SDU
University of
Southern Denmark

Cryptographic Engineering

Multiprecision Arithmetic I

Ruben Niederhagen
(based on content by Peter Schwabe)

Department of Mathematics and Computer Science (IMADA)

Multiprecision Arithmetic in Cryptography

- Asymmetric cryptography heavily relies on arithmetic on “big integers”.
- Example 1: RSA-2048 needs (modular) mult. and squaring of 2048-bit numbers.
- Example 2:
  - Elliptic curves defined over finite fields.
  - Typically use EC over large-characteristic prime fields.
  - Typical field sizes: (160 bits, 192 bits), 256 bits, 448 bits. . .
- Example 3: Poly1305 needs arithmetic on 130-bit integers.
- An integer is “big” if it is not natively supported by the machine architecture.
- Example: AMD64 supports up to 64-bit integers, multiplication produces 128-bit result, but not bigger than that.
- We call arithmetic on such “big integers” *multiprecision arithmetic*.
- For now mainly interested in 160-bit and 256-bit arithmetic.
- Example architecture for today (most of the time): AVR ATmega.

Available numbers (digits): $1, 2, 3, 4, 5, 6, 7, 8, 9$

Addition:	**Subtraction:**
$3 + 5 = ?$	$7 - 5 = ?$
$2 + 7 = ?$	$5 - 1 = ?$
$4 + 3 = ?$	$9 - 3 = ?$

- All results are in the set of available numbers.

$\Rightarrow$ No confusion for first-year school kids.

Available numbers (digits): $01, 2, \ldots, 255$

Addition:

```
uint8_t a = 42;
uint8_t b = 89;
uint8_t r = a + b;
```

Subtraction:

```
uint8_t a = 157;
uint8_t b = 23;
uint8_t r = a - b;
```

- All results are in the set of available numbers.
- Larger set of available numbers: `uint16_t`, `uint32_t`, `uint64_t`.
- Basic principle is the same; for the moment stick with `uint8_t`.

Crossing the ten barrier:

$$6 + 5 = ? \quad 11$$

$$9 + 7 = ? \quad 16$$

$$4 + 8 = ? \quad 12$$

- Inputs to addition are still from the set of available numbers.
- Results are allowed to be larger than 9.
- Addition is allowed to produce a *carry*.

Crossing the ten barrier:

$$6 + 5 = ? \quad 11$$

$$9 + 7 = ? \quad 16$$

$$4 + 8 = ? \quad 12$$

What happens with the carry?

- Introduce the decimal positional system: Write an integer A in two digits $a_1 a_0$ with:

$$A = 10 \cdot a_1 + a_0.$$

- Note that at the moment $a_1 \in 0, 1$.

Back to Programming

```
uint8_t a = 184;
uint8_t b = 203;
uint8_t r = a + b;
```

- The result r now has the value of 131, **not** 387!
- The carry is lost, what do we do?
- Could cast to `uint16_t`, `uint32_t` etc., but that does not solve the general problem.
- We really want to obtain the carry, and put it into another `uint8_t`.

Introducing the AVR ATmega

- 8-bit RISC architecture.
- 32 registers R0 . . . R31, some of those are “special”:
  - (R26,R27) aliased as X.
  - (R28,R29) aliased as Y.
  - (R30,R31) aliased as Z.
  - X, Y, Z are used for addressing.
  - 2-byte output of a multiplication always in R0 and R1.
- Most arithmetic instructions cost 1 cycle.
- Multiplication and memory access takes 2 cycles.

Computing 184 + 203 on AVR ATmega


```
LDI R5, 184
LDI R6, 203
ADD R5, R6   ; result in R5, sets carry flag
CLR R6       ; set R6 to zero
ADC R6, R6   ; add with carry, R6 now holds the carry
```

Department of Mathematics
and Computer Science (IMADA)

Addition:

$$42 + 78 = ?$$

$$789 + 543 = ?$$

$$7862 + 5275 = ?$$

$$\begin{array}{lr} & 7862 \\ + & 5275 \\ \hline = & 13137 \end{array}$$

- Once school kids can add beyond 1000, they can add arbitrary numbers.

Multiprecision addition is old

Oh Līlāvatī, intelligent girl, if you understand addition and subtraction, tell me the sum of the amounts 2, 5, 32, 193, 18, 10, and 100, as well as [the remainder of] those when subtracted from 10000.

— "Līlāvatī" by Bhāskara (1150)

AVR multiprecision addition...

- Input pointers `X`, `Y`, output pointer `Z`.
- Add two n-byte numbers, returning an $(n+1)$-byte result:

```
LD R5,X+
LD R6,Y+
ADD R5,R6
ST Z+,R5
```

```
LD R5,X+
LD R6,Y+
ADC R5,R6
ST Z+,R5
```

...

```
CLR R5
ADC R5,R5
ST Z+,R5
```

. . . and subtraction.

- Input pointers X, Y, output pointer Z.
- Use highest byte $= -1$ to indicate negative result.
- Subtract two n-byte numbers, returning an $(n + 1)$-byte result:

```
LD R5,X+
LD R6,Y+
SUB R5,R6
ST Z+,R5
```

```
LD R5,X+
LD R6,Y+
SBC R5,R6
ST Z+,R5
```

. . .

```
CLR R5
SBC R5,R5
ST Z+,R5
```

How about multiplication?

Consider multiplication of 1234 by 789:

$$
\begin{array}{rr}
 & 1234 \cdot 789 \\
\hline
 & 11106 \\
+ & 9872 \\
+ & 9638 \\
\hline
= & 973626
\end{array}
\qquad
\begin{array}{r}
1234 \cdot 789 \\
\hline
973626
\end{array}
$$

- This is also an old technique.
- Early reference again in the Līlāvatī (1150).

Let's do that on the AVR. . .

0x 4F E3 AB · 0x E7 A4 40

↑ X (at AB) ↑ Y (at 40)

R0		R8
R1		R9
R2		R10
R3		R11
R4		R12
R5	–	R13
R6	–	R14
R7		R15

Carry:

```
⇒ 1: LD R2, X+
⇒ 2: LD R3, X+
⇒ 3: LD R4, X+

⇒ 4: LD R7, Y+

⇒ 5: MUL R2, R7
⇒ 6: ST  Z+, R0
⇒ 7: MOV R8, R1
```

```
⇒ 8: MUL R3, R7
⇒ 9: ADD R8, R0
⇒10: CLR R9
⇒11: ADC R9, R1

⇒12: MUL R4, R7
⇒13: ADD R9, R0
⇒14: CLR R10
⇒15: ADC R10, R1
```

Let's do that on the AVR...

$0x\,4F\,E3\,AB \cdot 0x\,E7\,A4\,40$

(↑ X at AB, ↑ Y at 40)

R0		R8
R1		R9
R2		R10
R3		R11
R4		R12
R5	–	R13
R6	–	R14
R7		R15

Carry:

```
⇒16: LD   R7,  Y+
⇒17: MUL R2,  R7
⇒18: MOVW R12,R0

⇒19: MUL R3,  R7
⇒20: ADD R13, R0
⇒21: CLR R14
⇒22: ADC R14, R1

⇒23: MUL R4,  R7
⇒24: ADD R14, R0
⇒25: CLR R15
⇒26: ADC R15, R1

⇒27: ADD R8,  R12
⇒28: ST  Z+,  R8
⇒29: ADC R9,  R13
⇒30: ADC R10, R14
⇒31: CLR R11
⇒32: ADC R11, R15
```

Let's do that on the AVR. . .

0x 4F E3 AB · 0x E7 A4 40

X ↑ (at AB), Y ↑ (at 40)

R0		R8
R1		R9
R2		R10
R3		R11
R4		R12
R5	–	R13
R6	–	R14
R7		R15

Carry:

```
⇒33: LD  R7,  Y+
⇒34: MUL R2,  R7
⇒35: MOVW R12,R0

⇒36: MUL R3,  R7
⇒37: ADD R13, R0
⇒38: CLR R14
⇒39: ADC R14, R1

⇒40: MUL R4,  R7
⇒41: ADD R14, R0
⇒42: CLR R15
⇒43: ADC R15, R1
```

```
⇒44: ADD R9,  R12
⇒45: ST  Z+,  R9
⇒46: ADC R10, R13
⇒47: ADC R11, R14
⇒48: CLR R12
⇒49: ADC R12, R15

⇒50: ST Z+,R10
⇒51: ST Z+,R11
⇒52: ST Z+,R12
```

Department of Mathematics
and Computer Science (IMADA)

- Problem: Need $3n + c$ registers for $n \times n$-byte multiplication.
- Can add on the fly, get down to $2n + c$, but more carry handling.

Can we do better?

> *Again as the information is understood, the multiplication of 2345 by 6789 is proposed; therefore the numbers are written down; the 5 is multiplied by the 9, there will be 45; the 5 is put, the 4 is kept; and the 5 is multiplied by the 8, and the 9 by the 4 and the products are added to the kept 4; there will be 80; the 0 is put and the 8 is kept; and the 5 is multiplied by the 7 and the 4 by the 8 and the 9 by the 3, and the products are added to the kept 8; there will be 102; the 2 is put and the 10 is kept in hand . . .*

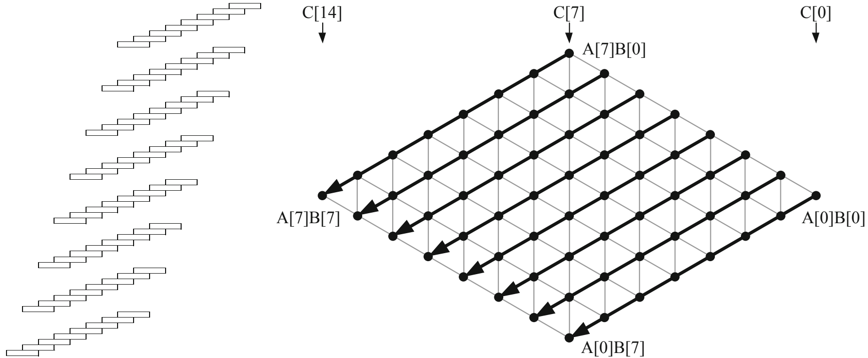
From "Fibonacci's Liber Abaci", Fibonacci, 1202 (Chapter 2).
(English translation by Sigler)

Product Scanning on AVR


```
LD R2, X+
LD R3, X+
LD R4, X+
LD R7, Y+
LD R8, Y+
LD R9, Y+

MUL R2, R7
MOV R13, R1
STD Z+0, R0
CLR R14
CLR R15

MUL R2, R8
ADD R13, R0
ADC R14, R1
MUL R3, R7
ADD R13, R0
ADC R14, R1
ADC R15, R5
STD Z+1, R13
CLR R16

MUL R2, R9
ADD R14, R0
ADC R15, R1
ADC R16, R5
MUL R3, R8
ADD R14, R0
ADC R15, R1
ADC R16, R5
MUL R4, R7
ADD R14, R0
ADC R15, R1
ADC R16, R5
STD Z+2, R14
CLR R17

MUL R3, R9
ADD R15, R0
ADC R16, R1
ADC R17, R5
MUL R4, R8
ADD R15, R0
ADC R16, R1
ADC R17, R5
STD Z+3, R15

MUL R4, R9
ADD R16, R0
ADC R17, R1
STD Z+4, R16

STD Z+5, R17
```

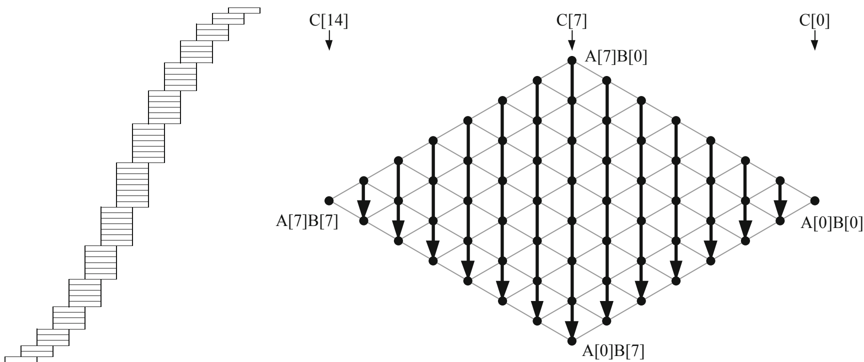
Department of Mathematics
and Computer Science (IMADA)

Operand scanning multiplication:



Hutter, Wenger: “Fast Multi-precision Multiplication for Public-Key Cryptography on Embedded Microprocessor”, J Cryptol (2018) 31:1164–1182

Product scanning multiplication:



Hutter, Wenger: “Fast Multi-precision Multiplication for Public-Key Cryptography on Embedded Microprocessor”, J Cryptol (2018) 31:1164–1182

Even better. . . ?

Products and sums are independent:

From “Treviso Arithmetic”, anonymous, 1478.

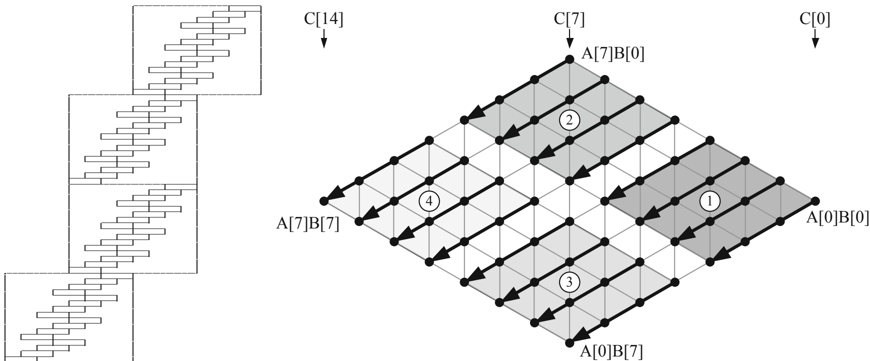
$\Rightarrow$ There is much flexibility in the order of the computation.

Hybrid Multiplication

- Idea: Chop whole multiplication into smaller blocks.
- Compute each of the smaller multiplications by schoolbook.
- Later add up to the full result.
- See it as two nested loops:
  - inner loop performs operand scanning,
  - outer loop performs product scanning.
- Originally proposed by Gura, Patel, Wander, Eberle, Chang Shantz, 2004.
- Various improvements, consider 160-bit multiplication:

Originally:	3106 cycles
Uhsadel, Poschmann, Paar (2007):	2881 cycles
Scott, Szczechowiak (2007):	2651 cycles
Kargl, Pyka, Seuschek (2008):	2593 cycles

Hybrid multiplication:

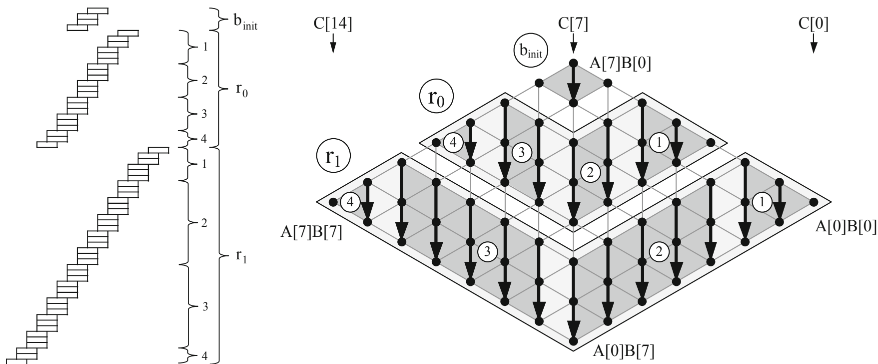


Hutter, Wenger: “Fast Multi-precision Multiplication for Public-Key Cryptography on Embedded Microprocessor”, J Cryptol (2018) 31:1164–1182

Operand-caching Multiplication

- Hutter, Wenger, 2011: More efficient way to decompose multiplication.
- Inside separate chunks use product-scanning.
- Main idea: re-use values in registers for longer.
- Performance:
  - 2393 cycles for 160-bit multiplication.
  - 6121 cycles for 256-bit multiplication.
- Followup-paper by Seo and Kim: “Consecutive operand caching”:
  - 2341 cycles for 160-bit multiplication.
  - 6115 cycles for 256-bit multiplication.

Operand-caching multiplication:

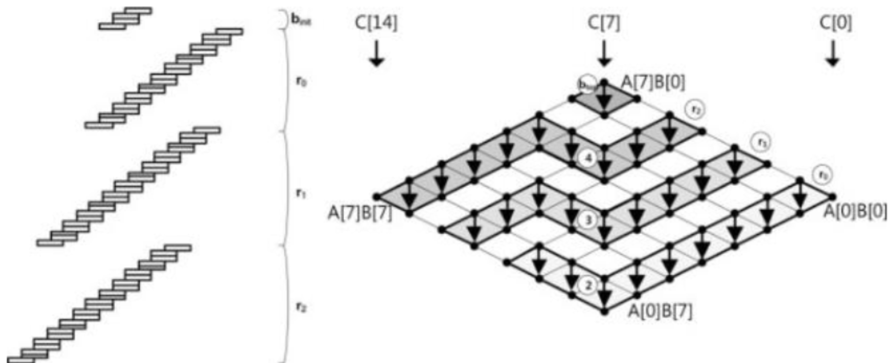


Hutter, Wenger: "Fast Multi-precision Multiplication for Public-Key Cryptography on Embedded Microprocessor", J Cryptol (2018) 31:1164–1182

Operand-caching Multiplication

- Hutter, Wenger, 2011: More efficient way to decompose multiplication.
- Inside separate chunks use product-scanning.
- Main idea: re-use values in registers for longer.
- Performance:
  - 2393 cycles for 160-bit multiplication.
  - 6121 cycles for 256-bit multiplication.
- Followup-paper by Seo and Kim: “Consecutive operand caching”:
  - 2341 cycles for 160-bit multiplication.
  - 6115 cycles for 256-bit multiplication.

Consecutive operand-caching multiplication:



Seo, Kim: “Consecutive Operand-Caching Method for Multiprecision Multiplication, Revisited”, J. Inf. Commun. Converg. Eng. (2015) 13(1): 27–25

Multiplication Complexity

- So far, multiplication of two n-byte numbers needs n^2 `MUL`s.
- Kolmogorov conjectured 1952:
  You can't do better, multiplication has quadratic complexity.
- Proven wrong by 23-year old student Karatsuba in 1960.
- Idea: write $A \cdot B$ as $(A_0 + 2^m A_1)(B_0 + 2^m B_1)$ for half-size A_0, B_0, A_1, B_1.
- Compute:

$$\begin{aligned} & A_0 \times B_0 \quad + 2^m (A_0 \times B_1 + A_1 \times B_0) \quad + 2^{2m} A_1 \times B_1 \\ = \; & A_0 \times B_0 \quad + 2^m ((A_0 + A_1) \times (B_0 + B_1) - A_0 \times B_0 - A_1 \times B_1) \quad + 2^{2m} A_1 \times B_1 \end{aligned}$$

- Instead of 4 mult and 3 add, we get 3 mult and 4 add and 2 sub.
  $\longrightarrow$ Savings iff mul more expensive than add/sub!
- Recursive application yields $O(n^{\log_2 3}) \approx n^{1.58}$ `MUL`s.

Straight-forward Approach

Consider multiplication of n-byte numbers

$$A \hat{=} (a_0, \ldots, a_{n-1}) \text{ and}$$
$$B \hat{=} (b_0, \ldots, b_{n-1})$$

- Write $A = A_l + 2^{8k} A_h$ and $B = B_l + 2^{8k} B_h$ for k-byte integers A_l, A_h, B_l, and B_h and $k = n/2$.
- Compute $L = A_l B_l \hat{=} (l_0, \ldots, l_{n-1})$.
- Compute $H = A_h B_h \hat{=} (h_0, \ldots, h_{n-1})$.
- Compute $M = (A_l + A_h)(B_l + B_h) \hat{=} (m_0, \ldots, m_n)$.
- Obtain result as $AB = L + 2^{8k}(M - L - H) + 2^{8n} H$.

Multiplication by the Carry in M

- There is an off number of 'limbs' in the middle multiplication.
- Expand carry to `0xff` or `0x00`.
- Use `AND` instruction for multiplication.
- Does not help for recursive Karatsuba!

Subtractive Karatsuba:

- Compute $L = A_l B_l \hat{=} (l_0, \ldots, l_{n-1})$.
- Compute $H = A_h B_h \hat{=} (h_0, \ldots, h_{n-1})$.
- Compute $M = |A_l - A_h| \cdot |B_l - B_h| \hat{=} (m_0, \ldots, m_{n-1})$.
- Set $t = 0$, if $M = (A_l - A_h) \cdot (B_l - B_h)$; $t = 1$ otherwise.
- Compute $\hat{M} = (-1)^t M = (A_l - A_h)(B_l - B_h) \hat{=} (\hat{m}_0, \ldots, \hat{m}_{n-1})$ (conditional negation).
- Obtain result as $A \cdot B = L + 2^{8k}(L + H - \hat{M}) + 2^{8n} H$.

Conditional Negation

The easy solution:

```
if(b) a = -1
```

- `NEG` instruction does not help for multiprecision.
- Can subtract from zero, but subtraction would ‘overwrite’ zero and needs a register.
- Even worse, the `if` would create a timing side-channel!

Reminder: Negative numbers

Commonly, two’s complement is used for negative numbers.

To get the negative of a number: 01001011_b

1. invert all the bits and 10110100_b
2. add one. 10110101_b

Conditional Negation

The easy solution:

```
if(b) a = -1
```

- `NEG` instruction does not help for multiprecision.
- Can subtract from zero, but subtraction would 'overwrite' zero and needs a register.
- Even worse, the `if` would create a timing side-channel!

The constant-time solution:

- Produce condition bit as byte `0xff` or `0x00`. $\Longleftarrow$
- `XOR` all limbs with this condition byte. $\Longleftarrow$
- Negate the condition byte and obtain `0x01` or `0x00`.
- Add this value to the lowest byte.
- Ripple through the carry (`ADC` with zero).

- Don't negate the condition byte.
- Subtract the condition byte from all bytes.
- Saves two `NEG` instructions and the zero register.

- Consider example of 4×4-byte Karatsuba multiplication:

l_0	l_1	l_2	l_3	h_0	h_1	h_2	h_3
	$-$	$\hat{m}_0$	$\hat{m}_1$	$\hat{m}_2$	$\hat{m}_3$		
	$+$	l_0	l_1	l_2	l_3		
	$+$	h_0	h_1	h_2	h_3		

- Karatsuba performs some additions twice.
- Refined Karatsuba: Do them only once.
- Merge additions into computation of H.
- Compute $\mathbf{H} \hat{=} (\mathbf{h_0}, \mathbf{h_1}, h_2, h_3) = H + (l_2, l_3, 0, 0)$, thus:

l_0	l_1	l_0	l_1	$\mathbf{h}_0$	$\mathbf{h}_1$	$\mathbf{h}_2$	$\mathbf{h}_3$
	$-$	$\hat{m}_0$	$\hat{m}_1$	$\hat{m}_2$	$\hat{m}_3$		
	$+$	$\mathbf{h}_0$	$\mathbf{h}_1$	$\mathbf{h}_2$	$\mathbf{h}_3$		

- Note that $\mathbf{H}$ cannot “overflow”.
- Consequence: fewer additions, easier register allocation.

48-bit Karatsuba on AVR


```
CLR R22
CLR R23
MOVW R12, R22
MOVW R20, R22

LD R2, X+
LD R3, X+
LD R4, X+
LDD R5, Y+0
LDD R6, Y+1
LDD R7, Y+2

MUL R2, R7
MOVW R10, R0
MUL R2, R5
MOVW R8, R0
MUL R2, R6
ADD R9, R0
ADC R10, R1
ADC R11, R23

MUL R3, R7
MOVW R14, R0
MUL R3, R5
ADD R9, R0
ADC R10, R1
ADC R11, R14
ADC R15, R23
MUL R3, R6
ADD R10, R0
ADC R11, R1
ADC R12, R15

MUL R4, R7
MOVW R14, R0
MUL R4, R5
ADD R10, R0
ADC R11, R1
ADC R12, R14
ADC R15, R23
MUL R4, R6
ADD R11, R0
ADC R12, R1
ADC R13, R15
STD Z+0, R8
STD Z+1, R9
STD Z+2, R10

LD R14, X+
LD R15, X+
LD R16, X+
LDD R17, Y+3
LDD R18, Y+4
LDD R19, Y+5

SUB R2, R14
SBC R3, R15
SBC R4, R16
SBC R26, R26

SUB R5, R17
SBC R6, R18
SBC R7, R19
SBC R27, R27

EOR R2, R26
EOR R3, R26
EOR R4, R26
EOR R5, R27
EOR R6, R27
EOR R7, R27

SUB R2, R26
SBC R3, R26
SBC R4, R26
SUB R5, R27
SBC R6, R27
SBC R7, R27

MUL R14, R19
MOVW R24, R0
MUL R14, R17
ADD R11, R0
ADC R12, R1
ADC R13, R24
ADC R25, R23
MUL R14, R18
ADD R12, R0
ADC R13, R1
ADC R20, R25

MUL R15, R19
MOVW R24, R0
MUL R15, R17
ADD R12, R0
ADC R13, R1
ADC R20, R24
ADC R25, R23
MUL R15, R18
ADD R13, R0
ADC R20, R1
ADC R21, R25

MUL R16, R19
MOVW R24, R0
MUL R16, R17
ADD R13, R0
ADC R20, R1
ADC R21, R24
ADC R25, R23
MUL R16, R18
MOVW R18,R22
ADD R20, R0
ADC R21, R1
ADC R22, R25

MUL R2, R7
MOVW R16, R0
MUL R2, R5
MOVW R14, R0
MUL R2, R6
ADD R15, R0
ADC R16, R1
ADC R17, R23

MUL R3, R7
MOVW R24, R0
MUL R3, R5
ADD R15, R0
ADC R16, R1
ADC R17, R24
ADC R25, R23
MUL R3, R6
ADD R16, R0
ADC R17, R1
ADC R18, R25

MUL R4, R7
MOVW R24, R0
MUL R4, R5
ADD R16, R0
ADC R17, R1
ADC R18, R24
ADC R25, R23
MUL R4, R6
ADD R17, R0
ADC R18, R1
ADC R19, R25

ADD R8, R11
ADC R9, R12
ADC R10, R13
ADC R11, R20
ADC R12, R21
ADC R13, R22
ADC R23, R23

EOR R26, R27
BRNE add_M

SUB R8, R14
SBC R9, R15
SBC R10, R16
SBC R11, R17
SBC R12, R18
SBC R13, R19
SBCI R23, 0
SBC R24, R24
RJMP final

add_M:
  ADD R8, R14
  ADC R9, R15
  ADC R10, R16
  ADC R11, R17
  ADC R12, R18
  ADC R13, R19
  CLR R24
  ADC R23, R24
  NOP

final:
  STD Z+3, R8
  STD Z+4, R9
  STD Z+5, R10
  STD Z+6, R11
  STD Z+7, R12
  STD Z+8, R13

  ADD R20, R23
  ADC R21, R24
  ADC R22, R24

  STD Z+9, R20
  STD Z+10, R21
  STD Z+11, R22
```

Larger Karatsuba Multiplication

- 48-bit Karatsuba is friendly; everything fits into registers.
- Remember that previous speed records were achieved by eliminating loads/stores.
- Karatsuba structure needs additional temporary storage.
- Good performance needs careful scheduling and register allocation.
- It is very important to compute $\mathbf{H} = H + (l_{k+1}, \ldots, l_{n-1})$ on the fly.
- Use 1-level Karatsuba for 48-bit, 64-bit, 80-bit, 96-bit inputs.
- Use 2-level Karatsuba for 128-bit, 160-bit, 192-bit inputs.
- Use 3-level Karatsuba for 256-bit inputs.

Cycle counts for *n*-bit multiplication:

Approach	**Input size *n***							
	48	64	80	96	128	160	192	256
Product scanning:	235	395	595	836	—	—	—	—
Hutter, Wenger, 2011:	—	—	—	—	—	2393	3467	6121
Seo, Kim, 2012:	—	—	—	—	1532	2356	3464	6180
Seo, Kim, 2013:	—	—	—	—	1523	2341	3437	6115
Karatsuba:	217	360	522	780	1325	**1976**	2923	**4797**
— w/o branches:	222	368	533	800	1369	2030	2987	4961

- 160-bit multiplication $>$ 18% faster.
- 256-bit multiplication $>$ 23% faster.

Department of Mathematics
and Computer Science (IMADA)

- Schoolbook multiplication has an asymptotic cost of $O(n^2)$.
- Karatsuba with $O(n^{\log_2(3)}) \approx O(n^{1.58})$ splits operands recursively into two terms; there are variants for other splits, e.g., 3-, 5-, 6-, and 7-terms.
  (Peter L. Montgomery, "Five, Six, and Seven-Term Karatsuba-Like Formulae", IEEE Transactions On Computers, Vol. 54, No. 3, 2005.)
- For larger integers, Toom-Cook becomes more efficient, e.g., Toom-3 with
  $O(n^{\log_3(5)}) \approx O(n^{1.46})$.
- For even larger integers, FFT (Schönhage–Strassen) becomes more efficient with
  $O(n \log n \log \log n)$.
- For "astronomically large values", Fürer's algorithm is even more efficient with
  $O(n \log n \cdot 2^{\log^* n})$.
- The algorithm by Harvey and van der Hoeven achieves $O(n \log n)$.

The conjectured lower bound is $\Omega(n \log n)$ — so that should be it...