Improving GMP Itanium multiplication times

Torbjörn Granlund

2011-01-10

1 Introduction

The Itanium processors have a very odd architecture, allowing 6-way issue in a completely static pipeline. The gap between easily achievable performance levels, and the full hardware capability is very large.

This note is about improving basic $O(n^2)$ multiplication in the context of GMP. It seems possible to approach 1 cycle/limb, but it is a daunting task, as is evident from this note.

Notation and conventions:

- For addmul_k, we have an n-limb number U and a k-limb number V. We reqire that routines work for any $n \geq k$.
- The inner loop necessarily handles k double-limb products. Usually, it handles wk double-limb, where w is the unrolling ways factor.
- We sometimes abbreviate addmul_k as am$k$ and mul_k as mk to save space.

2 Goals

We start with a summary. Possible speed for addmul_k:

k	**getf.sig/lim**	**cycles/limb**	**method/unrolling ways**
1	2	2	ad-hoc
1	1.5	1.75	semi-limping, add using xma
2	3	1.5	ad-hoc
3	3	$1.333 + \epsilon$	AACC/∞ i.e., 2 add, 2 cmp
4	4	$1.292 + \epsilon$	COLCY/∞ i.e., column carry
5	5	$1.233 + \epsilon$	COLCY/∞
6	6	$1.083 + \epsilon$	POPC/∞ i.e., table based popcnt
7	7	$1.048 + \epsilon$	POPC/∞ (enough pregs?)
8	8	$1.021 + \epsilon$	POPC (enough pregs?)
9	9	$1 + \epsilon$	POPC/∞ (enough pregs?)
10	10	1	POPC/1 (enough pregs?)

Possible speed for mul_k:

k	getf.sig/limb	cycles/limb	method/unrolling ways
1	2	2	ad-hoc
1	1.5	$1.5+\epsilon$	semi-limping
2	3	1.5	ad-hoc
3	3	$1.055+\epsilon$	AACC/∞
4	4	$1.083+\epsilon$	AACC/∞
5	5	$1.1+\epsilon$	AACC/∞
6	6	$1+\epsilon$	POPC/∞
7	7	1	POPC/1 (enough pregs?)
8	8	1	POPC/1 (enough pregs?)

In order to limit the life of loaded U limbs, we should multiply each U limbs by all v limbs in ascending order, without any other intervening multiplies. The xma addition operand should be a former xmahu result, except for addmul_1 and addmul_2 where the xmahu result will not yet be available.

If we load r limbs to gregs, and use earlier product limbs as additive input to the xma instructions, we can do with k getf.sig for addmul_k. Unfortunately, it seems difficult to cope with latency for $k < 3$.

It seems to make sense to implement addmul_6, since it is the first really fast primitive. For RSA-1024 signing, we will do many 8-limb operations, so we should have mul_3 as a complement, even if we do not provide mul_k for all $k \leq 6$.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
m1_am6 WDFI							2						2				
m2_am6 WDFI								2						2			
m3_am6 WDFI									2						2		
m4_am6 WDFI										2						2	
m5_am6 WDFI											2						2
m6_am6 WDFI												2					
am6 LO							3	3	3	3	3	3	3,5	3,5	3,5	3,5	3,5
am6 WDFI													4	4	4	4	4
am6 WD							4	4	4	4	4	4	6	6	6	6	6
m1	1						1						1				
m2		1						1						1			
m3			1						1						1		
m4				1						1						1	
m5					1						1						1
m6						1						1					
cycles/ulimb	2.00	3.25	4.25	5.50	7.00	7.00	9.00	10.25	11.25	????	????	????	????	????	????	????	????
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Table 1: Multiplication primitives and their invocation order in a prospective mul_basecase for $1 \leq n \leq 17$.

If we are to tweak things really well, we should add at least one more high-speed addmul predicate, perhaps addmul_5. Then we should avoid mul_1, except for size 1. Size 7 should use m2 + am5. Size 8 should use m3+am5. Size 9 should use m4+am5 or m3+am6, etc.

An important effect of that is that general multiplication performance will be more linear in $|V|$.

[These mul_k numbers are based on the addmul_k numbers. In reality, they could possibly be made faster.]

3 How to write addmul_k

3.1 addmul_3

The code for addmul_3 using a simple 2-add 2-cmp scheme becomes quite regular. One gets a 4-bundle block, of which 3 bundles have 2 xma instructions each. Unfortunately, there will be no bundle space for the loop branch instruction, meaning that we need to have that in its own bundle, taking up a full cycle.

To manage ldf8's latency for loading new limbs from U, it is necessary to have $w \geq 2$, or to use Itanium's modulo-scheduled loops. Due to the cost of branch control, we might want to unroll even more.

Instructions for addmul_k algorithm variant AACC:

$7wk$	xmal xmahu getfsig add cmpltu add1 cmpeqor
$3w$	ld8 ldf8 st8
1	br

Instructions for mul_k algorithm variant AACC:

$3wk$	xmal xmahu getfsig
$4w(k-1)$	add cmpltu add1 cmpeqor
$2w$	ldf8 st8
1	br

ways	code size	c/l
2	256	1.5
3	384	1.444
4	512	1.417
5	640	1.4
6	768	1.389
7	896	1.381
8	1024	1.375
∞	∞	1.333

Table 2: AACC with addmul_3

(Status: Completed for $w = 4$, not in repo.)

3.2 addmul_4

Generalising addmul_3's scheme to addmul_4 adds 7 instructions per way, but it seems difficult to keep to addmul_3's regular code layout. We can, however, put instructions in the loop branch bundle, which has 5 free slots, for up to 5-way unrolling, after an extra bundle causes things to slow down. We need a 10-bundle block plus one extra instruction.

ways	code size	c/l
2	336	1.375
3	512	1.333
4	672	1.3125
5	832	1.3
∞	∞	1.292

Table 3: AACC with addmul_4

We need to move the an add+cmp+add1 scheme, where in add1 increments a "column carry" variable that is later added like just another quantity to the next more significant column.

We thus save a cmp per double-limb product, but add another add+cmp+mov0+add1 per column. The mov0 is needed for initialising a new column carry variable. Unfortunately, for addmul_4 this does not bring any joy, unless we unroll very deeply.

Instructions for addmul_k algorithm variant COLCY:

$6wk$	xmal xmahu getfsig add cmpltu add1
$7w$	ld8 ldf8 st8 add cmpltu mov0 add1
1	br

Instructions for mul_k algorithm variant COLCY:

$6wk$	xmal xmahu getfsig add cmpltu add1
$3w$	ldf8 st8 mov0
1	br

ways	code size	c/l
2	336	1.375
3	496	1.333
4	656	1.312
5	816	1.3
∞	∞	1.292

Table 4: COLCY with addmul_4

One important caveat about this scheme is that the conditional add1 will need to have the same input and output register. This might seem like no problem, but if we go for a w-ways scheme where w is not a multiple of k ($k = 4$ for addmul_k) we will likely be forced to resort to using rotating registers.

If we postpone the add1, and instead collect k carry bits in k predicate registers, we can use just 5 instructions per double-limb product, at the expense of a movpr, a mask operation, and a population count, plus an add+cmp when adding this to the next more significant column.

We may use popcnt or a table lookup. The former needs fewer instructions, the latter gives shorter latency.

With popcnt, if the last cmpltu runs in cycle t_0, movpr can run in cycle $t_0 + 1$, extru in cycle $t_0 + 3$, popcnt in cycle t_0+?, add in cycle t_0+?, and its cmpltu in cycle t_0+?. Thus we have a forbidding recurrency path of 10 cycles, so we

should stay away from the popcnt instruction.

With table, if the last cmpltu runs in cycle t_0, movpr can run in cycle $t_0 + 1$, extru in cycle $t_0 + 3$, add for forming a table address in cycle $t_0 + 4$, ld1 in cycle $t_0 + 5$, add in cycle $t_0 + 6$, and its cmpltu in cycle $t_0 + 7$. Thus we have a recurrency path of 7 cycles.

Instructions for addmul_k algorithm variant POPC (7 cycle recurrency):

$5wk$	xmal xmahu getfsig add cmpltu
$9w$	ld8 ldf8 st8 add cmpltu movpr extru add ld1
1	br

Instructions for mul_k algorithm variant POPC (7 cycle recurrency):

$5wk$	xmal xmahu getfsig add cmpltu
$6w$	ldf8 st8 movpr extru add ld1
1	br

For addmul_4 both variants give too high latency.

3.3 addmul_5

For addmul_5 we cannot use popcnt beneficially, due to its latency.

ways	code size	c/l
1	224	1.400
2	416	1.300
5	1024	1.280
8	1632	1.275
∞	∞	1.267

Table 5: AACC with addmul_5

ways	code size	c/l
1	224	1.400
2	416	1.300
3	608	1.267
4	800	1.250
5	992	1.240
∞	∞	1.233

Table 6: COLCY with addmul_5

3.4 addmul_6

For addmul_6 we can finally use column population count, but plain column carry also works well.

3.5 addmul_8

Using AACC we get no joy, but COLCY is better:

ways	code size	c/l
1	256	1.333
2	480	1.250
3	704	1.222
4	928	1.208
5	1152	1.200
6	1408	1.222
7	1632	1.214
8	1856	1.208
∞	∞	1.194

Table 7: COLCY with addmul_6

ways	code size	c/l
1	224	1.167
3	640	1.111
5	1056	1.1
7	1472	1.095
∞	∞	1.083

Table 8: POPC with addmul_6

ways	code size	c/l
1	320	1.250
2	608	1.188
3	896	1.167
4	1184	1.156
5	1472	1.150
∞	∞	1.146

Table 9: COLCY with addmul_8

Using the popcnt scheme, we get very good numbers. However, it is unclear if we practically can implement this; the number of predicate registers needed is close to 64, but the rotating registers are just 44 (IIRC). An 8-way unroll would work, but this would be terribly large.

ways	code size	c/l
1	288	1.125
2	544	1.062
3	800	1.042
4	1056	1.031
5	1312	1.025
8	2112	1.031
∞	∞	1.021

Table 10: POPC with addmul_8

4 Final remarks

It would be possible, and perhaps practical in a mul_basecase setting, to use ldpf8 for leading from U. We should then have two loop variants for the two possible alignments; if U is not 16-byte aligned, we have to load 8 bytes initially and similarly we would need a final 8-bit load in the end for some alignments/sizes. We should absolutely not conditionally invoke an initial addmul_1 "rotated". We could perhaps run something like a rotated addmul_3 Using ldpf8 frees up $w/2$ instructions; we can only handle even w.

It would also be possible to reduce the insn count by identifying unlikely conditions, and use branches for those. This should not be considered, since it will leak in the side channel (i.e., bad for cryptography).

5 The semi-limping approach for mul_k

There is another interesting approach to mul_k, allowing us to com very close to 1 c/l for any k at the expense of software pipeline depth. Unfortunately, a consequence of the deep software pipeline is that we will require large n before these methods show their full power.

mul_1 0 1 2 3 4 5 6 7 8 9 10 11 n0 n2 n4 n6 a1 a3 a5 a7 8 dmul, 12 getf, 8 ldf8, 8 st8, 4 aacc, br, 12c 1.5 c/l n0 n3 n6 a1 a4 a7 a2 a5 a8 9 dmul, 12 getf, 9 ldf8, 9 st8, 3 aacc, br, 12c 1.333 c/l n0 n3 n6 n9 a1 a4 a7 a10 a2 a5 a8 a11 12 dmul, 16 getf,12 ldf8 12 st8, 4 aacc, br, 16c 1.333 c/l

mul_2 0 1 2 3 4 5 6 7 8 9 10 11

```
MPN_MUL_K(U, v_0, ..., v_{k-1}, n)
    In:
 1  for ℓ ← 0 ... w^2 − 1
 2      u_ℓ ← up[ℓ]                                  // Read new limbs from U
 3  for ℓ ← 0 ... w − 1
 4      for i = 0 ... k − 1
 5          plo_{wℓ,i} ← u_{wℓ} v_i  mod β
 6          phi_{wℓ,i} ← ⌊u_{wℓ} v_i/β⌋
 7  for j ← 1 ... w − 1
 8      for ℓ ← 0 ... w − 1
 9          for i = 0 ... k − 1
10              plo_{wℓ+j,i} ← (u_{wℓ+j} v_i + phi_{wℓ+j,i}  mod β
11              phi_{wℓ+j,i} ← ⌊(u_{wℓ+j} v_i + phi_{wℓ+j,i})/β⌋
12  Sum plo's and phi's
```

Algorithm 1: Compute UV where V is k limbs, using a w-phase method.

phases	code size	c/l
1	?	2
2	?	1.5
3	?	1.333
4	?	1.25
5	?	1.2
8	?	1.167
∞	∞	1

Table 11: LIMP with mul_1