CHAPTER 4

Data path

The *data path* is the nucleus of a computational system where calculations are executed (general block diagram in figure 4.1). In the most typical cases it is composed of arithmetic blocks (additions, subtraction, multiplication and sometimes division, square root, inversion, power) and combinational and logic blocks (logicals, shifters, comparators). Intermediate results are normally saved in registers, and sometimes counters are exploited in execution. In many cases special block of registers, e.g. register files, are used for this purpose.

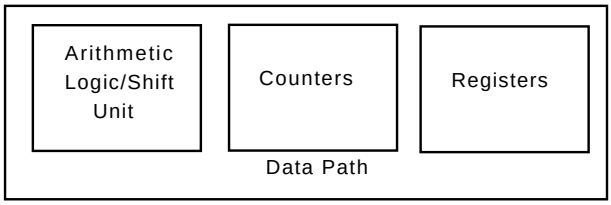


Figure 4.1: General Data Path components.

In the following (section 4.1) a few data path blocks present in three real processors will be examined. Picking some examples from these datapaths, specific cases of comparators, logical blocks and shifters (**??**), of arithmetical structures (section 4.2), and of registers and counters (sections 4.7 and 4.8) will be inspected.

The analysis is not exhaustive and aims at giving a few real-life cases. When necessary, elementary structures are recalled in order to describe the more complex one.

4.1 Data paths examples in real processors

4.1.1 Pentium 4 65nm Integer Unit

The 64-bit integer unit block diagram of the 65-nm Intel Pentium 4 processor is in figure 4.2. It uses a double frequency fast clock to enable single-cycle latency on the critical arithmetic and logic unit (ALU) bypass loops. Only a subset of the integer unit functionality operates at 2 frequency: the ALU adder input multiplexer (mux), ALU sparse tree adder, the Address generation Unit (AGU), the ALU shifter/rotator, and the Integer Register File (IRF). Longer latency ALU operations, such as integer multiply, are implemented outside the integer unit and operate at processor frequency. A monolithic 64-bit data path implementation is avoided since it will increase latencies of 32-bit ALU operations, Level 1 data cache loads, and store to load forwarding. For this reason, the 64-bit wide data path is implemented as two discrete 32-bit data paths. The latency between the lower and upper 32-bit data paths is 1/2 processor clock.

The write-back busses provide source operand data for the ALUs and AGU from the register file, data cache, ALUs, and the FSFB. The write-back busses that are outputs of the ALUs also write back

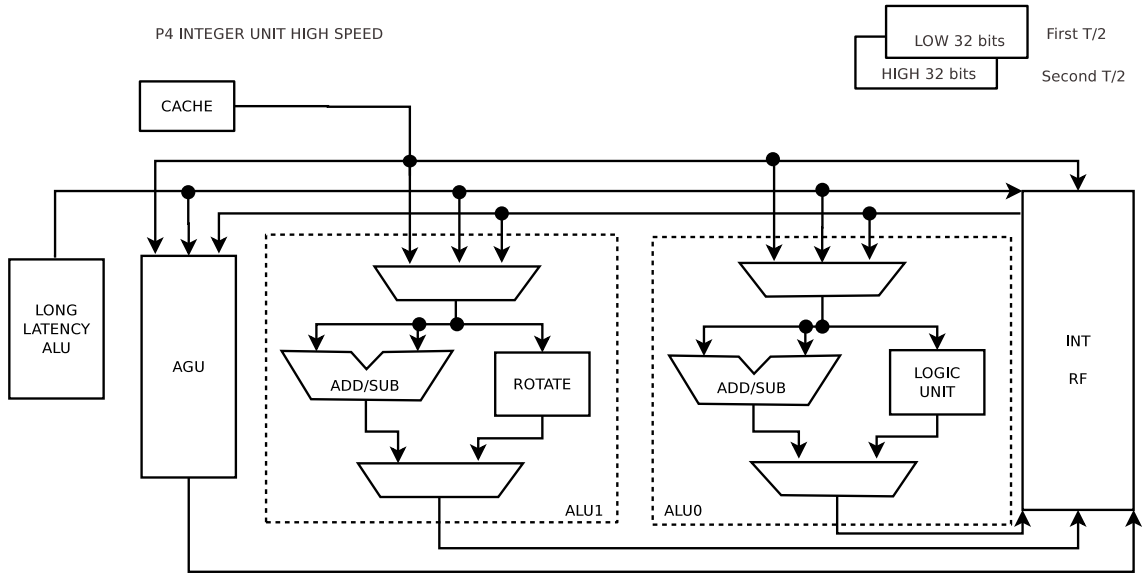


Figure 4.2: Pentium 4 Integer Unit block diagram

the results to the register file. These write-back busses are both latency and speed critical. A majority of these write-back busses operate at double frequency in order to match the double throughput of the AGUs, ALU, and integer register file. The double frequency integer register file can sustain a throughput of 12 reads and 6 writes per processor clock. The 2double frequency AGU can compute 1 load and 1 store address, interleaved, per processor clock. The 32-bit double frequency ALU loop latency is 1 processor clock cycle.

The integer unit contains two 64-bit ALUs (ALU0 and ALU1), each implemented using two discrete 32-bit data-path blocks, with one-FCLK (frequency fast clock) latency for communication between the upper and lower 32-bit units. In addition to ADD/SUB instructions, ALU0 and ALU1 execute logic and rotate operations respectively. Implemented in a 65nm technology it operates at 9GHz.

A detailed explanation of the sparse adder is reported in section 4.4.

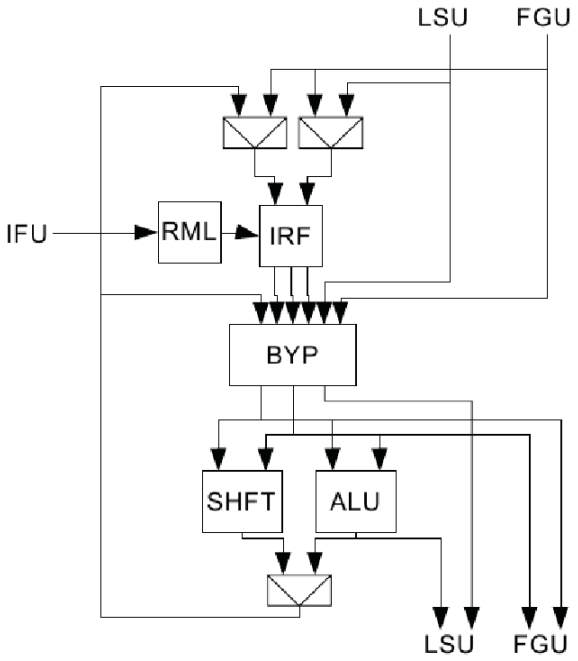


Figure 4.3: Niagara T2 Integer Unit block diagram

4.1.2 T2 EXU

The Execution Unit (EXU) of Sun Microsystem Niagara T2 block diagram is shown in figure 4.3. It executes all integer arithmetic and logical operations except for integer multiplies and divides. The EXU calculates memory and branch addresses. The EXU handles all integer source operand bypassing.

The EXU is composed of the following subunits: operand by-pass (BYP); arithmetic logic unit (ALU, analyzed in details in section 4.2.1); shifter (SHFT, discussed in section **??**); integer register file having two input ports and three output ports rs1, rs2 and rs3 (IRF, detailed in section4.8.3); register management logic (RML); result multiplexers and flip-flops (RM_FF).

4.1.3 T2 Floating Point and Graphical Unit (FGU)

OpenSPARC T2 contains one dedicated FGU per core. The OpenSPARC T2 floating-point and graphics unit (FGU, a generig block diagram in figure 4.5) implements the SPARC V9 floating-point instruction set, the SPARC V9 integer multiply and divide.

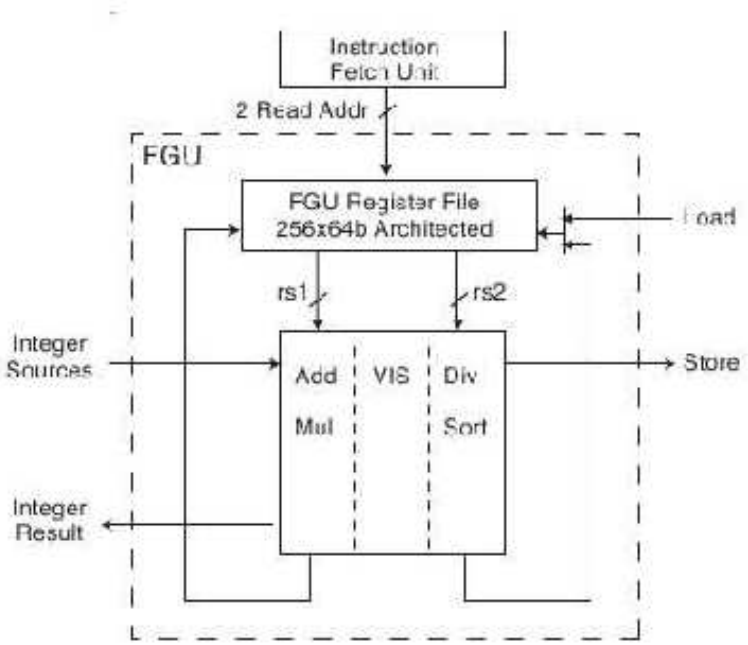


Figure 4.4: Niagara T2 FGU blocks scheme

This unit supports has support for IEEE 754 single precision (SP) and double precision (DP) data formats. All quad precision floating-point operations are unimplemented. FGU includes three execution pipelines:

- Floating-point execution pipeline (FPX);
- Graphics execution pipeline (FGX);
- Floating-point divide and square root pipeline (FPD).

The detailed diagram is in figure **??**. Up to one instruction per cycle can be issued to the FGU. Instructions for a given thread are executed in-order. A discussion of the multiplier implementation is in section **??**.

4.1.4 A few words on ARM VFP11 coprocessor

The VFP11 coprocessor is an implementation of the ARM Vector Floating-point Architecture (VFPv2). It provides low-cost floating-point computation that is fully compliant with the ANSI/IEEE Std 754-

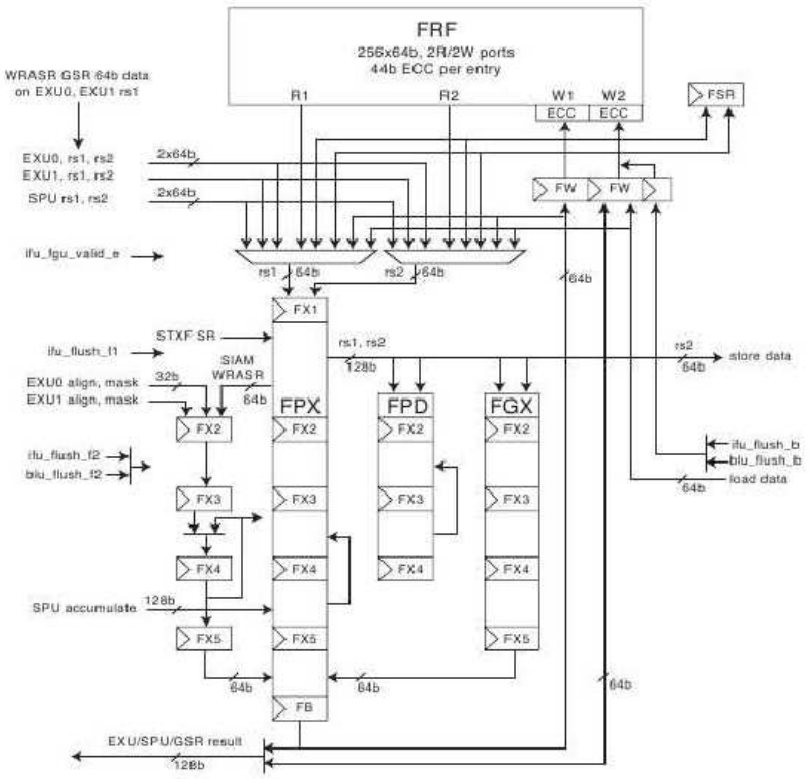


Figure 4.5: Niagara T2 FGU blocks scheme

1985, IEEE Standard for Binary Floating-Point Arithmetic, referred to in this document as the IEEE 754 standard.

The VFP11 coprocessor has three separate instruction pipelines:

- the Multiply and Accumulate (FMAC) pipeline
- the Divide and Square root (DS) pipeline (a discussion provided in section **??**)
- the Load/Store (LS) pipeline

Each pipeline can operate independently of the other pipelines and in parallel with them. Each of the three pipelines shares the first two pipeline stages, Decode and Issue. These two stages and the first cycle of the Execute stage of each pipeline remain in lockstep with the ARM11 pipeline stage but effectively one cycle behind the ARM11 pipeline. When the ARM11 processor is in the Issue stage for a particular VFP instruction, the VFP11 coprocessor is in the Decode stage for the same instruction. This lockstep mechanism maintains in-order issue of instructions between the ARM11 processor and the VFP11 coprocessor. The three pipelines can operate in parallel, enabling more than one instruction to be completed per cycle. Instructions issued to the FMAC pipeline can complete out of order with respect to operations in the LS and DS pipelines. This out-of-order completion might be visible to the user when a short vector FMAC or DS operation generates an exception, and an LS operation begins before the exception is detected. The destination registers or memory of the LS operation reflect the completion of a transfer. The destination registers of the exceptional FMAC or DS operation retain the values they had before the operation started.

Except for divide and square root operations, the pipelines support single-cycle throughput for all single-precision operations and most double-precision operations. Double-precision multiply and multiply and accumulate operations have a two-cycle throughput. The LS pipeline is capable of supplying two single-precision operands or one double-precision operand per cycle, balancing the data transfer capability with the operand requirements.

4.2 Arithmetic units

Arithmetic units in general must be able to execute most of the operations shown in figure 4.6; specific commands configure the unit for the operation to be executed.

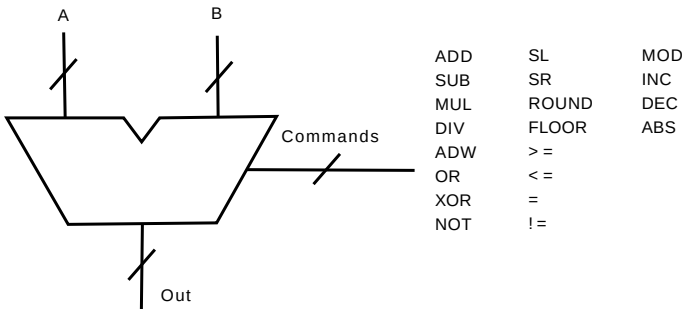


Figure 4.6: Generic alu.

In the following we are going to analyze how arithmetic units and thir subblocks are organized using some examples from the processor mentioned above.

4.2.1 T2 EXU-ALU

An example of a real ALU, the T2 ALU, is shown in figure 4.7.

Adder and subtractor are realized with the same component, which provides the sum between two 64-bits operands with carry in bit. The carry out bit is also available as output signal. In the subtractor the second operand is the output of an inverter, whose input is the rs2 data, and the carry in is set to 1 (this realizes the 2-complement of rs2, in order to compute subtractions with the same component that computes sums, see a specific example in section 4.4). The adder is implemented as a Carry Look Ahead (possible implementations mentioned in section 4.4).

The logical block is described in section 4.3.1.

In order to determine if the outcome of sum and subtraction is zero, zero detectors are included in the ALU block. They are described in section 4.3.4.

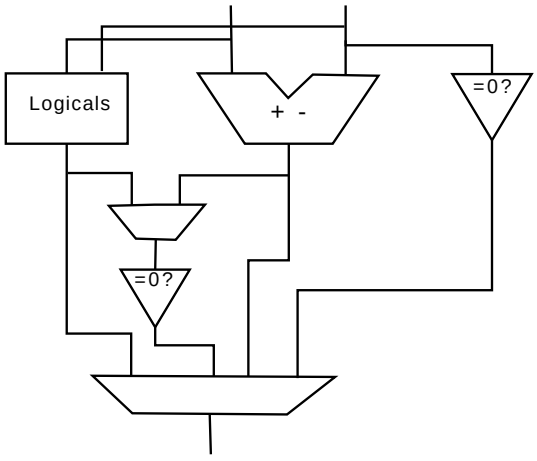


Figure 4.7: Niagara T2 ALU included in the Integer Unit.

4.3 Logicals, Shifters, Comparators

4.3.1 Logicals

The OpenSPARC T2 instruction set contains the following logical operations between two operands:

- bitwise AND, NAND
- bitwise OR, NOR
- bitwise XOR, XNOR

The logic unit that computes this operations is implemented with two NAND gates (*nand*) level: the first one has four 3-inputs *nands*, each input is 64-bits wide; the second level has a 4-inputs *nand*, whose inputs are the outputs of the first level gates. The block scheme is shown in figure 4.8.

Each first level gate has a select input (extended to 64 bits), called *selectN*, with N=0,1,2,3. The other two inputs are the R1 and R2 operands, or their negative value. In order to compute one of the logical instructions, the select signals are properly activated as follow:

- AND: *select0*=0, *select1*=0, *select2*=0, *select3*=1;
- NAND: *select0*=1, *select1*=1, *select2*=1, *select3*=0;
- OR: *select0*=0, *select1*=1, *select2*=1, *select3*=1;
- NOR: *select0*=1, *select1*=0, *select2*=0, *select3*=0;
- XOR: *select0*=0, *select1*=1, *select2*=1, *select3*=0;
- NXOR: *select0*=1, *select1*=0, *select2*=0, *select3*=1;

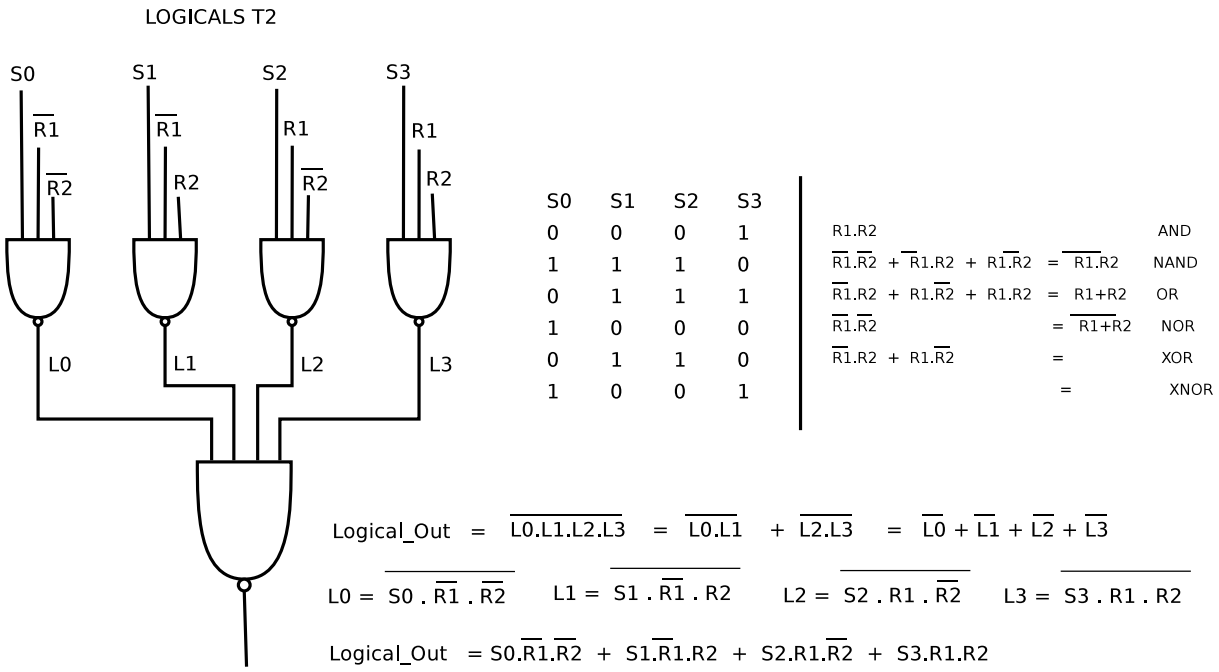


Figure 4.8: Logicals T2.

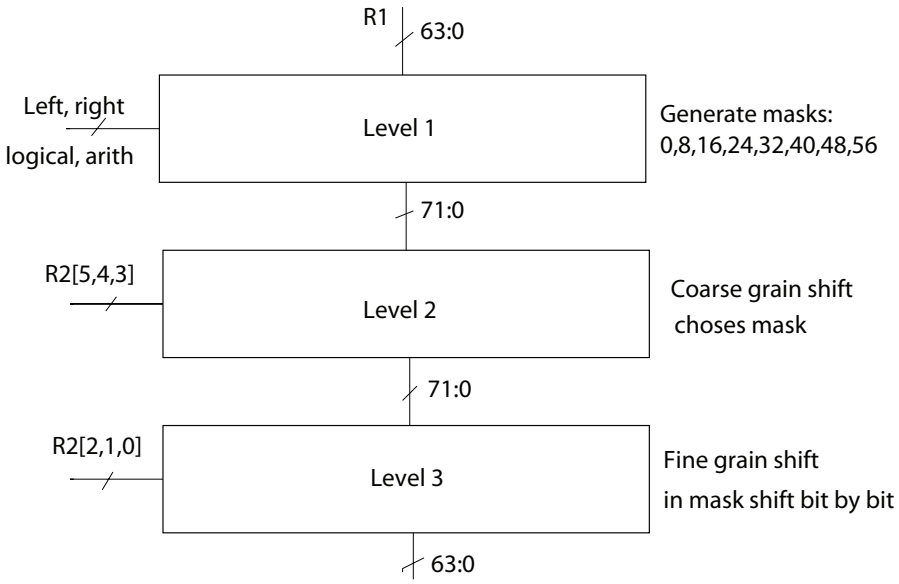


Figure 4.9: T2 shifter block diagram.

4.3.2 Shifters

The example we analyze is the **T2 shifter**. The T2 instruction set includes the following shift operations:

- *SLL* and *SLLX* (shift left logical) shift all 64 bits of the value in R[rs1] left by the number of bits specified by the shift count, replacing the vacated positions with zeroes, and write the shifted result to R[rd].
- *SRL* (shift right logical) shifts the low 32 bits of the value in R[rs1] right by the number of bits specified by the shift count. Zeroes are shifted into bit 31. The upper 32 bits are set to zero, and the result is written to R[rd].

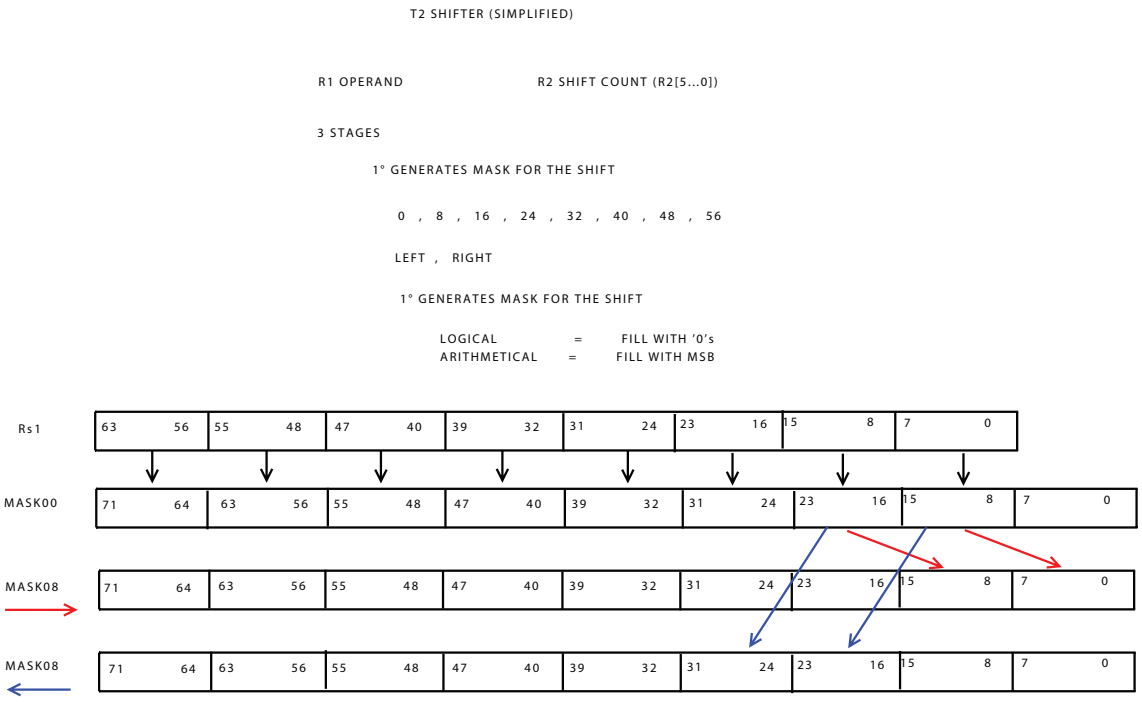


Figure 4.10: T2 shifter.

- *SRA* (shift right arithmetic) shifts the low 32 bits of the value in R[rs1] right by the number of bits specified by the shift count and replaces the vacated positions with bit 31 of R[rs1]. The high-order 32 bits of the result are all set with bit 31 of R[rs1], and the result is written to R[rd].
- *SRLX* (shift right logical extended) shifts all 64 bits of the value in R[rs1] right by the number of bits specified by the shift count. Zeroes are shifted into the vacated high-order bit positions, and the shifted result is written to R[rd].
- *SRAX* (shift right arithmetic extended) shifts all 64 bits of the value in R[rs1] right by the number of bits specified by the shift count and replaces the vacated positions with bit 63 of R[rs1]. The shifted result is written to R[rd].

The operand to be shifted is R1, while R2[5,4,3,2,1,0] is used for defining the shift amount. Other inputs define if the shift should be right, left, arithmetical (fill with MSB) or logical (fill with 0). The general shifter block diagram is in figure 4.9.

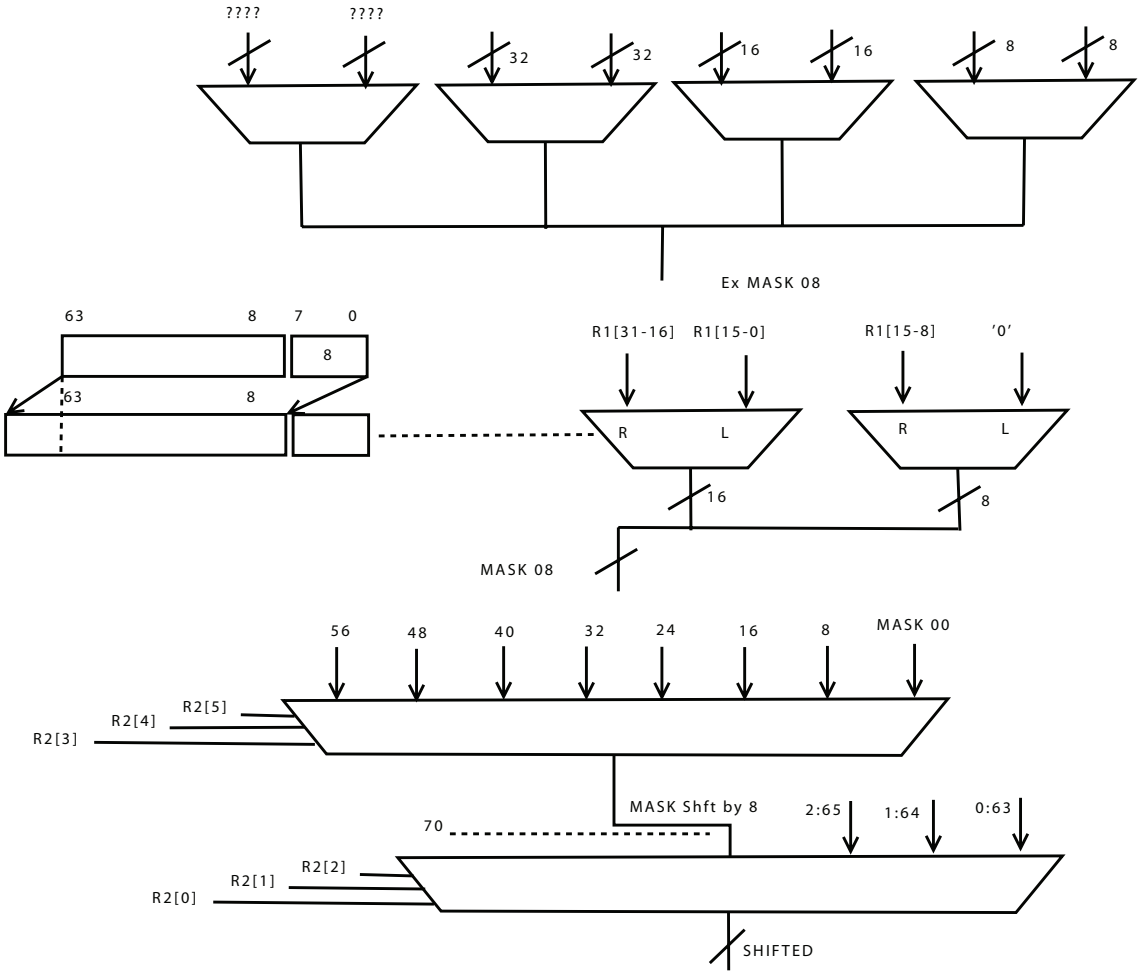


Figure 4.11: T2 shifter first stage.

It is organized in three levels. The first consist in preparing 8 possible "masks", each already shifted of 0,8, 16,....56, left or right depending on the configuration. The second level performs a coarse grain shift, that is it chooses among the 8 masks the nearest to the shift to be operated. For example, in case a shift by 4 is needed, then mask 0 is chosen, while if a shift by 11 is needed then

mask 8 is selected. The choice is operated using bit 5,4 and 3 of operand R2. Finally the third level receives such a mask and implements the real shift according to bits 2,1 and 0 of operand R2.

Figures 4.11 and 4.10 show the scheme of the first level multiplexers, for four example cases, and their concerning masks of possible shift types for these cases.

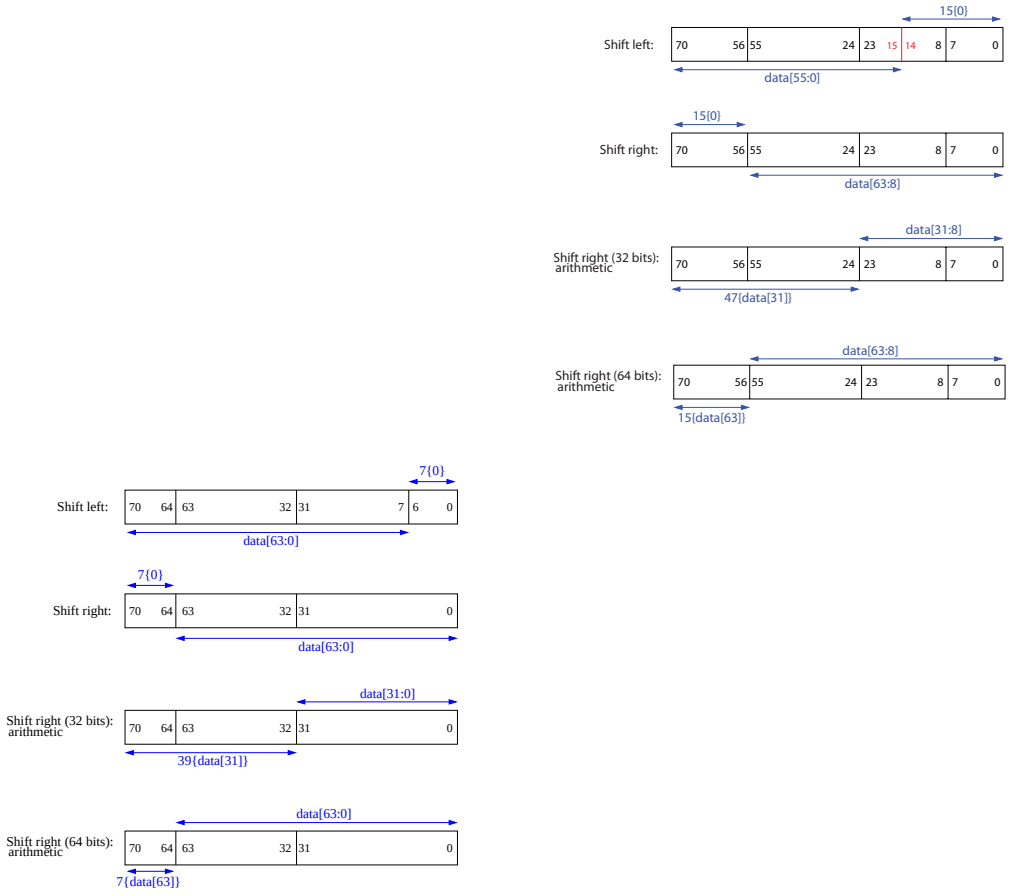


Figure 4.12: *Mask_mux00[70:0]* (left) and *Mask_mux08[70:0]* (right)

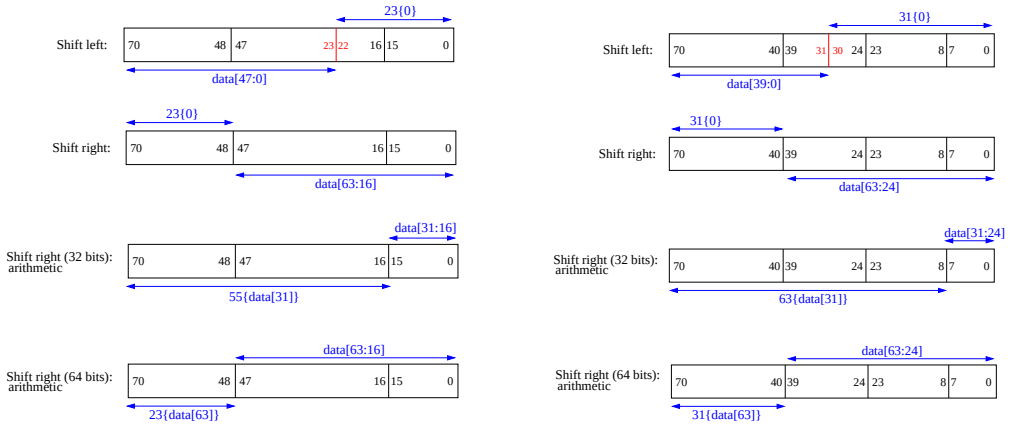


Figure 4.13: *Mask_mux16[70:0]* (left) and *Mask_mux24[70:0]* (right)

The selection signals for the multiplexers in the first level are generated according to the specific shift type, as shown in the following table:

	SLL	SLX	SRAX	SRA	SRLX	SRL
sel_lshift	1	1	0	0	0	0
sel_rshift	0	0	1	1	1	1
sel_rax	0	0	1	0	0	0
sel_ra	0	0	0	1	0	0
sel_rshiftx	0	1	0	0	0	0

The second level multiplexer chooses one of the eight mask signals (*mask_muxXX[70:0]*, with XX=00,08,..,56 indicates the shift count); the third level performs the intermediate shift count (from 1 to 7) on the selected mask signal.

The selection signals for these multiplexers are generated from the first 6 bits of the RS2 operand (*exu_rs2_data[5:0]*): *exu_rs2_data[5:3]* is the selection signal for the second level; *exu_rs2_data[2:0]* is for the third level in case of a right shift, while for left shift this signal is negated to generate the correct select.

The second and third level multiplexers scheme is shown in figure 4.14.

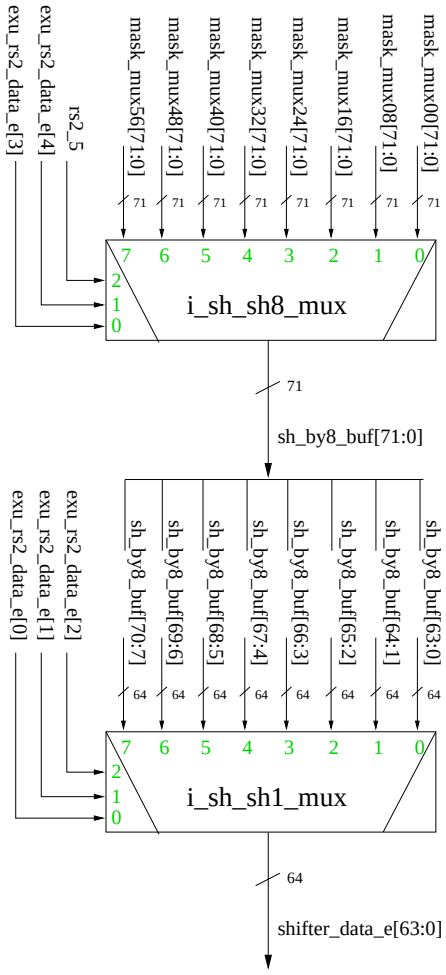


Figure 4.14: Shifter - Second and third level multiplexers

The outcome of shift operation is computed in two steps. For example, if shift left by 37 (100101 in a 6-bits binary representation) positions has to be performed: in the second level, *exu_rs2_data[5:3]* is equal to 4 (decimal representation) and chooses the mask related to the 32 positions left shift (*mask_mux32[70:0]*); the third level multiplexer computes the left shift by 5 position on *mask_mux32[70:0]* signal, in this case *exu_rs2_data[2:0]* is equal to 101, so its negated value (010, equal to decimal 2) selects *mask_mux32[65:2]*.

4.3.3 Comparators

Equality Comparator

Comparator based on binary numbers' Sum/Subtraction And an equality comparator decides whether A = B. You can easily do it with XNOR and 1's detector. The schematic of a Equality Comparator is shown in Figure 4.15.

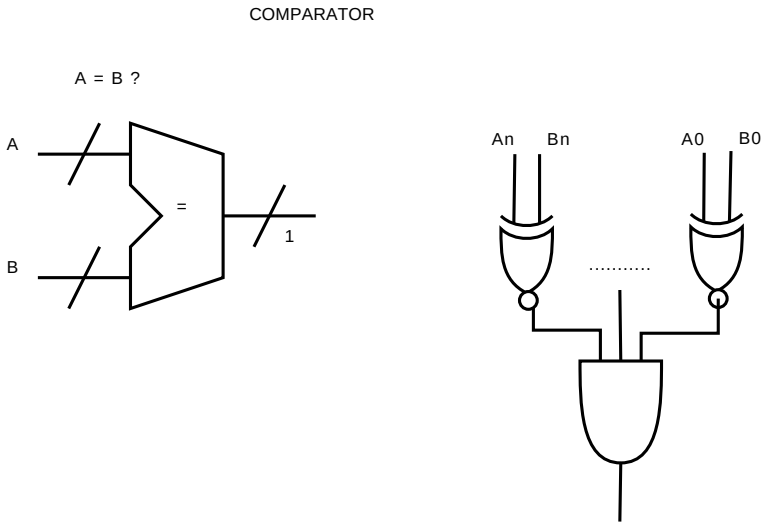


Figure 4.15: Schematic of a simple Equality Comparator.

In case not only equality is to be found, then, instead of using a specific hardware block, the idea is exploiting a structure based on the ALU.

For example:

- A = 7, B = 4. In this case $A > B$. The difference is A - B = 3;

$$\begin{array}{rrl} A & 0111 & + \\ \overline{B} & 1011 & \\ \hline A+\overline{B} & 10010 & \\ & 1 & + \\ \hline & 10011 & \end{array}$$

  Apart from the subtraction value (3) we notice that the carry out is '1'.

- A = 7, B = 8. In this case $A < B$. The difference is A - B = -1;

$$\begin{array}{rrl} A & 0111 & + \\ \overline{B} & 0111 & \\ \hline A+\overline{B} & 01110 & \\ & 1 & + \\ \hline & 01111 & \end{array}$$

  The carry out is '0' and all the other bits a '1'.

  And, besides:

- A = 7, B = 7. In this case $A = B$. The difference is A - B = 0.

$$\begin{array}{rrl} A & 0111 & + \\ \overline{B} & 1000 & \\ \hline A+\overline{B} & 1111 & \\ & 1 & + \\ \hline & 10000 & \end{array}$$

All the bits of sum are zero.

Thus, if $A \geq B \Rightarrow C_{out} = 1$; in case all others bits are 0 then $A = B$ If $A<B \Rightarrow C_{out} = 0$.(Only in the example all other bits are '1', but this is not true in general).

Exploiting an adder/subtractor and adding minimum hardware we can obtain a comparator. The circuit diagram in Figure4.16.

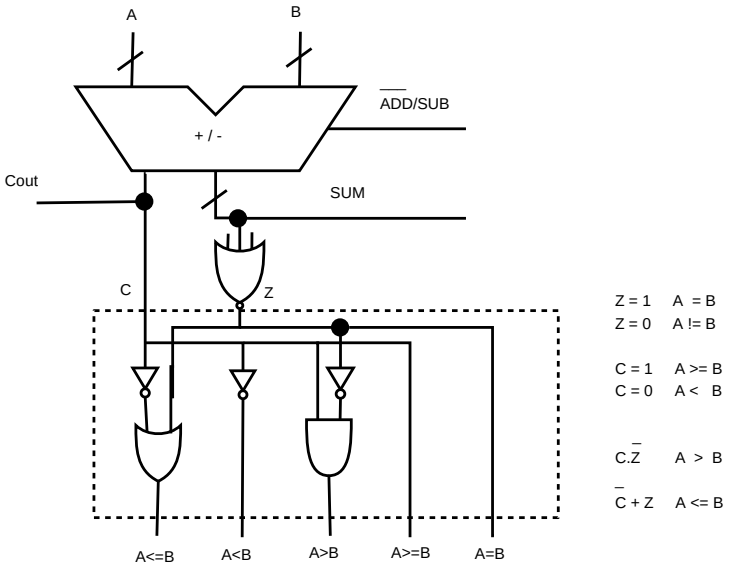


Figure 4.16: Schematic of Comparator and Corresponding Results.

The following table shows what will happen in different relations between A and B, it lists what the carry-in C_{i-1} should be for this to be true and what the carry-out C_i will be for each bit position i.

$$\begin{array}{lcll} \text{A} > \text{B} & \Rightarrow & C & and \quad \overline{Z} \\ \text{A} \geq B & \Rightarrow & C & \\ \text{A} < B & \Rightarrow & \overline{C} & \\ \text{A} \leq B & \Rightarrow & \overline{C} & or \quad Z \\ A = B & \Rightarrow & Z & \\ A \neq B & \Rightarrow & \overline{Z} & \end{array}$$

A+B=K Comparator

Sometimes,we need to compare the results of the sum of two addends (e.g. A,B) with a third entrance (e.g. K). If this calculation must be done frequently, then it is not good to perform the sum and compare the result with K, because this method implies the delay of propagation of carry. In order to understand how to get a dedicated circuit, we should know that C_i is a bit that should be the carry of array i-1, notice that data A_i and B_i could obtain K_i. If the required carry can be predicted, compare it with the actually generated carry, checking all of their bits whether equals to each other. The following table contains the data inputs A_i, B_i and K_i, which produce the expected carry and the generated carry out.

The schematic of A+B=K Comparator is shown in Figure4.17.

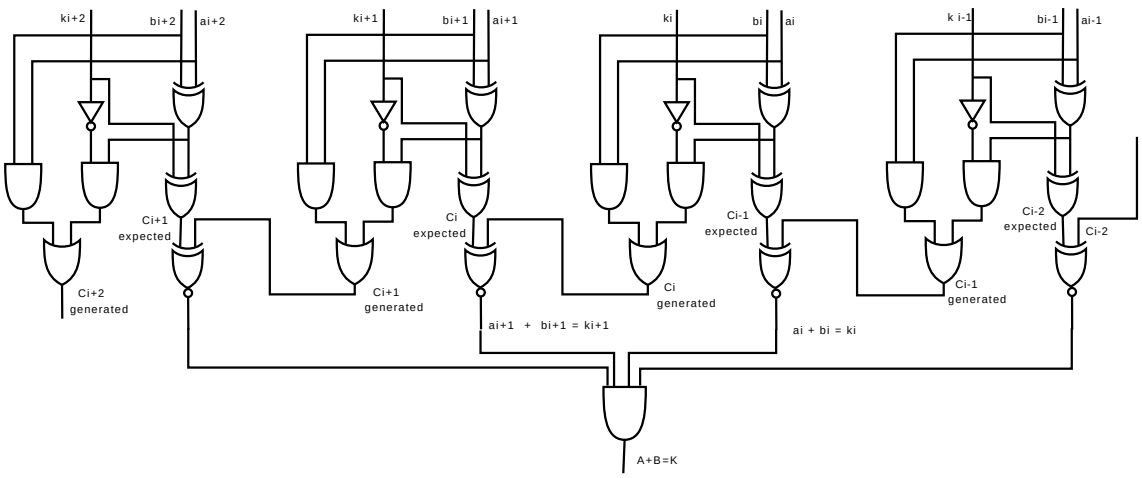


Figure 4.17: Schematic of A+B=K Comparator.

Ai	Bi	Ki	Ci-1 Required	Ci Generated
0	0	0	0	0
0	0	1	1	0
0	1	0	1	1
0	1	1	0	0
1	0	0	1	1
1	0	1	0	0
1	1	0	0	1
1	1	1	1	1

The truth table brings C_{i-1} REQUIRED and C_i GENERATED depending on the three inputs:

C_{i-1} REQUIRED $= A_i \oplus B_i \oplus K_i$ and C_i GENERATED $= (A_i \oplus B_i)\overline{K_i} + A_i \cdot B_i$

The result is the circuit shown in figure 4.17, where each C_{i-1} REQUIRED is compared with its corresponding GENERATED carry. The carry is not propagated to all of the required and generated, which are generated in parallel. The critical path is due to the depth of the logic of generation (the worst difference between generated and required) and comparison (XNOR and final AND).

For example:

A=1001,B=0100,K=1111

So,by using the truth table,we can easily got each C_{i-1} REQUIRED and C_i GENERATED.

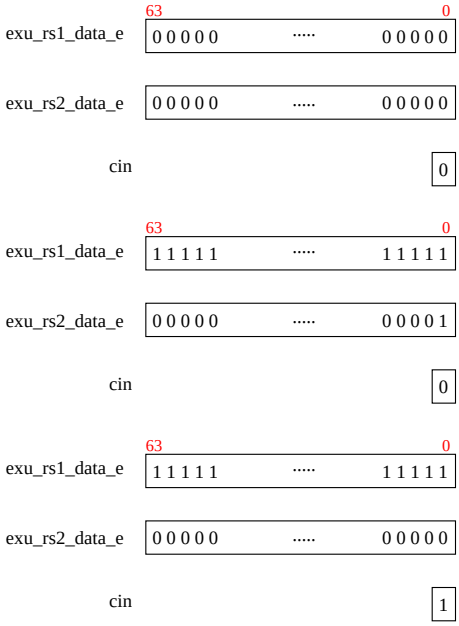
Ai	Bi	Ki	Ci-1 Required	Ci Generated
0	0	1	1	0
0	1	1	0	0
1	0	1	0	0

Each time,after we got C_{i-1} REQUIRED and C_i GENERATED, we do the XNOR between them and send all results to the final AND gate to get the final comparator result.Here, we got the result after each XNOR is 1101, then after AND,the answer will be 0(FALSE), means $A + B \neq K$.

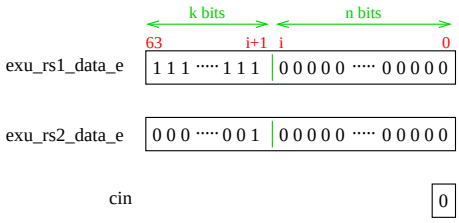
4.3.4 Zero detector

The example of the T2 ALU zero detector is here described. A zero is detected if:

- both the operands are 0 (extended to 64 bits) and carry in bit too;
- one operand has 64 bits set to 1 and the other one or the carry in bit is the equivalent to 1 in decimal base (borderline case in which the adder/subtractor outcome is 0 and the carry out 1, overflow);



- one operand has the k MSBs set to 1 and the other n LSBs to 0 (with k and n arbitrary values), the other operand has the LSBs set to 0 and the k MSBs set to the binary equivalent of a decimal base 1, the carry in set to 0 (borderline case in which the adder/subtractor outcome is 0 and the carry out 1, overflow).



The detector evaluates also if the 32-bits outcome is 0, in the same ways for 64-bits evaluation cases.

The block diagram of the zero detector is shown in figure 4.18.

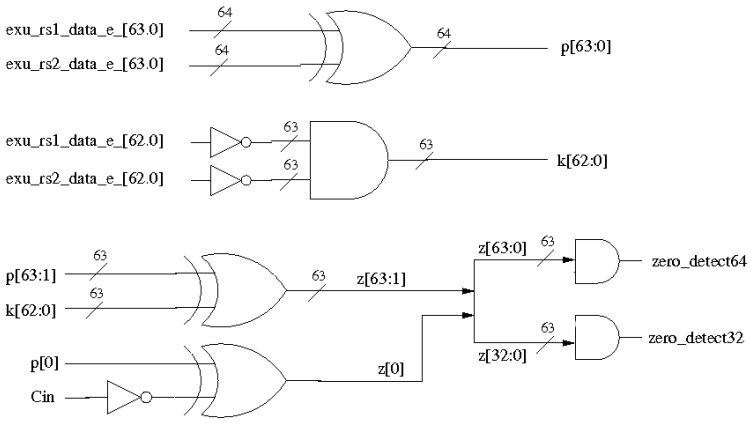


Figure 4.18: Adder/subtractor - Zero detector

4.4 Adders/Subtractor

In this section we want to focus on the adder used in the Pentium4. the general structure is as in figure 4.19. The subblocks are a carry select for the sum generation and a sparse tree for the carry generation. As the explanation is not straightforward, a step by step recall of the basic structures is necessary.

So first in the following we start recalling the simplest blocks, and then be go nearer and nearer to the P4-adder.

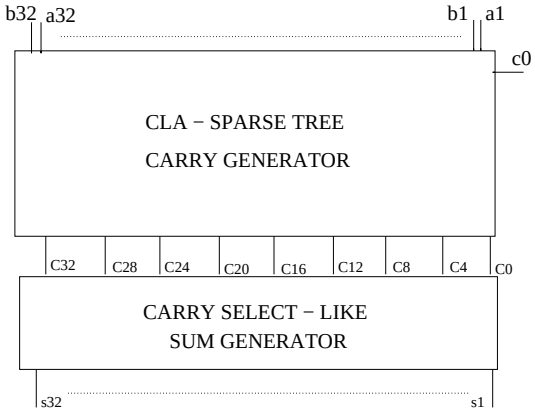


Figure 4.19: P4 adder general block diagram

4.4.1 Ripple Carry adder

The easiest and most straightforward implementation is the ripple carry adder, which, for two n-bit operands, requires n full adders (FA). Given inputs A_i, B_i and C_i, the sum S and the output carry C_o have as boolean functions:

$$S = A_i \oplus B_i \oplus C_i \qquad C_o = A_i \cdot B_i + A_i \cdot C_i + B_i \cdot C_i$$

The FA are cascaded so that the carry out of the $i-$th FA is connected to the carry in of the $(i+1)-$th FA. A 4 bits example is in figure 4.20.

From the performance point of view, the worst case occurs when a carry propagates from the Least Significant Bit (MSB) to the Most Significant Bit (LSB), thus the critical path delay is:

$$t_{carry-MSB} = (N-1) \cdot t_{carry} + t_{carry0}$$

where t_{carry} is the delay of a single FA in generating the carry out from the moment in which the three inputs are available. The inefficiency of this structure is then clear if an addition between two operands with a high number of bits is to be executed.

The same structure can be used as a subtractor slightly changing a part of the diagram thanks to an XOR gate. The XOR truth's table is

x	y	x⊕y
0	0	0
0	1	1
1	0	1
1	1	0

That is: if $x = 0$ then $x \oplus y = y$, if $x = 1$ then $x \oplus y = \overline{y}$. This property can be exploited for generating a two's complement subtraction, by connecting the FA array as in figure 4.21.

All the XOR gates are connected to input C_{IN} which assumes the significance of a $\overline{SUM}/SUB$:

$$\overline{SUM}/SUB = 0 \qquad S = A + B + 0$$

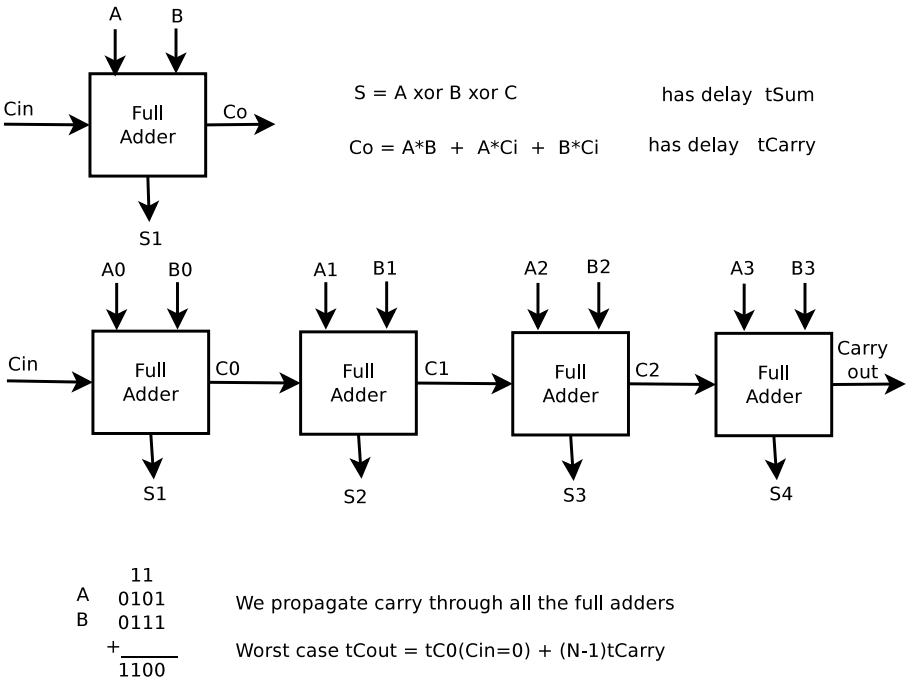


Figure 4.20: Ripple Carry adder.

that is a normal sum is executed

$$\overline{SUM}/SUB = 1 \qquad S = A + \overline{B} + 1$$

that is a two's complement subtraction is executed. This principle can be used whenever a subtraction is necessary, even if the adder implementation is different from a RCA one.

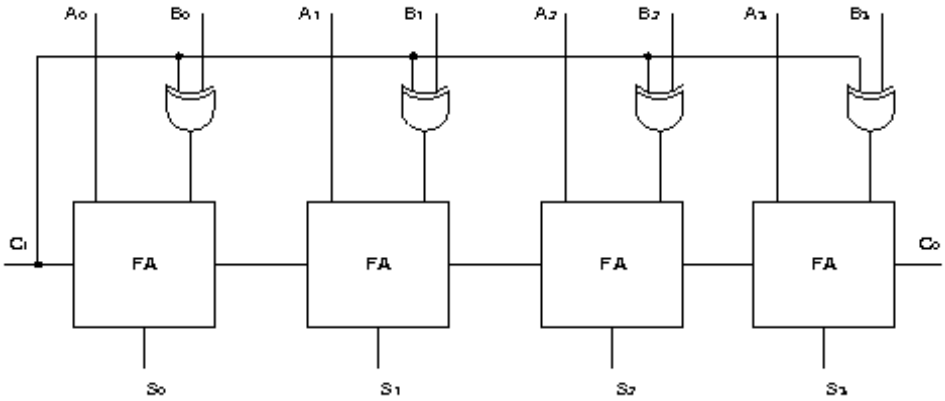


Figure 4.21: Subtractor based on RCA.

4.4.2 Carry Select adder

The underlying strategy of the carry select adder is that of using two identical adders, as sketched in figure 4.22, each generating a set of sum bits and an outgoing carry. One adder assumes that the incoming carry into the group is 0, while the other assumes that it is 1. Thus the two adders generate in parallel the results. When the incoming carry is assigned, its final value is selected out by means of two multiplexers, one for the sum and the other for the carry.

The advantage in using this structure comes when the m-bit adder is divided in k groups of n-bit carry select adders as for example in figure 4.23 where m=12, k=3 and n=4. The size of n is chosen

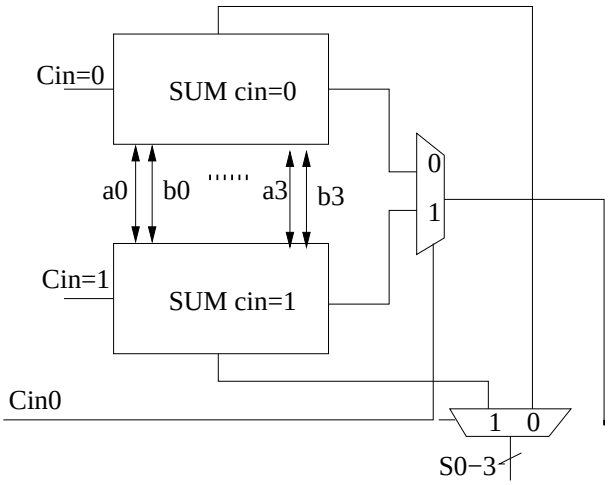


Figure 4.22: Carry Select adder

so as to equalize the delay of the carry to be generated in the n-bit adder and the delay of the carry select chain from adder 1 to adder k. So the delay is:

$$delay_{CP} = MAX\{n \cdot t_{carry}, k \cdot t_{mux}\}$$

In the P4 case the carry select is a simplified version, as the carry is generated directly from the top network (see later). For this reason the structure is like in figure 4.24.

OPTIONAL: Carry Skip adder

This structure is alternative to the carry select. Analisys is optional. Is is important though to read it for understanding he definition of propagate and generate terms.
The aim of the Carry Skip, as in the case of the carry select, is to reduce the worst case delay by reducing the number of FA through which the carry has to propagate. The adder again is divided into k groups if n bits, and the carry of a group $j+1$ is determined by one of the following conditions:

- the carry is propagated by group j, that is the carry-out of that group is equal to the carry-in of that group
- the carry is not propagated by the group, that is it is generated or killed inside the group

Consequently, to reduce the length of the propagation network of the carry a skip network is provided for each group of n bit so that when a carry is propagated by this group the skip network makes the carry bypass the group. Let's define the two terms:

$$g_i = a_i \cdot b_i \quad \text{generate} \tag{4.1}$$

$$p_i = a_i + b_i \quad \text{propagate} \tag{4.2}$$

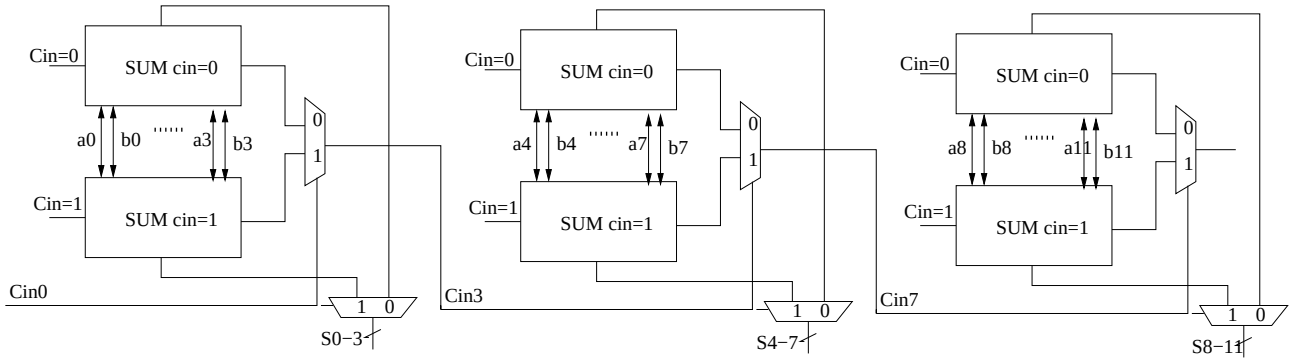


Figure 4.23: 16 bit Carry Select adder

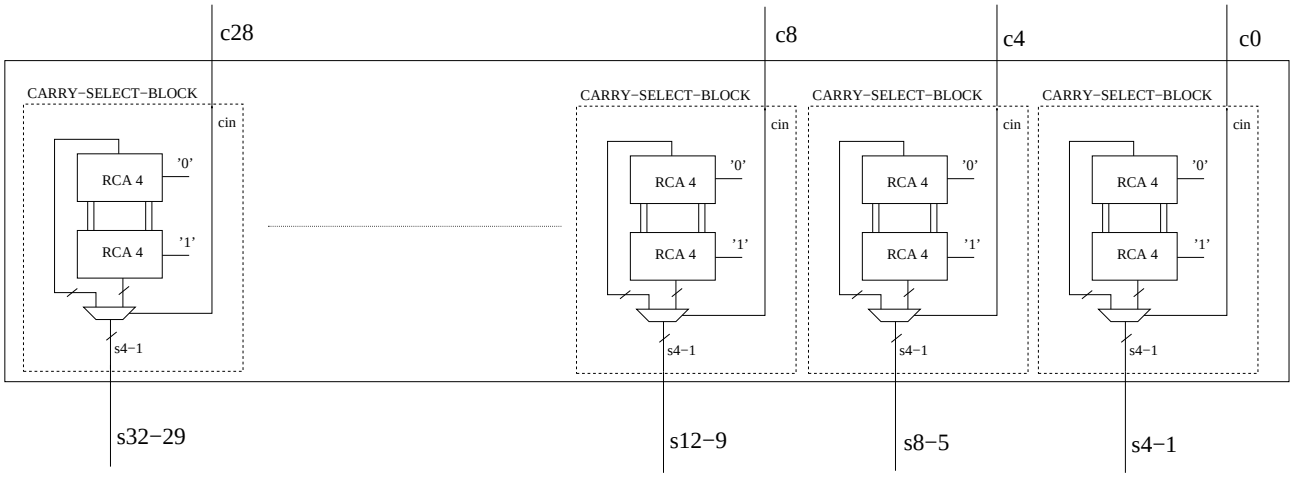


Figure 4.24: P4 Simplified Carry Select adder

Now a carry can be written as a function of propagate and generate terms as follows:

$$C_{i+1} = a_i \cdot b_i + a_i \cdot c_i + b_i \cdot c_i = g_i + p_i \cdot c_i$$

That is, if $g_i = 1$ then the carry is generated at step i, while if $p_i = 1$ then input carry is propagated. Iteratively:

$$C_{i+1} = g_i + p_i \cdot (g_{i-1} + p_{i-1} \cdot c_{i-2}) = g_i + p_i \cdot g_{i-1} + p_i \cdot p_{i-1} \cdot (g_{i-2} + p_{i-2} \cdot c_{i-3})) = ... \quad (4.3)$$

Thus, in a 4 bit RCA block, if p_0 and p_1 and p_2 and p_3 have as a result 1 then the carry in to the block (c_{IN}) is propagated. The structure can be then realized as in figure 4.25, where the and among all the propagate terms is called Carry Bypass (CB), which is in and with the carry in from previous block. The result is in OR with the generated carry. As the propagate terms can be generated in

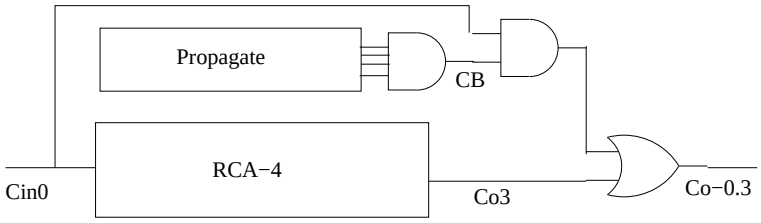


Figure 4.25: Carry Skip (OR).

parallel when the inputs are ready, then the propagation delay of this secondary chain is smaller than the chain through the whole n-bit RCA, as it involves one propagate gate (OR), one n-bit AND, and a 2-bit AND.

Again thus the structure is efficient when organized in k groups as in figure 4.26, where m=16, k=4 and n=4. the critical path corresponds in this case to the situation in which the carry is generated inside the first group, and is then propagated through the other two groups, and that the last block should determine the sum at the MSB.

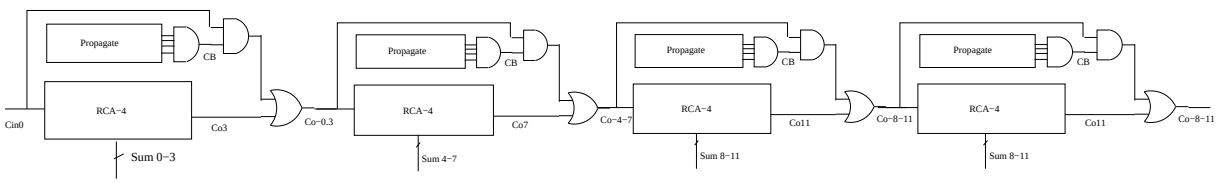


Figure 4.26: 16 bit Carry Skip (OR).

The carry skip can be found also in the configuration of figure 4.27, where the principle is the same, but the propagate term is defined in a more restrictive way:

$$p_i = a_i \oplus b_i$$

.

Looking at the following table this definition becomes more obvious:

a	b	c_i	sum	c_o	
0	0	0	0	0	generated
0	1	0	1	0	propagated
1	0	0	1	0	propagated
1	1	0	0	1	generated
0	0	1	1	0	generated
0	1	1	0	1	propagated
1	0	1	0	1	propagated
1	1	1	1	1	generated

The CB signal is built with the same philosophy as before, but is used as a control signal of a multiplexer which determines if the output carry should be the input carry or the generated one.

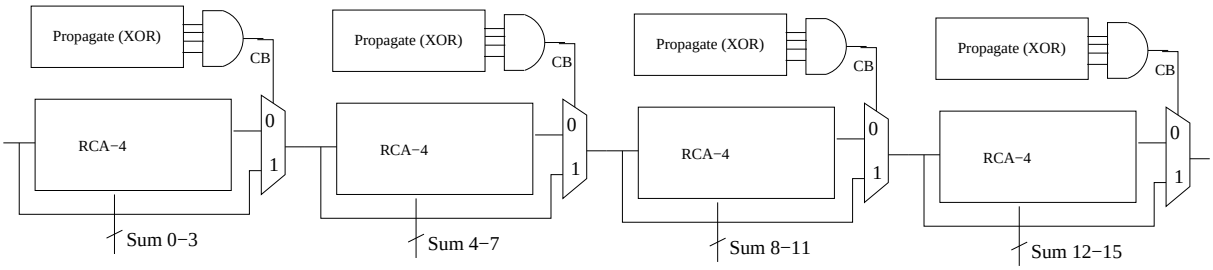


Figure 4.27: 16 bit Carry Skip (XOR).

4.4.3 Carry Look Ahead adder

The Carry Look Ahead (CLA) adder is the basis for every Tree adder: the P4 sparse tree adder is a particula version of tree adder.
The idea behind the CLA is to compute several carries simultaneously and to avoid waiting until the correct carry propagates from the stage of the adder in which it has been generated. It is thus based on the *generate* and *propagate* (XOR) terms defined before (carry skip) and depending on how the carry terms are obtained in equation (4.3) different structures are available with different performance.

In case the carry in is $c_0 = c_{IN}$, and the least significant bit of the operand has index 1 (a_1, b_1) then we can write

$$c_2 = g_2 + p_2 \cdot (g_1 + p_1 \cdot c_0) = g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot c_0 \tag{4.4}$$

Furthermore, the sum is

$$s_1 = a_1 \oplus b_1 \oplus c_0 = p_1 \oplus c_0$$

The structure so defined is reported in figure 4.28 for a few bits, where the generate and propagate gates are hidden (and and xor respectively), and where by convention $c_0 = c_{IN} = g_0$.

Three blocks are then defined: the PG-logic, the SUM-logic and the CARRY-logic, which is the critical one. In this implementation the CARRY logic is “flat” and each carry term is originated by the carry in. Clearly the complexity grows with the number of bit, but the carry network generates the carry as much in parallel as possible. Still there is a delay, for sure lower than the ripple carry adder, but for a big number of bit the complexity grows and the fan out for each propagate and generate terms increases, thus requiring ad hoc sizing to compensate the delay increment due to higher fan out.

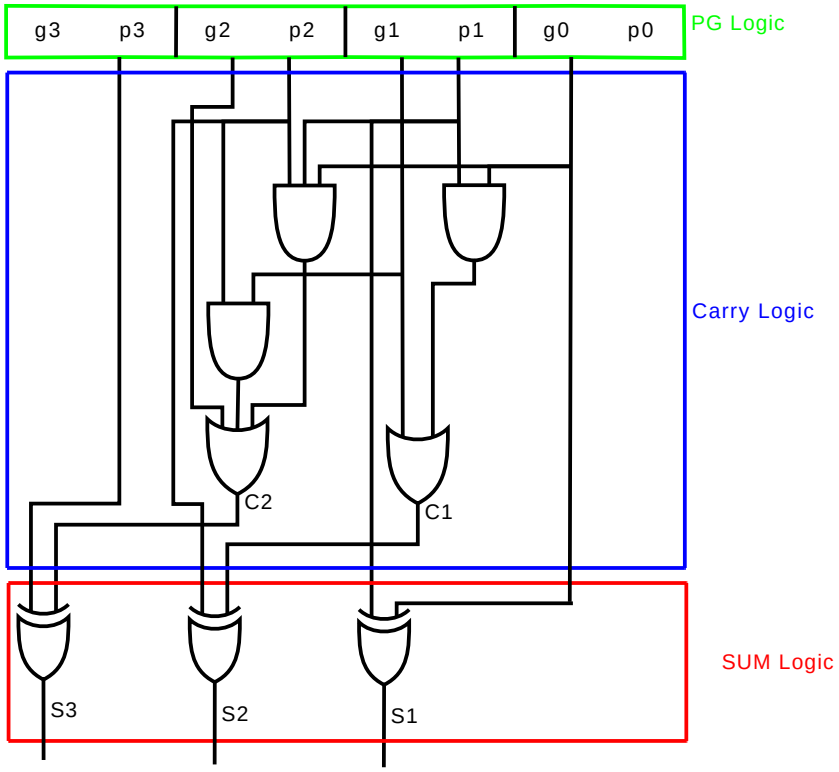


Figure 4.28: Flat Carry Look Ahead adder.

A different approach is followed in the structure shown in figure 4.29, where the dependency of c_3 on c_2 and the of c_2 on c_1 (and so on) is exploited. This is thus a ripple carry look ahead adder, in which carries are rippled, even if partially generate thanks to a PG-logic. The structure is extremely regular and thus the fan-out of all the gates is identical.

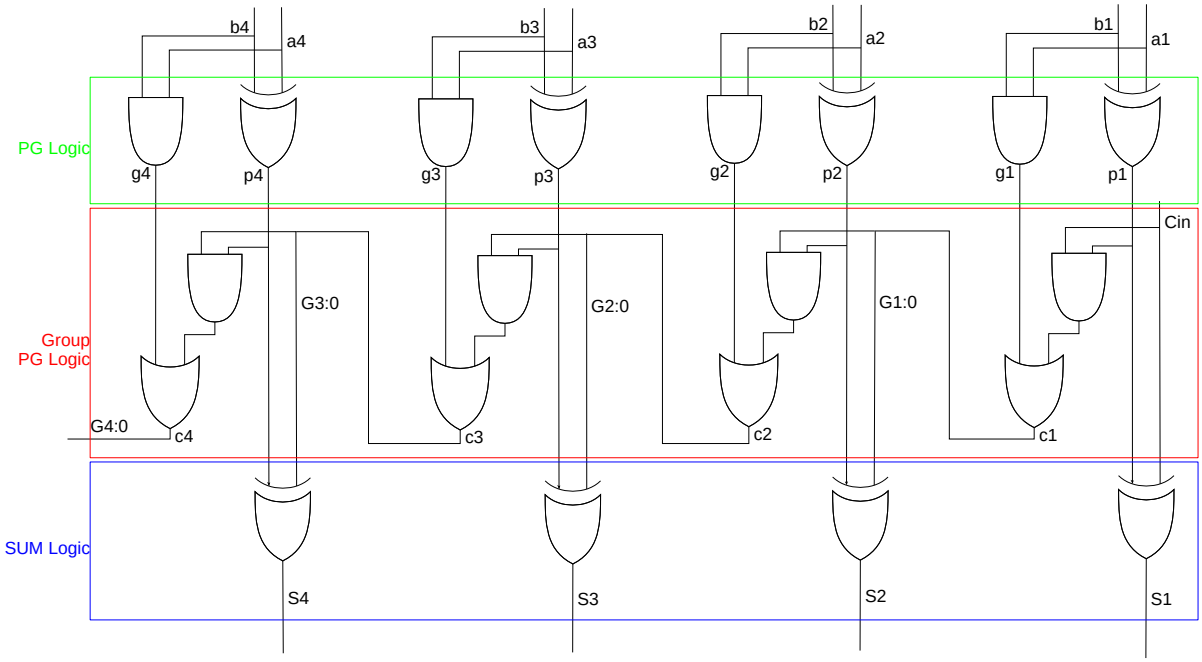


Figure 4.29: Propagate Carry Look Ahead adder.

Tree / Prefix adders

The Carry Look Ahead family is particularly rich of solutions differing in most of the cases for the carry-logic. Such a family is also named "Tree adders" or "Prefix adders".

The carry logic is organized by means of structures sometimes called "carry operators" defined as follows:

$$G_{i:j} = G_{i:k} + P_{i:k} \cdot G_{k-1:j} \tag{4.5}$$

$$P_{i:j} = P_{i:k} \cdot P_{k-1:j} \tag{4.6}$$

where

- $i \geq k > j$
- $G_{x:x} = g_x$ that is the previously defined generate term and $P_{x:x} = p_x$ that is the previously defined propagate term
- $g_0 = c_{IN}$ and $p_0 = 0$

Normally two blocks are used shown in figure 4.30, the first $G - block$ generating only $G_{i:j}$ and the other $PG - block$ generating both $G_{i:j}$ and $P_{i:j}$.

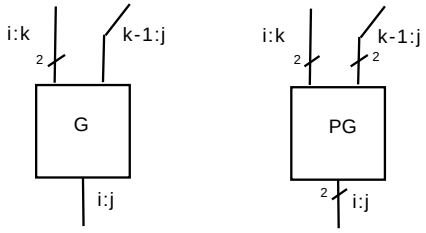


Figure 4.30: Carry operators: G and PG blocks.

Using this block the same ripple carry look ahead we have just analyzed can be represented as in figure 4.31. In this case only G-blocks are used and, for example,

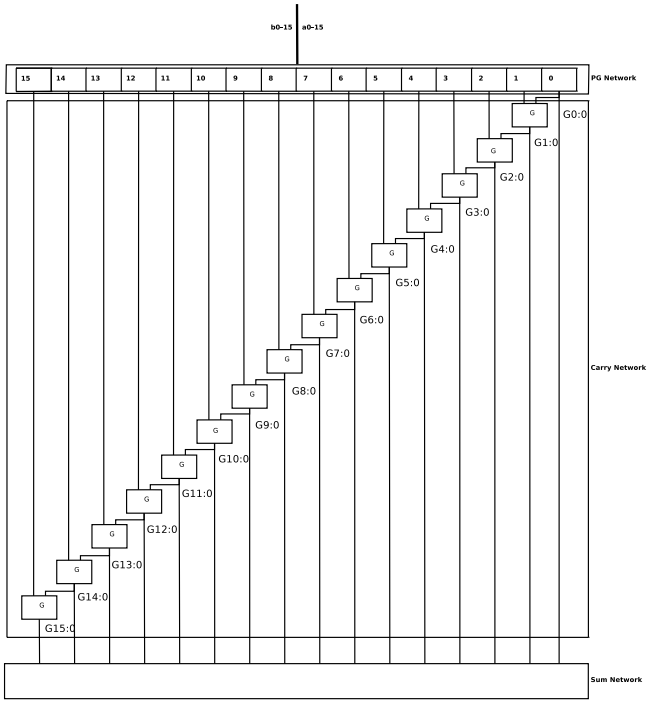


Figure 4.31: Tree description of a propagate Carry Look Ahead adder.

$$G_{1:0} = G_{1:1} + P_{1:1} \cdot G_{0:0} = g_1 + p_1 \cdot g_0$$

while thanks to the connections shown $G_{2:0}$ is obtained as

$$G_{2:0} = G_{2:2} + P_{2:2} \cdot G_{1:0} = g_2 + p_2 \cdot G_{1:0}$$

Using this notation thus a few tree adders can be analyzed. one is the Brent and Kung solution, sketched in figure 4.32. It computes prefixes (PG) for 2-bit groups. These are used to find prefixes for 4-bits groups, which in turn are used to find prefixes for 8-bit groups, and so forth.

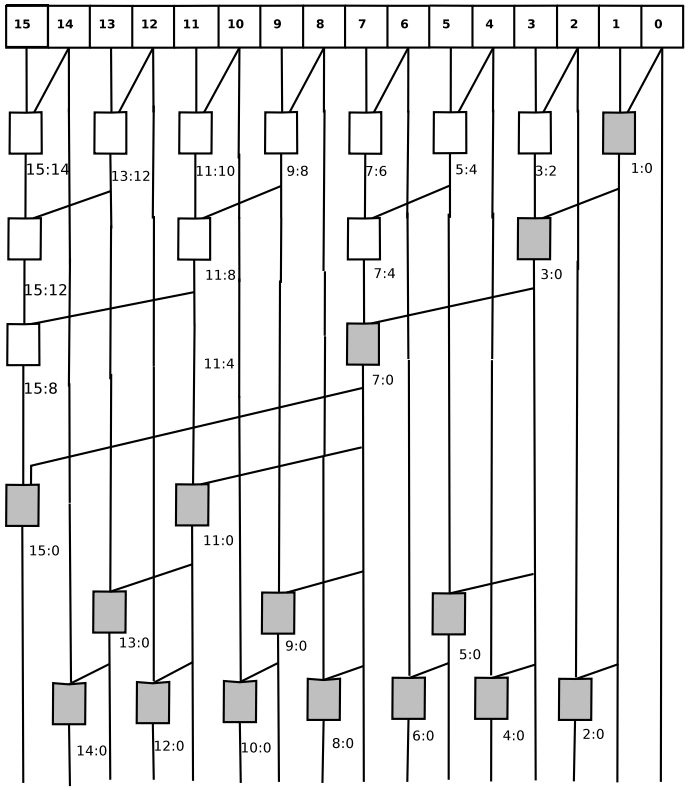


Figure 4.32: Brent and Kung propagate network.

A detail is in figure 4.33, where a few terms are also reckoned using previous definitions. This

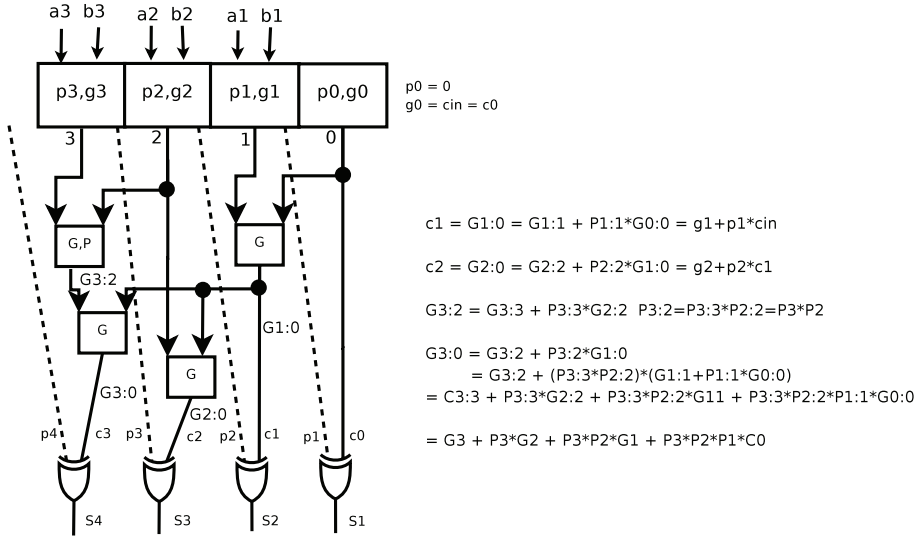


Figure 4.33: Brent and Kung propagate network detail.

structure allows a minimum area and has a logic depth given by $2(log_2 N - 1)$ where N is the operands number of bits. With respect to other solution it assures the minimum area and very attractive performance.

A different solution is the Kogge-Stone adder, reported in figure 4.34 (shadowed boxes are for G blocks, white boxes for PG blocks), which assures the minimum depth, and thus is in theory the faster.

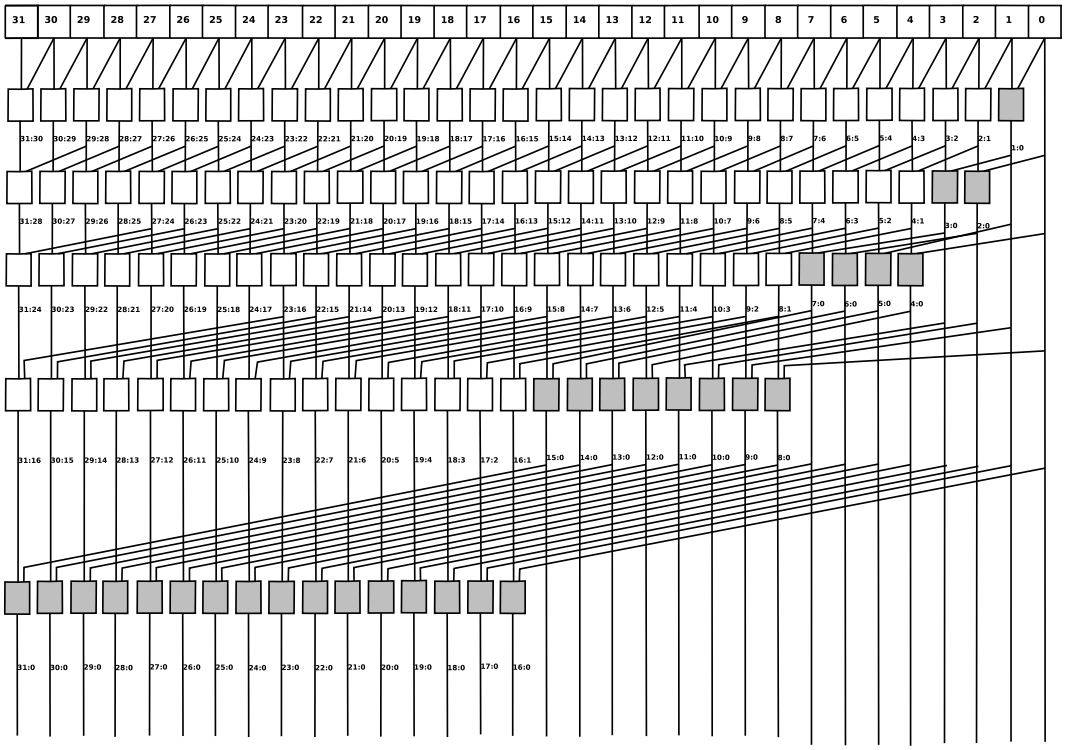


Figure 4.34: Kogge-Stone propagate network.

It results anyway in a large area and has a very complex interconnection system.

A special tree adder is the Sparse-Tree included in the P4 alu described at the beginning of the chapter. The 32-bit adder unit contained in each section of the ALU employs a sparse radix-2 carry-merge tree, reported in figure 4.35 that generates every fourth carry in the adder. This architecture speeds up the critical carry path by moving a substantial portion of carry-merge logic from the main carry-tree to a noncritical side-path.

Instead of generating the carry for each bit as in traditional KoggeStone approaches, the sparse tree generates every fourth carry. Consequently, the performance-setting carry-merge section reduces to a pruned tree that consists of an initial PG (propagate and generate) stage, followed by five stages of carry-merge logic. The sparseness of this tree results in 33%/50% reduction in P/G fan-outs per stage and 80% reduction in wiring complexity, providing a 20% delay reduction at equal energy with respect to a reference KoggeStone implementation.

The noncritical section of the adder consists of a 4-bit conditional sum generator (carry select) that produces sums assuming input carries of 0 and 1 (see before).

It is important to notice that the structure as shown, as also in the detail of figure 4.36 left, works correctly for integer sums. In case of subtractions or sum between non integer numbers then the carry in is necessary. The solution consists in changing the very first block on the right (see figure 4.36 right), that, instead of being a simple p-g structure generates not only the $p1$ and $g1$ terms, but also a $G1:0 = G1:1 + P1:1 \cdot G0:0 = g1 + p1 \cdot C_{in}$. As a consequence, in the following G terms the C_{in} is present.

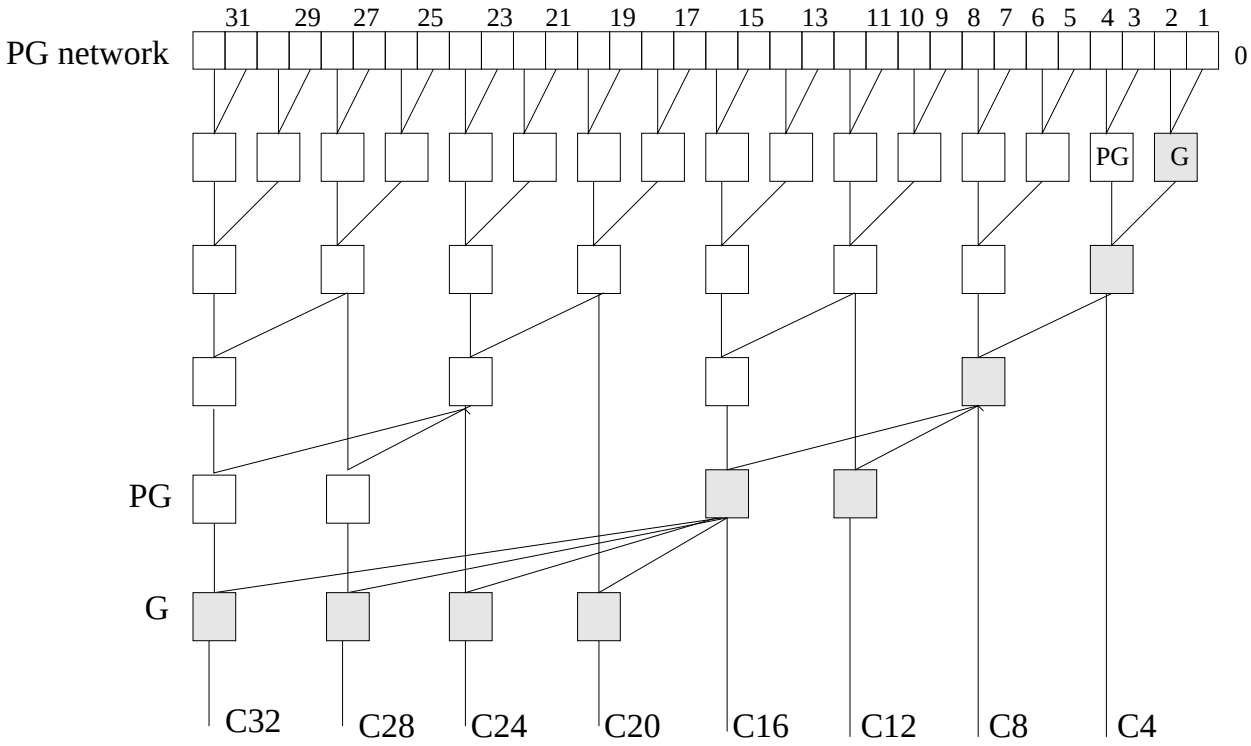


Figure 4.35: Pentium 4 sparse tree propagate network.

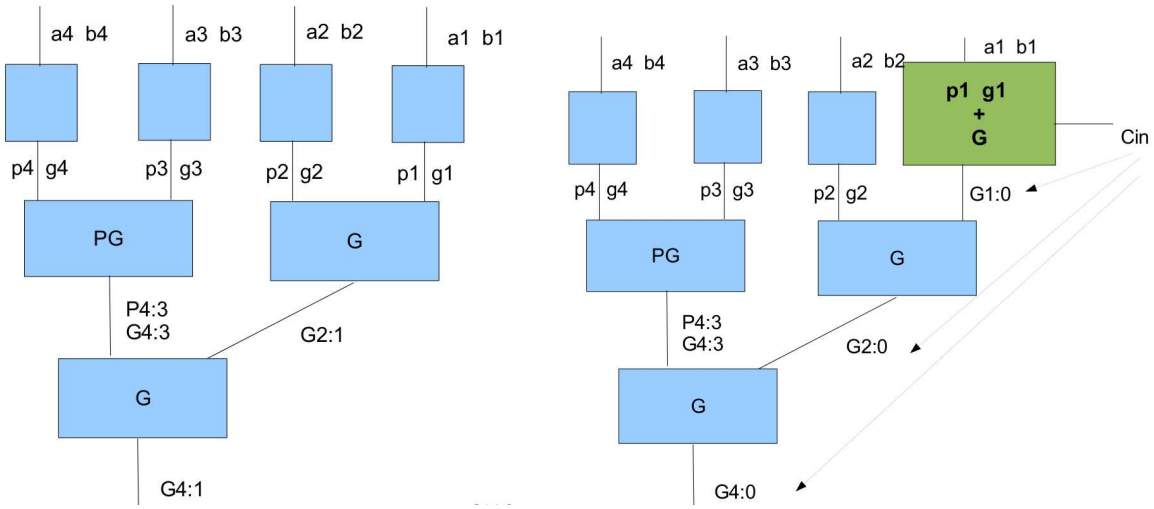


Figure 4.36: Detail for the first terms in the Pentium 4 sparse tree propagate network. On the right a modified version to allow the propagation of carry in.

OPTIONAL: Minimum resources adders

In case performance is not the main goal for an adder design, but resource allocation, and thus area, is the main concerns, then it a possible solution is sketched in figure 4.37.

If the number of bit is $N+1$, then $N+1$ clock cycles are necessary to execute the sum. Two shift registers (with parallel or serial input, see the second part of this section) store the two operands, which are shifted at each clock cycle. At cycle i the FA sum bits a_i and b_i together with carry out i_1 of previous sum. The result is stored in shift register S.

Clock period is limited by sum and carry generation.

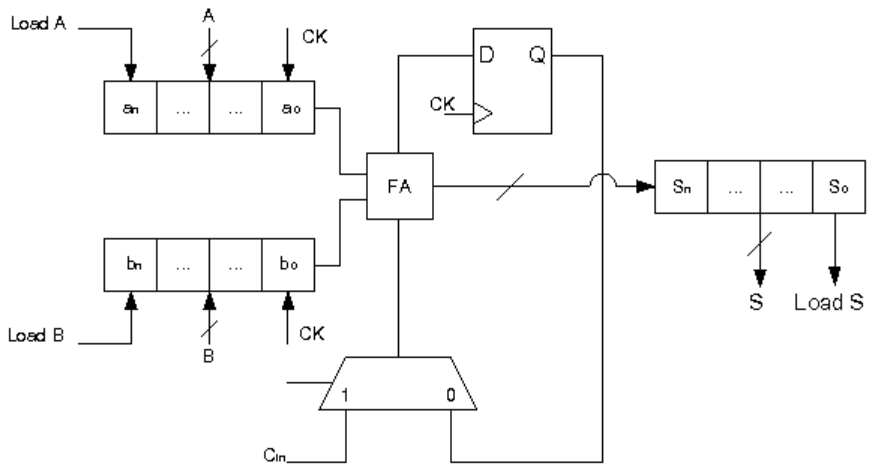


Figure 4.37: Minimum resources adders.

Carry save adders

This adder is used when the sum among more than two operands is needed. Normally it is exploited in the multiplication and division. The natural way is to cascade two adders so that the first as in figure 4.38 sums the first two operands, the second sums the third to the result of the previous addition. This means that the result will be available after a delay equal to the twice the delay of a single adder.

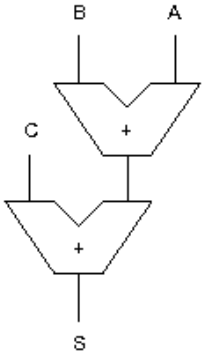


Figure 4.38: Cascaded sum.

The Carry Save Adder is based on the carry property of having a weight bigger of one position with respect to the weight of bits that generated it. It is thus possible to separate the carry vector and add it after the sum without carry has been generated, provided that a left shift is performed on it. Consider for example the sum among operands A,B,C and the final result S:

A	0001
B	0111
C	1101
S	10101

Now, organize the operands bit by bit and group separately sum and carry results in two separate vector as in figure 4.39. If now the two resulting vector are summed after a carry shift, the result is as expected the same as before. These two operations can be executed by the structure in figure 4.40, where an array of FA is used to generate sum and carry array forcing as carry in of each FA the third operand. Then in the second FA array a normal RCA connection is needed as the addition between the partial sum and the carry array may imply a carry propagation. In this structure the delay is

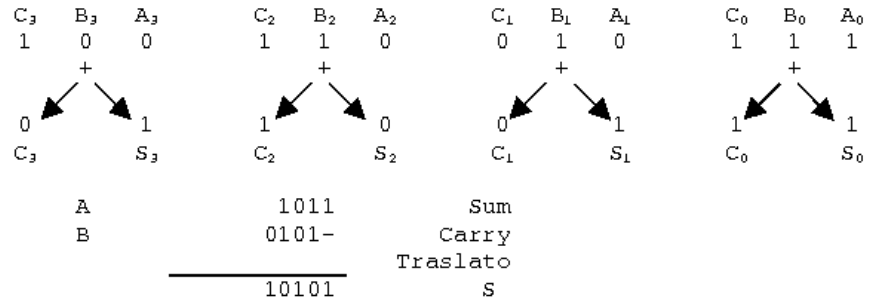


Figure 4.39: Sum and carry vector split in a three operand addition.

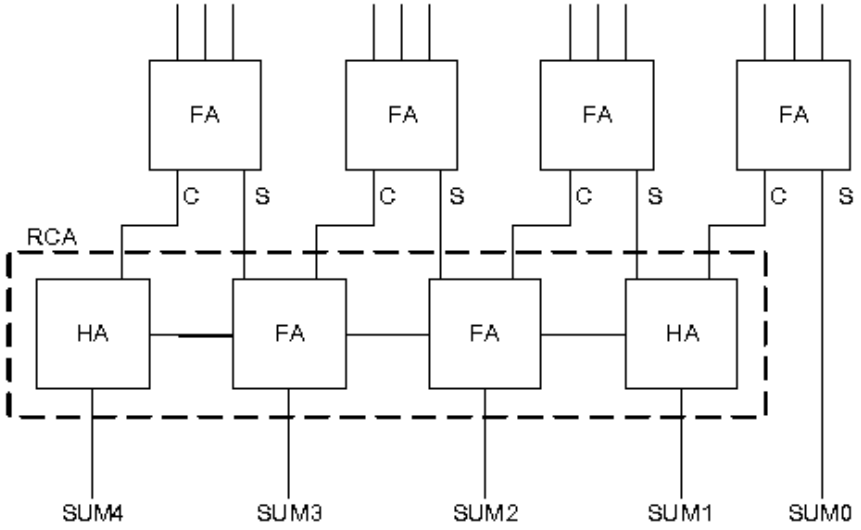


Figure 4.40: Carry Save Adder structure.

clearly reduced as the carry propagation is needed only once.

4.5 Multipliers

The focus is on the **multiplier of the T2 floating point unit**. It is based on two blocks which perform the *Booth's algorithms* for the lowest and highest 32 bits, respectively; they are then merged to obtain the final multiplication using a *Wallace tree*.

As this structure is not documented in details, we are going to analyze the two main principles: Booth's algorithms and Wallace tree. In the following this is done for the integer case, but it is easy to expand the principles to the floating point case.

Again, as in the addition case, we reach the essential points step by step, in order to understand the basic aspects bhind the algorithms and the implementations.

The simplest multiplication algorithm between two integers multiplicands consists in adding shifted partial products as in the by hands multiplication below.

				a_3	a_2	a_1	a_0	*
				b_3	b_2	b_1	b_0	
				$a_3 \cdot b_0$	$a_2 \cdot b_0$	$a_1 \cdot b_0$	$a_0 \cdot b_0$	+
			$a_3 \cdot b_1$	$a_2 \cdot b_1$	$a_1 \cdot b_1$	$a_0 \cdot b_1$	-	+
		$a_3 \cdot b_2$	$a_2 \cdot b_2$	$a_1 \cdot b_2$	$a_0 \cdot b_2$	-	-	+
	$a_3 \cdot b_3$	$a_2 \cdot b_3$	$a_1 \cdot b_3$	$a_0 \cdot b_3$	-	-	-	=
p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0	

Array multiplier

The implementation is in figure 4.41: the two arrays $A \times b_0$ and $A \times b_1$ are generated and added

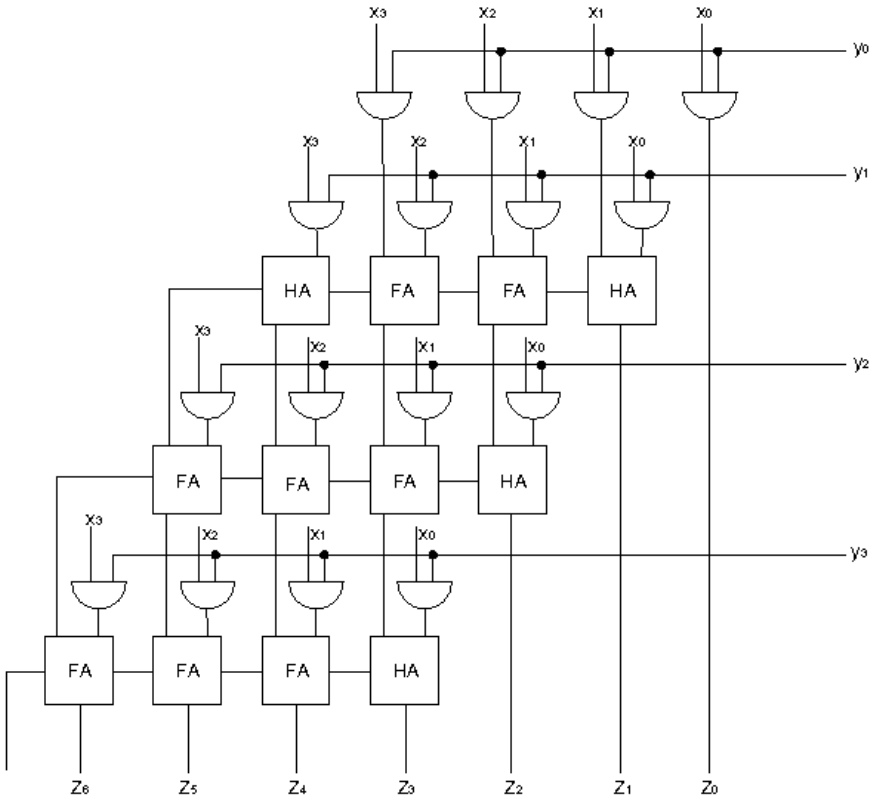


Figure 4.41: Array multiplier.

together with a shifted position for $A \times b_1$. The addition is performed by a simple RCA. The addition result is then added to the third array $A \times b_2$ (shifted) through a second RCA. The same structure is repeated for the fourth array $A \times b_3$.

There is not a unique critical path which is in any case due to the carry propagation along the three adder chains.

Carry Save multiplier

A solution is to use a Carry Save version so that the horizontal carry propagation is present only in the last row. The structure is in figure 4.42. The first CSA sums the three vectors $A \times b_0$, $A \times b_2$ and $A \times b_3$ with proper shifts, and generate a sum and a carry vector. All the other CSA add the previous row sum vector, the current partial product vector (in this case $A \times b_4$) and the previous carry vector properly shifted. A final Ripple Carry is necessary to adjust the carry propagation. This structure assures a lower delay and a higher regularity.

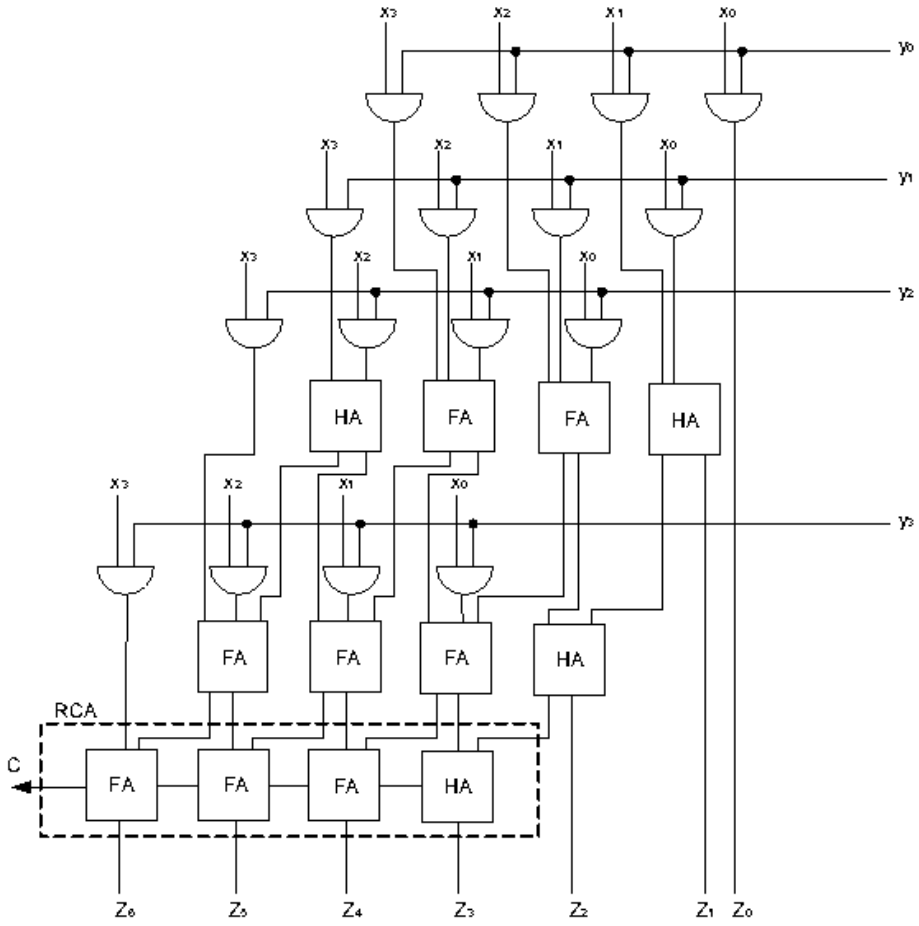


Figure 4.42: Carry Save multiplier.

Wallace tree multiplier

For reducing the multiplier depth and thus its critical path, especially when the multiplicand bit number is high, tree solutions can be used. An example is the Wallace tree sketched in figure 4.43 based on Carry Save Adders, where the input dots represent partial products $A \times b_i$ (each dot for a different i). The idea is to group the additions of partial product exploiting the carry shift principle as much as possible. Also in this case a final RCA is necessary.

Booth's algorithm multiplier

An important improvement consists in implementing the Booth's Algorithm which is based on the recoding of multiplied numbers. It is possible to reduce the number of partial products by half, by using the technique of radix 4 Booth recoding. The basic idea is that, instead of shifting and adding for every column of the multiplier term and multiplying by 1 or 0, we only take every second column, and multiply by ± 1, ± 2,or 0, to obtain the same results.

Consider a multiply A = 010 (N = 3 bit) and a multiplier B = 00011000 (M = 8 bits). Using an array standard solution as before is necessary to generate 8 partial products and 8 additions. The result is P=00000110000 (that is 48).And the non-zero partial products are: $A \times b_3 = 010000$, that is $A \times 8$, and $A \times b_4 = 0100000$, that is $A \times 16$. Thus it is possible to write the final product as:

$$P = A \times 8 + A \times 16 = A \times 24$$

This result can be also written as:

$$P = A \times 32 - A \times 8$$

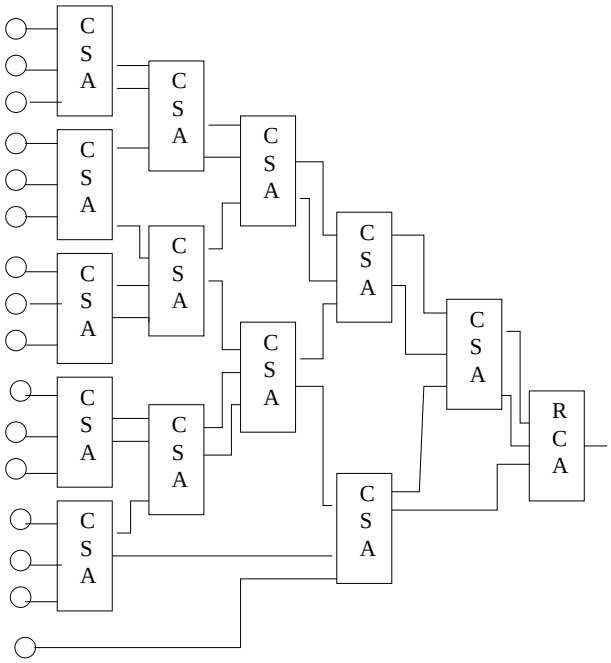


Figure 4.43: Wallace tree multiplier.

where 32 and 8 corresponds to the powers of 2 (5 and 3 exponents respectively) which include the consecutive presences of 1 in the multiplier. This example can be generalized and has the advantage of generating the product without really executing a product, but simply executing left shifts, as only powers of 2 are used.

According to the above example we can get the algorithm of Booth. Assuming for simplicity that M is even, the algorithm in pseudo-code can be written as:

```
i = 0
P = 0
while   i <M - 2      loop
        P <= P + vp(b[i+1], b[i], b[i-1])
        A <= A * 4
        i <= i + 2
end     loop
```

And as a convention, when i = -1, the b[-1] equals 0 and vp is a "partial value" corresponding to a tern of consecutive multiplicand defined as follows:

b[i+1]	b[i]	b[i-1]	vp
0	0	0	0
0	0	1	+A
0	1	0	+A
0	1	1	+2A
1	0	0	-2A
1	0	1	-A
1	1	0	-A
1	1	1	0

According to the above table, Booth recodes the multiplier term, with three bits in one block as b[i+1], b[i], b[i-1], such that each block overlaps the previous block by one bit. Grouping starts from the LSB, and the first block only uses two bits of the multiplier, assuming a zero for the third bit, shown as the figure 4.44. (since there is no previous block to overlap):

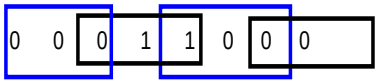


Figure 4.44: Grouping of bits from the multiplier term.

Verifying the previous example of A and B (A = 010, B = 00011000) with the Booth's algorithm, the process of the multiply shown as follow.

i = 0 P<= vp("0,0,0") = 0
i = 2 P<= 0 + vp("1,0,0") = 0 + (-2A)*4
i = 4 P<= 0 - A*8 + vp("0,1,1") = 0 - A*8 + (+2A)*4*4
i = 6 P<= 0 - A*8 + A*32 + vp("0,0,0") = -A*8 + A*32 + 0

which is exactly what we want to achieve. There are numerous implementations in hardware. A block version is shown in figure 4.45.

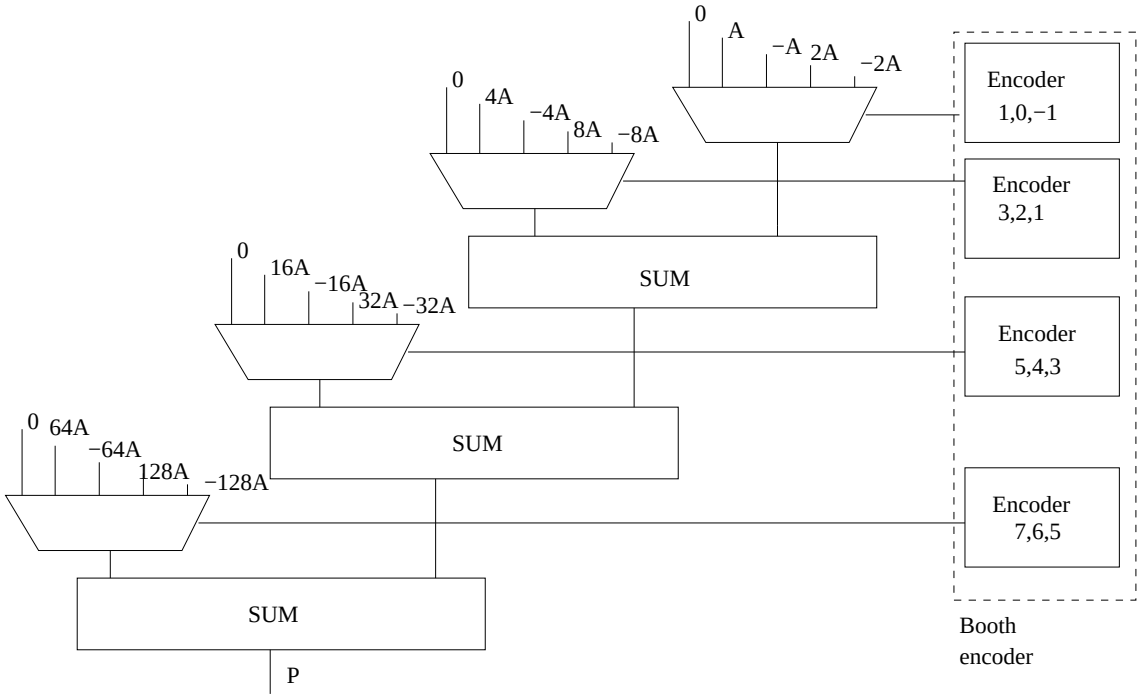


Figure 4.45: Booth's algorithm multiplier.

the number of sums that run halves than the beginning case, then the delay due to the sum of halves. Moreover the partial products (2A, 4A ...) can be obtained in parallel and easily, because it is simple shift to the left of multiplying. Finally, all the blocks making up the booth encoder process the bits of the multiplier in parallel, then the signals of selection are determined by the same delay on all the bits.
Notice that to generalize the algorithm it could be necessary to add zeros to the multiplier. Try for example to change B of previous example to 111000 and notice what happens if such value changes to 00111000.

OPTIONAL: Multiplier to the minimum

A version that uses fewer resources, slower than the parallel, but relatively faster than completely serial, is the Serial/Parallel version. And the structure of the Serial/Parallel is shown in figure 4.46.

The Flip Flop must be manually reset. Y is loaded in parallel; X is loaded in parallel and run in serial. The serial output is: for each clock period is available in output the product to start from the Least Significant Bit (LSB). Least Significant Bit (LSB), which is shown as figure 4.47.

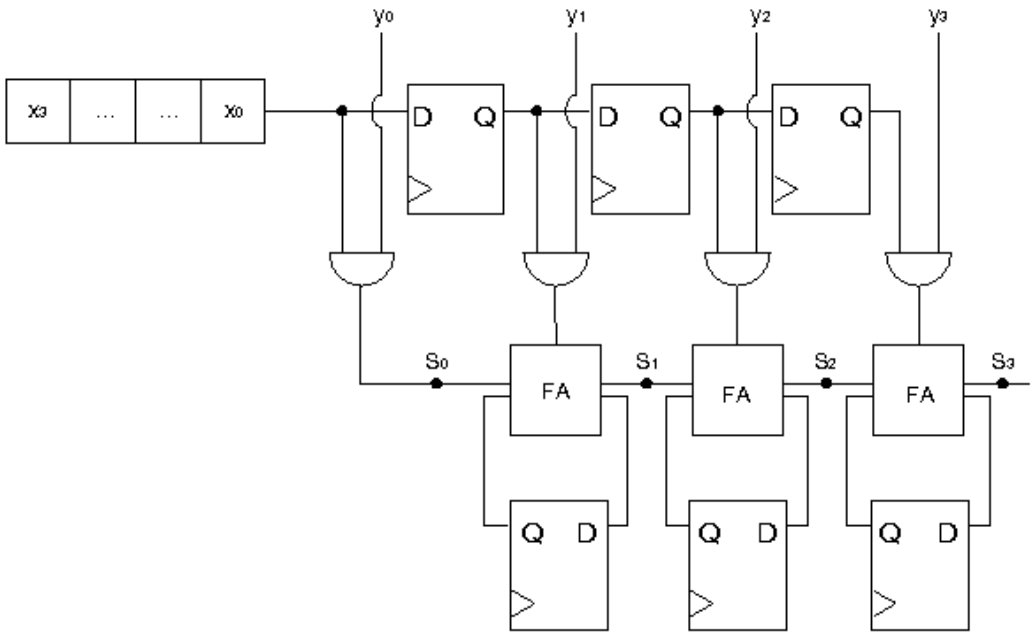


Figure 4.46: Schematic of the Serial/Parallel multiplier.

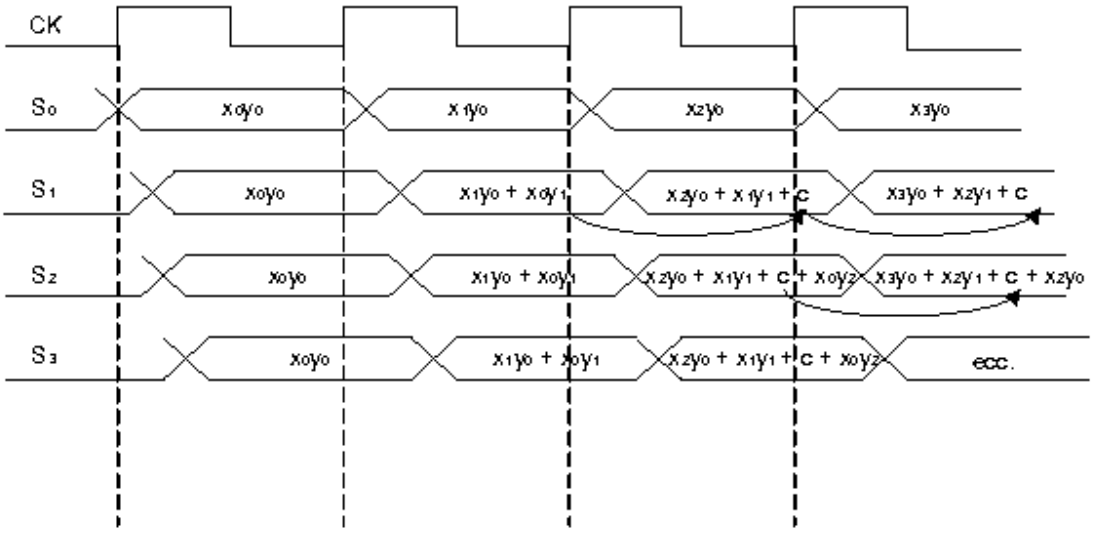


Figure 4.47: Graph of output of the Serial/Parallel multiplier.

4.5.1 ARM VFP11 FMAC: Multipliers and Adders merged

As mentioned at beginning of this chapter, the ARM VFP11 coprocessor has three pipelines. One is the FMAC: multiply and accumulate. This is typical example where adders and multipliers are used in a specific configuration different from what seen before: the **accumulator** (ACC) in figure 4.48 and the **Multiply and accumulate** (MAC).

The first allows to normally sum the operands A and B, or to iteratively sum the values of B storing the temporary values in the accumulator register (ACC).

The second allows to find a sum of products, that is to accumulate in ACC progressive sums of old stored values with new values found as the product $A \times B$. This is for example useful to rapidly calculate products between matrices, integrals, convolutions, applying filters. They are especially used in digital signal processing (DSP).

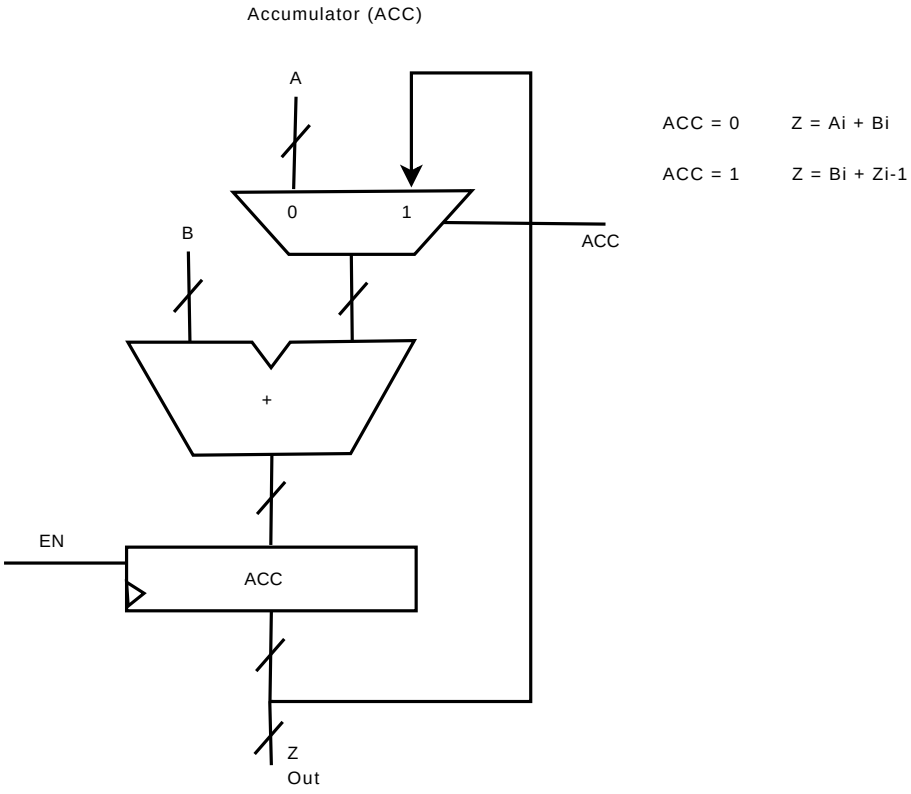


Figure 4.48: Accumulator.

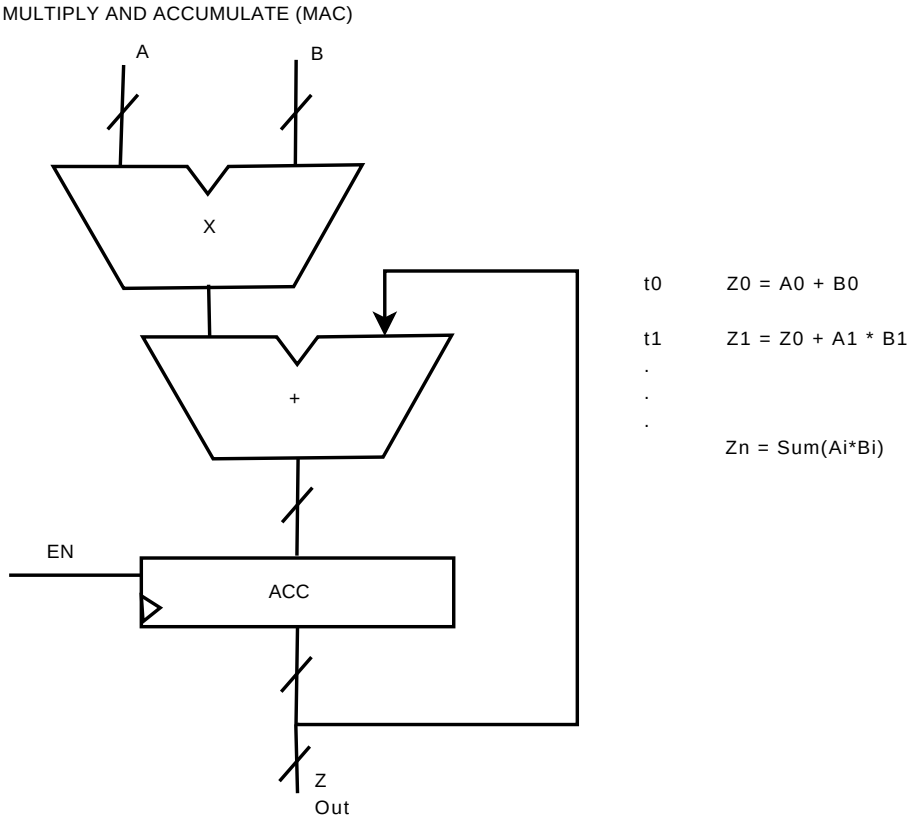


Figure 4.49: Multiply and accumulate.

In the VFP11 coprocessor the structure is more complicated, even though based on this very same principle. The real structure is in figure 4.50. The FMAC family of instructions (FMAC, FNMAC, FMSC, and FNMSC) perform a chained multiply and accumulate operation. The product is computed, rounded according to the specified rounding mode and destination precision, and checked for exceptions before the accumulate operation is performed. The accumulate operation is also rounded according to the specified rounding mode and destination precision and checked for exceptions. The final result is identical to the equivalent sequence of operations executed in sequence. Exception processing and status reporting also reflect the independence of the components of the chained operations.

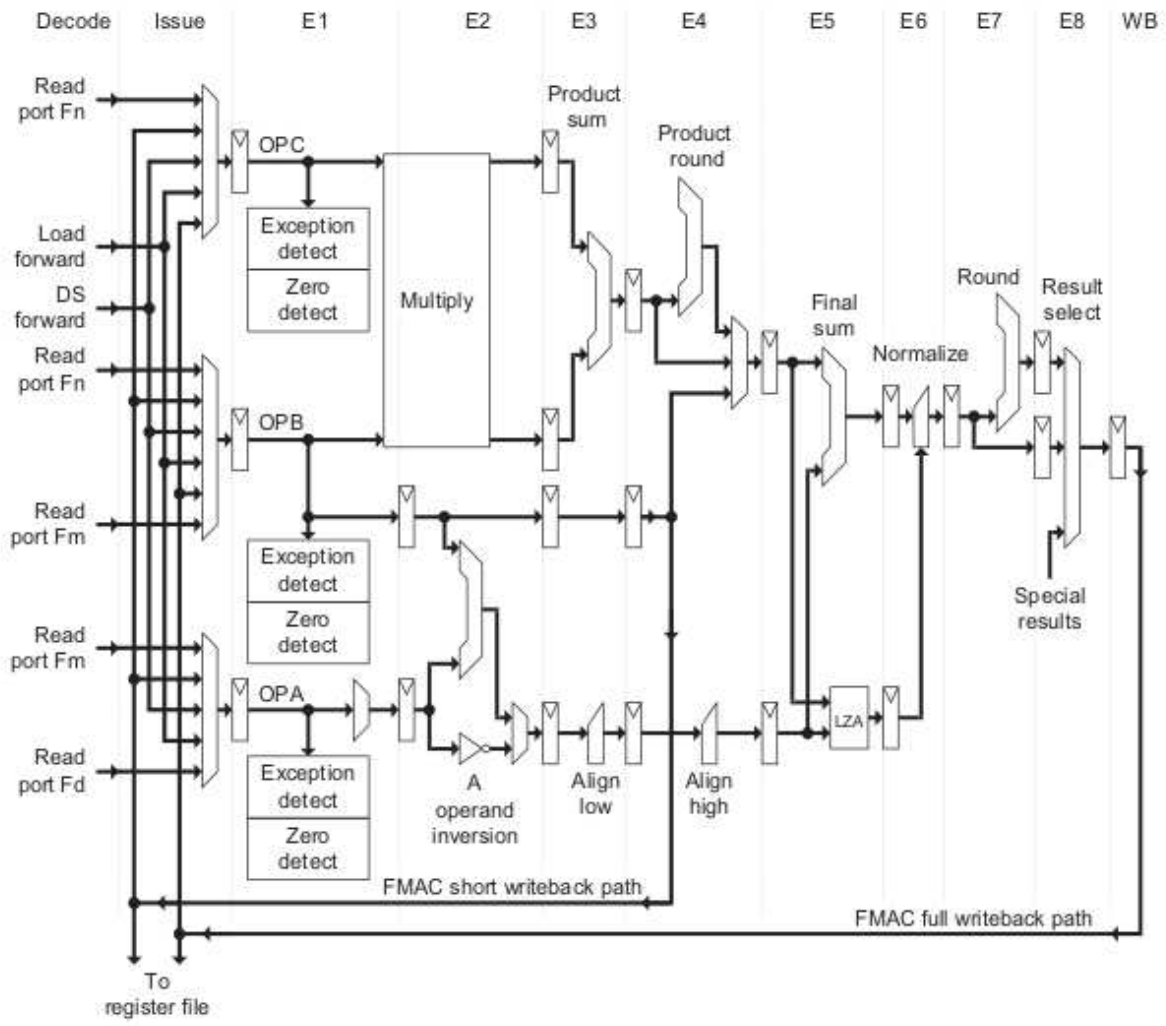


Figure 4.50: Multiply and accumulate.

As an example, the FMAC instruction performs a chained multiply and add operation with the following sequence of operations:

1. The product of the operands in the Fn and Fm registers is computed.
2. The product is rounded according to the current rounding mode and destination precision and checked for exceptions.
3. The result is summed with the operand in the Fd register.
4. The sum is rounded according to the current rounding mode and destination precision and checked for exceptions. If no exception conditions that require support code are present, the result is written to the Fd register.

4.6 Dividers

Several are the techniques for executing division. Here we will use as an example the implementation in the ARM VFP11 coprocessor.

The **DS pipeline** executes the following instructions:

- FDIV Divide.
- FSQRT Square root

The VFP11 coprocessor executes divide and square root instructions for both single-precision and double-precision operands with all IEEE 754 standard rounding modes supported. The DS unit uses a shared **radix-4 SRT algorithm** that provides a good balance between speed and chip area. DS operations have a latency of 19 cycles for single-precision operations and 33 cycles for double-precision operations. The throughput is 15 cycles for single-precision operations and 29 cycles for double-precision operations.

In the following then we will describe the radix-4 SRT algorithm and its hardware implementation in the ARM VFP11 (in simplified formulation). However, as the algorithm is not simple, we will reach its discussion step by step starting by a simple implementation.

4.6.1 From simple iterative to radix-4 SRT division

The algorithm at the basis of the SRT is the simple iterative one that we learned at school. The idea is to extract one by one the quotient numbers. Let's recall first without formalism how we work in decimal and binary.

For example, if our *dividend* $Z = 14_{10}$ has to be divided by our *divisor* $D = 3_{10}$ we start guessing, as in figure 4.51 what the first digit of quatient could be by considering the first digit of the dividend; the first guess is 0, which gives us a partial reminder of 14. By guessing again the second digit of the quotient we write a 4 and find the next reminder, 2, which is the final *reminder* R. the final *quotient* q_{10} is 4.

```
 14 | 3
 0  |--
----| 04
 14
 12
----
  2
```

Figure 4.51: Pen and paper decimal division

If we work in binary, the pen and paper mechanism is the same, as in figure 4.52. This time $Z = 14_{10} = 1110_2$ and $D = 3_{10} = 11_2$. We start by guessing the first quotient digit $q_1 = 1$ and find a partial reminder $R_1 = 00$. Then the next guess is $q_2 = 0$ and the related reminder is $R_2 = 01$, and finally the last quotient is $q_3 = 0$ witha final reminder $R_3 = 01$.

```
 1110 | 11
 11   |--
------| 100
  001
  00
 ----
   10
   00
  ---
   10
```

Figure 4.52: Pen and pencil bynary division

By thinking at what we have done we notice that i) the method is iterative and this means it can be easily executed in hardware if we accept of obtaining one new quotient digit at each clock cycle by

storing partial values in registers; ii) the difficult aspect in hardware is "guessing" the quotient, so we have to find a systematic method for this step.

The algorithm can be written considering the following iterative formulation:

$$R^{j+1} = r \cdot R^j - q_{j+1} \cdot D$$

where: R^j is the reminder at step j, r is the number of digits in the alphabet, q_j is the digit j-th of the quotient and D is the divider. To simplify the discussion, and without loosing generality, let's make the following assumptions: a) the operands are both positive, b) the operands are normalized, i.e. smaller than 1 in absolute value, c) the divident Z is smaller than the divisor D. Conditions b) and c) can be obtained by shifting the operands, and at the end of the operation they can be shifted back to recover the original form.

Let's say then that:

$$Q = .q_1 q_2 \cdots q_n \qquad D = .d_1 d_2 \cdots d_n \qquad Z = R^{(0)} = .r_1^{(0)} r_2^{(0)} \cdots r_n^{(0)}$$

where Q is the quotient, D is the dividend and $R^{(0)}$ is the initial reminder, that is the dividend, with the condition that $R^{(0)} < D$

As a first step let's verify that the iterative formulation really gives the requested quotient at the end of the execution.

$$\begin{aligned}
j = 0, \quad & R^{(1)} = r \cdot R^{(0)} - q_1 \cdot D \\
j = 1, \quad & R^{(2)} = r \cdot R^{(1)} - q_2 \cdot D = r \cdot \left[r \cdot R^{(0)} - q_1 \cdot D \right] - q_2 \cdot D = r^2 \cdot R^{(0)} - \left[r \cdot q_1 + q_2 \right] \cdot D \\
& .... \\
j = n-1, \quad & R^{(n)} = r^n \cdot R^{(0)} - \left(r^{n-1} \cdot q_1 + r^{n-2} \cdot q_2 + \cdots + r \cdot q_{n-1} + q_n \right) \cdot D
\end{aligned}$$

and as a consequence:

$$\frac{R^0}{D} = r^{-1} \cdot q_1 + r^{-2} \cdot q_2 + \cdots + r^{-(n-1)} \cdot q_{n-1} + r^{-n} \cdot q_n + \frac{R^n \cdot r^{-n}}{D}$$

where we easily recognize quotient Q and reminder R:

$$Q = \sum_{j=1}^{n} q_j \cdot r^{-j} \qquad R = r^{-n} \cdot R^{(n)}$$

The procedure that allows to find the q_j digit and the partial reminder $R^{(j)}$ is what differentiates one solution from another one.

The first to be understood is the **Conventional Restoring Division**.

Let's stay with binary numbers so that $r = 2$. As a consequence, the alphabet for the numbers representation is $S = \{0, 1\}$. The iterative formula becomes then:

$$R^{j+1} = 2 \cdot R^j - q_{j+1} \cdot D$$

According to the division with restoring the following condition is assumed:

$$0 \leq R^{(j+1)} < D$$

As a consequence the possible quotient is chosen in this way:

$$q_{j+1} = \begin{cases} 0 & if \quad 2 \cdot R^{(j)} < D \\ 1 & if \quad 2 \cdot R^{(j)} \geq D \end{cases}$$

From an operating point of view the algorithm supposes $q_{j+1} = 1$ in each iteration and the reminder is estimated executing the subtraction

$$R_{es}^{(j+1)} = 2\dot{R}^{(j)} - D$$

(as $q_{j+1} = 1$). At this point if the result of the subtraction is positive, then the reminder was guessed correctly

$$R^{(j+1)} = R_{es}^{(j+1)}$$

and so was the quotient digit. On the contrary it is the result of the guessing is negative, than the correct quotinet digit is $q_{j+1} = 0$ and the correct reminder must be *restored* by adding again the devider to the estimated reminder:

$$R^{(j+1)} = R_{es}^{(j+1)} + D = 2 \cdot R^{(j)}$$

This last step is the reason why it is called a restoring division. Figure 4.53 shows graphically (Robertson diagram) the criteria used while chosing the (j+1)-th reminder e of the quotient digit as a function of the partial dividend $2 \cdot R^{(j)}$.

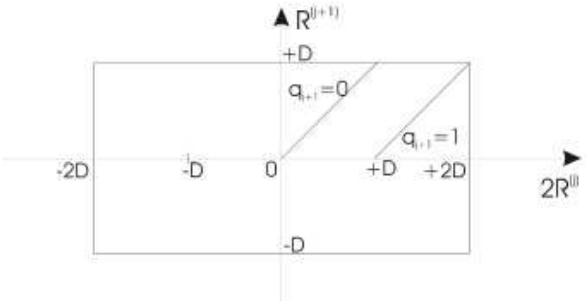


Figure 4.53: Robertson diagram for restoring division.

The x axis represents then the current partial reminder $R^{(j)}$, while the y axis represents the value of the reminder after one additional divide step. The graph shows how we decide what to set for the quotient bit and what is the next value of the remainder.

Let's make now an example with a particular case of A/B where A=0.01010110 and B=0.1100. In the following the unrolled procedure.

$$\begin{aligned}
A &= 0.0\,1\,0\,1\,0\,1\,1\,0 \\
B &= 0.1\,1\,0\,0 \\
R^{(0)} = A &= 0.0\,1\,0\,1\,0\,1\,1\,0
\end{aligned}$$

$$\begin{aligned}
2 \cdot R^{(0)} &= 0.1\,0\,1\,0\,1\,1\,0 \\
D &= 0.1\,1\,0\,0 \\
\hat{R}^{(1)} = 2 \cdot R^{(0)} - D &= 1.1\,1\,1\,0\,1\,1\,0 \quad \text{Neg} \quad \Rightarrow R^{(1)} = 2 \cdot R^{(0)},\ q_1 = 0
\end{aligned}$$

$$\begin{aligned}
2 \cdot R^{(1)} &= 1.0\,1\,0\,1\,1\,0 \\
D &= 0.1\,1\,0\,0 \\
\hat{R}^{(2)} = 2 \cdot R^{(1)} - D &= 0.1\,0\,0\,1\,1\,0 \quad \text{Pos} \quad \Rightarrow R^{(2)} = \hat{R}^{(2)},\ q_2 = 1
\end{aligned}$$

$$\begin{aligned}
2 \cdot R^{(2)} &= 1.0\,0\,1\,1\,0 \\
D &= 0.1\,1\,0\,0 \\
\hat{R}^{(3)} = 2 \cdot R^{(2)} - D &= 0.0\,1\,1\,1\,0 \quad \text{Pos} \quad \Rightarrow R^{(3)} = \hat{R}^{(3)},\ q_3 = 1
\end{aligned}$$

$$\begin{aligned}
2 \cdot R^{(3)} &= 0.1\,1\,1\,0 \\
D &= 0.1\,1\,0\,0 \\
\hat{R}^{(4)} = 2 \cdot R^{(3)} - D &= 0.0\,0\,1\,0 \quad \text{Pos} \quad \Rightarrow R^{(4)} = \hat{R}^{(4)},\ q_4 = 1
\end{aligned}$$

That gives:

$$\frac{A}{B} = 0.0111 + \frac{2^{-4} \cdot 0.0010}{0.1100} \qquad Q = 0.0111 \qquad R = 0.00000010$$

From the hardware implementation point of view this algorithm is not complex. It requires one register to store D, two registers to store and left shift (as we mutiply time 2) the partial reminder, ad adder/subtractor and a logic to pick the correct quotient digit and if necessary restoring after the decision.

In order to improve the execution efficiency a few variations have been proposed. the first is using a **redundant representation** for the quotient. Basically the alphabet is no more conventional ba includes also signed symbols. Let's consider to have alphabet $S = \{\overline{1}, 0, 1\}$ Exactly as for the other representations the following relation between number N and the digits used to represent it:

$$N = \sum_{i=0}^{k} n_i \cdot 2^i$$

where digits n_i assume values -1,0,1 depending on their being $\overline{1}, 0, 1$ respectively. The consequence is that an integer number has not a unique representation. For example number N=3 on 3 digitas has the following possible representations:

$$011 \Rightarrow 1 + 2 = 3 \qquad 10\overline{1} \Rightarrow 4 - 1 = 3 \qquad 1\overline{1}1 \Rightarrow 4 - 2 + 1 = 3$$

Notice that the sign of N is given by the sign of the first digit different from zero (always one in this case). Moreover, the opposite of N can be obtained by simply changing the sign of the digits in one of its representation:

$$0\overline{11} \Rightarrow -1 - 2 = -3 \qquad \overline{1}01 \Rightarrow -4 + 1 = -3 \qquad \overline{1}1\overline{1} \Rightarrow -4 + 2 - 1 = -3$$

An improvement to the restoring algorithm, the **Non Restoring Division**, uses a redundat representation considering q_j in the set $\{\overline{1}, 1\}$ for the quotient, and the normal representation for the rest.

In this case the criterion used to chose the next quotient and reminder is

$$|R^{(j+1)}| < D$$

and thus

$$R^{(j+1)} = \begin{cases} 2 \cdot R^{(j)} - D & if \quad 2 \cdot R^{(j)} > 0 \\ 2 \cdot R^{(j)} + D & if \quad 2 \cdot R^{(j)} < 0 \end{cases}$$

$$q_{j+1} = \begin{cases} 1 & if \quad 0 < 2 \cdot R^{(j)} < 2 \cdot D \\ \overline{1} & if \quad -2 \cdot D < 2 \cdot R^{(j)} < 0 \end{cases}$$

If $2 \cdot R^{(j)} = 0$ the rest is zero and the operation is concluded.

Basically, when we had to restore the remainder in the restoring division, we do not do so in the nonrestoring. We simply compensate for that in the next step of division by adding to the remainder instead of subtracting from it. Here is the demonstration.

In a given step j the reminder is $R^{(j)}$. In the restoring algorithm $R^{(j+1)} = 2 \cdot R^{(j)} - D$ is calculated. If $R^{(j+1)}$ is positive both the algorithms finish with the same reminder. If, on the contrary, the estimated reminder is negative, then the restoring algorithm restores $R^{(j+1)} = 2 \cdot R^{(j)}$ and thus in the next step

$$R^{(j+2)} = 2 \cdot R^{(j+1)} - D = 2(2 \cdot R^{(j)}) - D$$

In the non restoring algorithm $R^{(j+1)}$ is kept $2 \cdot R^{(j)} - D$ which is negative, and in the next step, as it is negative,

$$R^{(j+2)} = 2 \cdot R^{(j+1)} + D = 2(2 \cdot R^{(j)} - D) + D = 2(2 \cdot R^{(j)}) - 2D + D = 2(2 \cdot R^{(j)}) - D$$

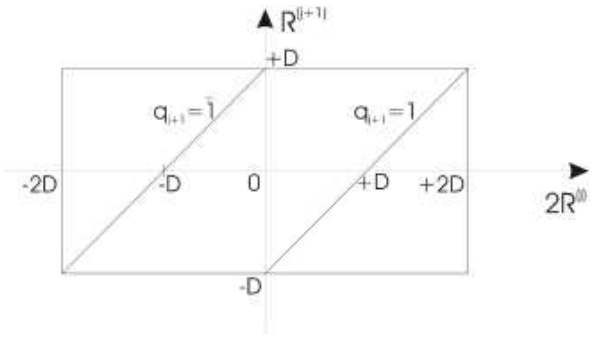


Figure 4.54: Robertson diagram for non restoring division.

Here is the same example as before based on the non restoring algorithm.

$$
\begin{aligned}
A &= 0.0101011\,0 \\
B &= 0.1100 \\
R^{(0)} = A &= 0.0101011\,0 \quad \text{Pos.} \quad \Rightarrow q_1 = 1 \\
\\
2\cdot R^{(0)} &= 0.1010110\,- \\
D &= 0.1100 \\
R^{(1)} &= 1.1101101\,0 \quad \text{Neg.} \quad \Rightarrow q_2 = \bar{1} \\
\\
2\cdot R^{(1)} &= 1.1011010\,+ \\
D &= 0.1100 \\
R^{(2)} &= 0.1001101\,0 \quad \text{Pos.} \quad \Rightarrow q_3 = 1 \\
\\
2\cdot R^{(2)} &= 1.0011010\,- \\
D &= 0.1100 \\
R^{(3)} &= 0.0111101\,0 \quad \text{Pos.} \quad \Rightarrow q_4 = 1 \\
\\
2\cdot R^{(3)} &= 0.1111010\,- \\
D &= 0.1100 \\
R^{(4)} &= 0.0010
\end{aligned}
$$

So finally

$$\frac{A}{B} = 0.1\bar{1}11 + \frac{0.0010 \cdot 2^{-4}}{0.1100}$$

it should be noticed that if the final reminder is negative, the devider must be added up (restored), as well as the wuotient must be modified (subtract 2^{-n}).

If then it is necessary to go back again to the conventional representation the following simple operation can be performed:

$$Q = 0.1\bar{1}11 = 0.1011 + 0.0\bar{1}00 = 0.1011 - 0.0100 = 0.0111$$

A block diagram of the harware implementation is in figure 4.55. This technique doe not require restoring (if not inthe lest step if needed). However, this procedure can be improved.

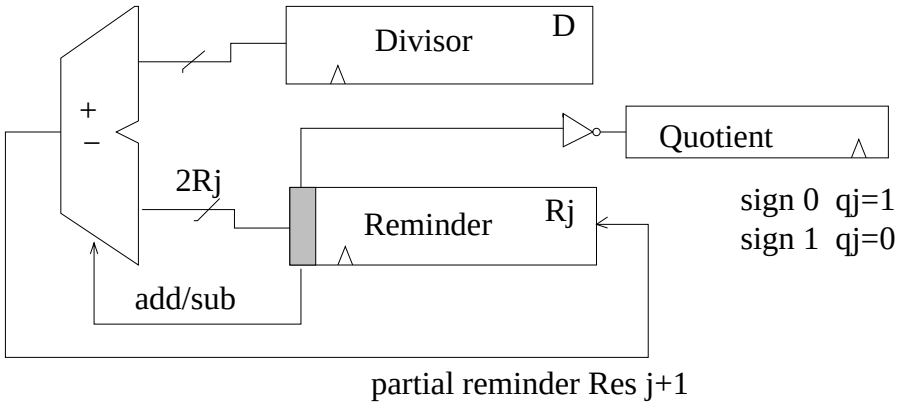


Figure 4.55: A block diagram of the non restoring iterative algorithm.

Ths improvement consists in the **SRT division algorithm**.
The alphabet is $S = \{\overline{1}, 0, 1\}$ and all three are the possible values of the quotient digit. Morever, a normalization takes place

$$\frac{1}{2} \leq |D| < 1 \qquad \frac{1}{2} \leq |2 \cdot R^{(0)}| < 1$$

This is simple to assure as is is sufficient to shift left the number until the leading bit is a 1.

The criteria adopted are:

$$R^(j+1) = \begin{cases} 2 \cdot R^{(j)} + D & if \quad 2 \cdot R^{(j)} < -D \\ 2 \cdot R^{(j)} & if \quad -D \leq 2 \cdot R^{(j)} \leq D \\ 2 \cdot R^{(j)} - D & if \quad 2 \cdot R^{(j)} > D \end{cases}$$

$$q_{j+1} = \begin{cases} \overline{1} & if \quad 2 \cdot R^{(j)} < -D \\ 0 & if \quad -D \leq 2 \cdot R^{(j)} \leq D \\ 1 & if \quad 2 \cdot R^{(j)} > D \end{cases}$$

It is worth noticing that in practice it is not necessary to really compare $2 \cdot R^{(j)}$ with D or -D, but 1/2 can be used instead, if we combine the criterion with the assumtion on D and the reminder. This is simpler from an hardware implementation point of view.

$$R^(j+1) = \begin{cases} 2 \cdot R^{(j)} + D & if \quad 2 \cdot R^{(j)} < -\frac{1}{2} \\ 2 \cdot R^{(j)} & if \quad -\frac{1}{2} \leq 2 \cdot R^{(j)} < \frac{1}{2} \\ 2 \cdot R^{(j)} - D & if \quad \frac{1}{2} \leq 2 \cdot R^{(j)} \end{cases}$$

$$q_{j+1} = \begin{cases} \overline{1} & if \quad 2 \cdot R^{(j)} < -\frac{1}{2} \\ 0 & if \quad -\frac{1}{2} \leq 2 \cdot R^{(j)} < \frac{1}{2} \\ 1 & if \quad \frac{1}{2} \leq 2 \cdot R^{(j)} \end{cases}$$

Figure 4.56 shows the Robertson diagram for this case.

This case is particularly convenient because camparing with $-\frac{1}{2}$ means $2 \cdot R^{(j)} = (1.0XXXX)_{2'scompl.}$ and comparing with $\frac{1}{2}$ means $2 \cdot R^{(j)} = (0.1XXXX)_{2'scompl.}$ and thus the comparison simply consists in considering the two most significant bits. In particular, when $q = 0$ the condition consists in searching for cases where two most significant consecutive digits are different. And this case does not require additions or subtractions.

Another important advantage of this method is that the value of the quotient to be computed in a given step only depends on the result of previous step; while in case of nonrestoring (and restoring)

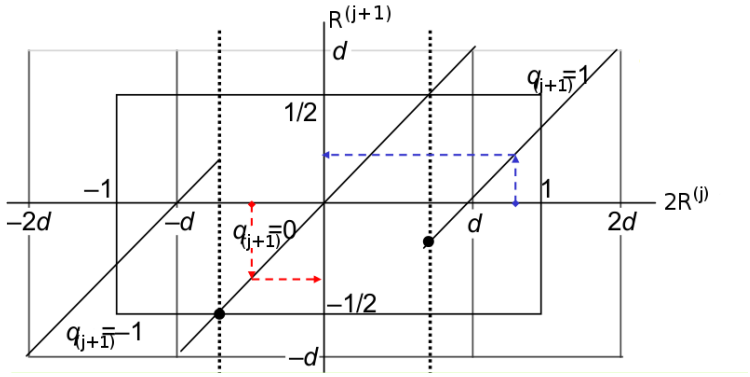


Figure 4.56: Robertson diagram for SRT division.

the quotinet to be computed in a step depends on the result of the operation executed in the same step.

The same example as before for the SRT algorithm is in the following.

$$A = 0.0101011 0$$
$$B = 0.1100$$
$$R^{(0)} = A = 0.0101011 0$$

$$2 \cdot R^{(0)} = 0.1010110 \quad 2 \cdot R^{(0)} \geq \frac{1}{2} \Rightarrow q_1 = 1$$

$$D = 0.1100$$
$$R^{(1)} = 1.1110110$$

$$2 \cdot R^{(1)} = 1.1101100$$

The most significant bits are eliminated until tow consecutive different bits are found. For each "elmination" a quetient digit = 0 is used

$$q_2 = 0, q_3 = 0$$

$$2 \cdot R^{(3)} = 1.0110 + \qquad 2 \cdot R^{(3)} < -\frac{1}{2} \Rightarrow q_4 = \bar{1}$$

$$D = 0.1100$$
$$R^{(4)} = 0.0010$$

$$\frac{A}{B} = 0.100\bar{1}11 + \frac{0.0010 \cdot 2^{-4}}{0.1100}$$

If then it is necessary to go back again to the conventional representation the following simple operation can be performed:

$$Q = 0.100\bar{1} = 0.1000 + 0.000\bar{1} = 0.1000 - 0.0001 = 0.0111$$

A block diagram of the hardware implementation is in figure 4.57

It is possible to further improve this method using **high-radix** algorithms. Let's assume again that the operands are represented in a normalized floating point format with n bit significands in

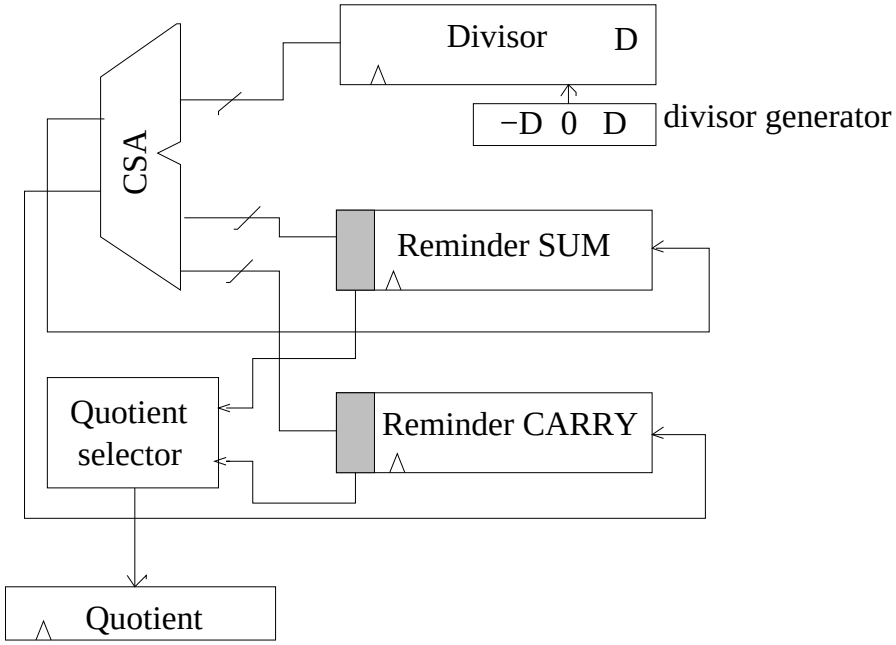


Figure 4.57: A block diagram of the SRT divider.

sign-and-magnitude representation. The quotient Q is defined to comprise k radix-r digits with

$$r = 2^b \qquad k = n/b$$

where a division algorithm that retires b bits of quotient in each iteration is said to be a readix-r algorithm. Such an algorithm requires k iterations to compute the final n bit result and thus a latency of k cycles.

The SRT algorithm analyzed before is a radix-2, as $r = 2^1$, which requires $k = n/1$ cycles to complete. The fundamental method of decreasing the overall latency (in machine cycles) of the algorithm is to increase the radix r of the algorithm, typically chosen to be a power of 2. However, this latency reduction does not come for free. As the radix increases, the quotient-digit selection becomes more complicated, which may increase the cycle time. Moreover, the generation of all required divisor multiples may become impractical for higher radices. it was demonstrated that the delay of quotient selection tables increases linearly with increasing radix, while the area increases quadratically. While prescaling of the input operands reduces table complexity at the expenseof additional latency, nevertheless the difficulty in generating all of the required divisor multiples for radix-8 and higher limits practical divider implementations to radix-2 and radix-4.

For a given choice of radix r, some range of digits is decided upon for the allowed values of the quotient in each iteration. The simplest case is where, for radix r, there are exactly r allowed values of the quotient. However, to increase the performance of the algorithm, a redundant digit set is used. This allows a quotient digit to be selected based upon an approximation of the partial remainder, permitting the useof a redundant remainder representation as discussed before. Small errors in the quotient due to the remainder approximation are corrected in later iterations. Such adigit set is composed of symmetric signed-digit consecutive integers, where the maximum digit is a. The digit set is made redundant by having more than r digits in the set. By using a larger number of allowed quotient digits, the complexity and latency of the quotient selection function is reduced. However, choosing a smaller number of allowed digits for the quotient simplifies the generation of the multiple of the divisor. Specifically, for radix-2, the digit set is $\{\overline{1}; 0; 1\}$. For radix-4, there are two typical choices for the digit set: minimally redundant

$\{-2; -1; 0; 1; 2\}$ and maxi-mally redundant $\{-3; -2; -1; 0; 1; 2; 3\}$. The quotient selection logic for a maximally-redundant radix-4 digit set is about 20% faster and 50% smaller than for a minimally-redundant digit set. However, maximally-redundant radix-4 requires the computation of the 3x divisor multiple, which typically requires extra initial delay and area.

4.6.2 OPTIONAL: The real ARM VFP11 HW implementation

The major design requirement was to achieve a logic depth of as close to 15 logic levels as possible so as to meet a variety of performance targets for the whole chip. This led to the minimum-redundancy radix-4 SRT algorithm being used because multiplicative solutions were unattractive at the required high clock rate for two reasons:

- fast multipliers are large and power-hungry it was infeasible to use the extant VFP11 multiplieraccumulator chain for performing division and square root operations, and wasteful in area and power terms to build a second multiplier dedicated to square root and division
- in a deeply-pipelined processor design, NewtonRaphson and Goldschmidt iterations take many cycles to complete due to dependencies between and within successive iterations. By contrast, using radix-4 SRT, VFP11 takes 15 cycles (single precision) or 29 cycles (double precision) to compute either correctly rounded quotients or square roots.

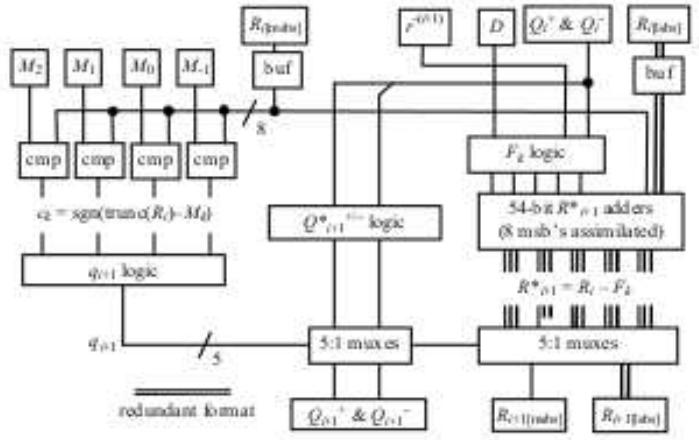


Figure 4.58: Schematic of the radix-4 SRT divider in ARM.

The recurrence equation for SRT division is:

$$Ri + 1 = r \cdot RiD \cdot qi + 1$$

and that for SRT square root is:

$$Ri + 1 = r \cdot Ri2Qi \cdot qi + 1qi + 1 \cdot r(i + 1)$$

where Rx is the remainder after the x iteration, r is the radix of the SRT algorithm, D is the divisor, qx is the xth digit of Rx, and Qx is the x-digit result computed after the xth iteration. In SRT division, the divisor is assumed to be in the range $1D < 2$ in keeping with the significand range of the IEEE floating-point standard. For consistency between the SRT division implementations, the root estimate is constrained to satisfy $12Qi < 2$, implying that the radicand must be in the range $0.25Ri < 1$. This range of radicand ensures that the exponent can always be even. To initialise the square root recurrence, q0 is forced to 1 so that if $Qi > u$, the redundancy factor defined as $qmax/(r1)$, the result is still obtainable. For minimumredundancy radix-4 SRT iterations, $u = 2/3$. These two equations are frequently combined into one unified expression by writing:

$$Ri + 1 = r \cdot RiFi \cdot qi + 1$$

where Fi is the appropriate value for division or square root derived from previous equations.

A block diagram of the divide and square root synthesisable macrocell is shown in figure 4.58, and Figures 4.59, 4.60 and 4.61 provide more details of the three critical logic blocks.

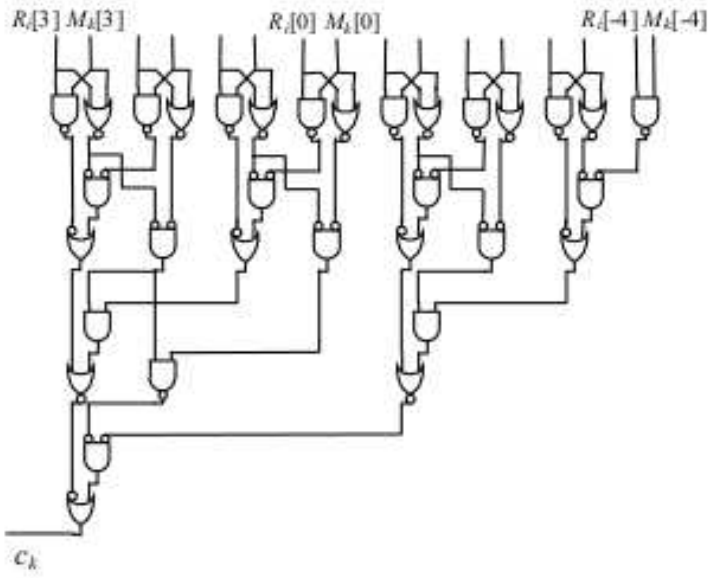


Figure 4.59: 8-bit comparator: ck = sign(Ri[3:-4] Mk[3:-4])

Before the first iteration, the four selection constants used to select qi+1, denoted Mk, are loaded into registers, and the D, Ri, Qi + and Qi registers are initialised with the appropriate values. Then, at each subsequent iteration, the top eight bits of the partial remainder, Ri, are compared with the four selection constants, Mk (Figure 2), and the four 1-bit results of these comparisons, ck, are combined to derive the value of the next result digit, qi+1, in a 1-hot encoding. In parallel with these comparisons, the five possible updated remainders R*i+1 = Ri Fk for k = -2 +2 are computed, but with the top 8 bits in non-redundant format, as shown in Figure 3. In this way, no 3:2 reduction is needed ahead of the Mk comparators on the next iteration, thus minimising the logic depth of the critical path through the comparators. As discussed in [4], these short carry-propagate additions across the m.s.b.s also have the effect of compressing the speculative signed-digit remainders, so that sign information is not lost when the top two bits of the remainder are discarded between iterations. Also as discussed in [4], the four possible updated subtrahends, Fk, (for k 2, 1, +1, +2 only) are formed by a circuit whose logic depth comprises a NOR gate driving into the data input of a 2:1 multiplexer. The multiplexer is needed to select the correct set of Fk values depending on whether a division or a square root operation is being executed. Finally, the new value of qi+1 as derived from the four values of ck selects the appropriate value of Ri+1 (in redundant format except for the 8 m.s.b.s) and the updated range estimates of the square root, Qi+1+ and Qi+1, and the iteration is complete (see Figure 4, where the logic that derives and concatenates the 2 l.s.b.s of the updated square root range estimates Qi+1+ and Qi+1 is not shown).

There were 18 stages of CMOS logic and buffers along the critical path (that ran through the qi+1 logic), which after allowing for clock insertion, translated to 142 ns per logic stage at 180nm. This was deemed near enough to the initial specification (of 15 CMOS stages) to be acceptable.

4.7 Registers, Counters

4.7.1 Registers

It is important to remember the basic sequential element shown in Figure4.62: the Flip-Flop, where a simple D-FF is sketched together with its time diagram. Remember that a FF is edge triggered (positive or negative) and thus it evaluates when a positive (negative) clock edge occurs and that a Latch is level triggered, that is it evaluates input during the half of the clock in which it stays high (low).

All of the Flip Flops and Latches have Reset and/or Preset. Apart from D-FF other FFs exist

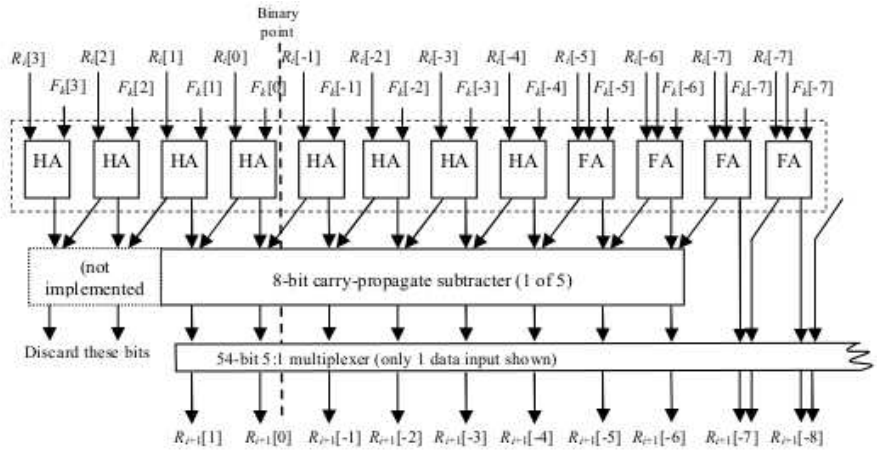


Figure 4.60: M.s.b.s of Ri+1 adder and 5:1 multiplexer.

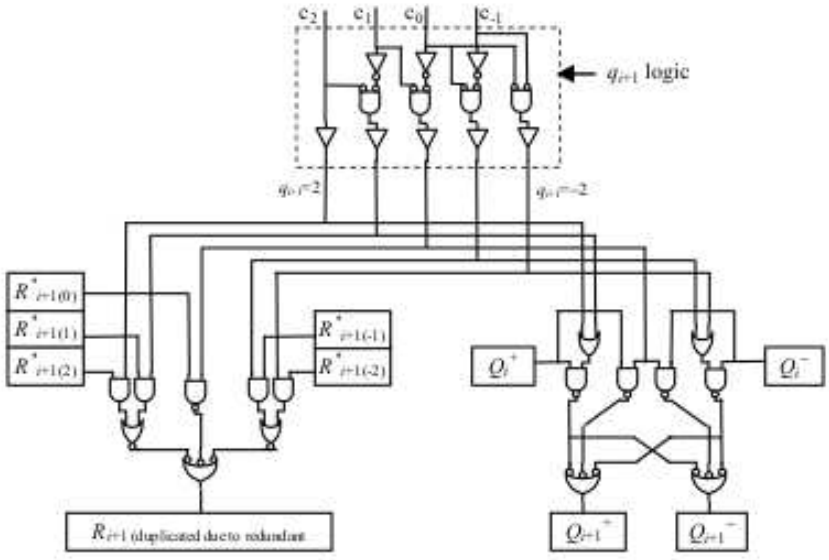


Figure 4.61: Logic diagram of qi+1 1-hot encoding and 5:1 multiplexers.

different for the function they perform (J-K, S-R, T...). In the following a few components based upon the reference FF are detailed.

SISO Register (Serial IN - Serial OUT)

This structure is also called shift-register. Basically at each clock cycle data are shifted from left to right or viceversa. The diagram of a SISO register is shown in Figure4.63.

For instance, let's assume there is a shifting pattern:1,1,0,1,0,0,0,0. Data will be stored in each flip-flop and available at the 'Q' port. There are four storage 'slots' available in this arrangement, let's imagine that the register holds 0000, means all storage slots are empty now. As 'Data In' presents the data with a pulse at 'Data Advance' each time. This is called clocking or strobing. The result is shown in the following table. The right hand column corresponds to the right-most flip-flop's output pin, and so on.

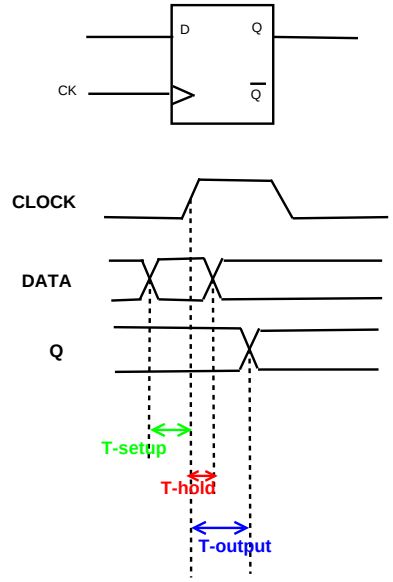


Figure 4.62: Flip-Flop

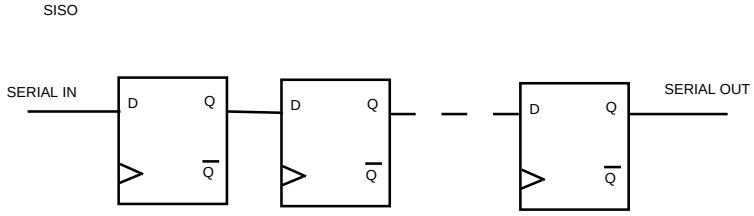


Figure 4.63: SISO register

0	0	0	0
1	0	0	0
1	1	0	0
0	1	1	0
1	0	1	1
0	1	0	1
0	0	1	0
0	0	0	1
0	0	0	0

So the serial output of the entire register is 11010000. As you may find, if we continue to input data, we would get exactly what be put in (true for D-ff only), but offset by four 'Data Advance' cycles. This offset is called "latency". This arrangement is the hardware equivalent of a queue. Also, at any time, the whole register can be set to zero by setting the reset pins high.

PIPO Register (Parallel IN C Parallel OUT)

In this case the evaluations are performed in the same moment for all the bits in the same time. The diagram of a PIPO is shown in Figure4.64.

PISO Register (Parallel IN C Serial OUT)

This structure "translates" data from parallel to serial. The diagram is shown in Figure 4.65.

In this configuration data are loaded in the register in parallel rising the $P/\overline{S}$ control line. To shift the data, the $P/\overline{S}$ control line is brought LOW. The arrangement now acts as a SISO shift register. However, as long as the number of clock cycles is not more than the length of the data-string, the

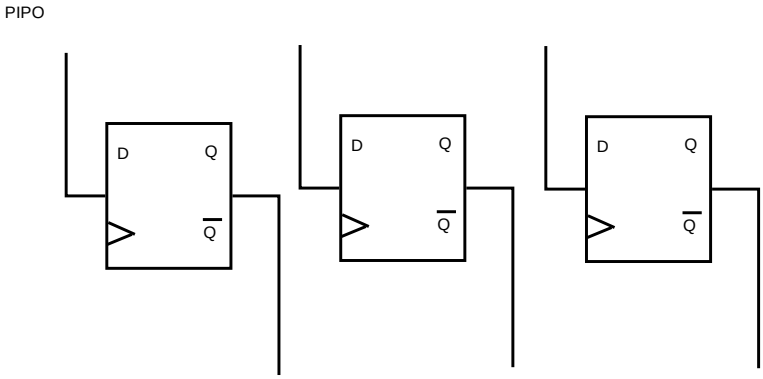


Figure 4.64: PIPO register

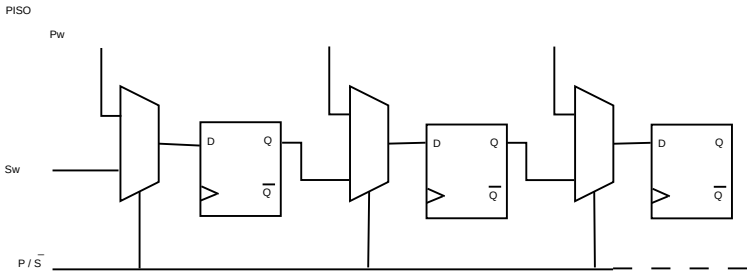


Figure 4.65: PISO register

Data Output, Q, will be the parallel data read off in order.

For example,use "011" as input from Pw,
Firstly,the registers are cleared,

0	0	0

Then we set the P/$\overline{S}$ to HIGH, write the parallel data into registers,

0	1	1

After that, we change the mode to shift and shift out data.

0	1	1	
0	0	1	1
0	0	0	1 1
0	0	0	0 1 1

SIPO Register (Serial IN C Parallel OUT)

The diagram is shown in Figure 4.66.

This configuration allows conversion from serial to parallel format. Data is fed serially, as described in the SISO section above. Once the data have been loaded, it may be either read off at each output simultaneously, or it can be shifted out and replaced.

Rotating Register

The schematic of Rotating Register is shown in Figure4.67.

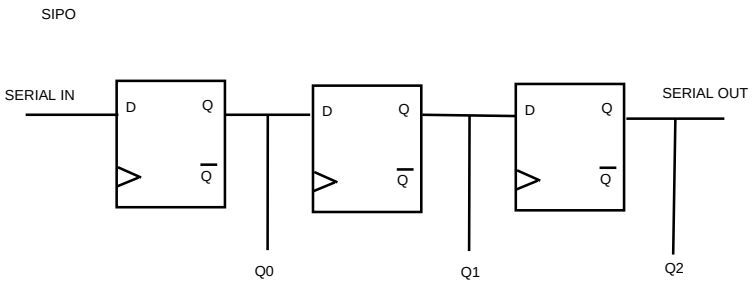


Figure 4.66: SIPO register

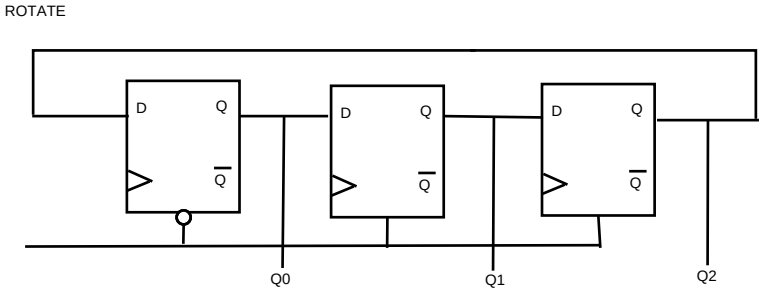


Figure 4.67: Shift Register

General Register

The schematic of an example of general register structure is shown in Figure4.68. It can be configured in order to perform different functions in this case mentioned in the following table.

Selection	Function
00	Parallel Entry
01	Mul or Div for power of 2 (2^2)
10	Serial Shift
11	Memory

LFSR (Linear Feed-Back Shift Register)

A linear feed-back shift register (LFSR) is a shift register whose input bit is a linear function of its previous state.

This is a rotating register, in which one of the Flip-Flops has a XOR as its input, an XOR among two or more outputs of the remaining Flip-Flops (the outputs connected to the XOR Gate are called **TAP**). There are two TAPs in the first graph of Figure 4.69.

The initial value of the LFSR is called "seed", and because the operation of the register is deterministic, the stream of values produced by the register is completely determined by its current (or previous) state. Likewise, because the register has a finite number of possible states, it must eventually enter a repeating cycle. However, an LFSR with a well-chosen feedback function can produce a sequence of bits which appears random and which has a very long cycle.

The circuit can be initialized with a different seed from Null vector.

In the example in Figure two of the three Flip-Flops are connected to the XOR, which is a input to the other Flip-Flop. Let's suppose that all bits are initialized to '1' after the reset. At each clock cycle the rotation continues and runs a sequence of pseudo random bits on the Flip Flop's outputs, which will be repeated at a given frequency. In this case the sequence will have a length of 7 as shown in the table.

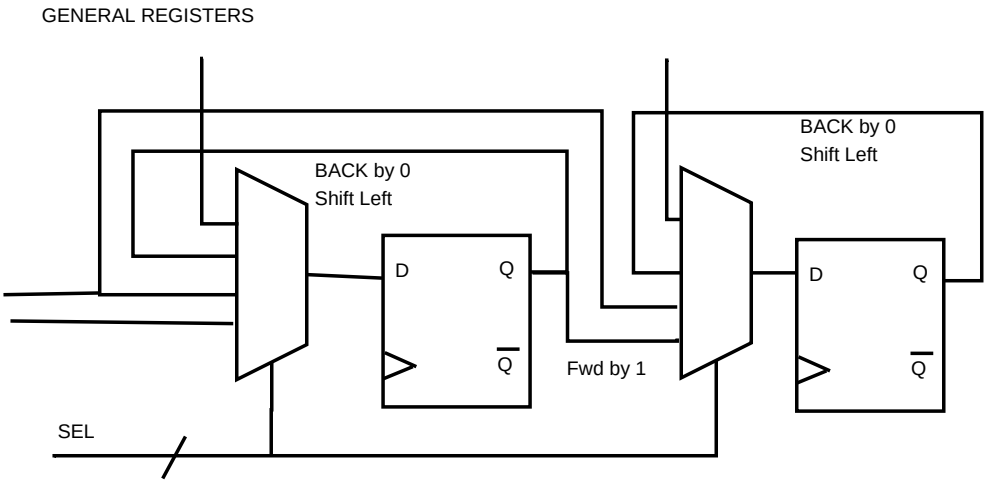


Figure 4.68: General Register

Q1	Q2	Q3
1	1	1
0	1	1
0	0	1
1	0	0
0	1	0
1	0	1
1	1	0
1	1	1

It can be demonstrated that the length of sequence is $2^n - 1$. The sequence is often associated to a polynomial where the terms different from zero are those with a position corresponding to the TAP. In this case P = $1 + x^2 + x^3$

Applications of LFSRs include generating pseudo-random numbers, pseudo-noise sequences, fast digital counters, and whitening sequences. Both hardware and software implementations of LFSRs are common.

4.7.2 Counters

Synchronous Counter T (or JK)

The schematic of one of the simplest synchronous counters is shown in Figure 4.70 and the related waveforms are in figure 4.71.

You can read Figure 4.71 by column. At each clock cycle, read the value of Q3Q2Q1Q0, which represent the value of the counter at this clock cycle.

In synchronous counter all flip-flops use the same synchronous clock signal. A counter is a state machine. Output counting is the Flip-Flop output.

If a counter has three bits the state table is:

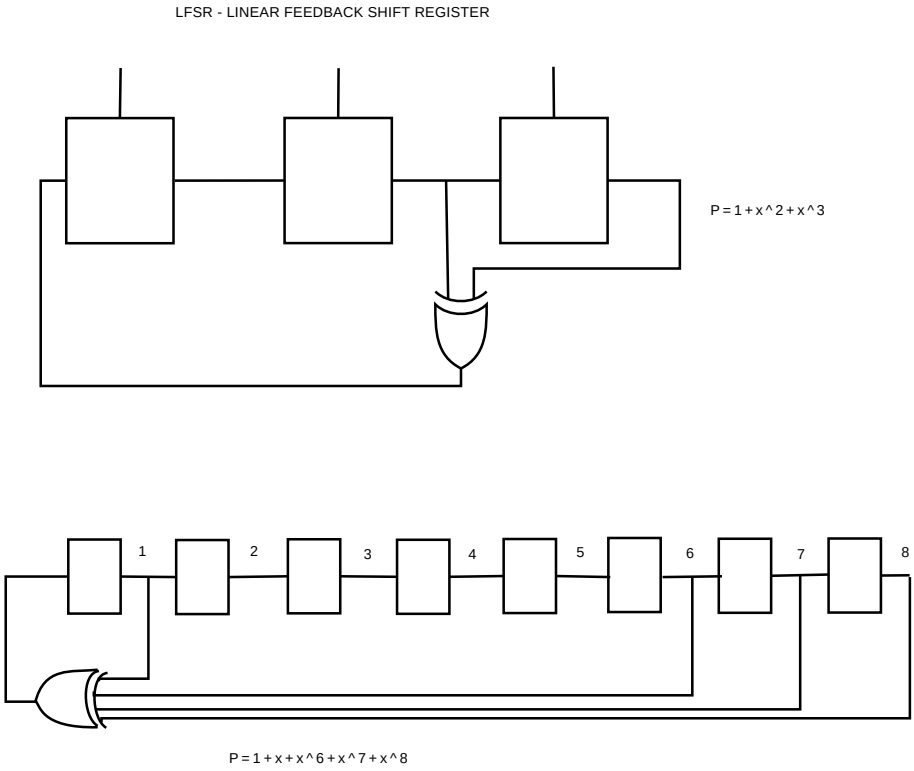


Figure 4.69: LFSR

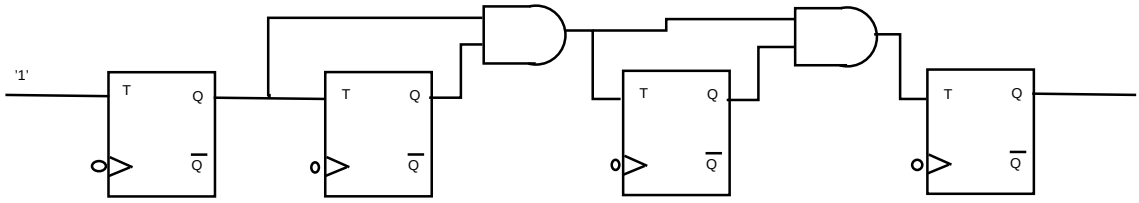


Figure 4.70: Synchronous counter.

Present State			Next State		
C	B	A	$C*$	$B*$	$A*$
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0

The critical point is the propagation of signals through the AND gates. In fact, the inputs to each AND gate are the outputs of previous ones, so the AND switches only when it propagates the signal on all previous AND, as shown in Figure 4.72.

Asynchronous Counter

The basic asynchronous counter is based on the concatenation of structures shown in Figure 4.73:

The output is a square wave signal with period twice the clock as shown in Figure 4.74.

With more Bit the structure becomes Figure 4.75:

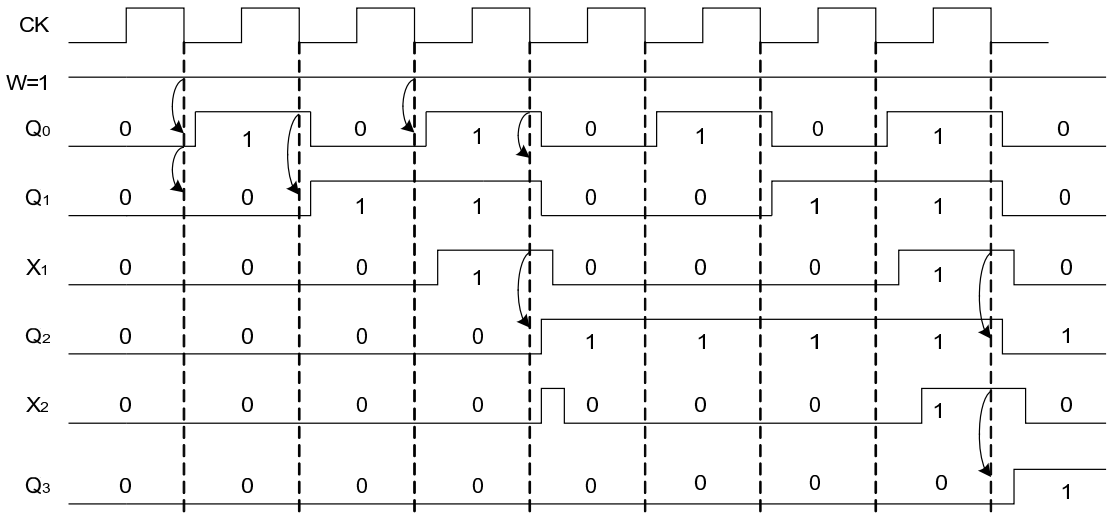


Figure 4.71: Synchronous counter's output.

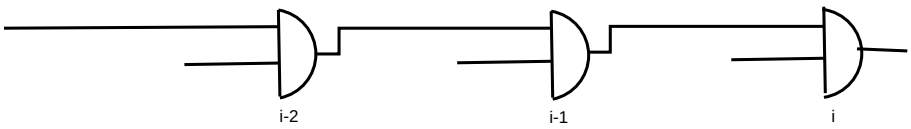


Figure 4.72: Cascading connection of AND in Synchronous Counter

where the output of the previous Flip-flop is the clock of the next.

In Figure 4.76, FF are positive triggered. For each Flip Flop output doubles the period of its clock, then, this is also meant as a **frequency divider**.

Reading Q outputs in parallel the circuit is a DOWN counter, on the contrary, reading $\overline{Q}$ it is an UP counter.

This counter is asynchronous in the as outputs are not synchronous with the clock and the delay in active progressive outputs of Flip Flop is:

$$\mathbf{n \cdot t(CK \rightarrow Q)}$$

where n is the number of the Flip Flop. If this delay is small with respect to the clock period, it can be used as a counter, otherwise it is not efficient in case a synchronous circuits is to used.

For the previous two structures it is possible to generalize and generate a UP / DOWN counter as shown in Figure 4.77:

Johnson Counter

Johnson counter is also called as shift counter. The output of the last Flip-flop goes to the input of the first Flip-flop after being inverted. The circuit of is shown as Figure 4.78.

This counter provides individual digit output rather than binary or BCD output.

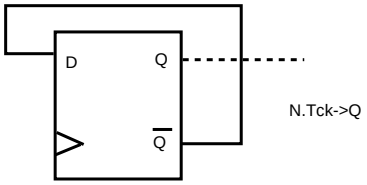


Figure 4.73: Basic asynchronous counter

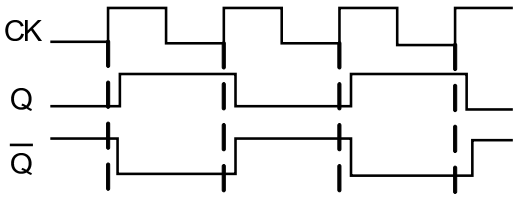


Figure 4.74: Graph of the asynchronous counter's output with more bit

Q0	Q1	Q2	Q3	count
0	0	0	0	0
1	0	0	0	1
1	1	0	0	2
1	1	1	0	3
1	1	1	1	4
0	1	1	1	5
0	0	1	1	6
0	0	0	1	7

When the output is 0000, it the digit count number is 0. In this way,the counter works as a decoder of digit numbers.

Half Adder Counter

The schematic of the Half Adder Counter is shown as Figure 4.79: The Graph of the Half Adder counter's output is shown as Figure 4.80:

The structure is very simple and regular. The glitch on SUM0, SUM1, SUM2 bits are masked by the Flip Flop. The propagation of the carry, however, is binding on the combinatorial and clock frequency. The glitch on carry can create problems as well: if they arise during the time between the Hold and Set-up of Flip Flop they can sample the wrong value.

For exercise, determine the critical path.

4.8 Register File

4.8.1 General structure

This is a structure around which the processing units in a microprocessor system are organized. It is composed by by a set of registers that can be read / written after the correct one can be accessed (address). It can have multiple access ports for reading / writing, each with an address decoder. The RF registers can be in the simplest case based on Flip Flop, while, in the high performance RF it is a real RAM with the necessary access ports read and write.

In typical applications (example in figure 4.81), there are two access ports in reading (with selection signals) and a access port for writing (with corresponding signals for the selection). The reason is related to the way the RF is connected to the Arithmetic Logic Unit (ALU), which usually have two

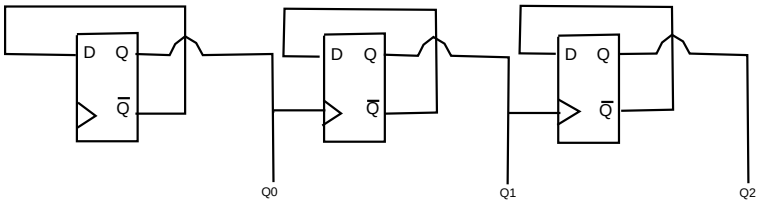


Figure 4.75: Asynchronous counter with more bit

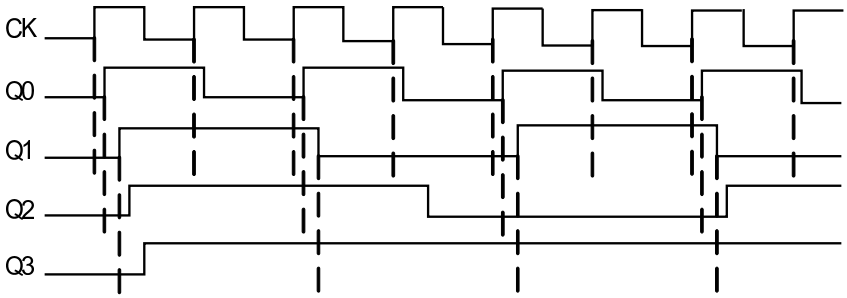


Figure 4.76: Graph of the asynchronous counter's output with more bit

ASYNCHRONOUS COUNTERS

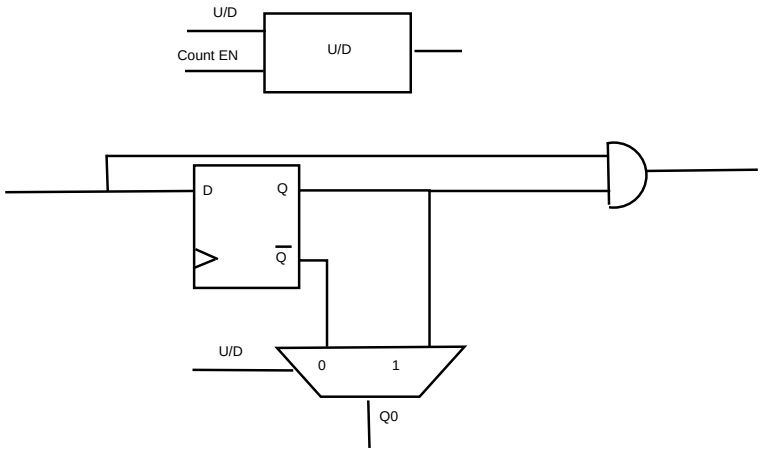


Figure 4.77: Schematic of a UP/DOWN counter.

JOHN JOHNSON COUNTER

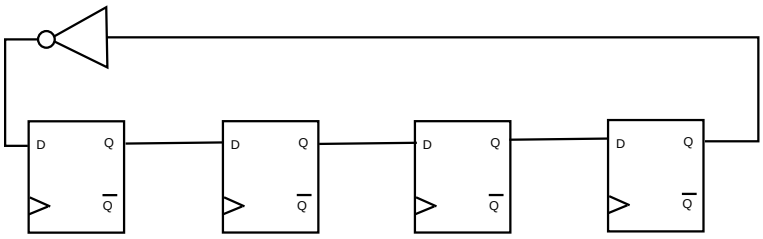


Figure 4.78: Johnson counter.

inputs and one output. One of the simplest possible structures for the reading selection is given in Figure 4.82.

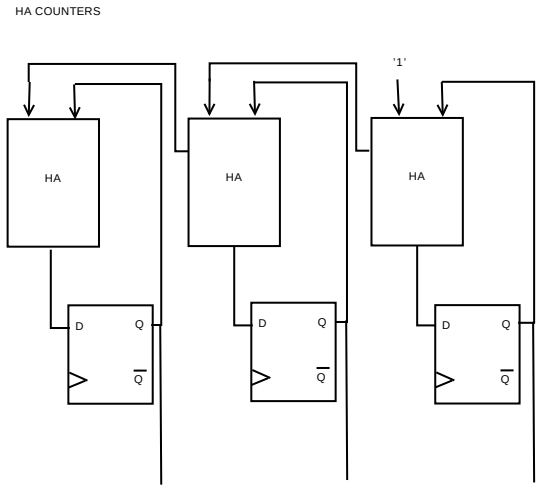


Figure 4.79: HA counter.

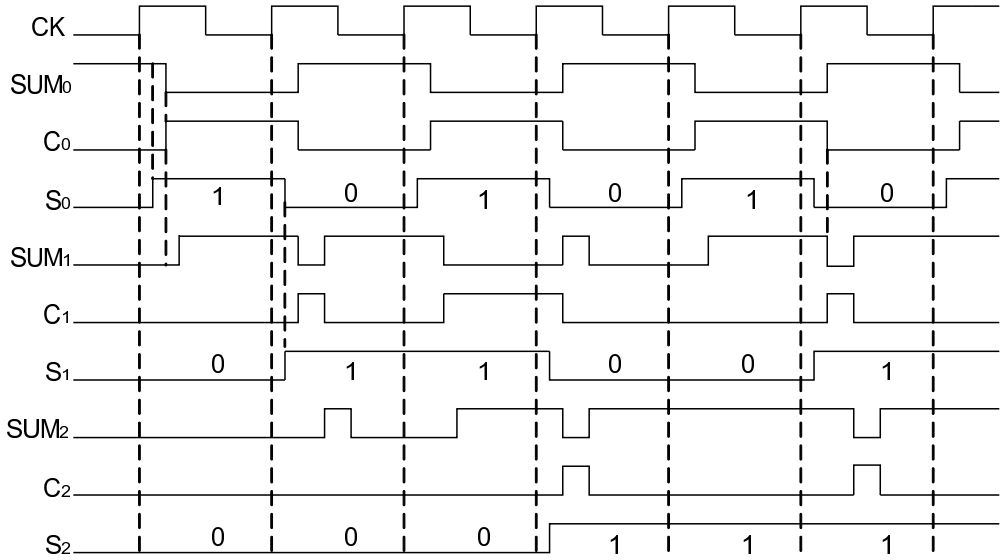


Figure 4.80: Graph of output of HA counter.

One possible implementation for the selection WRITING is given in figure 4.83

In Figure 4.84 two examples of simple 1-2 and 2-4 decoders are sketched. In the case of a large number of registers in the decoder is organized cascaded stages. The graph in Figure 4.85 shows the waveforms for the use of port input / output. Normally it is possible to write and read at the same clock cycle.

A simplified structure in the file register is the accumulator register. In current processors it is necessary to have records for at least 32 integers and 32 in the floating point. RF is at the highest level hierarchical memory, that is, the directly accessible memory. It can be also used for stacking or to saving all data in the event of a routine call (context switching), avoiding the expensive call or a memory cache. Thus, increasing the capacity of the RF is of broad utility. However, if it is too large:

- The output MUX output is complicated and has a bigger delay time

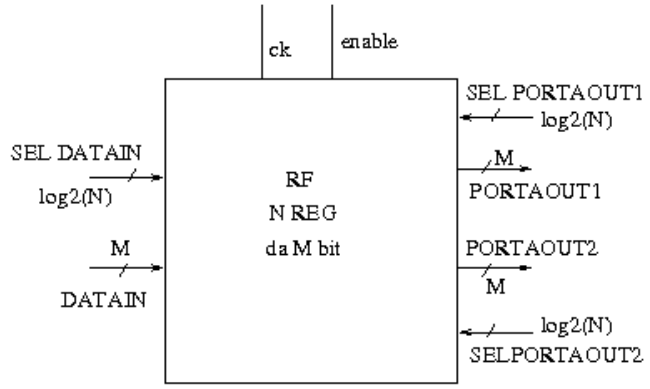


Figure 4.81: Register file general overview.

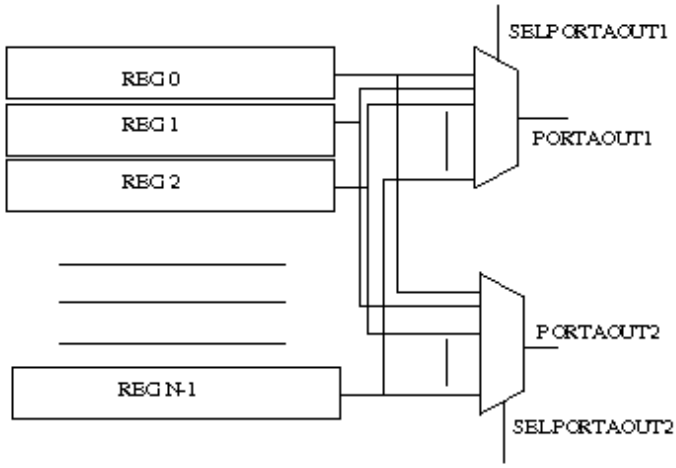


Figure 4.82: Register file output port.

- The decoder input is more complex and more time is needed for writing
- The address requires more bits and therefore requires longer instructions

An example: the MIPSR8000 has a RF with 32 registers of 64 bits, with 9 read ports and 4 write ports.

4.8.2 Register Windowing

A technique used to optimize overhead due to subroutine change context switching is the windowing one. The basic idea is to allocate different parts of the RF (windows) to different routines of the program to run. Two are the possible solutions classified as FIXED and VARIABLE windowing referred to window size. In the Berkeley RISC (fixed) the RF is composed of 64 registers, organized into 8 windows, and each of them has 8 registers. They are eligible and then up to 8 nested procedures using the same RF without using the memory for the stack.

The SPARC architecture uses fixed windowing with 8 registers dedicated to global variables, while all other organizations (number varies depending on the version of the SPARC) in windows of 3 blocks. In the most typical configuration each of them has 8 registers. An example is in figure 4.86. The first block is dedicated to the data inherited from the parent routine (IN), the second block is dedicated to variables local dedicated to routine activities (LOCALS), while the third block is dedicated to the variables to be passed to the child routine (OUT). When switching from a routine parent to a child, the active window shifts of 16 registers, so that the third block of the previous window (OUT) becomes the first block of the new window which contains its own data input (IN). A pointer stores

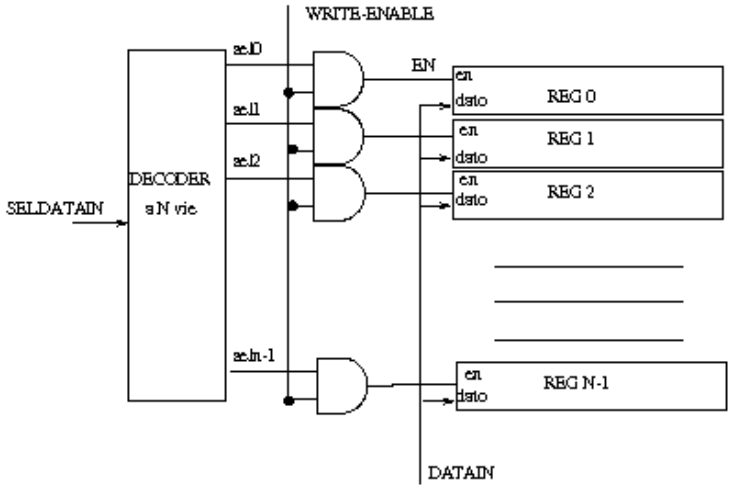


Figure 4.83: Register file decoder.

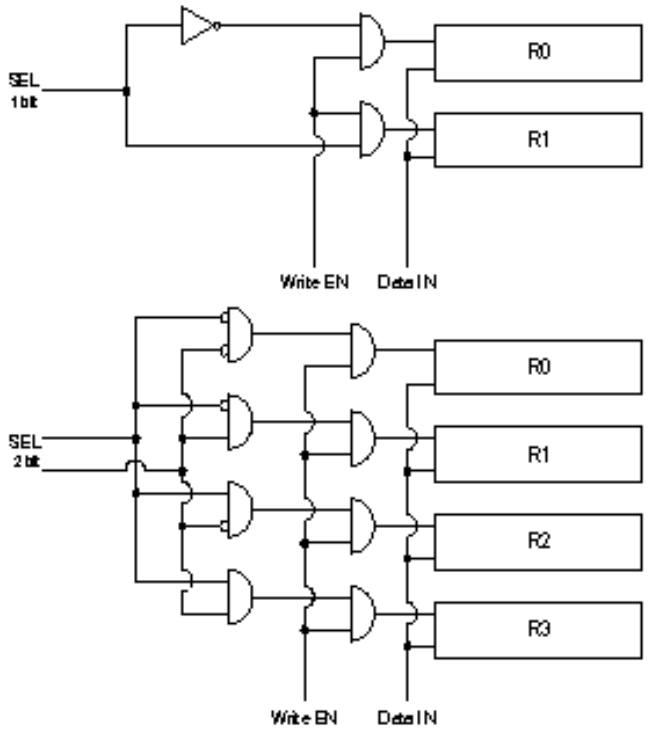


Figure 4.84: Register file decoder details.

the starting point of the current window (Current Window Pointer CWP) . This operation is iterated up to the number of possible windows. When the last window of available registers has been used then the mechanism is exploited again by spilling the oldest data in the RF (SUB0) in the main memory, and using the new available space. A specific pointer (SWP saved window pointer) stores the point starting from which data have been spilled. When a return from subroutine is to be executed, the active window shifts back from child to parent of 16 registers. In case the return imply the use of those old data that have been spilled, then a fill must be executed, and the SWP is used to know when this operation is necessary. In figure 4.87 the rotation of windows in the register file is sketched.

This structure is of simple implementation and present in all the modern RISC structure. The main problem consists in the absence of flexibility on the window size. A possible solution proposed is the VARIABLE windows one. It provides a structure where a pointer stores the starting point of

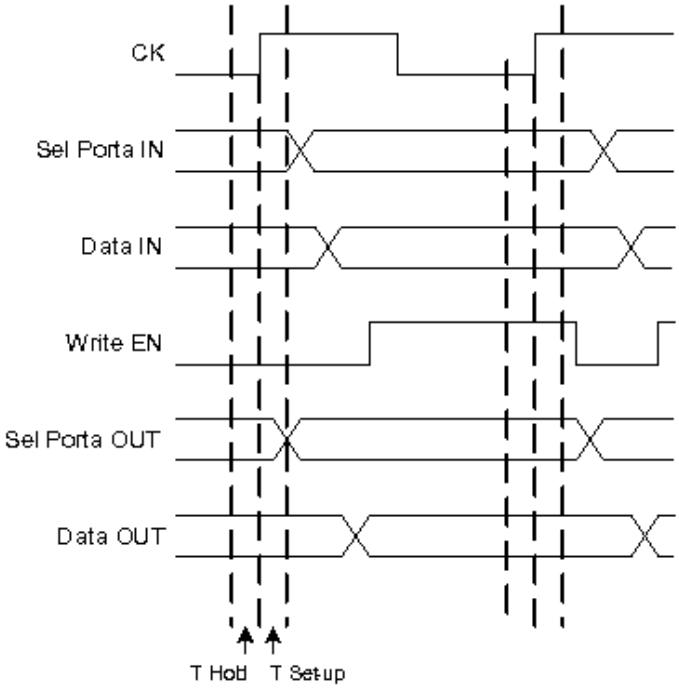


Figure 4.85: Register file signals.

current window (Start of Current Window), while another global register stores the offset to be added to the Start of Current Window in order to point to current register. This technique allows maximum flexibility and optimization of the capacity of the RF, but it is more critical in terms of performance as for each access it is necessary to execute a sum (offset + pointer).

4.8.3 T2 Register File

The integer register file of T2 provides the integer operands to EXU, LSU and FGU and stores the outcome of register-to-register instructions. A simplified structure is in figure 4.88.

The IRF contains 32 registers by 72 bits for each thread. Since the T2 core has two EXU, one for each 4 threads group, the IRF contains 128 registers (4 threads x 32 registers). These registers are implemented as an array of 128 registers called *active_window[127:0]*. This window is the only one accessible from the external core blocks. The 32 entries are split into 16 I/O registers, 8 local registers and 8 global registers.

WRITE and READ operations are possible between this window and the external blocks.

This *active_window* has a high speed bridge to the "shadowed" windows. The register file supports eight windows per thread, organized in local, I/O and global group of registers (each composed of 8 registers). A SWAP operation between registers in the active window for one thread and a shadowed window. It involves both SAVE (from active w. to shadowed w.) and RESTORE (from shadowed w. to active w.) operations pipelined internally and takes three cycles.

A SAVE is done by transferring the contents of the active register to one of the basic registers attached to it. Later when a RESTORE is done, the contents of this basic register are transferred back to the active register. The "in" register in an odd window becomes the "out" register in an even window and vice versa. Input register addresses must be translated to real register addresses before they are sent to the register file as shown in figure 4.89. In an even window, the input and real address are the same. In the odd window, address range [31:24] must be translated to [15:8], and the address range [15:8] will be translated to [31:24]. The global registers are the same across all windows. They are replicated to provide three sets of alternate global registers for use by privileged software only.

In summary, the IRF contains the following registers:

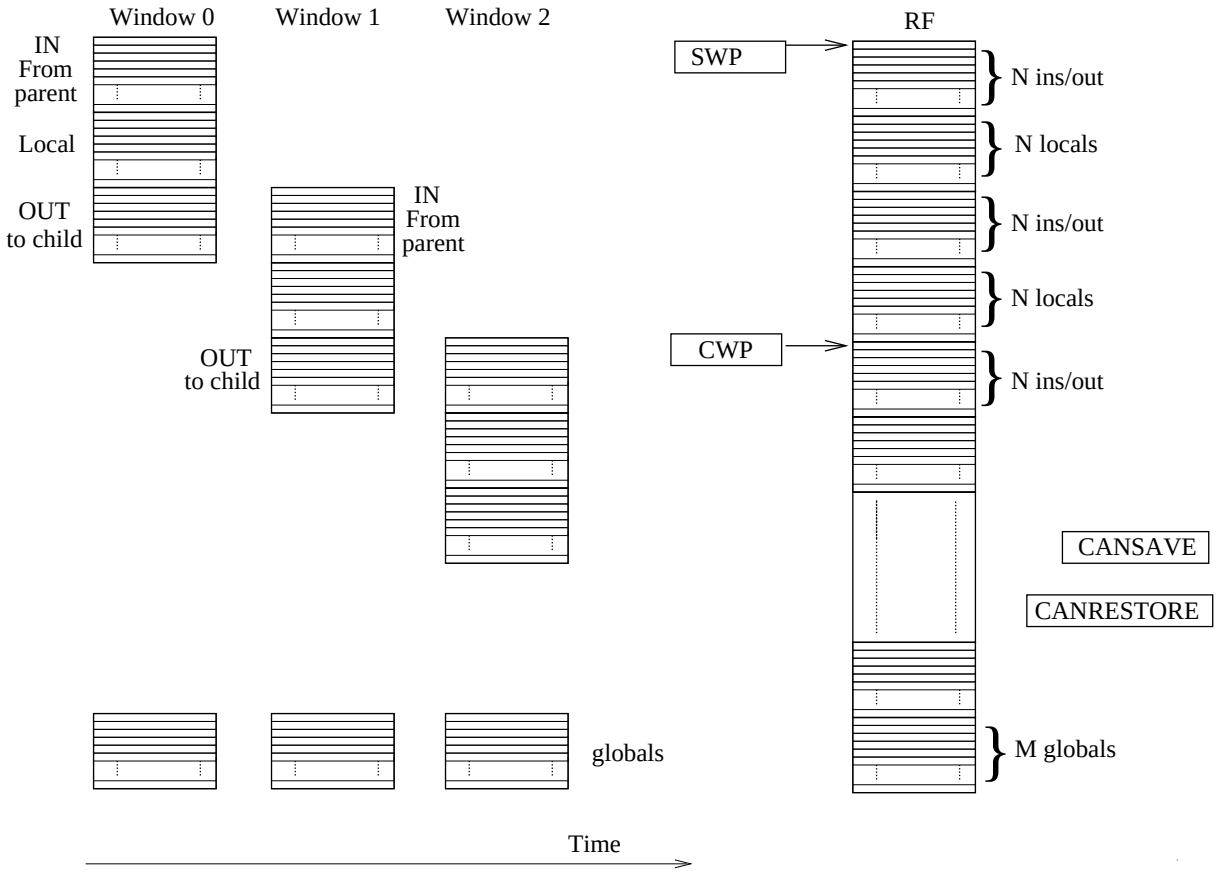


Figure 4.86: RISC-like windowing.

- 256 *locals[255:0]* (4 threads x 8 reg x 8 shadow);
- 128 *evens[127:0]* (4 threads x 8 reg x 4 shadow);
- 128 *odds[127:0]* (4 threads x 8 reg x 4 shadow);
- 128 *globals[127:0]* (4 threads x 8 reg x 4 shadow).

All registers are protected by error correcting code (ECC).

The IRF has 3 read ports and 2 write ports, with the respective enable signals. All three reads occur to the same thread, while the writes may be to different threads. During the normal access to IRF (*active_window[127:0]*), read and write operations are allowed simultaneously if read and write registers differ. When a read and write addresses match, the read output is set to X, like undefined value, while the write operation is discarded without changing the register content.

Read operations for each port are provided on the rising edge of the clock signal and allowed only if the enable signal is active high; while write operations are provided on the negative edge of the clock and if the write enables are active high.

As T2 works with interleaved multithread, for each clock cycle in one EXU, and thus one IRF, only one thread is active, that is, only the part of the active window related to that thread (THi) is being accessed by the external block. When this is happening, the access to the active window of the following thread (THj) is being prepared. In case it is known that the following thread will imply a subroutine call (or return), then the swap is performed for this thread (THj) while the READ or WRITE is occurring between THi active window and the external blocks. No SWAP is possible for the same thread in consecutive cycles.

The following figure (4.90) show the timing diagram for SWAP, READ and WRITE operations.

A SWAP instruction will send the current window pointer (CWP) and decoded SWAP (partial or full) or global control signals to the register file in E stage. The cancellation of the transition must

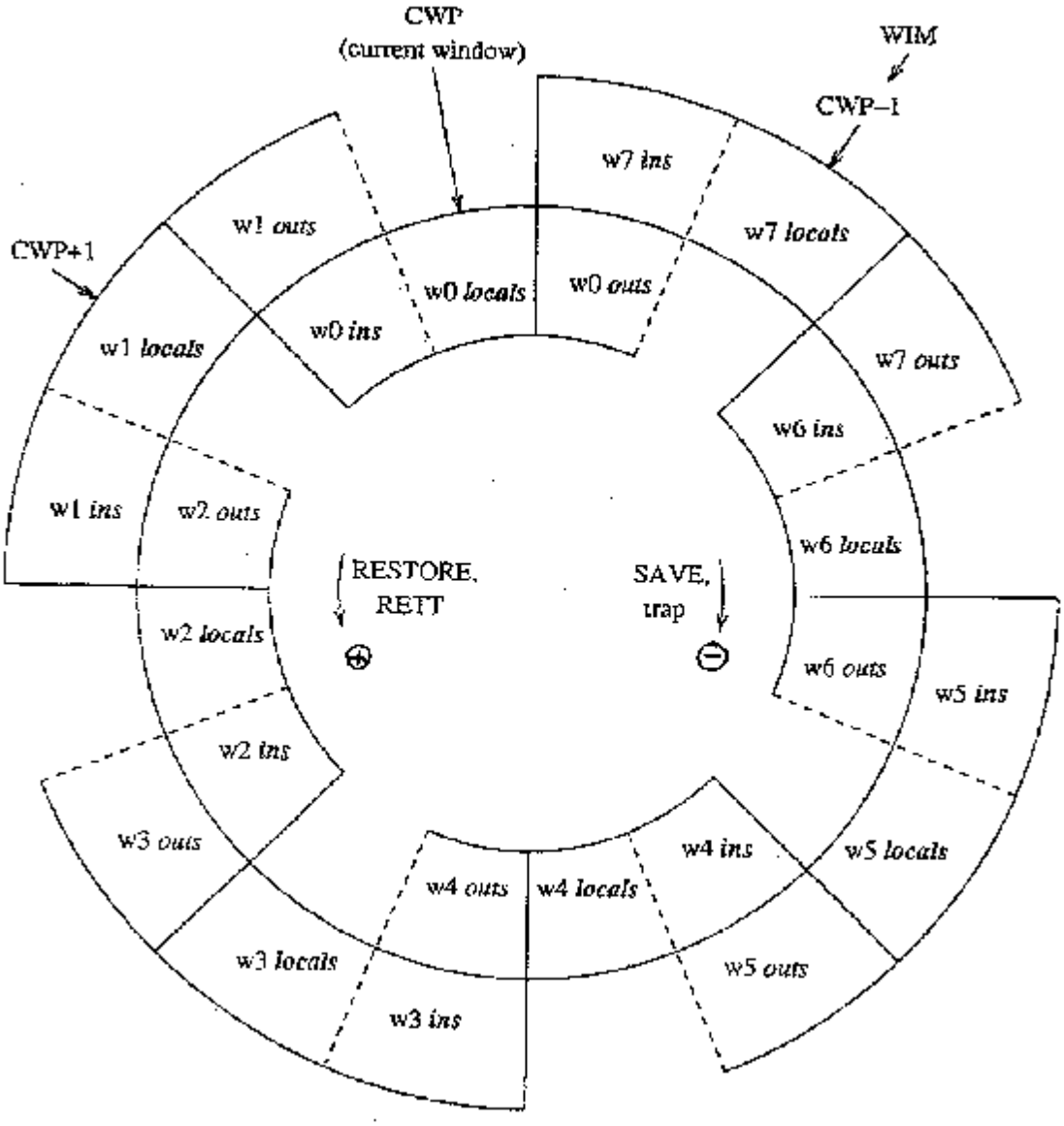


Figure 4.87: Conceptual windows organization in the SPARC RF.

be sent to the register file before the end of the W stage for the transition in the first (rising) phase of W2 stage and the switched window is available for read/write in the second (falling) phase.

The predecoded READ (WRITE) addresses and the associated thread are sent to the register file in the P (M) stage. Write enable will arrive late and set up to the falling edge of the W stage and have the (WRITE) operations done in the second phase of the (W) stage. Read occurs in the second phase of D stage.

4.9 Suggested readings

- Chapter 11 of J.Rabaey, A.Chandrakasan, B.Nikolic, "Digital Integrated Circuits", Prentice Hall
- Chapter 10 of N.Weste, D.Harris, "CMOS VLSI Design", Addison Wesley.
- Paper: "A 9-GHz 65-nm Intel Pentium 4 Processor Integer Execution Unit" (file P4integer65.pdf in Slides and Co directory)

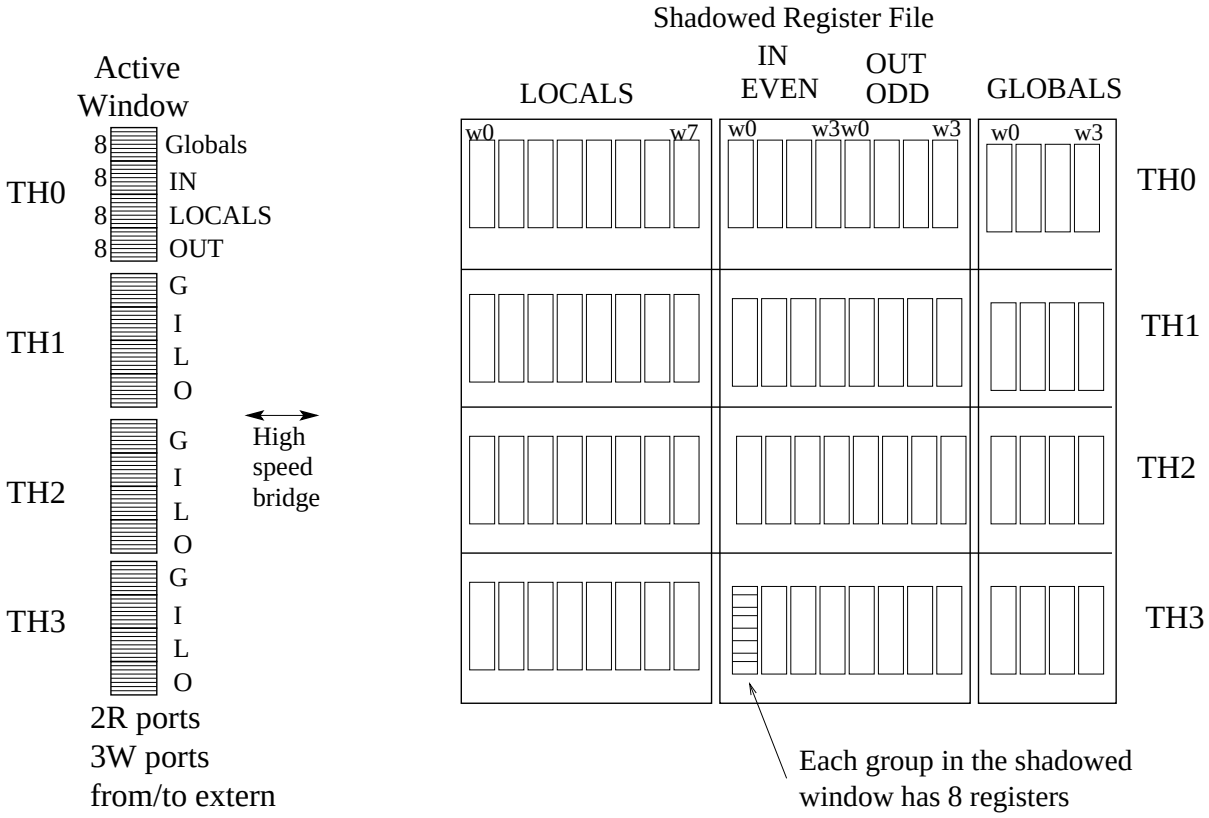


Figure 4.88: T2 simplified Register file

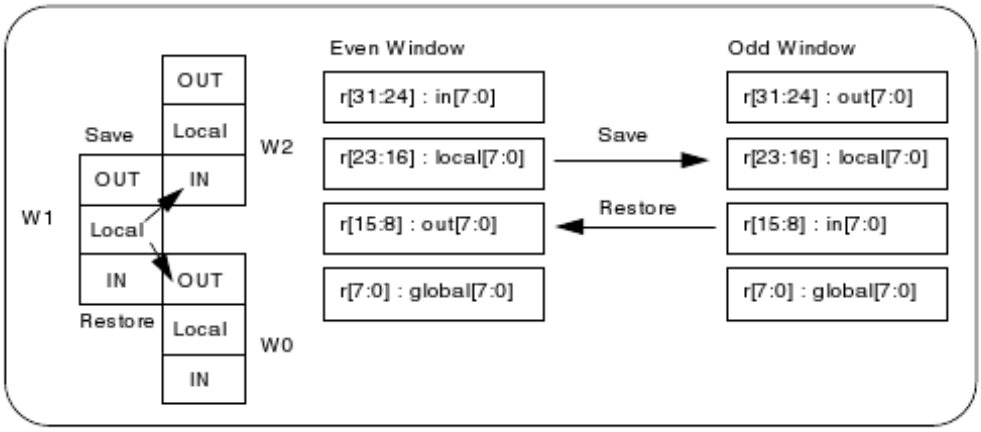


Figure 4.89: IRF window structure

- OpenSPARC T2 Core Microarchitecture Specification, (file OpenSPARCT2_Core_Micro_Arch.pdf in Slides and Co. directory)
- OpenSPARC T1 Core Microarchitecture Specification, (file OpenSPARCT1_Micro_Arch.pdf in Slides and Co. directory)
- Paper "Design of the ARM VFP11 Divide and Square Root Synthesisable Macrocell" (file FPunit-ARM.pdf)

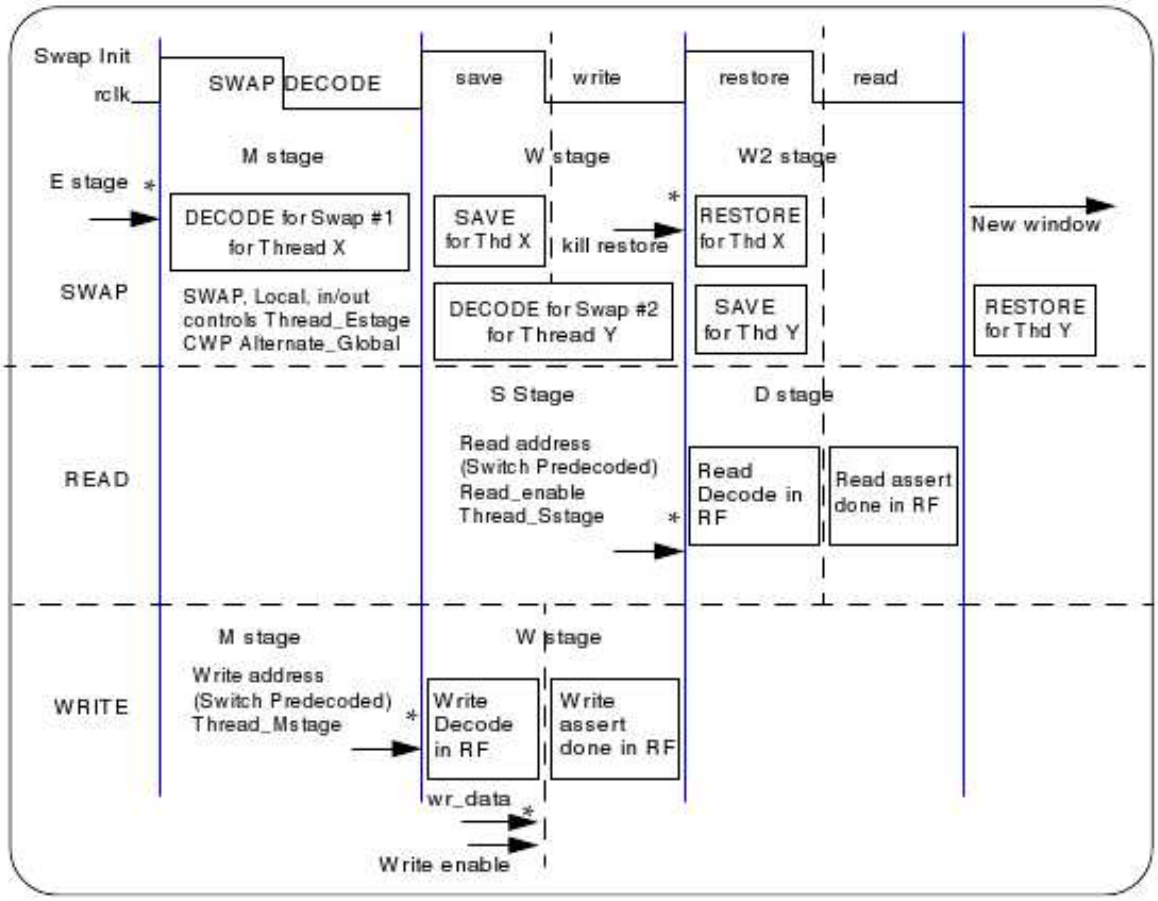


Figure 4.90: IRF timing diagram