Data slicing arithmetic operators

Bernard Goossens[1] Duc Thang Vu[2]
IBP-LITP,
Université Paris 7 Denis Diderot,
2 place Jussieu,
75251 Paris cedex 05,
France

Abstract

We present a set of data sliced integer operators: an adder, a multiplier and a divider. The main contribution of our proposal is to further reduce superpipelined processor cycle width, today fixed by the 64 bits integer arithmetic operations critical path. Computations in our architecture are handled via four pipe stages, with an even critical path based on a 16 bits carry select adder for the addition operator, and 4 levels of one bit carry save adders for the multiplier and the divider step operators. Multiplication step, based on radix 256, delivers 8 bits of result per cycle and division step, based on radix 16, gives 4 bits of quotient every two cycles. Step operators, used for product and division, allow small size operations, like successive and possibly dependent 64 bits * 8 bits products giving 64 bits results every two cycles. They can also be thought of the base of mantissa computation in modular floating point operations. The overall size of the three sliced operators should be less than 40000 transistors.

1 Introduction.

In order to increase RISC processor speed, one can work in two directions: decrease the CPI by superscaling or decrease the cycle by superpipelining. The former has its limit, due to the low level of parallelism inherent to the instruction flow [6]. This limit may be pushed forward a little by code reordering. The latter -our choice- is also limited, but this time hardware and technology are mainly concerned. Pushing forward the limit means dividing processor activity in more steps, made of thinner pipeline stages. Up to the 90's, the limit was imposed by external accesses, at least for fetch stage. Today, internal caches and 64 bits processors have moved the critical path from memory access to computing operations.

This is illustrated by the MIPS R4000 [2] pipeline, composed of 8 stages. Table 1 shows an estimation of the different pipe stages critical paths, including inter-stage latching, based on CMOS logic [10]. However the epd (elementary propagation delay, i.e. one CMOS transistor delay) values are only estimations (many implementation dependent features may have to be added to different stages, some increasing the critical path like

[1]email: bg@litp.ibp.fr
[2]email: vu@litp.ibp.fr

by-pass mechanism), it is clear that the critical stages are the ones containing arithmetic logic.
Since a 16 bits carry select adder propagation delay should be around 9 epd (to be precised in section 2), a critical path close to this value will be our goal, equalizing the time taken for each pipe stage. The main problem in designing a pipeline with such a constraint concerns the other arithmetic operations -multiplication and division-[3].

pipe stage object	most critic operator	epd
instruction fetch first half	64 bits adder (next address computation)	25
instruction fetch second half	20 bits equality comparator (virtual address translation)	14
instruction cache tag check	24 bits equality comparator (tag check)	14
ALU operation	64 bits adder (ALU)	25
data cache fetch first half	data cache address decoder	11
data cache fetch second half	20 bits equality comparator (virtual address translation)	14
data cache tag check	24 bits equality comparator (tag check)	14
register file write back	register number decoder (write enable)	8

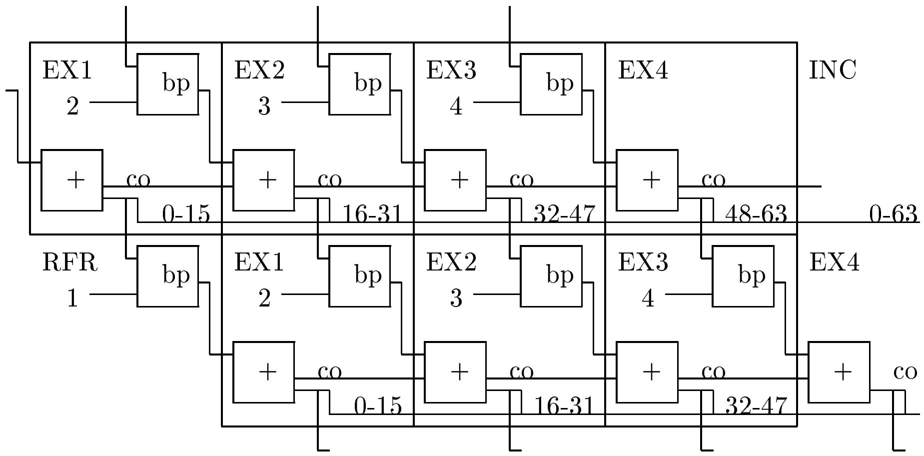
Table 1: R4000 pipeline (epd: elementary propagation delay)

Integer multiplication and division are often enough used in programs -for example with multidimensional arrays- to justify fast operators. However, they are known to be costly constructions. In today's RISC processors, they appear as step operators, the successive steps being linked together either by program instructions (SPARC-like) or by internal signals (MIPS-like). Because of the low radix employed, the operators are in most cases as little efficient as software computation with shift and add. This is what has been shown by the HP PA team [8].
Data slicing should allow much higher radix to be used (radix 256 for product and radix 16 for quotient). In order to obtain such a result, slices must neither be too small (as in bit-serial computation [7] which would give fast but too numerous pipe stages) nor too large (less but too slow stages requiring too costly structures). As was done for multiplication in the SPARC, we propose to include the three integer operators in the same pipeline structure. To allow both a faster cycle and a higher radix for multiplication and division, we propose -and this is our main contribution- to split the computation into four pipeline stages.

[3]It can be argued that PC update is also a problem, since it requires an adder to be performed. It was true as long as fetch needed an external memory access. Then, the whole PC value had to be prepared in a single cycle, in order to be ready for next fetch, starting as soon as next cycle. With internal cache accesses, only the common lower part of the virtual and physical addresses is used to read cache, the upper part being employed only for virtual to physical translation, followed by a comparison of the obtained physical address to the tag read from cache.

2 The adder.

The 64 bits adder is divided into four 16 bits slices, each making a pipeline stage. Stage EX_i ($1 <= i <= 4$) takes care of operands bits [16i-1..16(i-1)]. Each stage is linked to its immediate successor for carry and partial result propagation (see figure 1). Adder operands are obtained either from register file or from preceding instructions partial results (by-pass mechanism, bp on figure 1). As add operation, register file read is sliced too: 16 low order bits are read during decode stage (RFR on figure 1; bp operator entry tagged "1" on the figure; this entry comes either from preceding stages or from register file if no by-pass applies), second slice during first addition stage (EX_1; entry tagged "2"), third slice during second addition stage (EX_2; entry tagged "3") and last slice during third addition stage (EX_3; entry tagged "4"). Result is written into register file in one shot (Register File Write stage, after INC stage), to allow precise exception handling. Thus, every part of the result is forwarded from the stage it has been obtained to the write stage.



bp: by-pass; co: carry out; 0-63: result
0-15: first result slice; 16-31: second result slice
32-47: third result slice; 48-63: fourth result slice
RFR: Register File Read stage; read slice 0-15
EX1: EXecute first slice stage; addition slice 0-15; read slice 16-31
EX2: EXecute second slice stage; addition slice 16-31; read slice 32-47
EX3: EXecute third slice stage; addition slice 32-47; read slice 48-63
EX4: EXecute fourth slice stage; addition slice 48-63
INC: INdicator Compute stage; overflow, carry out
1: by-passed 0-15 slice from forwarded preceding EX1 results
2: by-passed 16-31 slice from forwarded preceding EX2 results
3: by-passed 32-47 slice from forwarded preceding EX3 results
4: by-passed 48-63 slice from forwarded preceding EX4 results

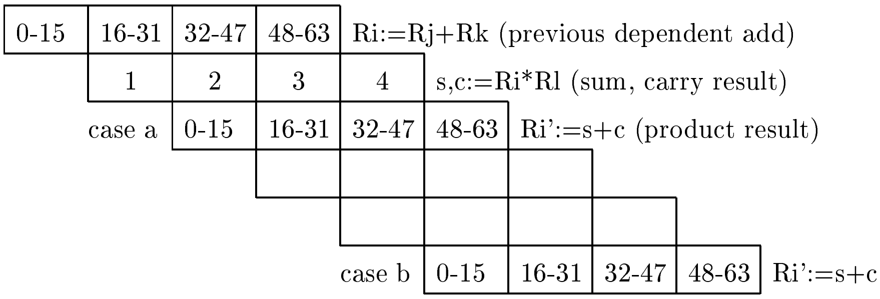
Figure 1: sliced adder

Indications like sign, null or out of range result are computed during INC stage (outside critical path). Subtraction is built from addition by inverting second operand and asserting carry in. Each 16 bits adder is made of seven 4 bits Manchester carry chains, linked by a carry select logic.

The addition stage (EX_i) propagation delay includes the operator delay, output usage delay (possibly used for next instruction same stage operand preparation, in case by-pass conditions are met) and latch delay (mainly, next instruction operands latching). The estimation is 15 epd: 10 epd for addition (3 epd for propagate and generate signals including add/subtract second operand preparation; 4 epd for 4 bits Manchester carry chain; 3 epd for 3 levels of carry selection multiplexers), 2 epd for by-pass (i entry of stage EX_{i-1} bp box on figure 1; 2 levels of multiplexers to allow two sorts of operands: general registers or special s and c registers; see section 3) and 3 epd for latch.
Without by-pass logic, the adder is composed of about 5000 transistors including forwarding latches. Such an adder may be driven by instructions like "ADD R_i, R_j, R_k" ($R_i := R_j + R_k$) or "SUB R_i, R_j, R_k" ($R_i := R_j - R_k$) at a rate of one 64 bits operation per cycle.

3 The multiplier.

A 64*64 bits multiplication requires 32 radix 4 additions to be performed. They will be computed with a carry save adder (CSA) array. In order not to go too far beyond the limit of 15 epd yet imposed by the adder, only 4 additions of Booth recoded radix 4 multiples will be held in what will constitute a multiply step. Eight successive steps will complete the product in sum and carry form. One final addition (through the adder) will give the 64 bits low part of the product. The upper part if desired will need another add. We will neither explain nor justify the well known Booth method ([3], [9]); for a CMOS implementation of a radix 4 multiplier, see [10]). We will only show how data slicing is performed in order to achieve radix 256 computation.



1: do not use Ri[63:16]; case a: use [15:0]; case b: use none
2: do not use Ri[63:32]; case a: use [31:16]; case b: use none
3: do not use Ri[63:48]; case a: use [47:32]; case b: use none
4: compute all that has not been computed in preceding stages
case a: use [63:48]; case b: use [63:0]

Figure 2: multiply computation pipelining

The multiply step must be sliced to allow full pipelining of add and multiply operations. Because multiplicand bits [63..16] might be under modification by a preceding instruction (see figure 2 showing a previous addition instruction, followed by a multiply step; two cases are presented: case "a" applies when step computation is sliced and case "b"

when computation is done during last stage), stage EX_1 for step 0 may not use other multiplicand bits than [15..0]. The same kind of restriction applies to other stages. Still, a non sliced multiplier could be built (figure 2 case "b"), postponing computation until stage EX_4. But then, the last step would have a 3 cycles latency before the final addition (see case "b" final add instruction $R_{i'}$:=s+c). Thus, successive operations full pipelining, required for efficiency, implies slicing.
Slicing can be performed by cutting the CSA array in diagonal (left part in figure 3) or with straight lines (right part in figure 3). The former, however more regular, is not appropriate since it imposes a one cycle delay between two successive steps (the lower half of stage EX_i sum and carry out must be forwarded to next step previous stage EX_{i-1}). Straight lines lead to uneven stages. They all have 4 CSA rows, but of different width. EX_1 has 8 CSA in its row 0 (high order for step 1), 12 CSA in row 1, 12 CSA in row 2 and 10 CSA in row 3. EX_2 and EX_3 have 4 rows of 16 CSA. EX_4 has 58 CSA in row 0, 60 CSA in row 1, 62 CSA in row 2 and 64 CSA in row 3 (for a total of 414 CSA).

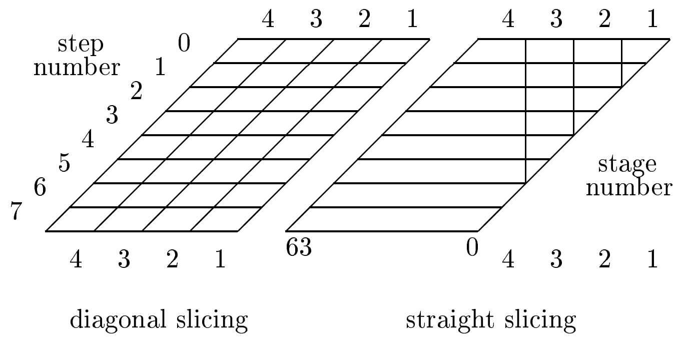


Figure 3: diagonal and straight slicing

Figure 4 is a general overview of the multiplier usage during two successive steps. Multiplicand and multiplier sources (1 and 2 in the figure) will be precised in a moment. S and c outputs (sum and carry out of the last CSA row) are forwarded to next step same stage (by-pass mechanism). C_r[r] bits (leftmost carry out of CSA row r) are forwarded to same step next stage. Stage EX_1 of step i saves its 8 lower carry and sum bits out of last CSA row in sl[8i+7..8i] and cl[8i+7..8i] (sl and cl are s and c lower halves; more precisely, the 8 bits values are forwarded to the last pipeline stage and inserted into sl and cl registers at this moment). Sl and cl will contain the product lower part after last step. S[127..72] and c[127..72] are reset to zero at the end of step 0 while s[71..0] and c[71..0] receive step 0 sum and carry results.
Each stage must be prepared, for its inputs as well as its outputs, to be used by 8 different steps. For each step, stage inputs vary, as shown in table 2. For example, step 2 computes its result in 3 stages (see figure 3). Stage EX_1 does not compute anything. It receives 16 zeros as its multiplicand entry x. Stage EX_2 uses multiplicand slice x[15..0] and multiplier slice y[23..15]. The 16 multiplicand bits are completed with 8 low order zeros ("z" in table 2). Stage EX_3 uses x[31..9] and y[23..15]. Stage EX_4 uses x[63..25] and y[23..15]. Multiplicand entry is sign extended for signed multiplication, 0 extended otherwise, with 25 bits ("s" in table 2).

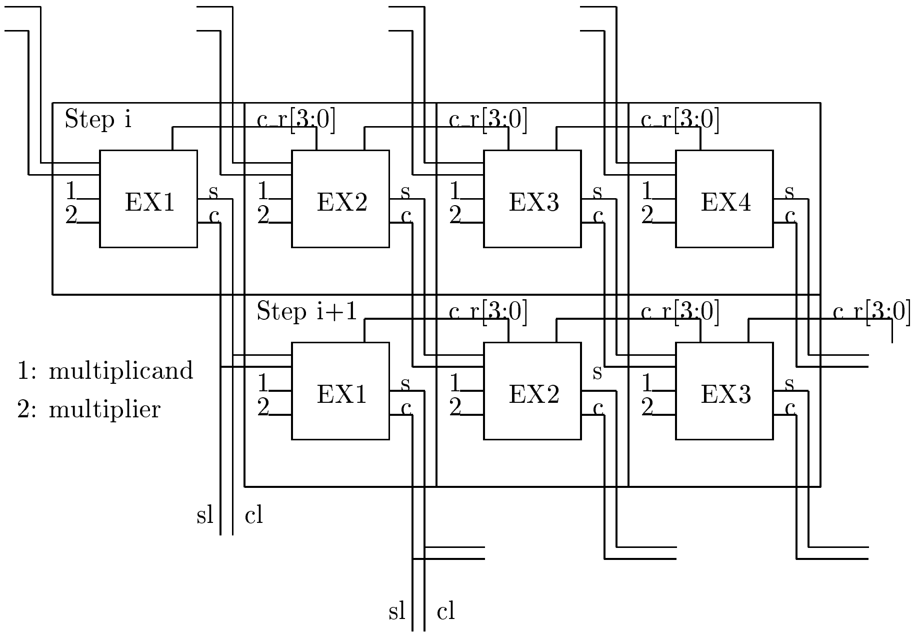


Figure 4: sliced multiplier general overview

		step 0	step 1	step 2	step 3	step 4	step 5	step 6	step 7
EX1	x	15:0	7:0,8*z	16*z	16*z	16*z	16*z	16*z	16*z
	y	7:0,z	15:7	23:15	31:23	39:31	47:39	55:47	63:55
EX2	x	31:9	23:1	15:0,7*z	7:0,15*z	23*z	23*z	23*z	23*z
	y	7:0,z	15:7	23:15	31:23	39:31	47:39	55:47	63:55
EX3	x	47:25	39:17	31:9	23:1	15:0,7*z	7:0,15*z	23*z	23*z
	y	7:0,z	15:7	23:15	31:23	39:31	47:39	55:47	63:55
EX4	x	41s,63:41	33s,63:33	25s,63:25	17s,63:17	9s,63:9	1s,63:1	63:0	63:0
	y	7:0,z	15:7	23:15	31:23	39:31	47:39	55:47	63:55

Table 2: multiplicand (x) and multiplier (y) slices

Table 2 content can be interpreted this way: whatever is the current step, stage EX_1 receives 16 bits from multiplicand (possibly less but completed with low order zeros), stage EX_2 and EX_3 receive 23 bits and stage EX_4 receives 64 bits (possibly less but sign extended). From step to step, what is changed for each stage is the multiplicand slice received: same size but different bits. It would be too costly in time to choose these operands during the stage they are used. It must be done during the preceding cycle and kept in a latch.
The selection process for multiplicand bits is made of multiplexer levels, controlled by the step number. At most, 5 levels will be needed. It is important to notice that the selection must take into account a possible by-passed slice coming from the preceding instruction, if it is an addition and the current multiplication corresponds to step 0. For this reason,

this choice will be the last one, implying a selection time of 11 epd (10 epd for addition and 1 epd for selection). For the multiplier operand, the same technique is applied with at most 5 levels. Same technique again for sum and carry inputs of each stage first CSA row, with 3 levels, by-passed value from preceding instruction being selected last (these operands are the most critical ones: the by-passed value coming from preceding multiplication step operation is not available before 11 epd (CSA array crossing; detailed in a moment), thus is selected after 12 epd and latched after 15 epd).
Figure 5 illustrates an 8 steps product (64 bits * 64 bits). In each cell, the upper line corresponds to the x multiplicand and the lower line to the y multiplier.

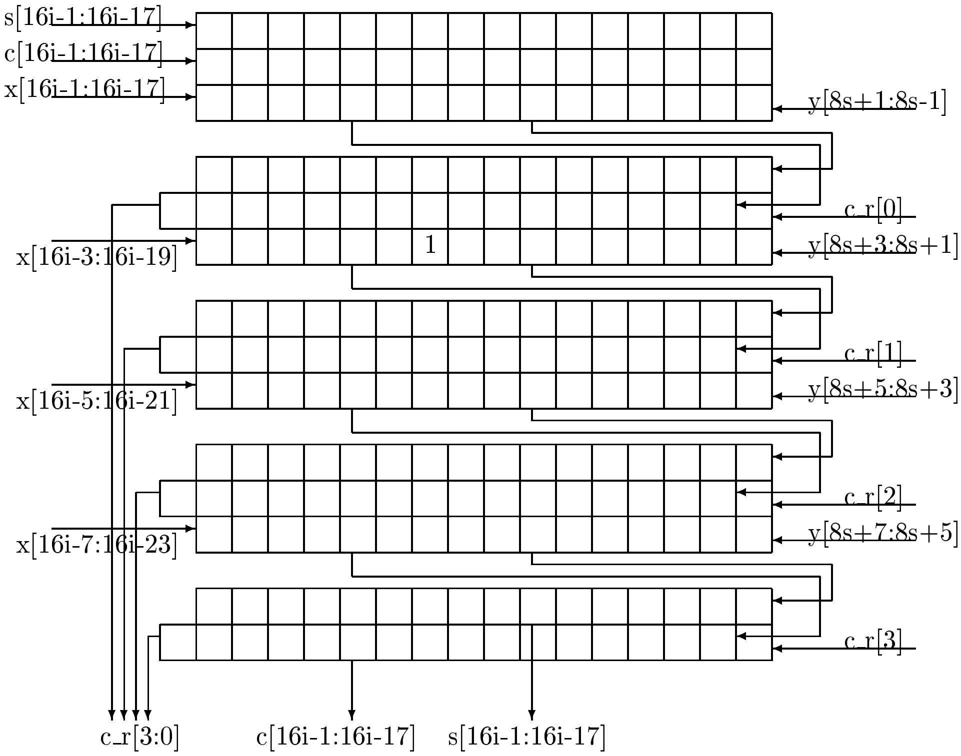
cycle:	0	1	2	3	4	5	6	7	8	9	10
step: 0	15:0 7:0,z	31:9 7:0,z	47:25 7:0,z	63:41 7:0,z							
1		7:0 15:7	23:1 15:7	39:17 15:7	63:33 15:7						
2			-	15:0 23:15	31:9 23:15	63:25 23:15					
3				-	7:0 31:23	23:1 31:23	63:17 31:23				
4					-	-	15:0 39:31	63:9 39:31			
5						-	-	7:0 47:39	63:1 47:39		
6							-	-	-	63:0 55:47	
7								-	-	-	63:0 63:55

Figure 5: 64 bits * 64 bits product in 8 steps

Figure 6 shows the inside of stage EX_i (more precisely, the represented width applies to EX_2 or EX_3 only, but stage EX_1 is built on the same pattern, with CSA rows shortened on their right edge, and stage EX_4 also, with CSA rows extended on their left edge). For each multiplicand CSA entry, a selection is made, corresponding to the operation to be performed, according to the current radix 4 multiplier digit.
The stage operation critical path is composed of three parts: preparation of the CSA first row inputs, addition in the CSA array and result latch. The first CSA row inputs are obtained after 3 transformations: choose between x[c] and x[c-1], then nullify choice or not, then invert it or not. The cost is 3 epd. Crossing the CSA array takes 8 epd[4] and

[4]CSA sum and carry outputs are considered to need 2 epd to be stabilized. In order to obtain such a fast CSA, we must build it with CMOS switch logic. We must also supply it with both normal and inverted inputs and for cascading reasons it must compute both normal and inverted outputs. Such a CSA is just 4 transistors bigger than an ordinary one.

latching lasts 3 epd. This gives a 14 epd operation critical path (still the stage critical path is given by the operand preparation delay (15 epd)).



stage i ($1 <= i <= 4$), step s ($0 <= s <= 7$), row r ($0 <= r <= 3$)
column c ($16i - 17 <= c <= 16i - 1$)
1: choose operand from 1,#x[c],#x[c],#x[c-1],x[c-1],x[c],x[c],0
according to y[8s+2r+1:8s+2r-1]
c_r[3:0]: to be propagated to next stage same step
c_r[3:0]: coming from previous stage same step
s[16i-1:16i-17],c[16i-1:16i-17]: to be propagated to next step same stage
s[16i-1:16i-17],c[16i-1:16i-17]: coming from previous step same stage

Figure 6: EXi multiplier stage

The material cost of the method can be estimated to 414 CSA, registers s (128 bits), c (128 bits), latches for multiplicand, multiplier, partial result propagation, c_r[r], CSA third input selection (per CSA: one multiplexer -choose between x[c] and x[c-1]-, followed by one AND gate -nullify or not-, ended by one XOR gate -invert or not-) and operand selection multiplexers. Altogether, it represents 414*24 (CSA) + 638*6 (latches and registers) + 414*11 (MUX, AND, XOR) + 1862*4 (operand selection)= 25766 transistors. The most of it will serve for division.
Such a multiplier may be driven by an instruction like "MUL s, R_j, R_k" where "s" (from 0 to 7) is the step number. The instruction set allows different size multiplications. For example, 64*8 bits product is obtained in just 2 instructions:

MUL 0, R_j, R_k ;(s[71..0], c[71..0]):= R_j[63..0] * R_k[7..0]
ADD R_i, sl, cl ;R_i[63..0]:= s[63..0] + c[63..0]

A $16 * 16 => 32$ bits product would be coded as:

MUL 0, R_j, R_k ;(s[71..0], c[71..0]):= R_j[63..0] * R_k[7..0]
MUL 1, R_j, R_k ;(s[79..0], c[79..0]):= R_j[63..0] * R_k[15..0]
ADD R_i, sl, cl ;R_i[63..0]:= s[63..0] + c[63..0]

A $64 * 64 => 128$ bits product is obtained after 8 steps and two final additions (the second addition that computes the product upper part must be started 4 cycles after the last step, introducing 3 delay cycles, one of them filled by the first addition):

MUL 0, R_j, R_k ;(s[71..0], c[71..0]):= R_j[63..0] * R_k[7..0]
MUL 1, R_j, R_k ;(s[79..0], c[79..0]):= R_j[63..0] * R_k[15..0]
MUL 2, R_j, R_k ;(s[87..0], c[87..0]):= R_j[63..0] * R_k[23..0]
MUL 3, R_j, R_k ;(s[95..0], c[95..0]):= R_j[63..0] * R_k[31..0]
MUL 4, R_j, R_k ;(s[103..0], c[103..0]):= R_j[63..0] * R_k[39..0]
MUL 5, R_j, R_k ;(s[111..0], c[111..0]):= R_j[63..0] * R_k[47..0]
MUL 6, R_j, R_k ;(s[119..0], c[119..0]):= R_j[63..0] * R_k[55..0]
MUL 7, R_j, R_k ;(s[127..0], c[127..0]):= R_j[63..0] * R_k[63..0]
ADD R_i, sl, cl ;R_i[63..0]:= s[63..0] + c[63..0]
–2 cycles delay
ADDcy R_{i+1}, sh, ch ;R_{i+1}[63..0]:= s[127..64] + c[127..64] + cy

Signed multiplication is properly handled via a signed step operator:

SMUL s, R_j, R_k

4 The divider.

Division is based on the SRT radix 2^n method [1]. Because we plan to reuse most of the multiplier, our divider will look like a cellular array one [4]. The step operator must estimate a new quotient from a prefix of both the remainder and the divider. Then, it has to compute the new remainder for the next step. Successive steps full pipelining is not possible within 15 epd stages. The estimation of the quotient can be performed in a single stage (as shown later). Thus, the new remainder computation may not start before the second stage. So, even though we fix new remainder high sum and carry parts during stage EX_2 instead of EX_4, next quotient q[i+1] selection cannot begin less than 2 cycles after the choice of q[i]. Pipeline stages usage for division is shown on figure 7.
The radix must be fixed as high as possible in order to decrease the number of steps. However, the radix choice has a serious impact on the first stage critical path. Stage EX_1 must first establish the remainder high part estimation (rhpe) from its sum and carry value out of previous step EX_2 stage. Then, rhpe serves to address a table which delivers the new quotient digit. The higher the radix, the larger the sum and carry addition and the larger the table. Radix 16 is appropriate: the addition is 7 bits wide and the table

can be reduced to 1K*4 bits, addressed first with 4 divider bits (not in critical path) then with estimated remainder 6 higher bits. Both operations are done in 12 epd. Radix 16 is also exactly suited to new remainder computation through the four CSA rows already provided by the multiplier. This allows an extension of the quotient value interval to its maximum -15..+15.

cycle:	j	j+1	j+2	j+3	j+4	j+5
step i	q[i]	rem high	rem med	rem low		
	delay	-	-	-	-	
		step i+1	q[i+1]	rem high	rem med	rem low

Figure 7: divider step operator slicing

As shown on figure 7, remainder is computed high order slice first. It will give a correct sum and carry result only if slices overlap. Because medium slice sum and carry high order bits are not available when high slice CSA array starts its computation, it will output four incorrect low order sum and carry bits (see figure 8). They will be recomputed by medium slice. In the same idea, we must take care of the fact that c_r bits will not be propagated correctly. Thus, low order bits computed by medium slice will not be accurate. To alleviate this problem for low slice (computed by EX_4 stage), stage EX_3 will receive only 12 bits (completed with 4 high order zeros: c_r input bits for EX_4 will all be 0). Therefore, stage EX_2 takes care of high slice 48-63 producing a correct sum slice 52-63 and carry slice 53-63. Stage EX_3 operates on medium slice 40-51, with accurate result slices 44-51 for sum and 45-52 for carry. Stage EX_4 computes slices 0-43 (sum) and 0-44 (carry) from slice 0-43.

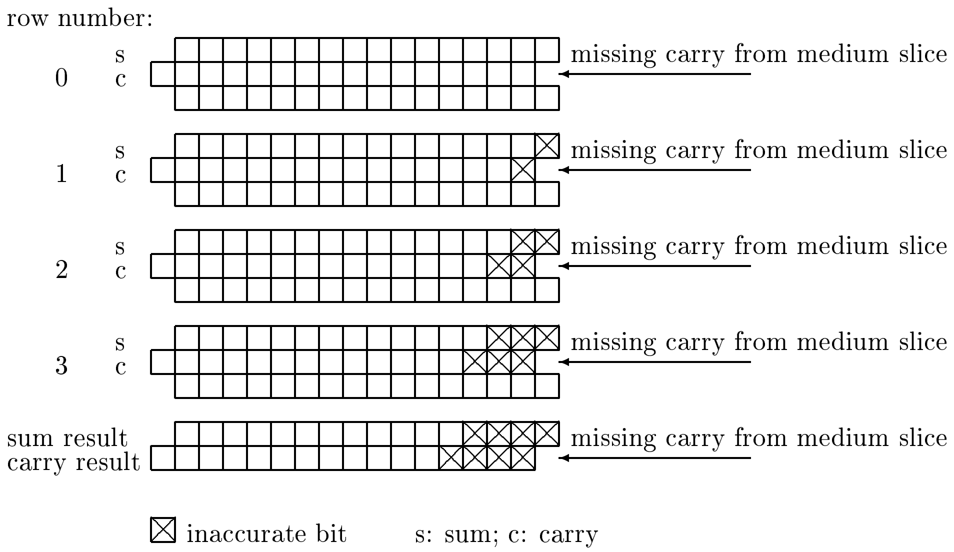


Figure 8: remainder high slice CSA computation

For this reason, next step quotient selection must be based on at most the 11 high order bits (53-63) of the new remainder sum and carry representation. In fact, only

7 signed bits are needed as shown by a program checking that quotient valid intervals [16*(r+2)/d-1, 16*(r+2)/d+1] and [16*r/(d+1)-1, 16*r/(d+1)+1] (for remainder $r >= 0$; [16(r+2)/(d+1)-1, 16(r+2)/(d+1)+1] and [16r/d-1, 16r/d+1] for $r < 0$) always include at least one common integer. The first interval is derived from the largest possible division for truncated d (divider 5 upper bits, including the leftmost 1 due to divider normalization) and r (remainder 7 upper signed bits), with r in CSA notation. The second interval corresponds to the smallest division.
Because we use a full redundancy for quotient representation (-15..+15), the $2^7 * 2^5$ quotients table is symetric to 0. Thus, it may be reduced to its positive part, providing that it is addressed with abs(r). Then, sign of r serves for q. The $2^6 * 2^4$ four bits unsigned quotients are given in table 3.
Positive quotients are saved in sl register and negative ones in absolute value, in cl register. The quotient value is obtained after a final subtraction of cl from sl.
The critical path for stage EX_1 (quotient selection) is composed of a 7 bits addition (remainder sum + carry high bits; s+c and #s+#c+2 (6 epd), select positive result (1 epd) to provide abs(r) in 7 epd), last selection from quotients table access (remainder 3 high bits decode followed by read enable; 5 epd) and a latch (latch q; 3 epd): 15 epd.
Such a divider may be driven by two instructions like "DIV 0, R_j, R_k" for step 0 and "DIV s, sh, ch, R_k" for other steps "s" (from 1 to 15). The instruction set allows different size divisions. For example, an 8 bits quotient is obtained in just 4 instructions (after normalization, not considered in this paper):

–shift R_k left until negative; keep shift number s_n;
–(normalize R_k)
–shift R_j left s_n positions;
DIV 0, R_j, R_k
; s[63..4]:= s[59..0]; s[3..0]:= q+, c[63..4]:= c[59..0]; c[3..0]:= q-
; (sh[63..0], ch[63..0]):= remainder
–1 cycle delay
DIV 1, sh, ch, R_k
; s[63..4]:= s[59..0]; s[3..0]:= q+, c[63..4]:= c[59..0]; c[3..0]:= q-
; (sh[63..0], ch[63..0]):= remainder
SUB R_i, sl, cl ;R_i:= sl - cl (quotient)
ADD 0, $R_{i'}$, sh, ch ;$R_{i'}$:= sh[63..0] + ch[63..0] (remainder)
–if $R_{i'} < 0, R_{i'} := R_{i'} + R_k; R_i := R_i$-1
–shift $R_{i'}$ right s_n positions

The divider includes latches for q and sign (5 bits latch per stage for forwarding), two 7 bits adders, a 4K bits table (one transistor per bit for a ROM structure), its column and row decoders. Multiplier must be upgraded to allow division operand selection for CSA inputs. The size of the division part represents about 7000 transistors.

16	0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9 a a b b c c d d e e f f
	f f f f f f f f f f f f f f f f f f f f f f f f f f f f f f f f
17	0 0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 7 8 8 9 9 a a b b c c d d e e
	f f f f f f f f f f f f f f f f f f f f f f f f f f f f f f f f
18	0 0 0 1 1 2 2 3 3 3 4 4 5 5 6 6 7 7 7 8 8 9 9 a a b b b c c d d
	e e f f f f f f f f f f f f f f f f f f f f f f f f f f f f f f
19	0 0 0 1 1 1 2 2 3 3 4 4 4 5 5 6 6 7 7 7 8 8 9 9 9 a a b b c c c
	d d e e f f f f f f f f f f f f f f f f f f f f f f f f f f f f
20	0 0 0 1 1 1 2 2 3 3 3 4 4 5 5 5 6 6 7 7 7 8 8 9 9 9 a a b b b c
	c d d d e e f f f f f f f f f f f f f f f f f f f f f f f f f f
21	0 0 0 0 1 1 2 2 2 3 3 3 4 4 5 5 5 6 6 7 7 7 8 8 8 9 9 a a a b b
	b c c d d d e e f f f f f f f f f f f f f f f f f f f f f f f f
22	0 0 0 0 1 1 1 2 2 3 3 3 4 4 4 5 5 5 6 6 7 7 7 8 8 8 9 9 9 a a b
	b b c c c d d d e e f f f f f f f f f f f f f f f f f f f f f f
23	0 0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6 6 7 7 7 8 8 8 9 9 9 a a
	a b b b c c c d d d e e f f f f f f f f f f f f f f f f f f f f
24	0 0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6 6 6 7 7 7 8 8 8 9 9 9 a
	a a b b b c c c d d d e e e f f f f f f f f f f f f f f f f f f
25	0 0 0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6 6 6 7 7 7 7 8 8 8 9 9
	9 a a a b b b c c c d d d e e e f f f f f f f f f f f f f f f f
26	0 0 0 0 0 1 1 1 2 2 2 3 3 3 3 4 4 4 5 5 5 6 6 6 7 7 7 7 8 8 8 9
	9 9 a a a b b b b c c c d d d e e e f f f f f f f f f f f f f f
27	0 0 0 0 0 1 1 1 1 2 2 2 3 3 3 4 4 4 4 5 5 5 6 6 6 7 7 7 7 8 8 8
	9 9 9 9 a a a b b b c c c c d d d e e e f f f f f f f f f f f f
28	0 0 0 0 0 1 1 1 1 2 2 2 3 3 3 3 4 4 4 5 5 5 5 6 6 6 7 7 7 7 8 8
	8 9 9 9 9 a a a b b b b c c c d d d d e e e f f f f f f f f f f
29	0 0 0 0 0 0 1 1 1 2 2 2 2 3 3 3 3 4 4 4 5 5 5 5 6 6 6 7 7 7 7 8
	8 8 8 9 9 9 a a a a b b b b c c c d d d d e e e f f f f f f f f
30	0 0 0 0 0 0 1 1 1 1 2 2 2 3 3 3 3 4 4 4 4 5 5 5 5 6 6 6 7 7 7 7
	8 8 8 8 9 9 9 9 a a a b b b b c c c c d d d d e e e f f f f f f
31	0 0 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3 4 4 4 4 5 5 5 5 6 6 6 7 7 7
	7 8 8 8 8 9 9 9 9 a a a a b b b b c c c c d d d d e e e f f f f
d	
r	0 … 31
	32 … 63

Table 3: quotient in t[d[31:16], r[63:0]]

Discussion and conclusion.

We have proposed a set of integer arithmetic sliced operators. They may form the computing part of a superpipelined processor, with a cycle time of 15 epd (1 epd= 1 CMOS transistor propagation delay; today, between 100 and 200 ps, which would allow a 500Mhz clocked CPU; actual superpipelined processors have a pipeline stage width of at least 26 epd (64 bits addition, next operand preparation -result by-pass- and latching)). Successive 64 bits additions can be pipelined at a rate of one per cycle. Multiplications and divisions are handled by step operators. Multiply step produces a 64*8 bits sum and carry result every cycle -radix 256 multiplier- and divide step gives 4 bits of quotient every two cycles -radix 16 divider-.
The size of the 64 bits adder is about 5000 transistors. The size of the 64*8 bits step multiplier is about 26000 transistors. The size of the SRT radix 16 step divider is about 7000 transistors. Altogether, the integer unit is less than 40000 transistors.
Integer operators in today's RISC processors are not efficient. It is usually preferred to have simple integer multiply and divide operators (like in the SPARC or the MIPS) giving a one bit result per cycle, and to rely on floating point coprocessors for faster

computation. First, we think that integer products and quotients are often enough used in programs to justify fast integer units, for example when involving multidimensional arrays. Second, 90% of integer multiplications involve a constant which tends to be small (80% are one byte large [8]). In this context, our multiplication operator should perform much better than simulation by shift and add (as in the HP PA [8]: an average of 20 instructions are needed per product) or low radix operator like in the R4000 (20 cycles for a double precision product) or the DEC alpha (23 cycles [5]). It should also perform better than fast multipliers like in the superSPARC (4 cycles per product, through the floating point coprocessor), but with a much lower cycle width, due to superpipelining.
This improvement is not due to the use of space costly structures. The CSA array is kept small, thanks to step operating, and shared by multiplier and divider.
To conclude, we think that this type of operator may form the base of modular floating point units, working with data dependent algorithms, neglecting as often as possible certain steps like operand scaling, result rounding or sign transformations.

References.

[1]: D.E. Atkins: Higher-radix division using estimates of the divisor and partial remainders, IEEE Trans. Comput., vol. C-17 (1968)

[2]: A. Bashteen, I. Lui, J. Mullan: A superpipeline approach to the MIPS architecture, IEEE (1991)

[3]: A.D. Booth: A signed binary multiplication technique, Quart. J. Mech. Appl. Math., vol. 4 (1951)

[4]: M. Cappa, V.C. Hamacher: An augmented iterative array for high speed binary division, IEEE Trans. Comput., vol. C-22 (1973)

[5]: Digital Equipment Corporation: Alpha architecture handbook, DEC (1992)

[6]: N.P. Jouppi, D.W. Wall: Available instruction-level parallelism for superscalar and superpipelined machines, ACM (1989)

[7]: R.F. Lyon: Two's complement pipeline multipliers, IEEE Trans. Commun. (1976)

[8]: D.J. Magenheimer, L. Peters, K.W. Pettis, D. Zuras: Integer multiplication on the HP precision architecture, IEEE Trans. Comput., vol 37 (1988)

[9]: E. Swartzlander: Computer arithmetic, Dowden, Hutchinson and Ross (1980)

[10]: N. Weste, K. Eshraghian: Principles of CMOS VLSI design, Addison-Wesley (1992)