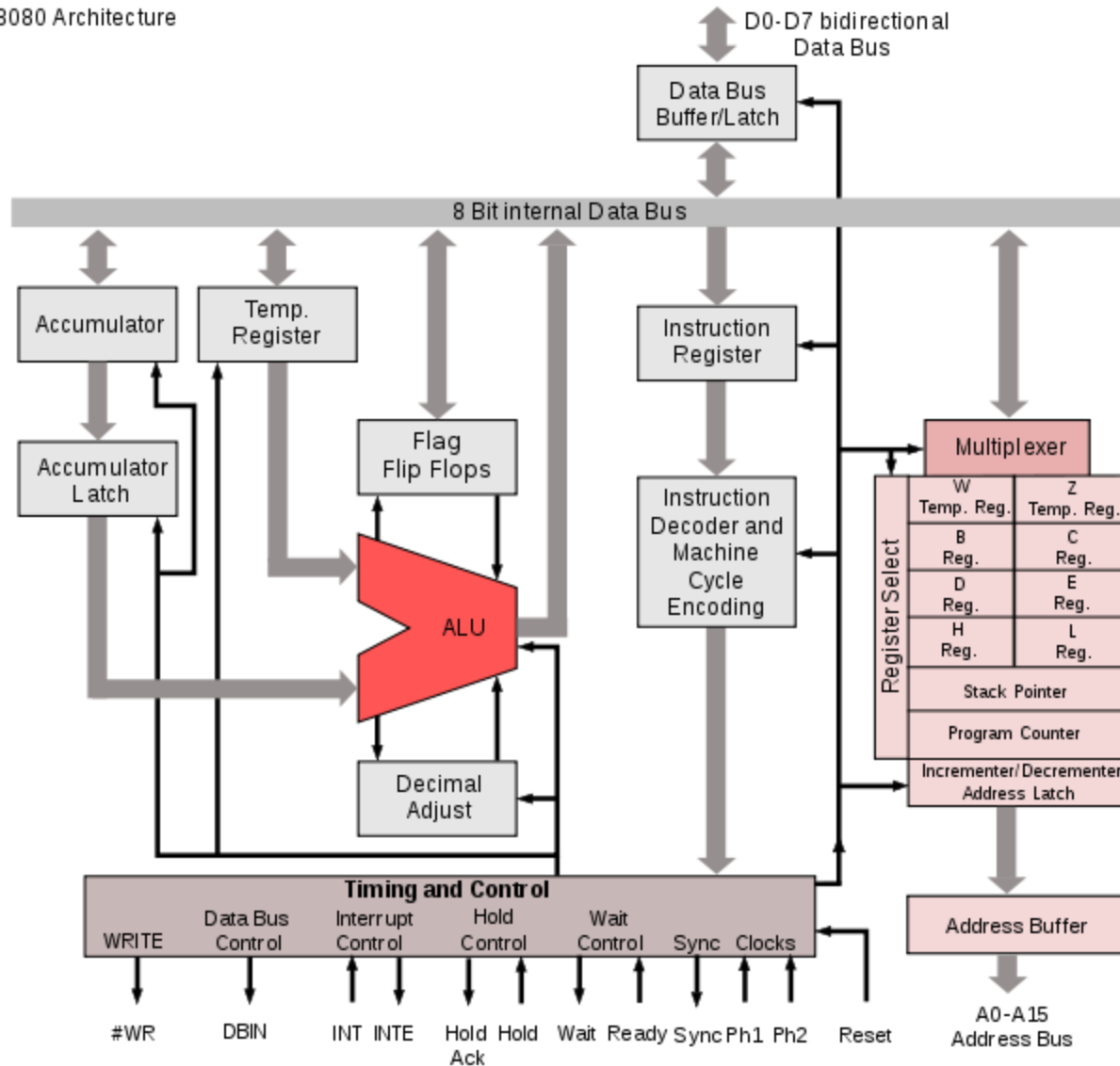
3 internal datapaths;
A bus,
B bus,
ALU bus.

additional ALU functionality;
multiplier
and barrel shifter.



Intel 8080 CPU: forerunner of Pentium

Intel 8080 Architecture



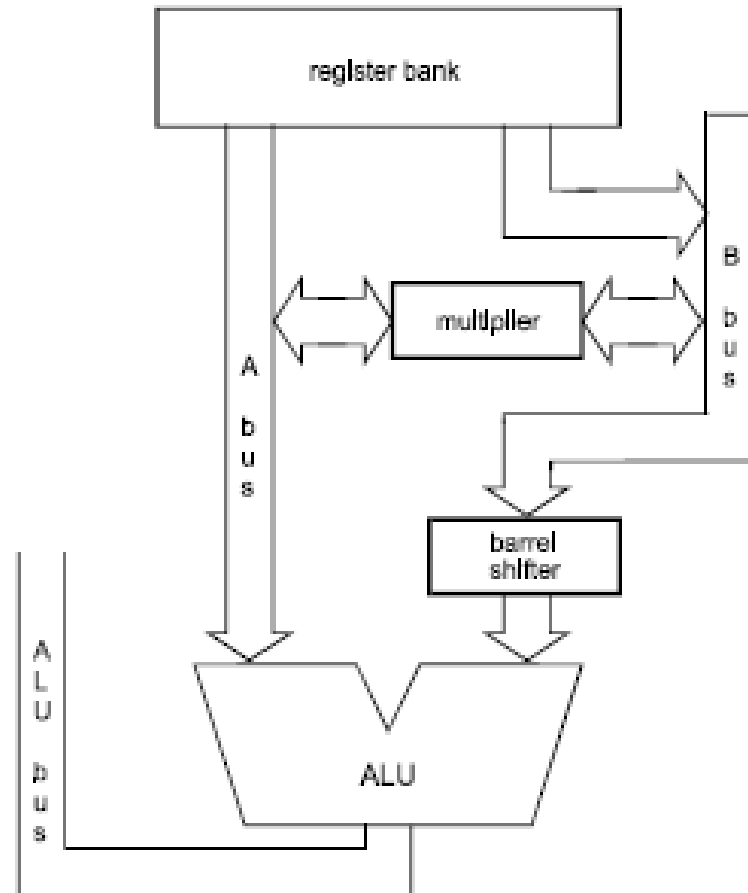
Source: Wikipedia

Arithmetic Logic Unit: ALU

The ALU is an important part of any microprocessor.

The ARM microprocessor divides the ALU functions into three blocks;

- a multiplier (that uses Booth's algorithm),
- the 'barrel shifter'
- and the rest of the ALU including the adder and logic functions. (This third block is generally referred to as the ALU.)



ALU circuits

An ALU can be considered to be one very large combinational logic circuit.

For Example – oversimplified case

Suppose we have 3-bit numbers

- (X and Y as inputs, Z as output),
- a 2-bit code, K, to indicate what the ALU is to do
 - (e.g. ADD, SUBTRACT, AND, OR)
- and a single flag CY to indicate carry.

We can write down the output for every possible combination of inputs as the rules are straight forward.

Truth table for a simple ALU circuit

K Code		Y digits			X digits			Result digits			CY	Comments
0	0	0	0	0	0	0	0	0	0	0	0	$0 + 0 = 0$
0	0	0	0	0	0	0	1	0	0	1	0	$0 + 1 = 1$
0	0	0	0	0	0	1	0	0	1	0	0	$0 + 2 = 2$
0	0	0	0	0	0	1	1	0	1	1	0	$0 + 3 = 3$
0	0	0	0	0	1	0	0	1	0	0	0	$0 + 4 = 4$
0	0	0	0	0	1	0	1	1	0	1	0	$0 + 5 = 5$
0	0	0	0	0	1	1	0	1	1	0	0	$0 + 6 = 6$
0	0	0	0	0	1	1	1	1	1	1	0	$0 + 7 = 7$
0	0	0	0	1	0	0	0	0	0	1	0	$1 + 0 = 1$
0	0	0	0	1	0	0	1	0	1	0	0	$1 + 1 = 2$
0	0	0	0	1	0	1	0	0	1	1	0	$1 + 2 = 3$
0	0	0	0	1	0	1	1	1	0	0	0	$1 + 3 = 4$
0	0	0	0	1	1	0	0	1	0	1	0	$1 + 4 = 5$
0	0	0	0	1	1	0	1	1	1	0	0	$1 + 5 = 6$
0	0	0	0	1	1	1	0	1	1	1	0	$1 + 6 = 7$
0	0	0	0	1	1	1	1	0	0	0	1	$1 + 7 = 0$ & overflow
0	0	0	1	0	0	0	0	0	1	0	0	$2 + 0 = 2$
												etc.

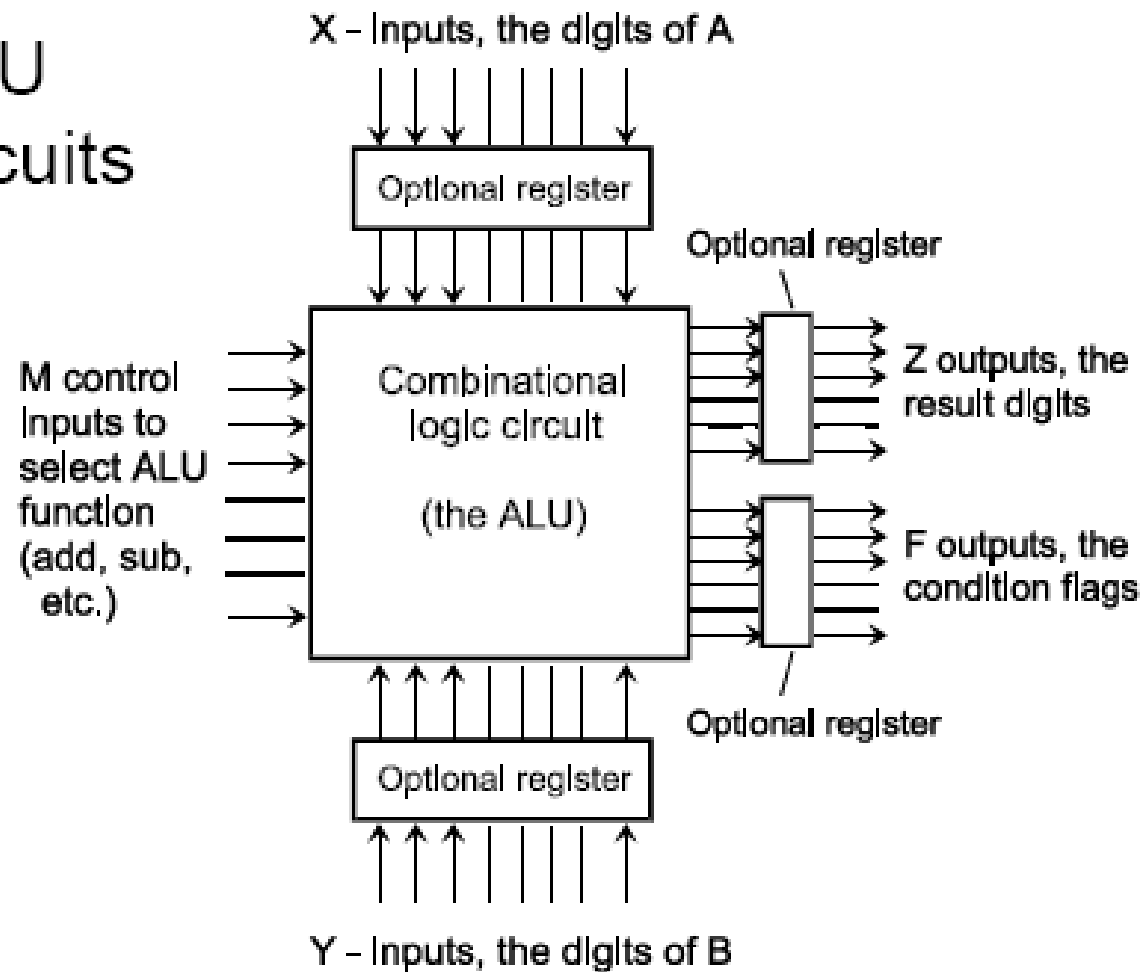
ALU circuits: design problem

The table is straightforward but is large for 3-bit numbers (has $2^8 = 256$ rows). This example is an unrealistic trivial case;

the design problem becomes extremely difficult in many cases.

A real ALU will have inputs with many more bits and K will be larger e.g. the ARM ALU has 32 bit inputs and at least 10 different functions (4 bit code) so the truth table would have at least $2^{(32+32+4)} \approx 2.95 \times 10^{20}$ rows.

ALU circuits



Combinational logic circuit design

Design of large combinational (e.g. arithmetic) circuits

– **some approaches**

- modular methods (e.g. cell technique)
- replace by connected more simple modules used in sequence – convert to a sequential circuit.

ALU circuit design

Modular - the cell technique

- break the circuit into separate modules – **cells**
- the chosen cells are small enough to be designed and built
- if possible only a small number of different types of cell are used.
- the cells are interconnected to form the complete circuit.

i.e. use cells which are larger building blocks than logic gates.

How to choose the cells?

ALU adder design

for arithmetic circuits is to examine how the arithmetic operation would be performed manually.

e.g. addition of the two 4-bit numbers 0110 + 0011

3	2	1	0	← Bit number
0	1	1	0	Addend (digits of A)
<u>0</u>	<u>0</u>	<u>1</u>	<u>1</u>	<u>Augend (digits of B)</u>
0	1	1	0	<u>Carry to next column</u>
1	0	0	1	Sum

The carry from the MSB column is the carry flag for a complete ALU.

ALU adder design

Taking each column separately

LSB column is to combine LSBs (bit 0) of A and B; Result is LSB (bit 0) of the **sum** and the **carry out** to next column

Next column is to combine bits position 1 and **carry out** from the LSB column (**carry in** to this column); Result is bit 1 of the **sum** and the **carry out** to column for bit 2

Next column is to combine bits position 2 and **carry out** from column 1 (**carry in** to this column)

etc.

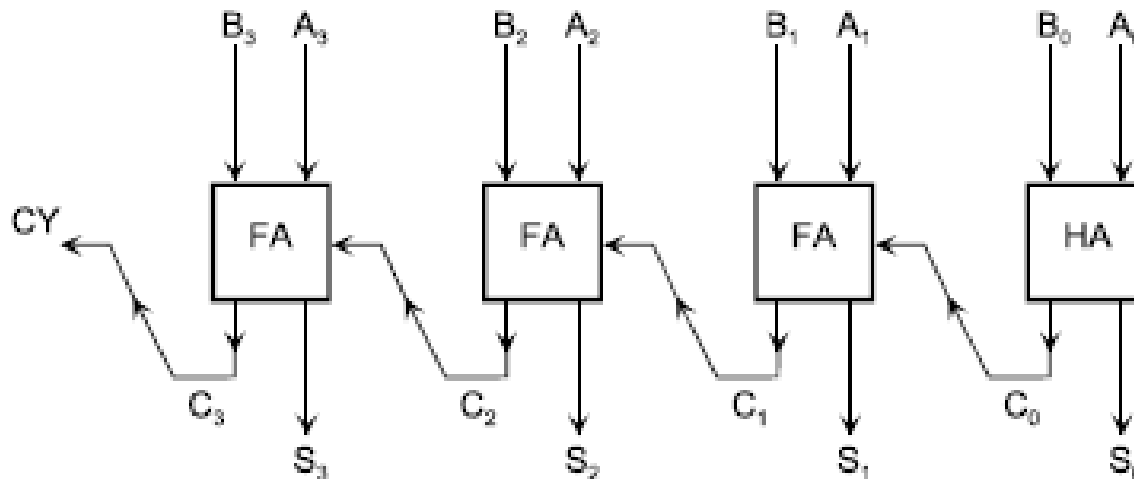
Ripple carry adder design

The truth table for each column is easily derived.

A **half-adder** for the LSB and a **full-adder** for the other bits.

The adders are obvious choices for the cells.

Connect the cells to give a **ripple-carry adder**



Problem with ripple-carry adder

- Each cell causes a propagation delay
- Cell for bit 1 cannot give a correct result until the cell for bit 0 has produced the carry output.
- Cell for bit 2 has to wait for the carry from the bit 1 cell
- and so on.

For an adder with many bits, the delays become very long.

E.g. $0xFFFFFFFF + 0x00000001$ would only produce a valid result after 32 propagation delays.

One bit full adder circuit

A 'full adder' adds two bits, A and B, and a 'carry-in' bit to produce a 'sum' bit and a 'carry-out' bit.

The truth table is as follows:

A	B	Cin	Cout	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$\text{Sum} = (A \oplus B) \oplus \text{Cin}$$

$$\text{Cout} = A.B + A.\text{Cin} + B.\text{Cin}$$

Carry out logic

The 'carry-out' logic can be implemented using a 'generate'/'propagate' approach. If 'generate' $G = 1$ then $C_{out} = 1$. $G = 1$ if both A and B are logic 1. If 'propagate' $P = 1$ then $C_{out} = 1$ if $C_{in} = 1$.

$$C_{out} = G + P.C_{in}$$

The truth table for G and P is as follows:

A	B	P	G
0	0	0	0
0	1	1	0
1	0	1	0
1	1	X	1

$$G = A.B$$

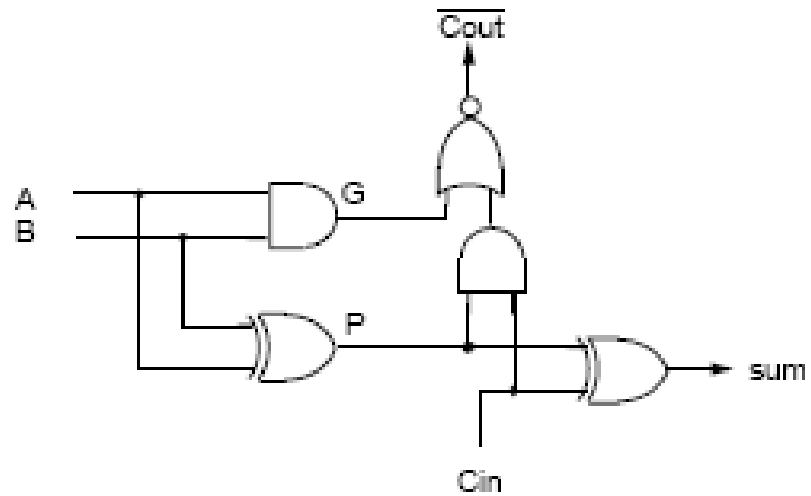
$$P = A + B \quad \text{or} \quad P = A \oplus B$$

ARM1 ripple-carry adder circuit

The original ARM adder circuit used the 1 bit ripple carry adder shown.

Note that the carry out is inverted.

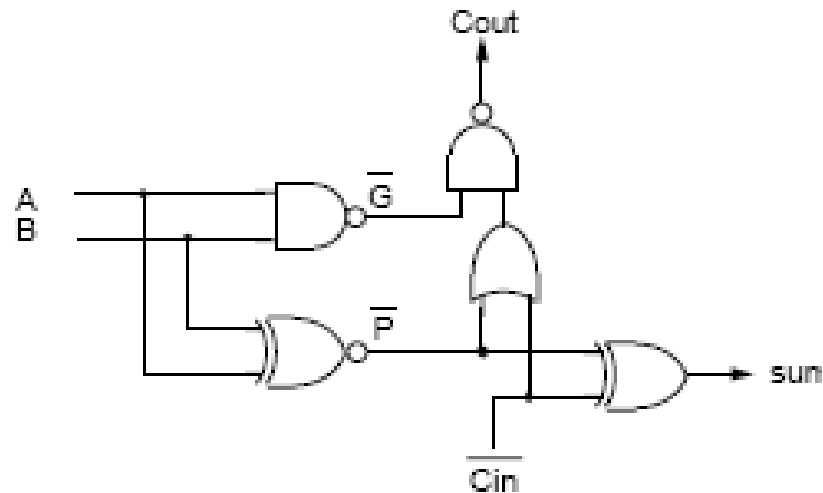
This is because the AND-OR-NOT function can be implemented as a single logic gate in CMOS hence reducing the critical Cin to Cout delay to one propagation delay.



ARM1 ripple-carry adder circuit

To take account of the inverted carry from the previous stage, the circuit shown is used for every alternate stage.

The OR-AND-NOT function can also be implemented as a single logic gate in CMOS so that a 32 bit adder would have a critical carry path of 32 propagation delays.



$$Cout = \overline{\overline{G} \cdot (\overline{P} + \overline{Cin})}$$

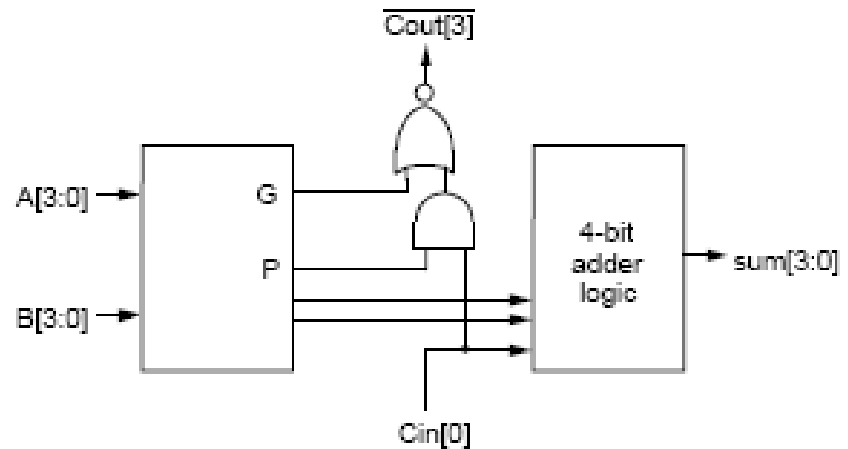
ARM2 4-bit carry look-ahead

The total propagation delay can be reduced by using a 'carry look ahead' scheme.

Now numbers are added in 4 bit blocks and the carry is computed for each block only.

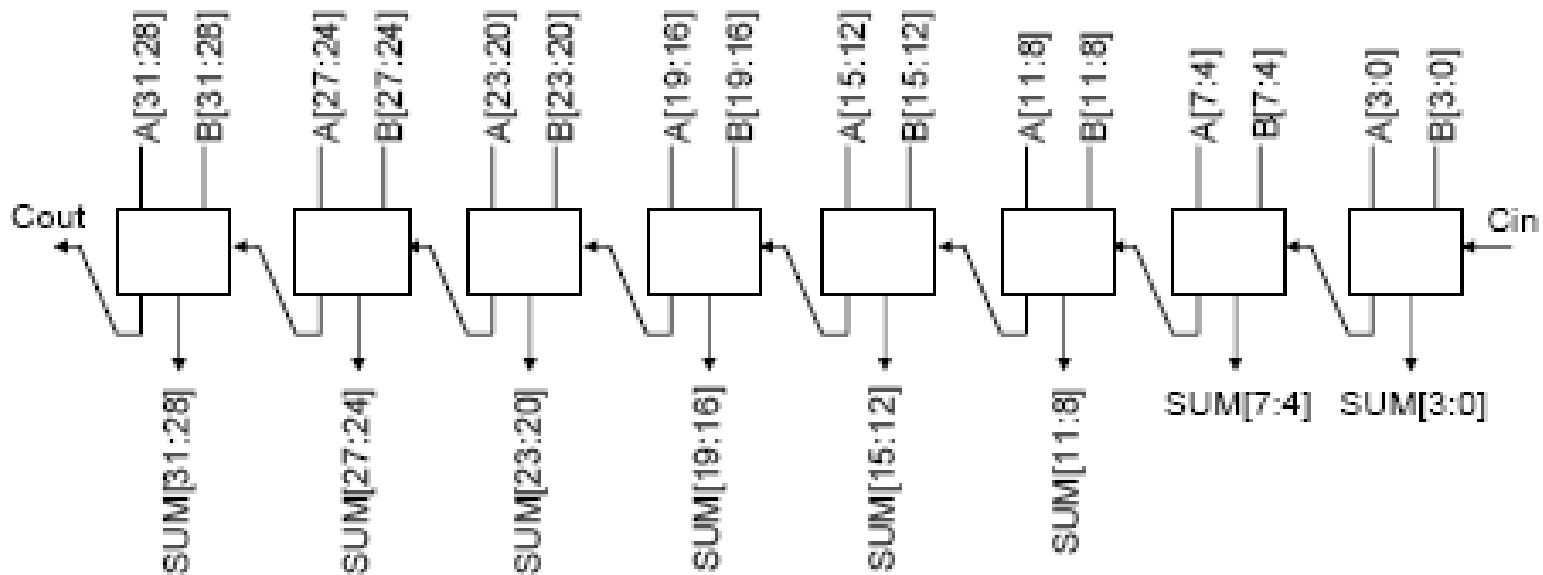
$G = 1$ if $(A+B) > 15$

$P = 1$ if $(A+B) = 15$



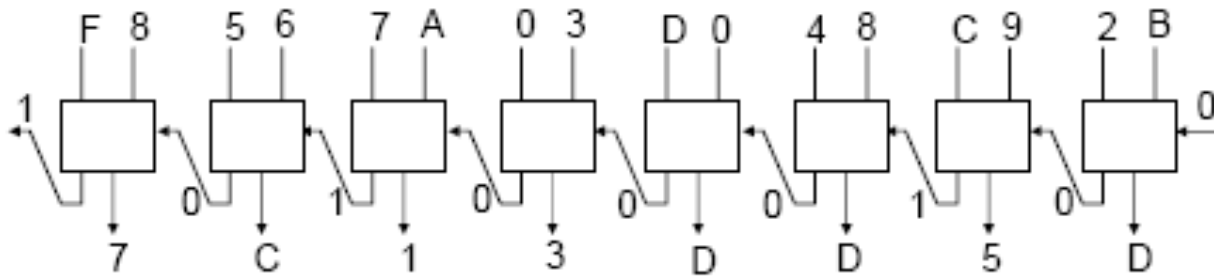
ARM2 4-bit carry look-ahead

The critical carry path now has 8 propagation delays for a 32 bit adder.



Carry look-ahead: example

For example adding 0xF570D4C2 to 0x86A3089B (with $C_{in} = 0$):



The sum is 0x7C13DD5D with $C_{out} = 1$.

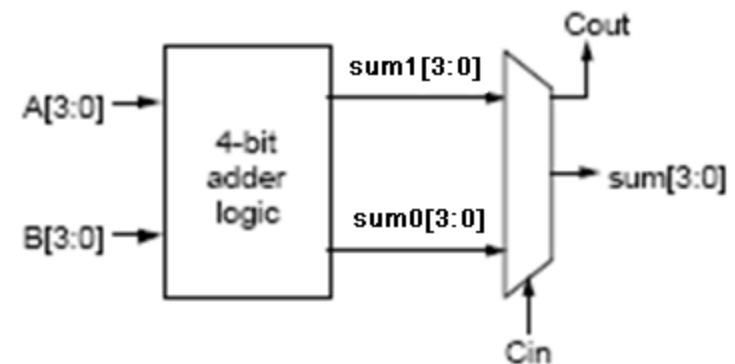
ARM6 carry select adder

The propagation delay can be further reduced by using a 'carry select' scheme.

The 4-bit adder logic produces two results; sum0 is simply $(A+B)$ whereas sum1 is the sum; $(A+B+1)$.

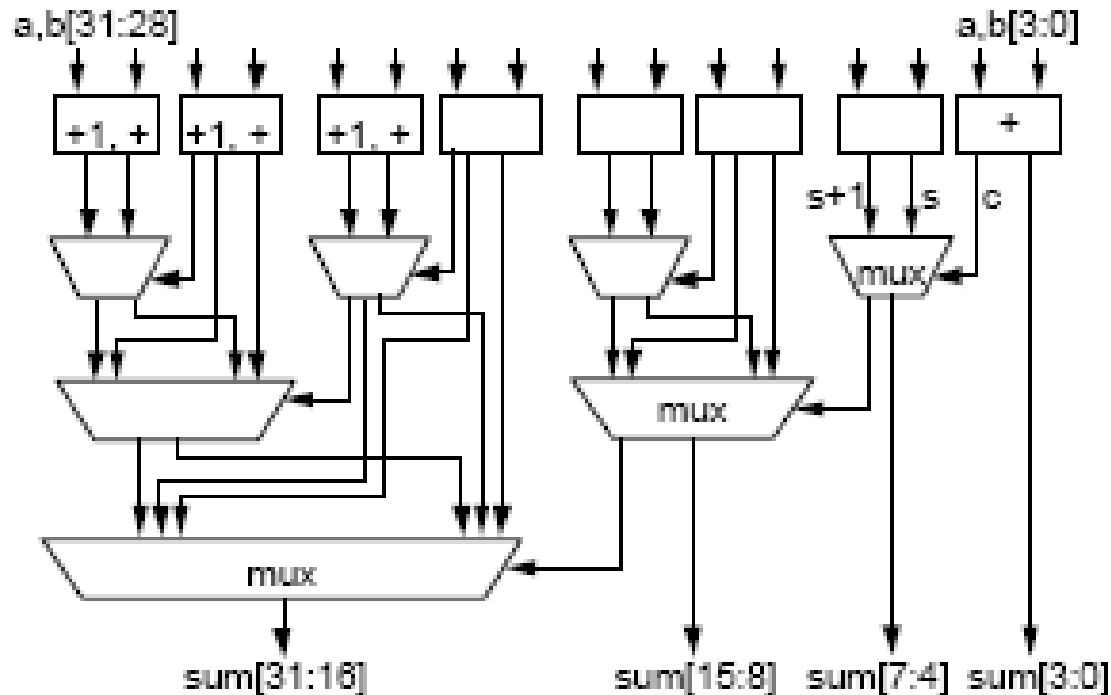
The carry-in is used to select one of these two results in a multiplexer.

The output of the multiplexer is the sum and carry-out chosen by the value of the carry-in.



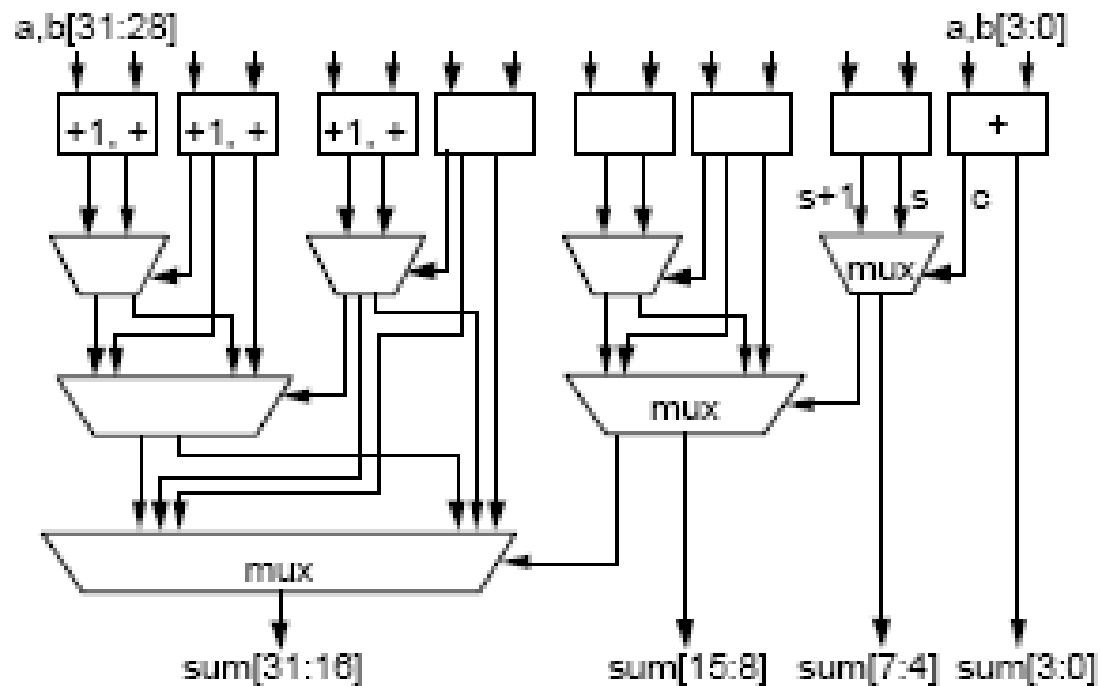
ARM6 carry-select adder

The adders and multiplexers are connected as shown for a 32 bit adder.



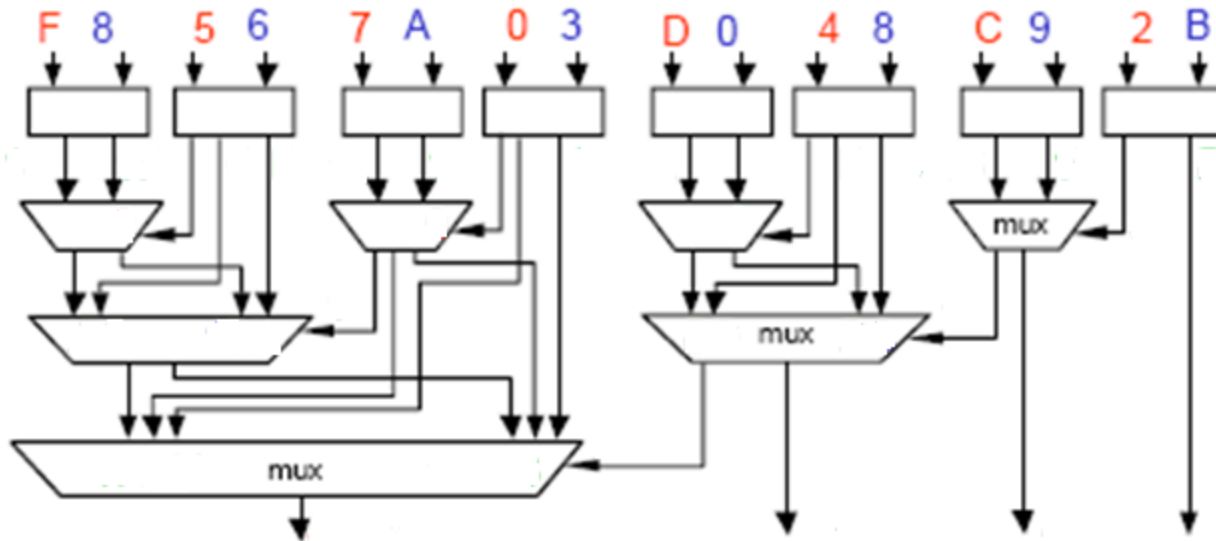
ARM6 carry-select adder

For a 32 bit adder there are a maximum of three propagation delays in the carry path.



Carry-select: example

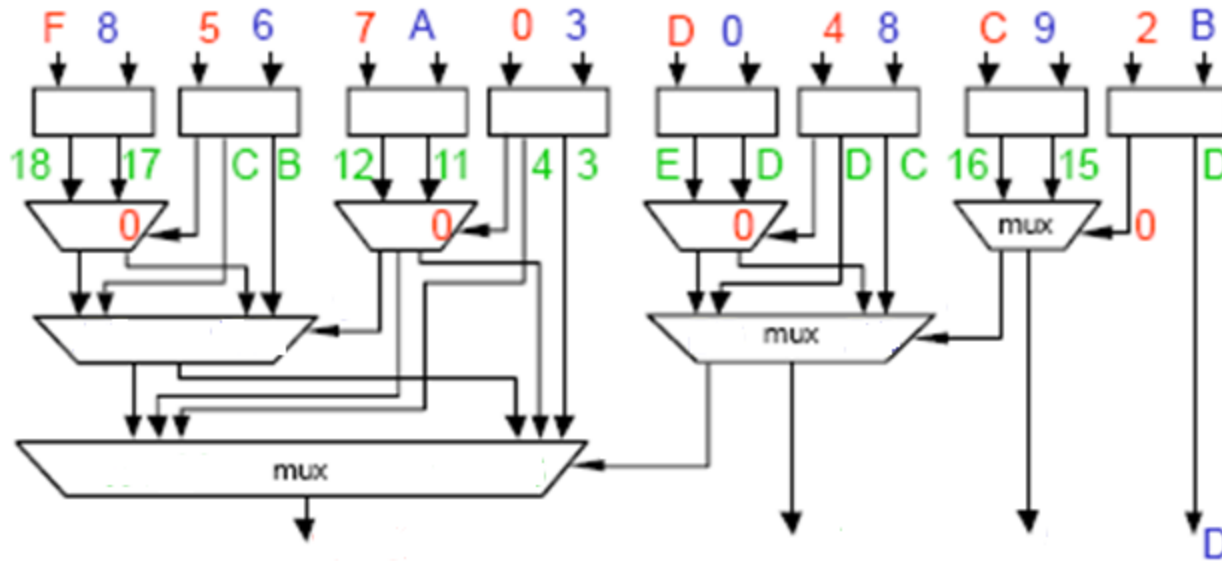
For example adding 0xF570D4C2 to 0x86A3089B:



The sum is 0x7C13DD5D with Cout = 1.

Carry-select: example

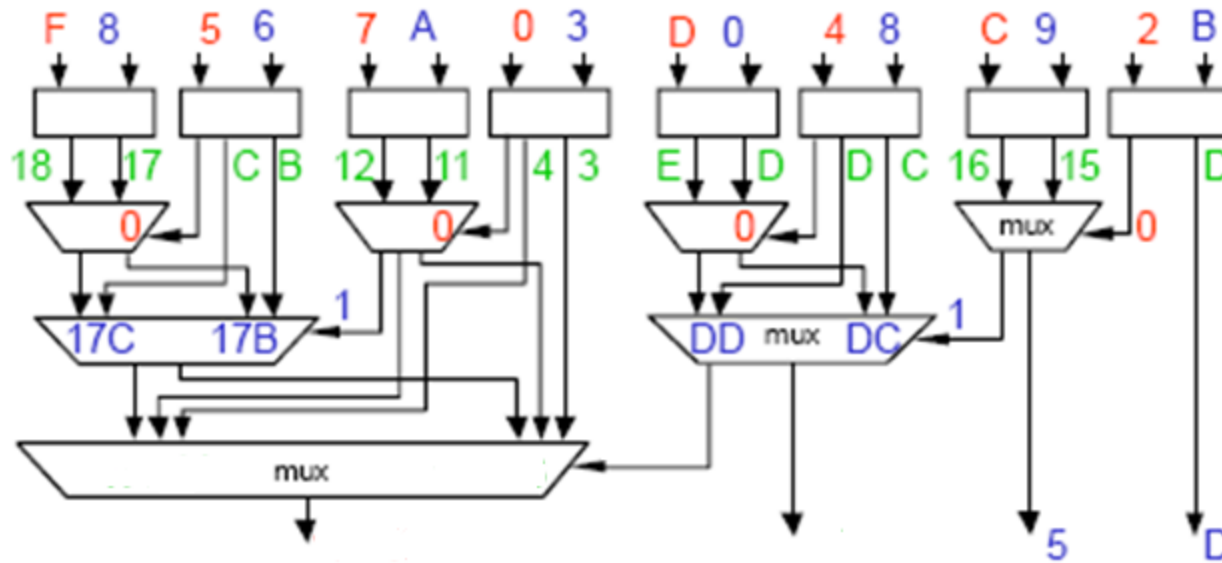
For example adding 0xF570D4C2 to 0x86A3089B:



The sum is 0x7C13DD5D with Cout = 1.

Carry-select: example

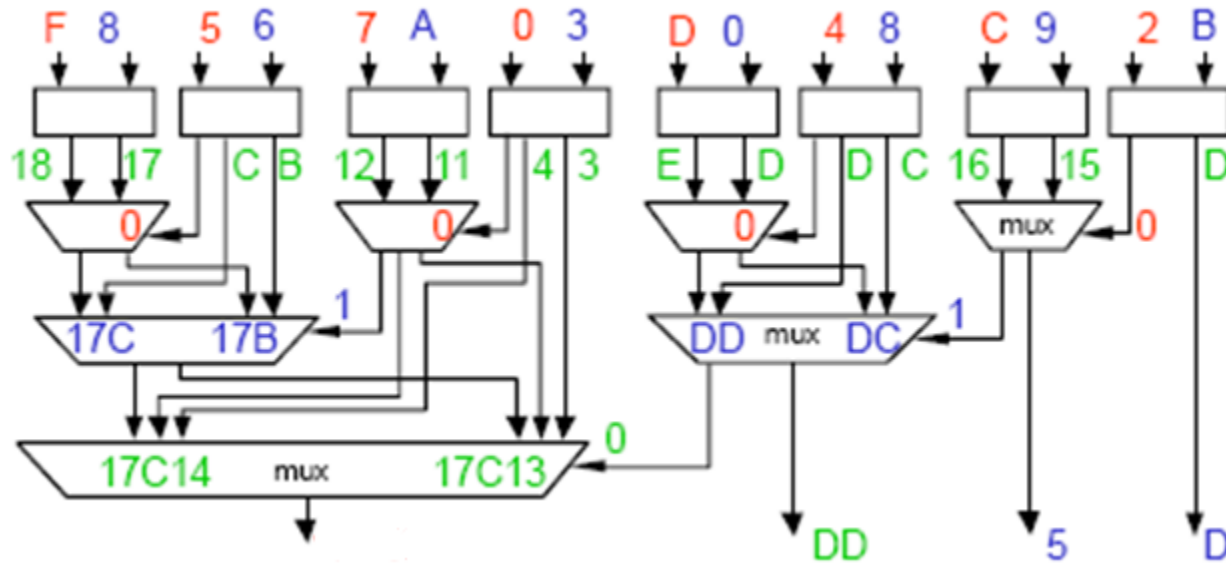
For example adding 0xF570D4C2 to 0x86A3089B:



The sum is 0x7C13DD5D with Cout = 1.

Carry-select: example

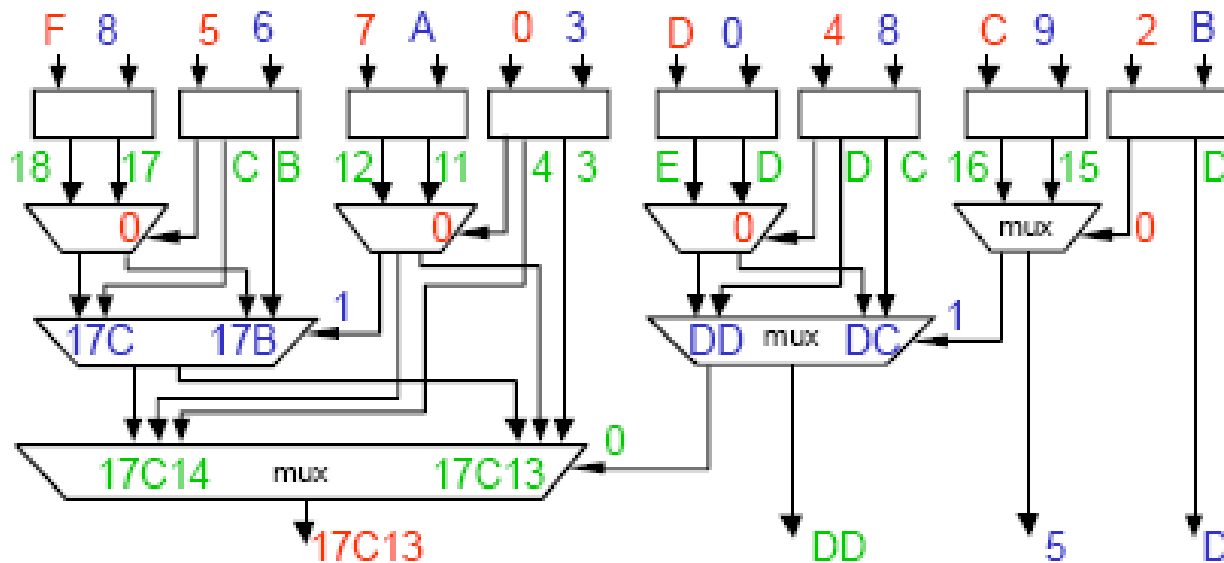
For example adding 0xF570D4C2 to 0x86A3089B:



The sum is 0x7C13DD5D with Cout = 1.

Carry-select: example

For example adding 0xF570D4C2 to 0x86A3089B:



The sum is 0x7C13DD5D with Cout = 1.

Performance comparison

The following table gives the number of propagation delays on the critical carry path for the three different types of adder (assuming the carry look ahead and carry select adders use 4 bit adder blocks).

The performance improvement is at the expense of more complicated circuits using more electrical power.

Size of adder	Ripple carry	Look ahead	Carry select
4 bits	4	1	1
8 bits	8	2	1
16 bits	16	4	2
32 bits	32	8	3
64 bits	64	16	4

Logic functions in the ALU

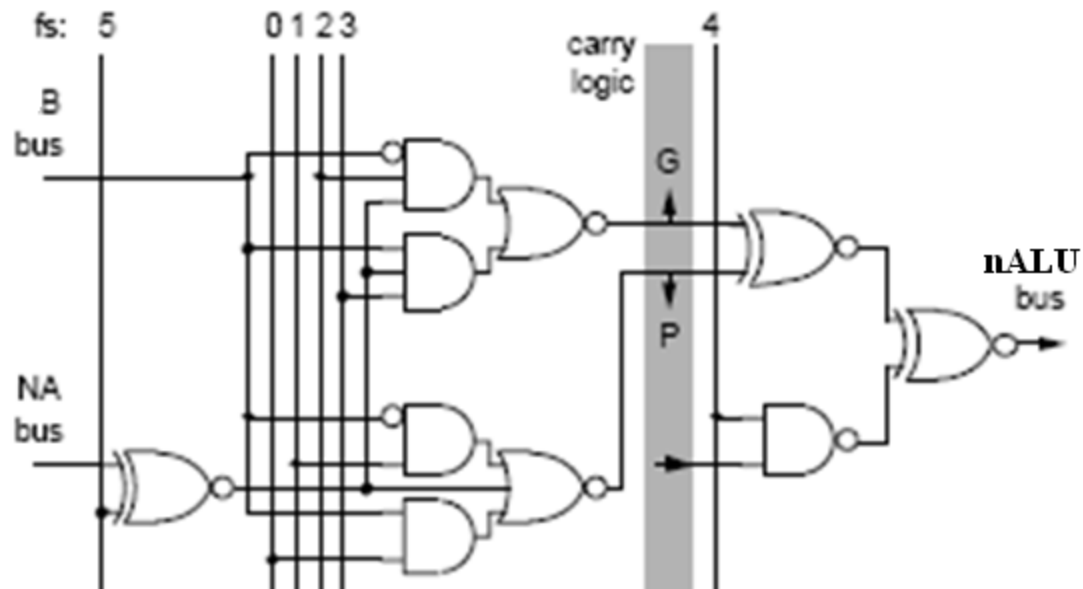
The ALU performs logic functions (AND, ORR, EOR, BIC) as well as arithmetic (ADD, SUB, etc.)

Logic functions can be combined in the same circuit as addition if a ripple carry adder is being used.

If addition uses 4 bit blocks, a separate logic function block is needed. The output of the ALU is then selected in a multiplexer depending upon the type of function being implemented.

ARM2 ALU logic for one bit

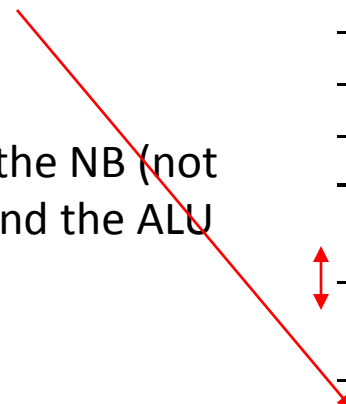
This circuit does 1 bit logic/addition functions depending upon the values of the 'function select' inputs; e.g. fs4 = 1 selects addition.



ARM2 ALU function codes

Note that the table given in Furber (Table 4.1) is incorrect.

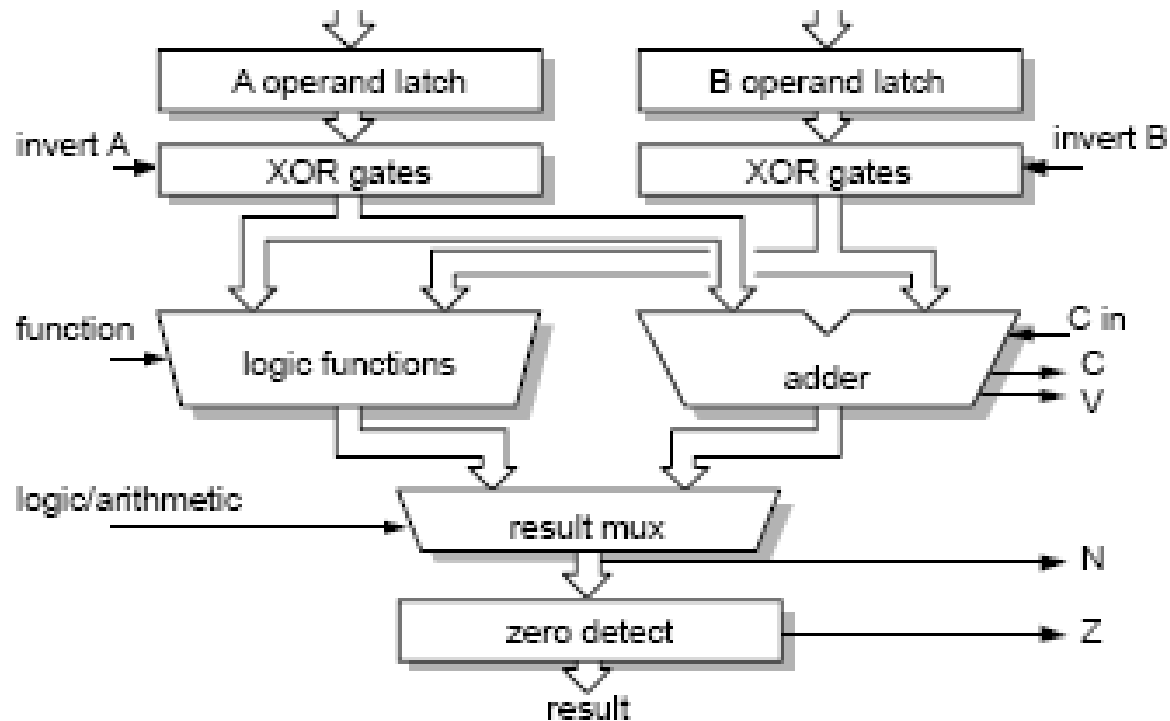
Also in Furber Fig. 4.12, the NB (not B) bus should be B bus and the ALU output is inverted (nALU).



fs5	fs4	fs3	fs2	fs1	fs0	ALU output
0	0	0	1	0	0	A and B
0	0	1	0	0	0	A and not B
0	0	1	0	0	1	A xor B
0	1	0	1	1	0	A plus not B plus carry
0	1	1	0	0	1	A plus B plus carry
1	1	1	0	0	1	not A plus B plus carry
0	0	0	0	0	0	A
0	0	0	0	0	1	A or B
0	0	0	1	0	1	B
0	0	1	0	1	0	not B
0	0	1	1	0	0	zero

ARM6 ALU organization

Separate addition and logic circuits:



XOR invert: ones complement

The XOR gates give the 'ones complement' of the input if selected.

Ones complement is found by switching every bit from logic 0 to logic 1 or from logic 1 to logic 0.

Also to change $+X$ to $-X$ subtract X from 2^N-1 , N is the number of bits in the number.

The MSB automatically becomes the sign bit.

A	B	A xor B	
0	0	0	B
0	1	1	
1	0	1	not B
1	1	0	

Twos complement

The 'twos complement' can be found by adding one to the ones complement.

Also to change $+X$ to $-X$ subtract X from 2^N where N is the number of bits in the number.

The MSB automatically becomes the sign bit.

This is a very common form of negative number representation and most CPUs have features to support its use (e.g. overflow flag).

Other arithmetic functions

Subtraction – using twos complement, an adder will perform subtraction. Or use ones complement and set 'carry-in' to logic 1.

Multiplication – complicated, consider 0110×1001

$$\begin{array}{r} 0110 \text{ Multiplicand} \\ 1001 \text{ Multiplier} \\ \hline 0110) \\ 0000) \text{ Partial products} \\ 0000) \\ 0110) \\ \hline 00110110 \text{ Total of partial products} \end{array}$$

Multiplication circuits

For a circuit the multiple additions in the columns would be difficult, instead add each partial product into a total as it is formed.

0 1 1 0	Multiplicand
<u>1 0 0 1</u>	Multiplier
0 1 1 0	First partial product
<u>0 0 0 0</u>	Second partial product
0 0 1 1 0	Partial sum
<u>0 0 0 0</u>	Third partial product
0 0 0 1 1 0	Partial sum
<u>0 1 1 0</u>	Fourth partial product
0 0 1 1 0 1 1 0	Total of partial products

Multiplication circuits

Now consider the general case for 4-bit numbers

A_3	A_2	A_1	A_0	Multiplicand
B_3	B_2	B_1	B_0	Multiplier
<hr/>				
$A_3 \cdot B_0$	$A_2 \cdot B_0$	$A_1 \cdot B_0$	$A_0 \cdot B_0$	First partial product
$A_3 \cdot B_1$	$A_2 \cdot B_1$	$A_1 \cdot B_1$	$A_0 \cdot B_1$	Second partial product
<hr/>				
PS	PS	PS	S_1	Partial sum
$A_3 \cdot B_2$	$A_2 \cdot B_2$	$A_1 \cdot B_2$	$A_0 \cdot B_2$	Third partial product
<hr/>				
PS	PS	PS	S_2	Partial sum
$A_3 \cdot B_3$	$A_2 \cdot B_3$	$A_1 \cdot B_3$	$A_0 \cdot B_3$	Fourth partial product
<hr/>				
S_7	S_6	S_5	S_4	Total of partial products

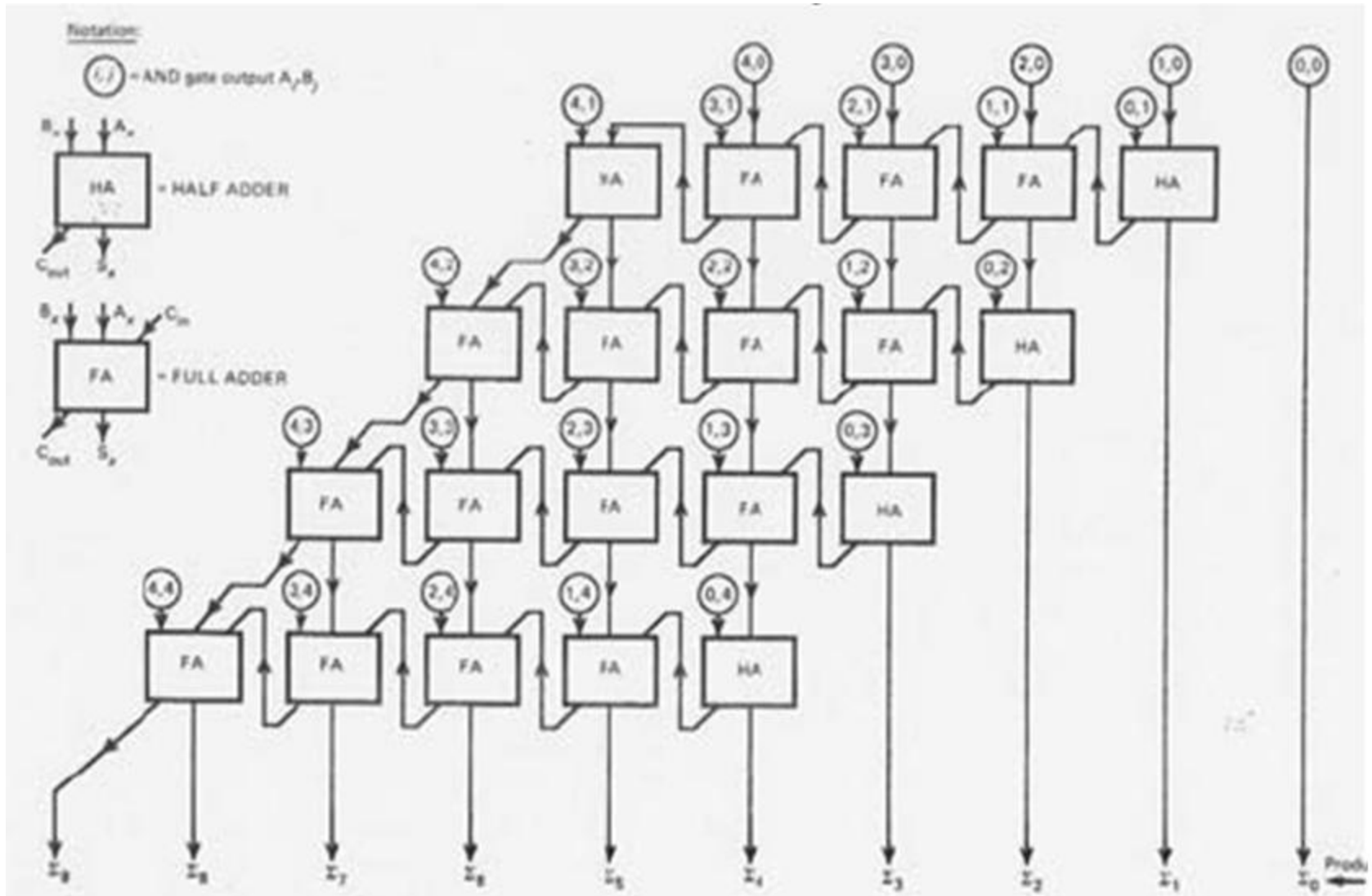
Multiplication circuits

Partial products are just AND functions (every possible AND of one digit of A and one digit of B).

Each partial sum can be generated by an adder – putting it all together we get a matrix multiplier.

A matrix multiplier for 32-bit numbers needs 32×32 AND gates and 31 (32 bit) adders.

Matrix Multiplier



Sequential approach

Sequential approaches to large combinational circuits

Because some arithmetic operations e.g. multiplication are complicated, circuits can break such operations into a sequence of steps

- An element of sequential behaviour is incorporated into a circuit that is essentially combinational.

Sequential implementation of multiplier

An alternative is a circuit that uses the adder and performs one step of the process at a time.

In simple form it requires double length registers and a shifter.

1. Set a total to zero.
2. Examine the LSB of the multiplier, if it is 1 add the multiplicand to the total otherwise do nothing.
3. Shift the multiplicand one place left moving in 0 as LSB.
4. Shift the multiplier one place right discarding the old LSB and moving in zero.
5. Repeat from 2 until all bits of multiplier tested (alternatively can end when multiplier is zero).

Multiplication circuits

There are numerous possible circuit implementations for realizing multiplication in an ALU.

E.g. a simple approach could use a combinational ‘matrix multiplier’;

for 32-bit numbers this needs 32×32 AND gates and 31 (32 bit) adders.

This would require a large number of logic gates and it would suffer from the problem of multiple propagation delays on the carry path.

Most multiplier circuits use sequential logic.

Simple Multiplication: example

$$200_{10} \times 100_{10} = 1100\ 1000_2 \times 0110\ 0100_2$$

In each step, one bit of the multiplier $0110\ 0100_2$ is selected.

If the bit is logic 1 the multiplicand, $1100\ 1000_2$ is shifted left and added to the running total.

LSL	Multiplier	ALU	Running total
0	$\times 0$	A+0	0000 0000 0000 0000
1	$\times 0$	A+0	0000 0000 0000 0000
2	$\times 1$	A+B	0000 0011 0010 0000
3	$\times 0$	A+0	0000 0011 0010 0000
4	$\times 0$	A+0	0000 0011 0010 0000
5	$\times 1$	A+B	0001 1100 0010 0000
6	$\times 1$	A+B	0100 1110 0010 0000
7	$\times 0$	A+0	0100 1110 0010 0000

Sequential implementation of multiplier

This has used **sequential** actions to perform an operation that is essentially **combinational**.

It uses mostly existing circuits, a shifter and adder, so does not add much to the gate count of the ALU.

However to perform a 32 bit multiplication would need 32 steps i.e. 32 clock cycles – too long.

An improved rule (algorithm) - Booth's algorithm

This is another possible sequence using an existing adder and a shifter.

Booth's multiplication algorithm

Initialization

1. Set a running total, T , to zero
2. Put the multiplier in a special register M
3. Set a carry bit, C_{in} , to 0
4. Set a cycle counter, N , to 0

THIS is the LOOP ENTRY point

5. Take the two LSBs of M , call this value B and add value of C_{in} to B
6. **Next action is one of**
 - i. if $C_{in} + B = 000_2$, do nothing and set $C_{out} = 0$.
 - ii. if $C_{in} + B = 001_2$, left shift the multiplicand $2N$ places & add this to running total T , set $C_{out} = 0$
 - iii. if $C_{in} + B = 010_2$, left shift the multiplicand $2N+1$ places & add this to running total T , set $C_{out} = 0$
 - iv. if $C_{in} + B = 011_2$, left shift the multiplicand $2N$ places & **subtract** this from running total T , set $C_{out} = 1$
 - v. if $C_{in} + B = 100_2$, do nothing and set $C_{out} = 1$.
7. Add 1 to N and copy C_{out} to C_{in} .
8. Right shift M two places (discarding bits shifted out)
9. If not dealt with all bits of multiplier go back to step 5.

Booth's algorithm: Nth cycle

Booth's algorithm works by replacing $\times 3$ by $\times (4-1)$ so that each cycle multiplies by a 2 bit value.

N bit multiplications can be completed in $N/2$ cycles.

The table right summarizes the action on the Nth cycle.

Table 4.3 in Furber also replaces $\times 2$ by $\times (4-2)$ for $C_{in}=0$ as this makes the control logic slightly simpler.

C_{in}	Multiplier	LSL #	ALU	C_{out}
0	$\times 00_2$	-	$A+0$	0
0	$\times 01_2$	$2N$	$A+B$	0
0	$\times 10_2$	$(2N+1)$	$A+B$	0
0	$\times 11_2$	$2N$	$A-B$	1
1	$\times 00_2$	$2N$	$A+B$	0
1	$\times 01_2$	$(2N+1)$	$A+B$	0
1	$\times 10_2$	$2N$	$A-B$	1
1	$\times 11_2$	-	$A+0$	1

xx ... xx 00 00 ... 00 00

N^{th} cycle

$2N$ 0s

$c_{\text{in}} = 0$

xx:

00 do nothing
 01 shift to left $2N$
 10 shift to left $2N + 1$
 11 = $100 - 01$

1 : handle it in next cycle
 00 : do nothing in this cycle
 -01 : shift to left $2N$ and subtract from running total

$c_{\text{in}} = 1$

xx:

00 \rightarrow 01 shift to left $2N$
 01 \rightarrow 10 shift to left $2N+1$
 10 \rightarrow 11 \rightarrow 100 - 01
 11 \rightarrow 100 handle it in next cycle

1 : handle it in next cycle
 00 : do nothing in this cycle

Booth's algorithm: example

$$200_{10} \times 100_{10} = \dots 0000\ 1100\ 1000_2 \times \dots 0000\ 0110\ 0100_2$$

The multiplier, $0110\ 0100_2$ is broken into 2 bit blocks and depending upon the value of the bits, one of four actions are implemented.

The multiplicand, $1100\ 1000_2$, is shifted and added onto the running total.

N	C _{in}	Multiplier	LSL	ALU	C _{out}	Running total
0	0	$\times 00_2$	-	A+0	0	0000 0000 0000 0000 0000 0000 0000 0000
1	0	$\times 01_2$	#2	A+B	0	0000 0000 0000 0000 0000 0011 0010 0000
2	0	$\times 10_2$	#5	A+B	0	0000 0000 0000 0000 0001 1100 0010 0000
3	0	$\times 01_2$	#6	A+B	0	0000 0000 0000 0000 0100 1110 0010 0000

Booth's algorithm: a further example

$$100_{10} \times 743_{10} = \dots 0000\ 0110\ 0100_2 \times \dots 0010\ 1110\ 0111_2$$

N	C _{in}	Multiplier	LSL	ALU	C _{out}	Running total
0	0	$\times 11_2$	#0	A-B	1	1111 1111 1111 1111 1111 1111 1001 1100
1	1	$\times 01_2$	#3	A+B	0	0000 0000 0000 0000 0000 0010 1011 1100
2	0	$\times 10_2$	#5	A+B	0	0000 0000 0000 0000 0000 1111 0011 1100
3	0	$\times 11_2$	#6	A-B	1	1111 1111 1111 1111 1111 0110 0011 1100
4	1	$\times 10_2$	#8	A-B	1	1111 1111 1111 1111 1001 0010 0011 1100
5	1	$\times 00_2$	#10	A+B	0	0000 0000 0000 0001 0010 0010 0011 1100

The multiplication can end when only zeros are left in the multiplier.

Booth's algorithm for negative numbers

Booth's multiplication algorithm also works with negative numbers in the two's complement format.

For instance multiplying a positive number, x , by $-y$. In 32 bit two's complement format $-y$ is represented by $(2^{32}-y)$ so that the product is $(x \cdot 2^{32} - x \cdot y)$. Taking the lowest 32 bits only, the result is $-x \cdot y$.

Likewise multiplying $-x$ by $-y$, the product is $(2^{32}-x) \cdot (2^{32}-y) = (2^{64} - (x+y) \cdot 2^{32} + x \cdot y)$ which is $x \cdot y$ when only the lowest 32 bits are considered.

The problem of 64 bit results will be considered later.

Booth's algorithm: negative number

$$200_{10} \times (-100_{10}) = \dots 00\ 1100\ 1000_2 \times (1111\dots 1111\ 1001\ 1100_2)$$

N	C _{in}	Multiplier	LSL	ALU	C _{out}	Running total
0	0	$\times 00_2$	-	A+0	0	0000 0000 0000 0000 0000 0000 0000 0000
1	0	$\times 11_2$	#2	A-B	1	1111 1111 1111 1111 1111 1100 1110 0000
2	1	$\times 01_2$	#5	A+B	0	0000 0000 0000 0000 0001 0101 1110 0000
3	0	$\times 10_2$	#7	A+B	0	0000 0000 0000 0000 0111 1001 1110 0000
4	0	$\times 11_2$	#8	A-B	1	1111 1111 1111 1111 1011 0001 1110 0000
5	1	$\times 11_2$	-	A+0	1	1111 1111 1111 1111 1011 0001 1110 0000
....	1	$\times 11_2$	-	A+0	1	1111 1111 1111 1111 1011 0001 1110 0000
15	1	$\times 11_2$	-	A+0	1	1111 1111 1111 1111 1011 0001 1110 0000

Booth's algorithm: 2 negative numbers

$$(-100) \times (-200) = (11\dots11\ 1001\ 1100_2) \times (11\dots11\ 0011\ 1000_2)$$

N	C _{in}	Multiplier	LSL	ALU	C _{out}	Running total
0	0	$\times 00_2$	-	A+0	0	0000 0000 0000 0000 0000 0000 0000 0000
1	0	$\times 10_2$	#3	A+B	0	1111 1111 1111 1111 1111 1100 1110 0000
2	0	$\times 11_2$	#4	A-B	1	0000 0000 0000 0000 0000 0011 0010 0000
3	1	$\times 00_2$	#6	A+B	0	1111 1111 1111 1111 1110 1010 0010 0000
4	0	$\times 11_2$	#8	A-B	1	0000 0000 0000 0000 0100 1110 0010 0000
5	1	$\times 11_2$	-	A+0	1	0000 0000 0000 0000 0100 1110 0010 0000
....	1	$\times 11_2$	-	A+0	1	0000 0000 0000 0000 0100 1110 0010 0000
15	1	$\times 11_2$	-	A+0	1	0000 0000 0000 0000 0100 1110 0010 0000

Further performance improvements

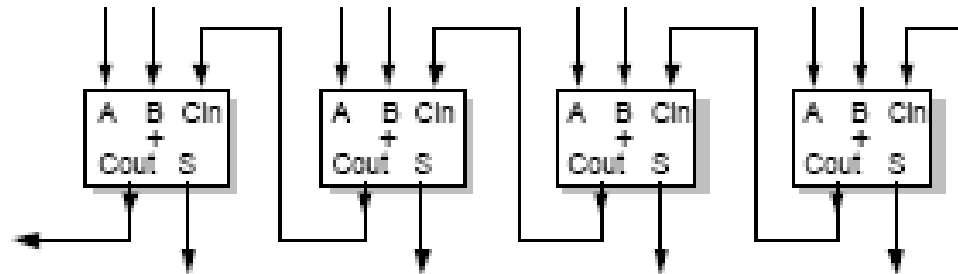
Using just the existing adder and barrel shifter a 32 bit multiplication can be completed in 16 cycles using Booth's algorithm.

However this is still not fast enough. The ARM uses a special adder circuit that adds five 32 bit numbers together in one combinational logic circuit.

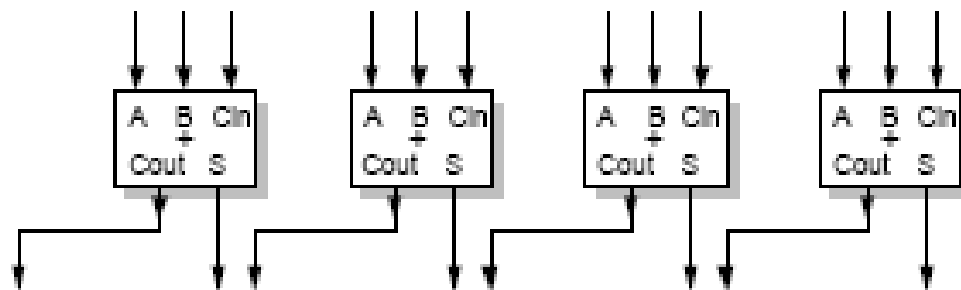
This adder is similar to a matrix multiplier circuit but differs by using a 'carry save' scheme.

carry propagate / carry save

Carry propagate is identical to a ripple carry adder.

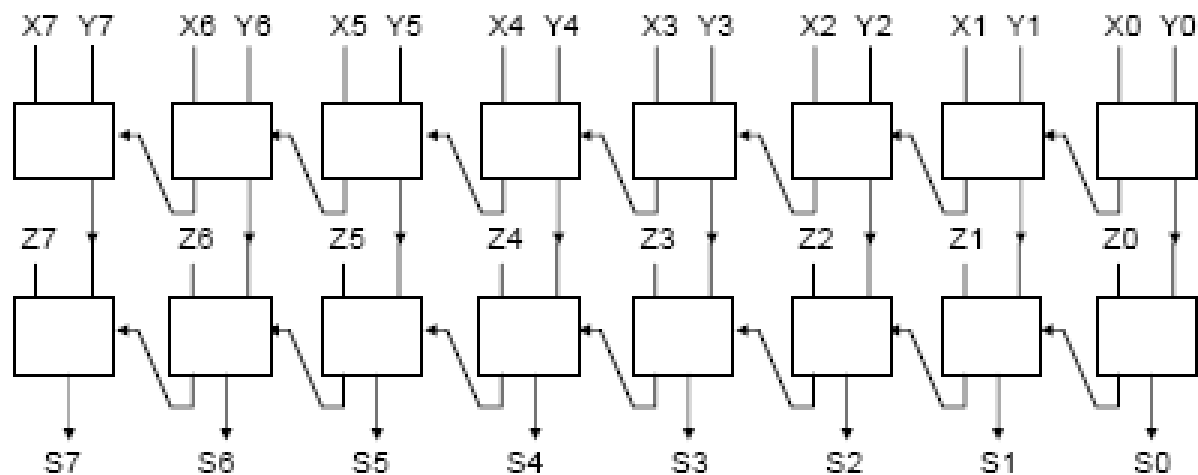


Carry save passes the carry values onto the next adder stage and has two outputs, a partial sum and a partial carry.



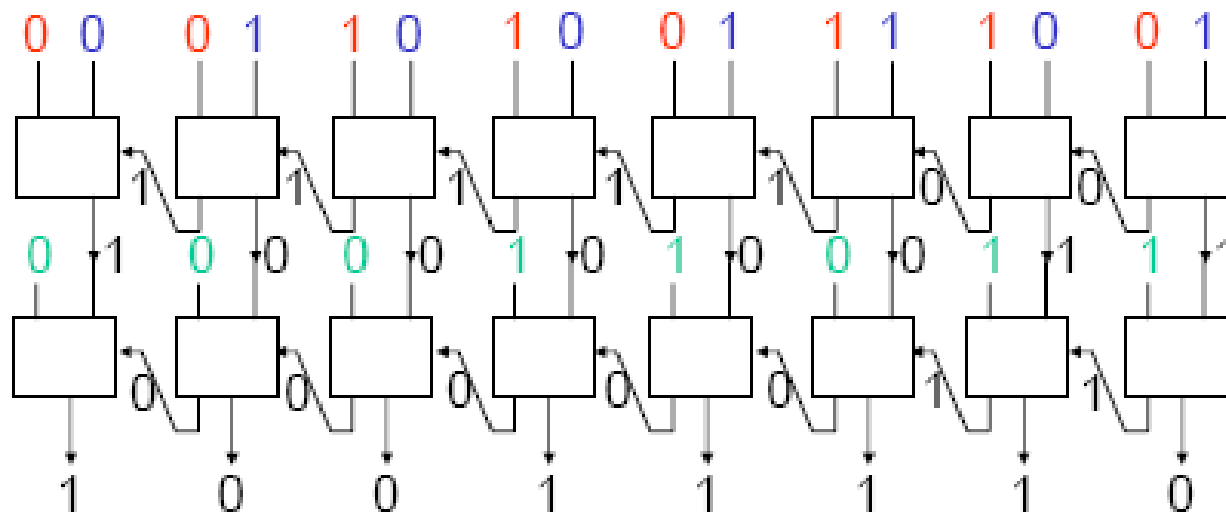
2×8 bit 'carry propagate' adder

The standard matrix multiplier uses the 'carry propagate' structure shown below. Each cell is a one bit full adder with carry in and carry out. The output S is $(X + Y + Z)$.



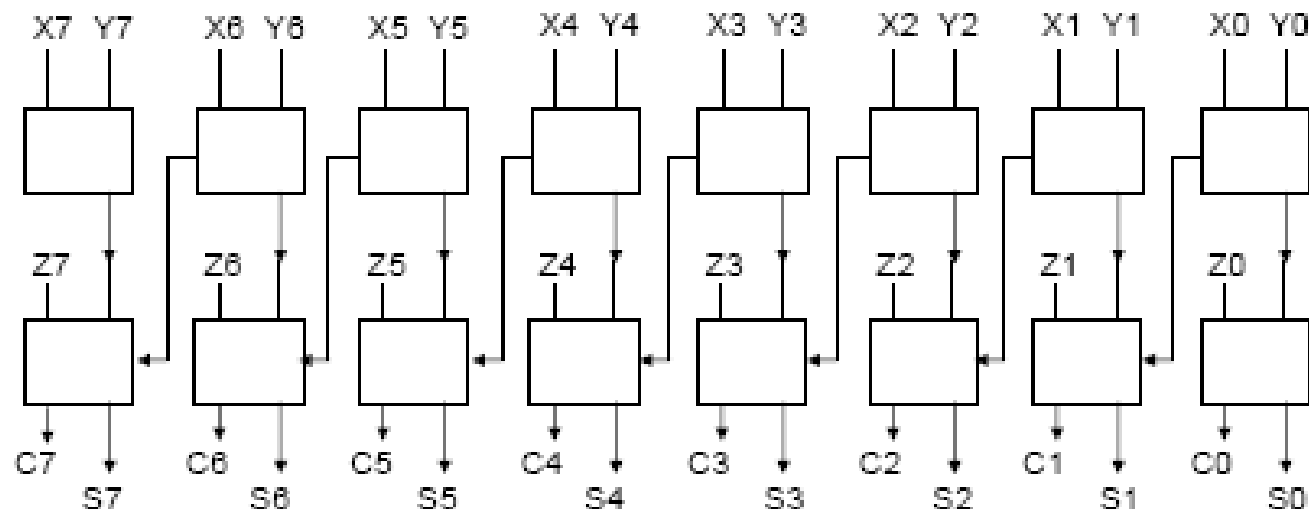
'carry propagate' adder: example

In the following example $X = 00110110$, $Y = 01001101$ and $Z = 00011011$. This structure suffers the same problem as the ripple carry adder; multiple propagation delays.



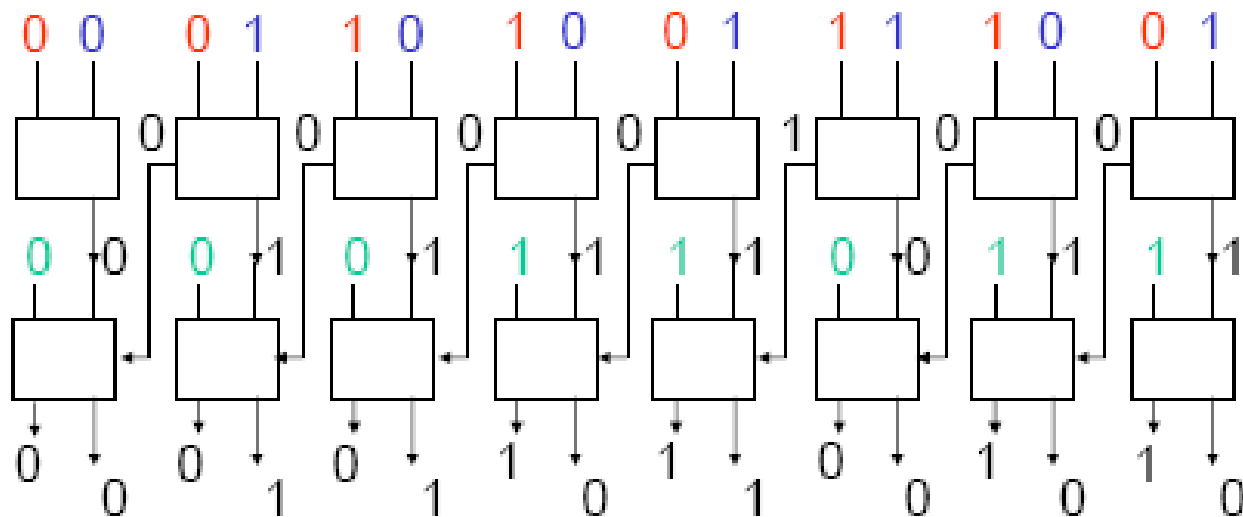
2×8 bit 'carry save' adder

The 'carry save' structure shown below. The sum $(X+Y+Z)$ can be found by adding the partial sum S to the partial carry C . The total propagation delay is greatly reduced.



'carry save' adder: example

If $X = 00110110$, $Y = 01001101$ and $Z = 00011011$ then the partial sum $S = 01101000$ & the partial carry $C = 00011011$. C is left shifted once and added to S ; $(S + 2C) = 10011110$.



ARM high performance multiplier

The ARM high performance multiplier uses a 4×32 bit carry save adder to add six numbers, a running total (partial sum and partial carry) and four values produced using Booth's algorithm.

The maximum propagation delay is a carry path of 4 gates; short enough to be completed in one clock cycle.

Each pass through the adder gives a multiplication of an 8 bit number with a 32 bit number.

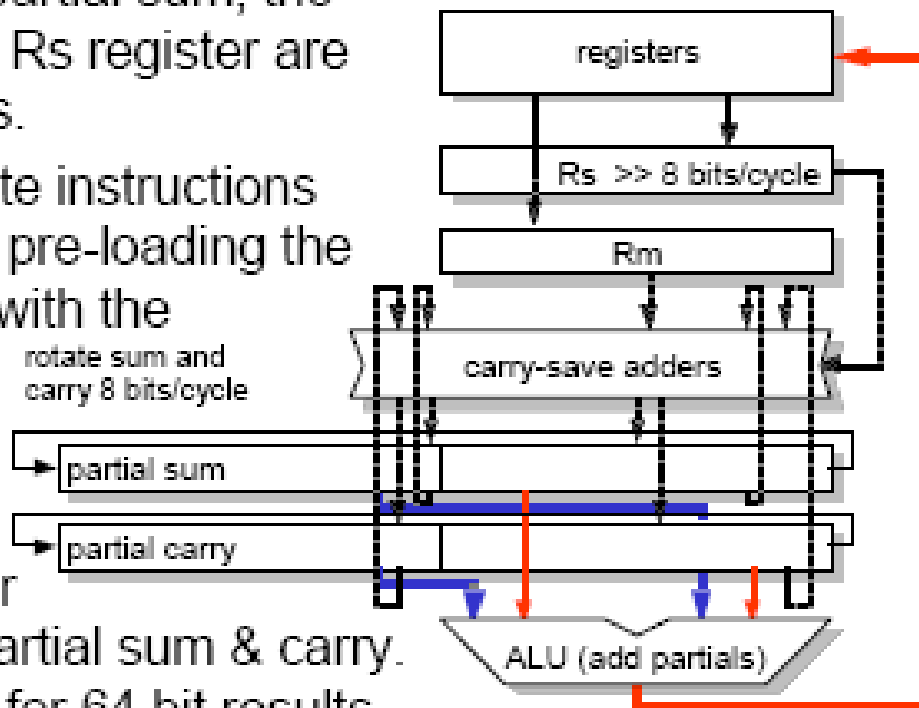
After 4 passes through the carry save adder, the standard ALU adder is used in a 5th cycle to add the partial sum to the partial carry giving a 32 bit by 32 bit product.

ARM high-speed multiplier

On each cycle, the partial sum, the partial carry and the Rs register are right shifted by 8 bits.

Multiply & accumulate instructions are implemented by pre-loading the partial sum register with the accumulate value.

On the last cycle (shown in red) the standard ARM adder is used to add the partial sum & carry. Path in blue is used for 64 bit results.



Double length products

The product of two numbers; one with n bits and the other with m bits, will have $(n + m - 1)$ or $(n + m)$ bits.

E.g. $11 \times 101 = 1111$, $11 \times 110 = 10010$

So if $(n + m)$ is greater than 32 the product will be too big for a single register.

Many microprocessors, including the ARM, have two multiplication instructions, one for a single register result and a 'long' multiplication which uses two registers for the result.

For long multiplication, the format of the numbers is important i.e. a two's complement product is different from the product of unsigned integers.

Signed and unsigned products

For example, take 0xFFFFFFFF9C multiplied by 0xFFFFFFFF38.

If these are unsigned integers then this is $4,294,967,196_{10} \times 4,294,967,096_{10}$ - the long product is 0xFFFFFED400004E20

But as two's complement numbers, this is $(-100) \times (-200)$ and the long product is 0x0000000000004E2 = 20,000.

The ARM microprocessor supports both forms of long multiplication with two different instructions; UMULL for unsigned integers and SMULL using 2's complement.

ARM multiplier - summary

The ARM multiplier uses Booth's algorithm with a dedicated carry save adder to complete a 32 bit multiplication in 5 cycles.

It can complete execution in less than 5 cycles if multiplier has leading zeros.

The multiplication circuit is large i.e. about 10% of the transistors in the ARM7 core but a simpler circuit would take many more clock cycles – a classic design compromise / trade off.