Custom VLSI Design of a Programmable Digital Filter

Quinn Jacobson Eric Rotenberg

December 11, 1995

Abstract

A custom VLSI implementation of a 12-bit programmable digital filter is presented. The core of the filter is a 2-stage fused multiply-add unit. Our design has the following important characteristics:

- The organization results in a datapath with minimal latches for the function being implemented.
- The organization of the multiplier carry-save tree is optimized to minimize the total number of full adders, but more importantly to minimize the number of full adders in series (five).
- The organization takes advanatage of the fact that a fused multiply-add unit can be made faster than distinct, serialized multiply and add operations.
- To meet timing requirements and save area, the vast majority of the logic is implemented in dynamic CMOS; key circuit techniques include dynamic CVS, simple domino logic, skewed inverters, and a small amount of pseudo-NMOS in the critical path.
- The entire design was layed out by hand, resulting in a fairly tight, clean layout.

These combined features results in a *3430x2839 micron (9.74 sq mm)* design which can operate at a *15.2 MHz* sample rate.

1 Introduction

This paper presents a custom VLSI design of a 12-bit programmable digital filter, from architecture to layout. Section 2 gives the architectural definition of the filter, including the functions performed and the interface provided to the user.[1] Section 3 describes the organization of the filter: the datapath and pipeline timing are detailed in this section. In Section 4, the control logic required to sequence data through the unit is described. The core of the design – the multiply-add unit – is presented in Section 5: a more detailed view of the pipeline is given with dynamic clocking in mind, the carry-save tree is described in detail, followed by a discussion of the final carry-propagate design. Section 6 discusses the critical paths, which in turn determine the overall cycle time of the filter. The final section wraps up with a summary.

2 Architectural Definition

2.1 Function Performed

The function computed by the programmable digital filter is of the form:

$$u(kT) = a_0 e(kT) + a_1 e((k-1)T) + a_2 e((k-2)T), \quad (1)$$

for $k = 1, 2, \ldots$. The constants a_0, a_1, and a_2 are positive 12-bit numbers. The input sample $e(kT)$ is assumed to be a positive 12-bit number for all k. Further, the constants and inputs are such that $a_0 e(kT)$, $a_1 e((k-1)T)$, and $a_2 e((k-2)T)$ can also be represented in 12 bits.

In the event of *overflow*, bit 12 of the output will be set to “1”.

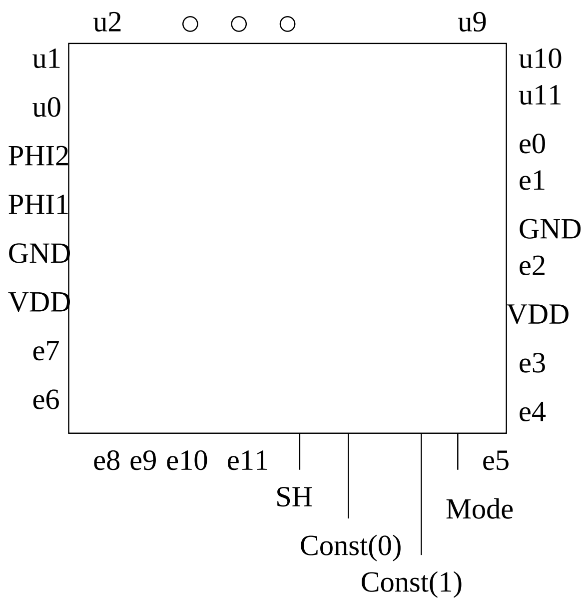

Figure 1: Pinout of the 12-bit programmable digital filter.

2.2 Interface and Modes of Operation

The digital filter has a 34-pin interface, shown in Figure 1.

The filter operates in one of two modes: *configure mode* (Mode = 1) and *run mode* (Mode = 0). The programmable constants are set during configure mode. In run mode the filter samples at a fixed rate, and at every sample produces a new output (corresponding to an earlier input sample).

In configure mode, the user can change the values for a_0, a_1, and a_2. The constant being changed is on Const(1:0) (refer to Table 1) and the new value is loaded from the bus E(11:0). The precise interface timing specification is given in Figure 2. The user must drive the Mode signal high off of the rising edge of ϕ_2 and hold it a full cycle for each constant. Const(1:0) and E(11:0) are driven with the same timing as the Mode signal.

Const(2:0)	Constant	Description
-00	—	All constants retain their previous values.
-01	a_0	The new value of a_0 is on E(11:0).
-10	a_1	The new value of a_1 is on E(11:0).
-11	a_2	The new value of a_2 is on E(11:0).

Table 1: Encoding of the Const(2:0) input during configuration mode.

Note that the original project specification indicates that the user should enter the values of sample $-T$ and 0 during configure mode, using special encodings of Const(2:0) to control this process. We instead redefine the "starting point", so that the initial two samples are loaded just as any sample is loaded – *during run mode. The only difference between the initial two sample points and arbitrary sample points is that no outputs are produced during the first two sample points.* In other words, there is nothing special about the first two sample points, so there is no need to load them during configuration mode.[2]

[1] The term *architecture* in the traditional sense refers to the interface between hardware and software, i.e. the view of hardware given to the programmer, or in our case the *user* of the chip.

[2] This does not violate the project specification, since full freedom is allowed in defining interface timing. Further, by moving the loading of the first two samples from configuration to run mode, we have eliminated an unnecessary special case. Note also that this makes bit 2 of Const(2:0) a don't-care.

In run mode, the filter continuously computes Equation 1. The filter activates a sample/hold (SH) signal once every T, indicating when the user is to place a new sample on the bus E(11:0) as well as when to latch the new output value from bus U(11:0). The precise run mode interface timing specification is given in Figure 3. The SH signal will go high for 1 cycle, from the rising edge of ϕ_1 to the next rising edge. This indicates to the user to place a new input on E(11:0) driven by the rising edge of ϕ_2 and held until the next rising edge. The output U(11:0) is valid during the second half of the SH cycle (falling ϕ_1 to rising ϕ_1). Therefore the user is to latch this output using the falling edge of ϕ_2.

3 Organization

Equation 1 can be divided into three smaller equations, as follows:

(2) $$out = a_0 e(kT) + temp_x$$

(3) $$temp_x = a_1 e((k-1)T) + temp_y$$

(4) $$temp_y = a_2 e((k-2)T)$$

From these equations we can make several observations. First, the above equations indicate in particular the need for an *accumulate* type of operation – that is, a multiply followed by an add. A fused multiply-add unit is an ideal candidate for this function. As will become more apparent in Section 5, bringing a third operand into the multiply process allows the accumulate step (the add part) to essentially proceed in parallel with the multiplication. That is, a fused multiply-add unit will outperform the combined latency of distinct multiply and add operations.

The second observation is that generating a new output requires *three* samples. It would appear that in order to compute outputs correctly, multiple sample values have to be kept around, requiring additional storage. However, this need not be the case with a good algorithm. During each sample period, three new values can be computed, but these values are all computed from the current sample (contrary to the three equations). One of these values will be used in the computation of the current output, the other two values will be used in the subsequent sample period. *In other words, the current sample period computes its own output as well as values contributing to the next two outputs.* This algorithm is depicted in Figure 4. It will be demonstrated in following sections how this algorithm can be implemented (1) without storage of temporary values ($temp_x$, $temp_y$) and (2) using a straightforward datapath.

3.1 Implementation of Algorithm

The algorithm is implemented using a two-stage multiply-add unit. Using a two-stage design is motivated by two factors:

- There is only a single multiply-add unit, yet 3 multiply-adds must occur during a sample period. Therefore a non-pipelined unit requires the latency of the unit to be $\frac{T}{3}$, which is not realizable in our technology (for our goal of T). A pipeline with two stages relaxes the latency requirement to $\frac{2T}{3}$. (Using even more stages could improve the throughput of the filter, but we found that the performance overhead of adding latches between stages is quite significant – up to 10% of a cycle.)
- The two-stage design lends itself naturally to the algorithm: the timing is such that a value is produced by one sample period exactly in time for it to be consumed by the next sample period, *so that these intermediate computations need not be stored in temporary registers.* Stated another way, the pipeline latches themselves store temp values on the fly. This leads to an efficient datapath (next section).

The timing alluded to is shown in Figure 5.

3.2 Datapath

A high-level view of the datapath is shown in Figure 6. It is "high-level" in that it shows the datapath in terms of master-slave registers. This is done for clarity; in reality we use dynamic logic, so a stage is actually composed of two substages, one clocked by phase ϕ_1 and one clocked by ϕ_2. However, conceptually the pipeline is only two stages deep since its latency is two full clock cycles. A more detailed datapath is given in Section 5. At that time we will also show tricks that can be used to eliminate latches feeding the top of the pipeline.

4 Control Logic

In configuration mode, all control is provided by the user via the Mode signal and the Const(1:0) bus.

In run mode, all control is provided by a 3-bit shift register. This shift register produces a one hot 3-bit value which drives the MUX selects, a "zero" signal used to zero the third operand into the multiply-add unit, the sample input latch enable, and the SH signal.

The shift register is implemented using 3 *pairs* of latches, each pair forming a master-slave latch. Because of the use of transparent ϕ_1 and ϕ_2 latches throughout the design (described in the next section), different control signals may need to be generated off of different clock phases. Therefore outputs are provided from both the master and slave latches of the shift register.

5 Multiply-Add Unit

The core of the digital filter is a two-stage pipelined, fused multiply-add unit (MADD). The unit performs a 12-bit by 12-bit multiply and adds in a 12-bit operand. However, as specified in the architecture, the unit produces only a 12-bit result; *this means only half of the multiplier array is actually implemented.*

While conceptually the pipeline is two stages deep, there are actually 4 half-cycle stages. This is because the pipeline uses a transparent latch design rather than a master-slave latch design. This facilitates dynamic logic.

5.1 Datapath

The detailed dataflow of the MADD is shown in Figure 7. The multiplier is implemented as a tree of full and half-adders, i.e. a carry-save adder tree. The carry-save tree is 5 full adders deep. The pipeline is divided into its four half-cycle stages as follows:

- *stage 1* – 1st and 2nd levels of adders.
- *stage 2* – 3rd and 4th levels of adders.
- *stage 3* – 5th level of adders, propagate and generate logic (preparation for carry-propagate).
- *stage 4* – carry-propagate logic, final sum logic.

The full and half adders are implemented using dynamic CVS logic, described in more detail in Section 5.4. The 4th stage features a dynamic Manchester Carry Chain and psuedo-NMOS XOR gates for the final sums, described in more detail in Section 5.5.

Figure 7 contains a lot of information, and so it should be mentioned how to read information from the diagram:

- The shaded region indicates static logic. Logic and latches outside of this region are dynamic, save some logic in stage 4.
- The number of logic elements and latches are indicated by a number in each block of the diagram. Adder stages have a pair of numbers; the first number in the pair is for full adders, the second is for half adders.
- Latches are labelled with either "phi_1" or "phi_2", indicating their clock phase.
- Special comments are included for the shaded region of the diagram; these comments are relevant for the next section.

5.2 Stage 1 "Tricks"

Here are some details that deserve special mention:

- Note that the first stage of the MADD pipeline does *not* actually include computing the bit-products. This computation is hidden in the precharge phase of stage 1. This is possible because the bit-product AND gates (and MUX) are static logic, hence they evaluate immediately after the rising edge of ϕ_1 (while stage 1 precharges).

- The third operand must be zeroed out when the a_2 multiplication begins. Similar to the previous bullet, it turns out that the AND gates used to block the low 5 bits must be static so that their delay is hidden in the stage 1 precharge phase. However, the upper 7 bits can use dynamic ANDs, because they feed into the second level of adders and therefore do not add delay to the critical path.

- Note that unlike the third operand, the multiplicand and multiplier do not feed into the first full adder stage directly off of a ϕ_1 latch. This would appear to be a timing violation. It is valid, however, due to two reasons. First, we gaurantee that the ϕ_1 clock driving the constant and sample input latches looks like the slave clock of a master-slave latch; therefore, the output of the latches appear valid at the rising edge of ϕ_1. Second, all logic between these latches and stage 1 of the pipeline is *static*. It is for these two reasons that the logic before stage 1 is "hidden" (during precharge), and why we can omit latches just before stage 1.

5.3 Carry-Save Tree

Rather than show the detailed array of full and half adders, complete with interconnections, we present the organization of the carry-save tree in a more compact form. Figure 8 shows the array of bits that must be added together at each stage of the pipeline. Grouped bits are added together using a full or half adder. The organization of the carry-save tree is optimized to minimize the total number of full adders, but more importantly to minimize the number of full adders in series (five).

Note that no adders are used in bit position 12 (indicated by shading in the diagram). If more than one "1" bit shows up in bit position 12, the result is *overflow*. Therefore, we need only OR all carry bits into position 12 to detect the overflow condition.

5.4 Full Adder Design

The full adder is perhaps the most important part of the design to get right because it is the building block for the largest part of the design. So it is important both in terms of area and performance.

We experimented with many full adder designs: many static variations, pseudo-NMOS, and dynamic CVS. We settled with a dynamic CVS implementation because it yields the best performance.

Most of the benefit of "dynamic CVS" comes from the "dynamic" part; the "CVS" part is incidental. This circuit technique produces both positive and negative outputs which can drive other dynamic gates, an essential requirement for the sum XOR logic. However, this could also be achieved by using two complementary domino gates. In fact, *a dynamic CVS gate is essentially two complementary domino gates merged into one.* The merging sometimes yields a reduction in the total number of n-transistors required.

Dynamic CVS – as with all dynamic logic – yields better performance over static CMOS due to the elimination of the p-transistor pullup network.[3] We combined two other circuit techniques with dynamic CVS to improve the design:

- The outputs of dynamic CVS (as with domino logic) go through static inverters. These inverters can be *skewed* to improve the speed of the circuit. In other words, we can take advantage of the fact that only a single transition is possible in the evaluate phase, and speed up this transition. The p-transistor of the inverter can be made significantly larger than the n-transistor, making the inverter switch quickly on an input transition of 1 to 0. (This may be at the expense of noise margins.)

- Dynamic logic suffers from charge-sharing. We chose to deviate from traditional CVS by precharging all internal nodes high, thus eliminating charge-sharing.

5.5 Carry-Propagate Adder Design

The final stage of the multiply-add unit requires a carry-propagate adder of some sort (since we cannot carry-save forever). Our multiplier design requires only a 6-bit adder (the 5 low order bits are in their final form after the carry-save tree, and bit 12 is a cumulative OR for overflow detection, as previously described).

[3] However, dynamic logic in this case requires double the n-transistors, in order to generate complementary outputs. Still, n-transistors are better than p-transistors.

5.5.1 Dynamic Manchester Carry Chain

Because the carry-propagate adder is only 6 bits wide, we decided to use a single dynamic Manchester Carry Chain to generate all carries. The only other serious candidate is a distinct carry lookahead gate per carry.

We saw no major motivation for using separate carry lookahead gates to generate the six carry signals independently. A dynamic Manchester chain is essentially a merging of the separate CLA gates. The only possible downsides of this merging are:

- Each internal node must be precharged. This means a p-transistor hanging off every internal node, possibly slowing down the evaluate transition due to additional diffusion capacitance. This effect is definitely noticeable, and can be observed by experimenting with the sizes of the precharge transistors.
- Propagate signals *must* be generated by XORs rather than ORs. This disadvantage has no impact in our design since propagates are computed in a previous stage.

The obvious advantage of a single Manchester chain over separate CLA gates is reduced area.

The delay through the carry chain is non-trivial, because of the long propagate "stack" (7 n-transistors in series). A common transistor sizing technique is therefore required: increased transistor widths further down the stack. This design results in a triangular layout.

5.5.2 Final Sum Logic

The final sum bits can be produced after the carries are generated by the Manchester chain. The sum is an XOR of the propagate bit and the carry bit. Because the carry arrives late in the cycle, the XOR must be quite fast. Unfortunately, a dynamic implementation of the XOR is not an option since we do not have complementary dynamic carry outputs from the Manchester chain (this would require duplicating the chain to produce the negative carries). Static XOR implementations perform poorly. Thus we turned to pseudo-NMOS techniques to drive performance.

The pseudo-NMOS XOR gate is shown in Figure 9. Pseudo-NMOS was chosen because it is the closest we can get to dynamic performance (elimination of p-transistor network). Further, by observing characteristics of the inputs to the XOR gate, the transistors can be sized in such a way to improve performance even further:

- *behavior of propagate signals* – the propagate signal and its complement come directly off of a latch, and so are constant throughout evaluation.
- *behavior of carry signals* – $\overline{C}$ is precharged high in the Manchester chain, and thus the only possible transition the XOR gate will see is from 1 to 0; the opposite holds for C.

When P is high, the value of $\overline{C}$ determines the XOR output; since $\overline{C}$ can only transition low, only a 0 to 1 transition is possible at the output node. This transition can be sped up if the left pulldown path *yields* quickly to the p-transistor during its release. This is achieved by sizing the left pulldown path smaller than the pullup. When P is low, the value of C determines the XOR output; since C can only transition high, only a 1 to 0 transition is possible at the output node. This transition can be sped up if the right pulldown path is made much stronger than the p-transistor.

These optimizations have significant impact on the performance of the final sum logic, and are therefore implemented in this design (at the cost of static power). However, the sizing must not be "overdone" to the point that the output voltage is in a grey region (neither high nor low) for the non-transitioning cases.

6 Performance

In this section we present the critical paths in the design which will determine the overall cycle time of the filter.

6.1 Stage 4: Carry-Propagate Adder

The final carry-propagate stage of the design is the worst critical path in the design. This is in spite of the fact that the propagate and generate signals are computed in the previous stage and available off of latches. Figure 10 shows the path. Two test patterns stress the path. Both patterns have all generates set low except for the lowest generate bit $G0$. All propagates from $P1$ to $P4$ are set high. This combination of generates and propagates causes a single carry to be generated out of the low-order bit and propagated through the chain. One test pattern sets $P5$ to 0 and the other sets it to 1. Basically we are testing the critical path to the high order sum bit, both rise and fall propagation delays. The low-to-high propagation delay is 10.94 ns, and the high-to-low propagation delay is 10.89 ns. This sets a half cycle at about 11 ns, resulting in a cycle time of 22 ns and a sample period T of 66 ns.

Two diagrams are attached to this paper which show the *Lsim* timing verification of this path.

6.2 Single Stage of Carry-Save Tree

The next cycle time limiting path is the delay through any of the other pipeline stages, i.e. the delay through 2 full adders followed by a dynamic latch, the latch being loaded by a full adder.

A single *Lsim* diagram attached to this paper shows that this path evaluates and precharges well within the 11 ns half-cycle limit imposed by the carry-propagate adder.

7 Summary

To summarize, we have implemented a 12-bit programmable digital filter with an area of 9.74 sq mm and a sample rate of 15.2 MHz.

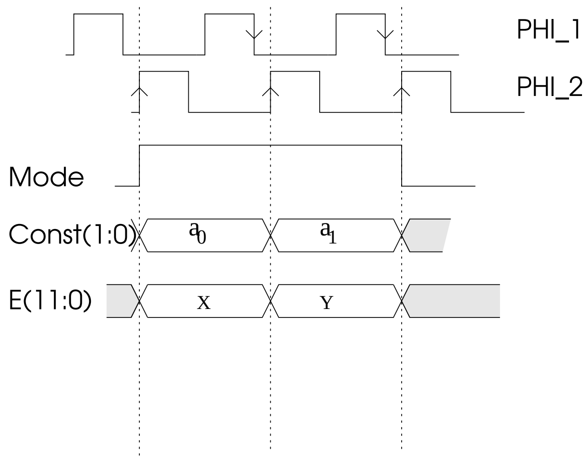

Figure 2: This diagram shows the interface timing for configure mode.

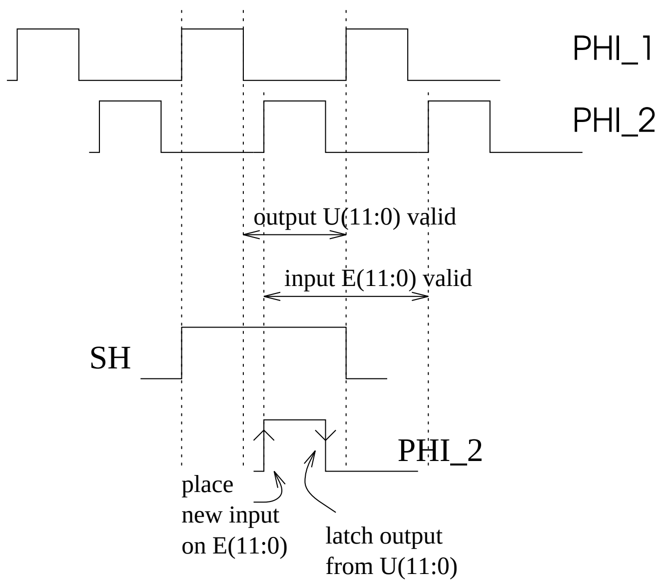

Figure 3: This diagram shows the interface timing for run mode.

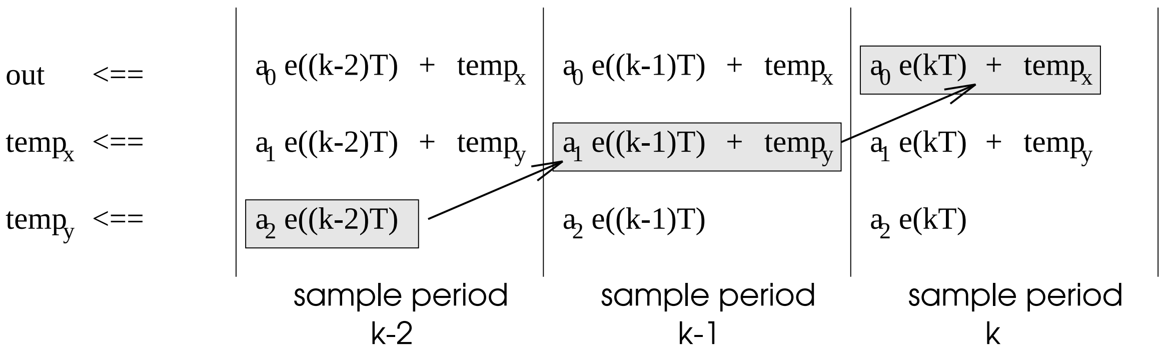

Figure 4: This diagram shows the algorithm for implementing the filter function. The key idea is to perform all computations that use a sample value during that sample period, so that samples don't have to be saved across sample periods. The result is that two of the computed values must be passed to the subsequent sample period.

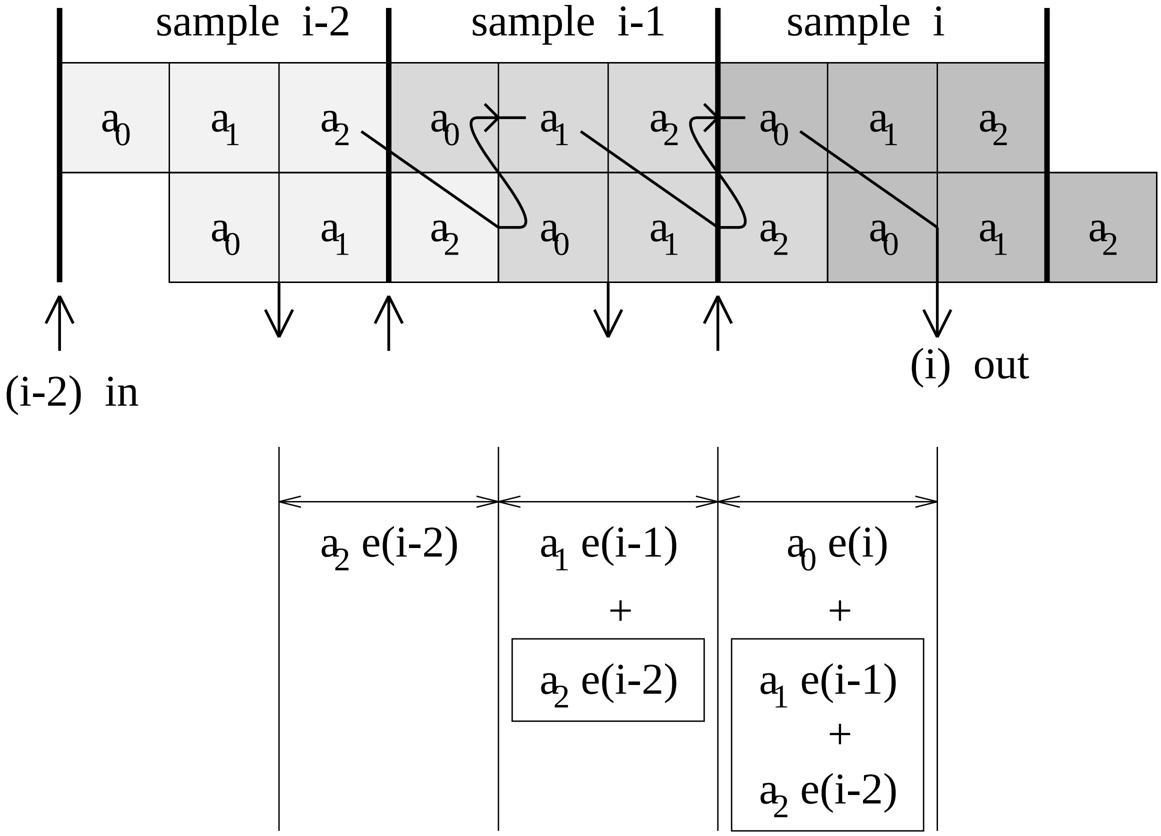

Figure 5: This timing diagram shows how multiply-add computations are sequenced from one sample period to the next; after two such producer-consumer transfers, a result is output.

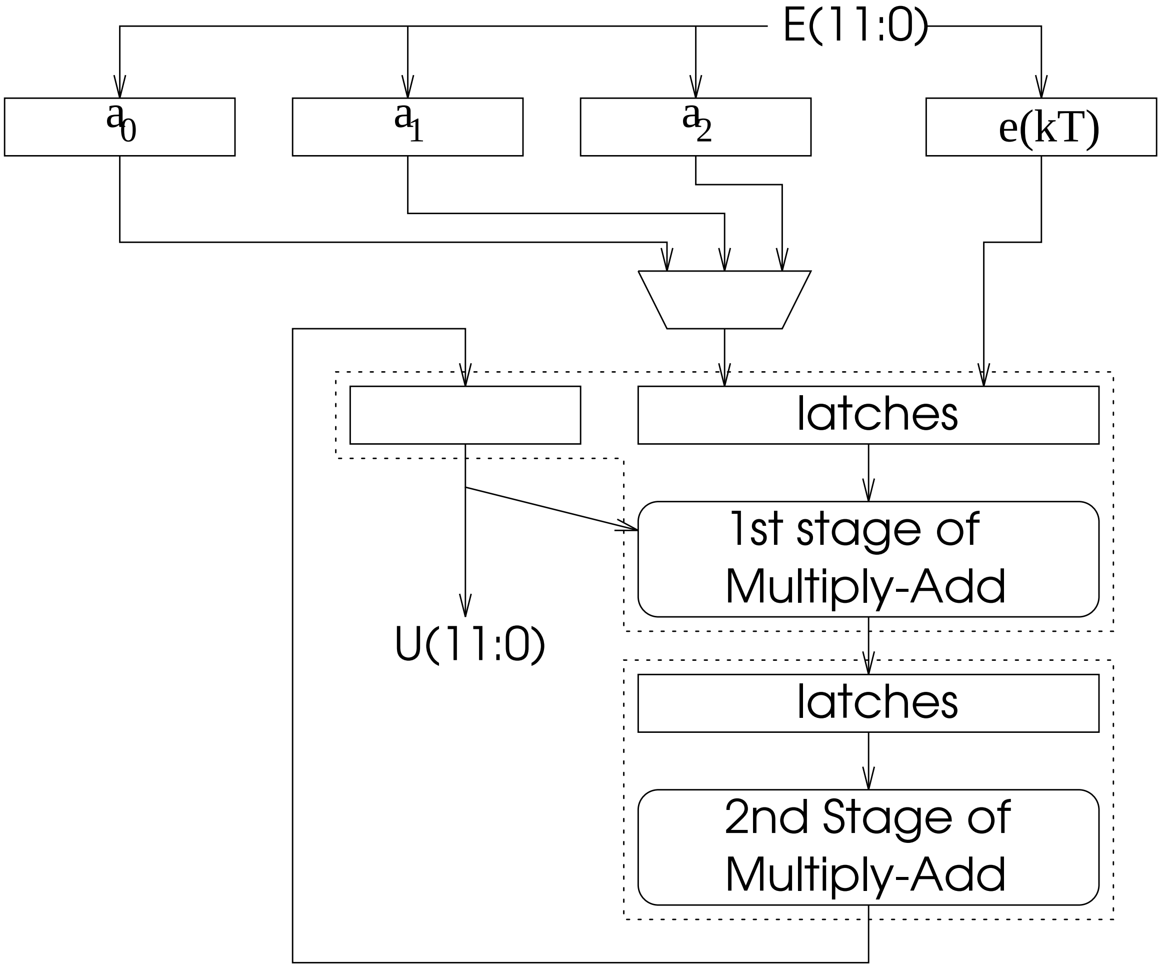

Figure 6: High-level view of the datapath depicting the two-stage design.

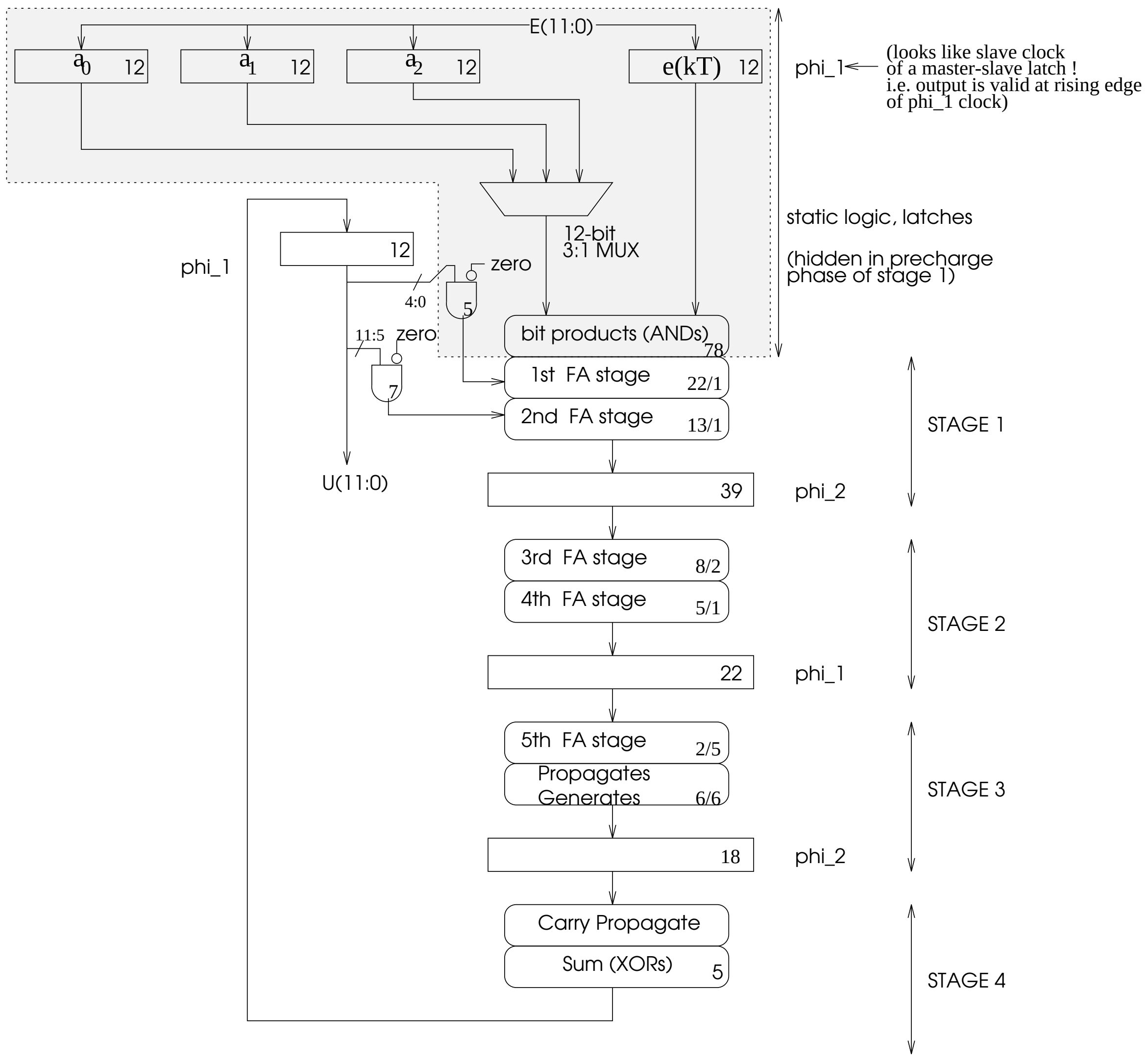

Figure 7: Detailed datapath of the multiply-add unit. (1) The shaded region indicates static logic, (2) element counts are shown, (3) latch phases are indicated to the side.

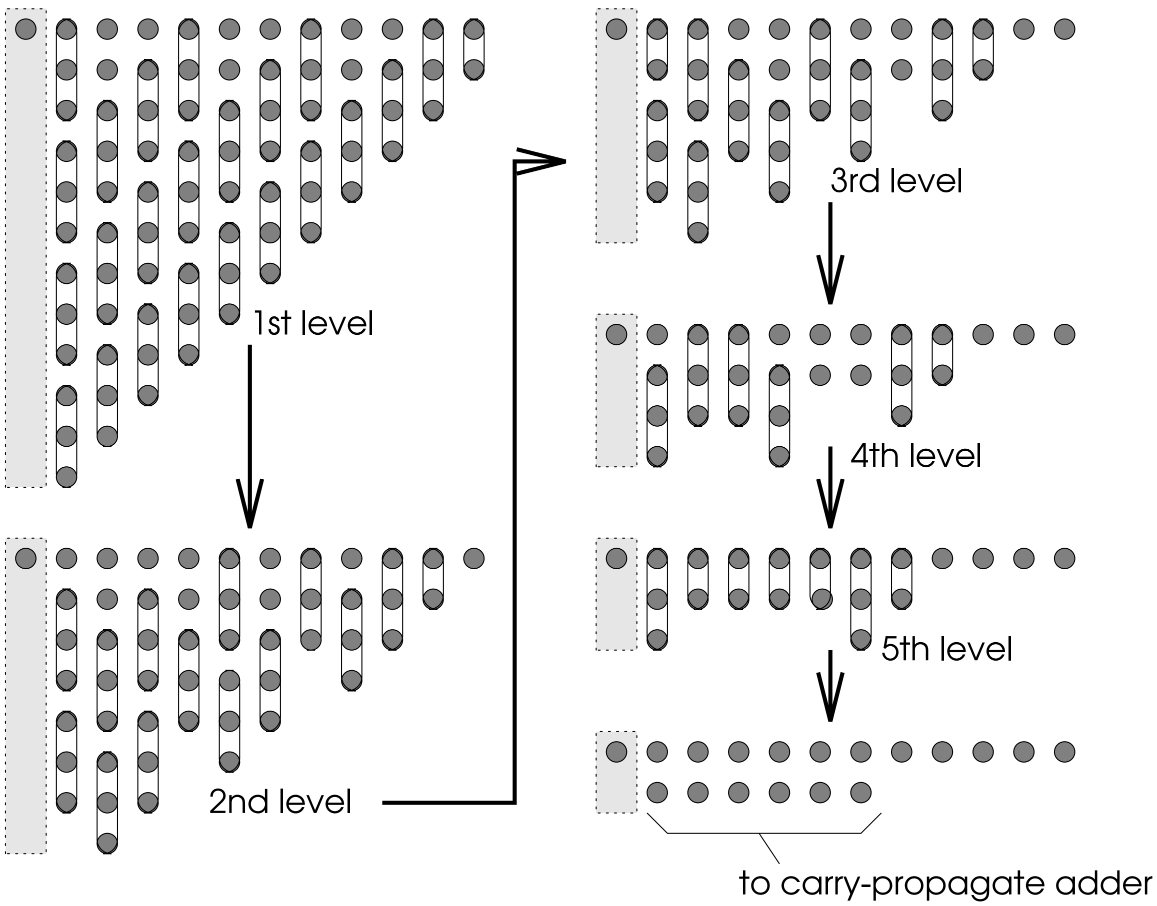

Figure 8: Organization of the carry-save tree, presented in compact form. Dots represent bits to be added. Grouped bits are added together using a full (three) or half (two) adder. All carries into bit position 12 are ORed together to detect overflow; there is no need for any adders in this bit position.

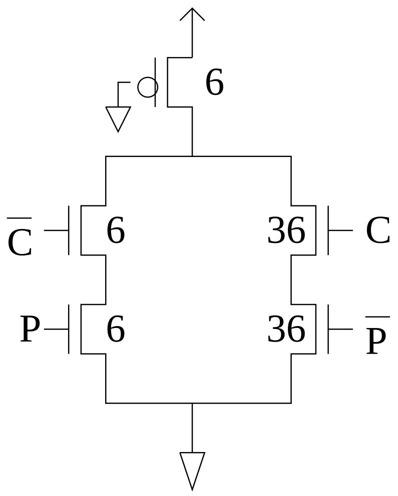

Figure 9: The pseudo-NMOS XOR gate used to produce the final sum result. When taking into account the behavior of the inputs to the gate, independent sizing of the two pulldown paths with respect to the pullup can be quite effective in improving the speed of the circuit.

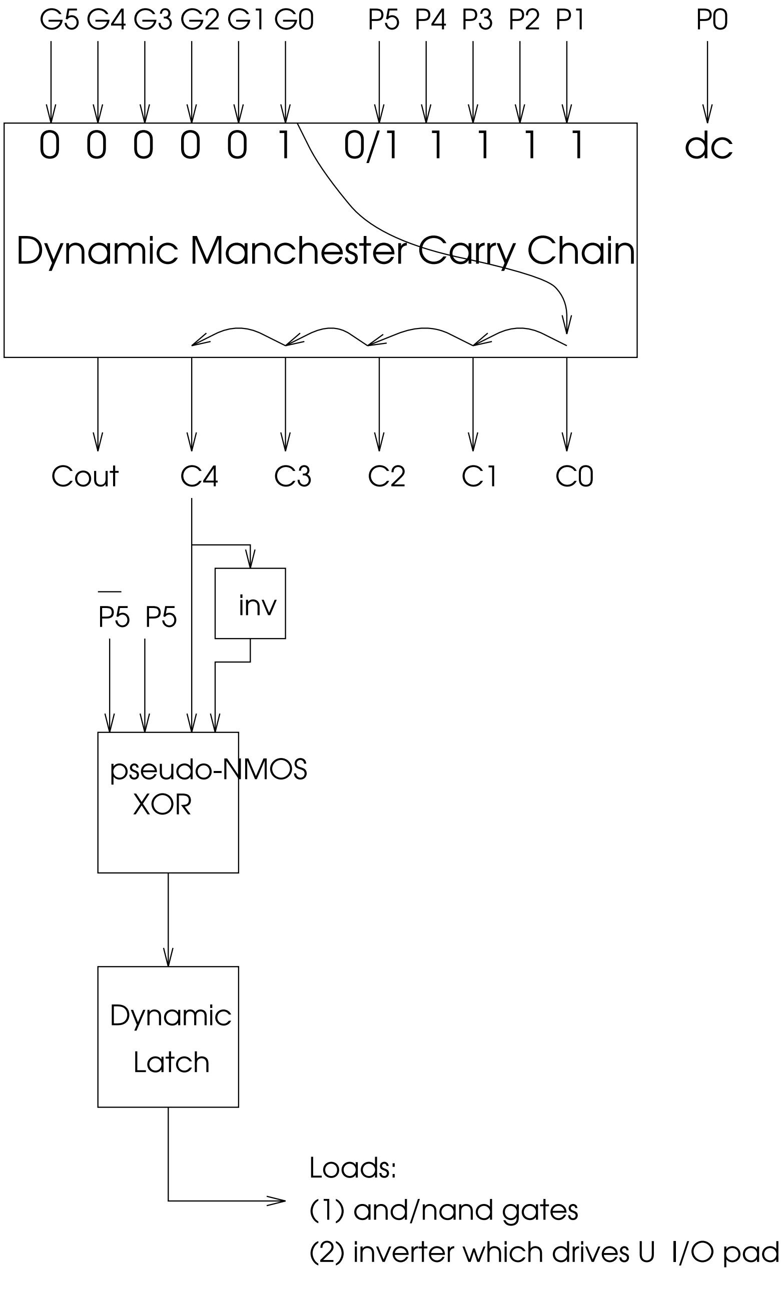

Figure 10: The worst critical path is through the carry-propagate adder.