Divide-and-Conquer Approach for Designing Large-Operand Functions on Reconfigurable Computers

Miaoqing Huang, Esam El-Araby, and Tarek El-Ghazawi

The George Washington University

RSSI 2008

Outline

- Background
  - **Motivation**
  - **Challenges and Approach**
- A generic architecture and case studies
  - **Large-size-operand functions**
    - **Multiplication**
  - **Large-number-of-operands functions**
    - **FFT**
- Experimental results on SRC-6
- Conclusions

Motivation

- Numerical non-robustness
  - **Recurring phenomenon in scientific computing caused by**
    - **Numerical errors**
    - **Small-precision arithmetic in integer and/or floating-point computations**
  - **Applications include**
    - **Most problems in computational sciences and engineering**
      - Public-key cryptography (e.g. Long-integer modular multiplications in RSA)
      - Computational metrology & Coordinate Measuring Machines (CMMs)
      - Computation of fundamental mathematical constants such as π to millions of digits
      - Rendering Fractal images with an extremely high magnification
    - **Computational Geometry**
      - Geometric editing and modeling
- Slower arithmetic performance when compared to small-precision arithmetic
- Limited hardware support

Challenges and Approach

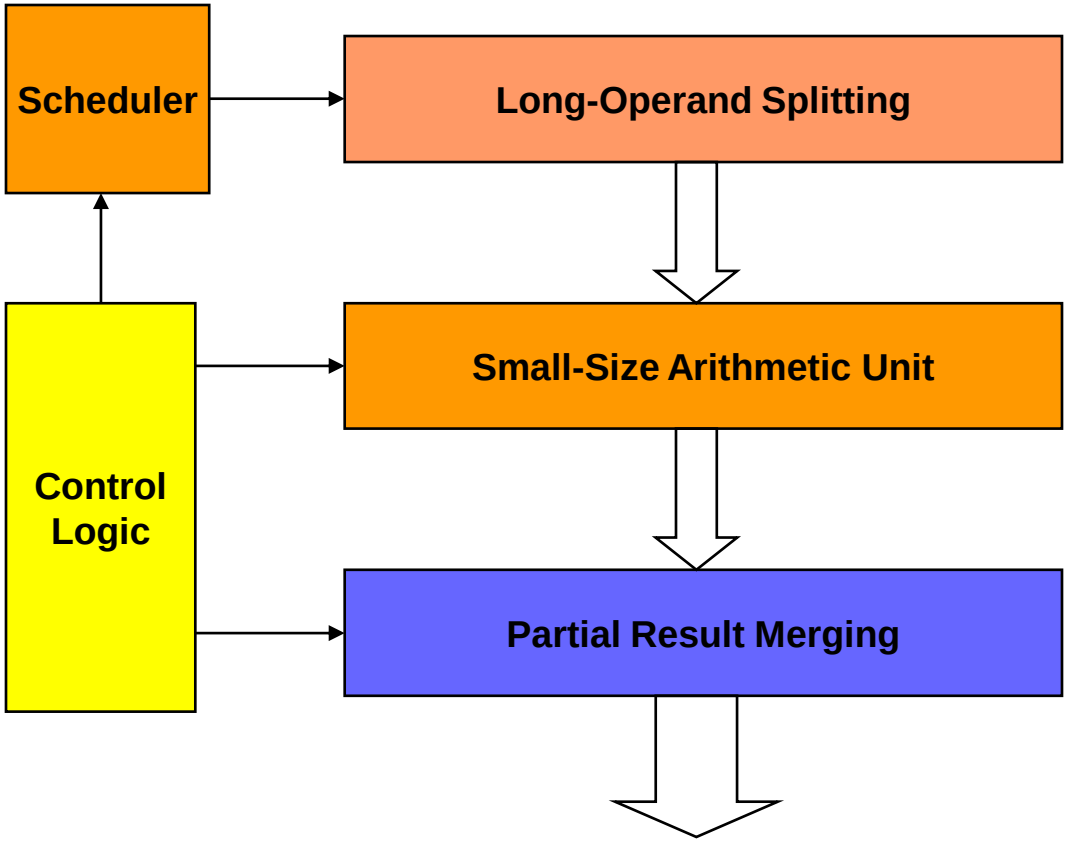
- Challenges
  - **FPGA devices have limited resource**
  - **Two types of large-operand functions**
    - **Large-size-operand functions**
      - Number of operands is small
      - Precision of operands is large
      - Typical example of this category: reduction operations
        - Multiplications: 2 inputs → 1 output
    - **Large-number-of-operands functions**
      - Number of operands is large
      - Precision of operands is small
      - Typical example of this category: transformation operations
        - FFT and sorting: *n* inputs → *n* outputs
- Approach
  - **Divide-and-Conquer technique**
  - **Generic architecture for large-operand arithmetic**
    - **Small-operand unit**
    - **Scheduler to fetch the small suboperands**
    - **Merger for partial results into final results**

Outline

- Background
  - Motivation
  - Challenges and Approach
- A generic architecture and case studies
  - **Large-size-operand functions**
    - **Multiplication**
  - **Large-number-of-operands functions**
    - **FFT**
- Experimental results on SRC-6
- Conclusions

A Generic Architecture

- **Components**
  - Large operands are split into suboperands
  - A small-size arithmetic unit performs the real operations
  - Scheduler fetches the suboperands for the small-size unit
  - Partial results may need to be merged
    - Reduction operations
- **General fetching pattern**
  1. Divide each operand into two suboperands
  2. If these suboperands can be further divided, go back to step 1
  3. Perform the operations among these suboperands
     - Size of suboperands matches the small-size unit



512bit×512bit Multiplier: a case of reduction operations

- Reuse a 32bit×32bit multiplier to perform the large-size multiplications
- A merging step is required

Karatsuba Algorithm: Multiplication using small multiplier

$$
\begin{aligned}
A &= A_1 \cdot 10^k + A_0; && /* A_0 < 10^k */ \\
B &= B_1 \cdot 10^k + B_0; && /* B_0 < 10^k */ \\
C &= A \cdot B \\
&= (A_1 \cdot 10^k + A_0) \cdot (B_1 \cdot 10^k + B_0) \\
&= A_1 \cdot B_1 \cdot 10^{2k} + (A_1 \cdot B_0 + A_0 \cdot B_1) \cdot 10^k + A_0 \cdot B_0;
\end{aligned}
$$

A: $A_{15}A_{14}A_{13}\ldots A_1A_0$ $\quad A_n$:32bit
B: $B_{15}B_{14}B_{13}\ldots B_1B_0$ $\quad B_n$:32bit

$$AB = A_{15\text{-}8}B_{15\text{-}8} \bullet 2^{512} + (A_{15\text{-}8}B_{7\text{-}0} + A_{7\text{-}0}B_{15\text{-}8}) \bullet 2^{256} + A_{7\text{-}0}B_{7\text{-}0}$$

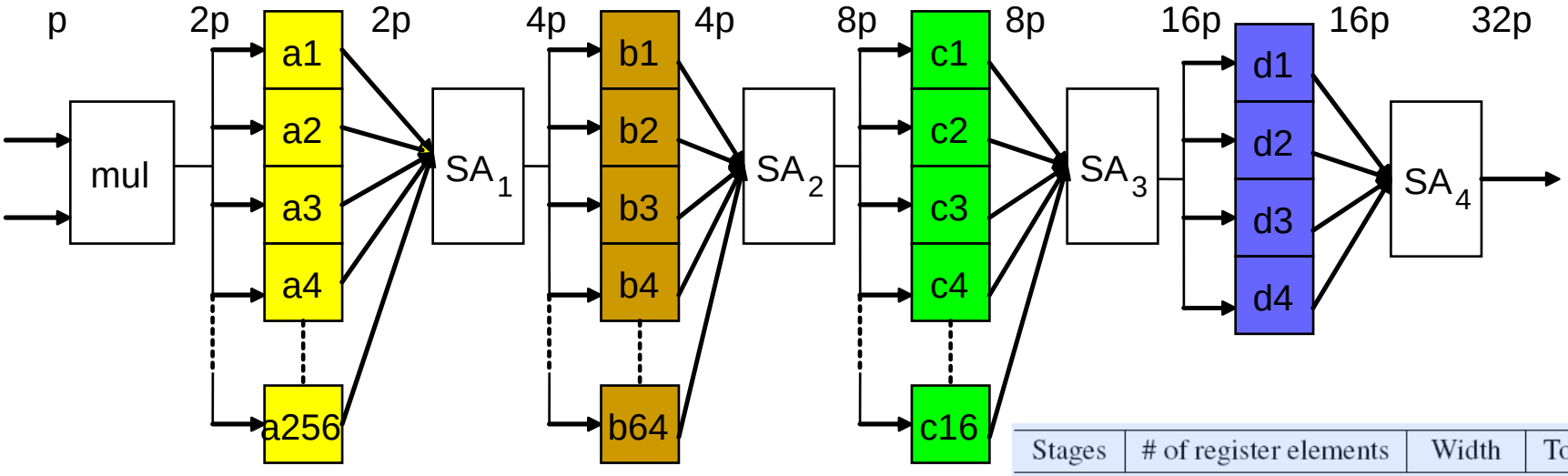
$$A_{7\text{-}0}B_{7\text{-}0} = A_{7\text{-}4}B_{7\text{-}4} \bullet 2^{256} + (A_{7\text{-}4}B_{3\text{-}0} + A_{3\text{-}0}B_{7\text{-}4}) \bullet 2^{128} + A_{3\text{-}0}B_{3\text{-}0}$$

$$A_{3\text{-}0}B_{3\text{-}0} = A_{3\text{-}2}B_{3\text{-}2} \bullet 2^{128} + (A_{3\text{-}2}B_{1\text{-}0} + A_{1\text{-}0}B_{3\text{-}2}) \bullet 2^{64} + A_{1\text{-}0}B_{1\text{-}0}$$

$$A_{1\text{-}0}B_{1\text{-}0} = A_1B_1 \bullet 2^{64} + (A_1B_0 + A_0B_1) \bullet 2^{32} + A_0B_0$$

512bit×512bit Multiplier *(cont.)*

- The merging step is responsible for processing the partial products → the final result
- Two approaches to implement the merging step
  - **Performance oriented → Merge on the fly (MoF)**
    - **Store the intermediate results in registers**

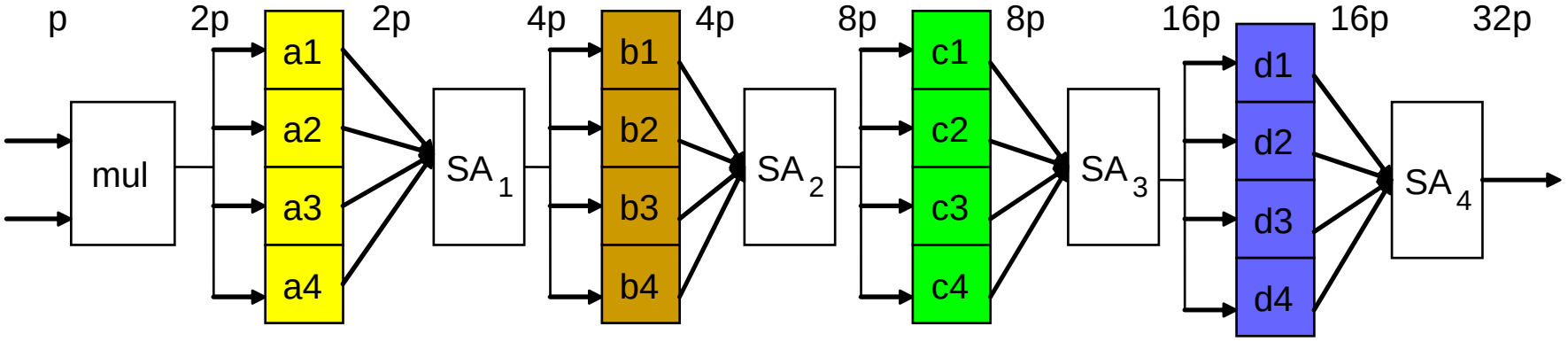


  - **Resource oriented → Merging in memory (MiM)**
    - **Store the intermediate results in memory**

Stages	# of register elements	Width	Total bits
4^{th}	4	512-bit	2,048
3^{rd}	16	256-bit	4,096
2^{nd}	64	128-bit	8,192
1^{st}	256	64-bit	16,384

512bit×512bit Multiplier *(cont.)*

- The merging step is responsible for processing the partial products → the final result
- Two approaches to implement the merging step
  - **Performance oriented → Merge on the fly (MoF)**
    - **Store the intermediate results in registers**



  - **Resource oriented → Merging in memory (MiM)**
    - **Store the intermediate results in memory**

512bit×512bit Multiplier *(cont.)*

- Generating the fetching schedule

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A																
B																

$(i_3, j_3, i_2, j_2, i_1, j_1, i_0, j_0) = (0,0, 0,0, 0,0, 0,0)$

$(a_n, b_n) = (0,0)$

Pseudo Code for Generating the Schedule

```
for i3, j3 = 0 to 8 step 8 do
    for i2, j2 = 0 to 4 step 4 do
        for i1, j1 = 0 to 2 step 2 do
            for i0, j0 = 0 to 1 step 1 do
                an = i0 + i1 + i2 + i3;
                bn = j0 + j1 + j2 + j3;
```

512bit×512bit Multiplier *(cont.)*

- Generating the fetching schedule

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A																0
B															1	

$(i_3, j_3, i_2, j_2, i_1, j_1, i_0, j_0) = (0,0, 0,0, 0,0, 0,1)$

$(a_n, b_n) = (0,1)$

Pseudo Code for Generating the Schedule

```
for i3, j3 = 0 to 8 step 8 do
    for i2, j2 = 0 to 4 step 4 do
        for i1, j1 = 0 to 2 step 2 do
            for i0, j0 = 0 to 1 step 1 do
                an = i0 + i1 + i2 + i3;
                bn = j0 + j1 + j2 + j3;
```

512bit×512bit Multiplier *(cont.)*

- Generating the fetching schedule

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A															1	
B																0

$(i_3, j_3, i_2, j_2, i_1, j_1, i_0, j_0)$ = (0,0, 0,0, 0,0, 1,0)

(a_n, b_n) = (1,0)

Pseudo Code for Generating the Schedule

```
for i3, j3 = 0 to 8 step 8 do
    for i2, j2 = 0 to 4 step 4 do
        for i1, j1 = 0 to 2 step 2 do
            for i0, j0 = 0 to 1 step 1 do
                an = i0 + i1 + i2 + i3;
                bn = j0 + j1 + j2 + j3;
```

512bit×512bit Multiplier *(cont.)*

- Generating the fetching schedule

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A																
B																

$(i_3, j_3, i_2, j_2, i_1, j_1, i_0, j_0) = (0,0, 0,0, 0,0, 1,1)$

$(a_n, b_n) = (1,1)$

Pseudo Code for Generating the Schedule

```
for i3, j3 = 0 to 8 step 8 do
    for i2, j2 = 0 to 4 step 4 do
        for i1, j1 = 0 to 2 step 2 do
            for i0, j0 = 0 to 1 step 1 do
                an = i0 + i1 + i2 + i3;
                bn = j0 + j1 + j2 + j3;
```

512bit×512bit Multiplier *(cont.)*

- Generating the fetching schedule

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A																**0**
B														**2**		

$(i_3, j_3, i_2, j_2, i_1, j_1, i_0, j_0) = (0,0, 0,0, 0,2, 0,0)$

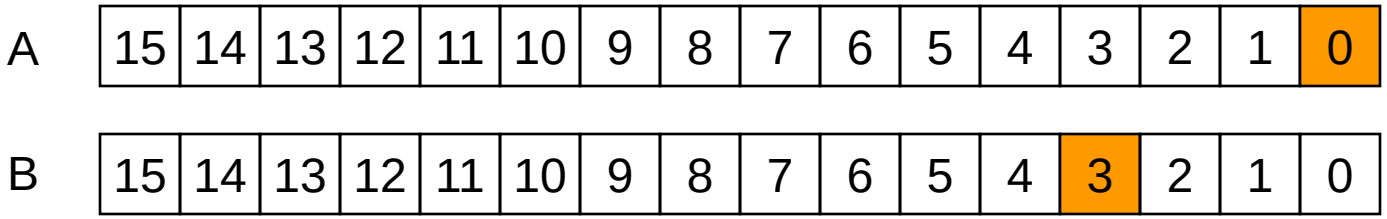
$(a_n, b_n) = (0,2)$

Pseudo Code for Generating the Schedule

```
for i3, j3 = 0 to 8 step 8 do
    for i2, j2 = 0 to 4 step 4 do
        for i1, j1 = 0 to 2 step 2 do
            for i0, j0 = 0 to 1 step 1 do
                an = i0 + i1 + i2 + i3;
                bn = j0 + j1 + j2 + j3;
```

512bit×512bit Multiplier *(cont.)*

- Generating the fetching schedule



$(i_3, j_3, i_2, j_2, i_1, j_1, i_0, j_0)$ = (0,0, 0,0, 0,2, 0,1)

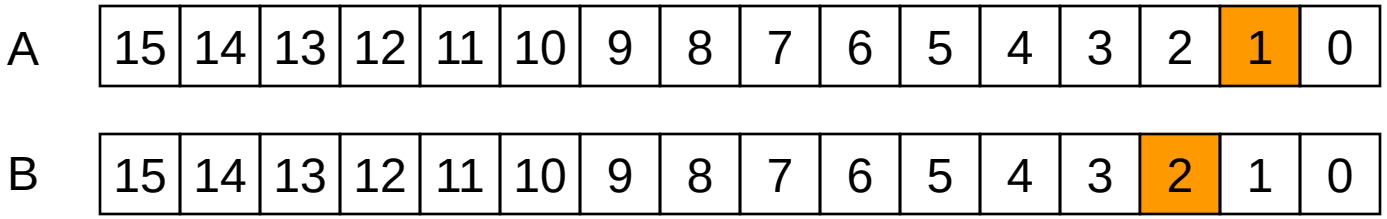
(a_n, b_n) = (0,3)

Pseudo Code for Generating the Schedule

```
for i3, j3 = 0 to 8 step 8 do
    for i2, j2 = 0 to 4 step 4 do
        for i1, j1 = 0 to 2 step 2 do
            for i0, j0 = 0 to 1 step 1 do
                an = i0 + i1 + i2 + i3;
                bn = j0 + j1 + j2 + j3;
```

512bit×512bit Multiplier *(cont.)*

- Generating the fetching schedule



$(i_3, j_3, i_2, j_2, i_1, j_1, i_0, j_0)$ = (0,0, 0,0, 0,2, 1,0)

(a_n, b_n) = (1,2)

Pseudo Code for Generating the Schedule

```
for i3, j3 = 0 to 8 step 8 do
    for i2, j2 = 0 to 4 step 4 do
        for i1, j1 = 0 to 2 step 2 do
            for i0, j0 = 0 to 1 step 1 do
                an = i0 + i1 + i2 + i3;
                bn = j0 + j1 + j2 + j3;
```

512bit×512bit Multiplier *(cont.)*

- Generating the fetching schedule

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A															**1**	
B													**3**			

$(i_3, j_3, i_2, j_2, i_1, j_1, i_0, j_0)$ = (0,0, 0,0, 0,2, 1,1)

(a_n, b_n) = (1,3)

Pseudo Code for Generating the Schedule

```
for i3, j3 = 0 to 8 step 8 do
    for i2, j2 = 0 to 4 step 4 do
        for i1, j1 = 0 to 2 step 2 do
            for i0, j0 = 0 to 1 step 1 do
                an = i0 + i1 + i2 + i3;
                bn = j0 + j1 + j2 + j3;
```

512bit×512bit Multiplier *(cont.)*

- Generating the fetching schedule

A	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

$(i_3, j_3, i_2, j_2, i_1, j_1, i_0, j_0)$ = (8,8, 4,4, 2,2, 0,0)

(a_n, b_n) = (14,14)

Pseudo Code for Generating the Schedule

```
for i3, j3 = 0 to 8 step 8 do
    for i2, j2 = 0 to 4 step 4 do
        for i1, j1 = 0 to 2 step 2 do
            for i0, j0 = 0 to 1 step 1 do
                an = i0 + i1 + i2 + i3;
                bn = j0 + j1 + j2 + j3;
```

512bit×512bit Multiplier *(cont.)*

- Generating the fetching schedule

A	15	**14**	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B	**15**	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

$(i_3, j_3, i_2, j_2, i_1, j_1, i_0, j_0)$ = (8,8, 4,4, 2,2, 0,1)

(a_n, b_n) = (14,15)

Pseudo Code for Generating the Schedule

```
for i3, j3 = 0 to 8 step 8 do
    for i2, j2 = 0 to 4 step 4 do
        for i1, j1 = 0 to 2 step 2 do
            for i0, j0 = 0 to 1 step 1 do
                an = i0 + i1 + i2 + i3;
                bn = j0 + j1 + j2 + j3;
```

512bit×512bit Multiplier *(cont.)*

- Generating the fetching schedule

A	**15**	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B	15	**14**	13	12	11	10	9	8	7	6	5	4	3	2	1	0

$(i_3, j_3, i_2, j_2, i_1, j_1, i_0, j_0)$ = (8,8, 4,4, 2,2, 1,0)

(a_n, b_n) = (15,14)

Pseudo Code for Generating the Schedule

```
for i3, j3 = 0 to 8 step 8 do
    for i2, j2 = 0 to 4 step 4 do
        for i1, j1 = 0 to 2 step 2 do
            for i0, j0 = 0 to 1 step 1 do
                an = i0 + i1 + i2 + i3;
                bn = j0 + j1 + j2 + j3;
```

512bit×512bit Multiplier *(cont.)*

- Generating the fetching schedule

A	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

$(i_3, j_3, i_2, j_2, i_1, j_1, i_0, j_0)$ = (8,8, 4,4, 2,2, 1,1)

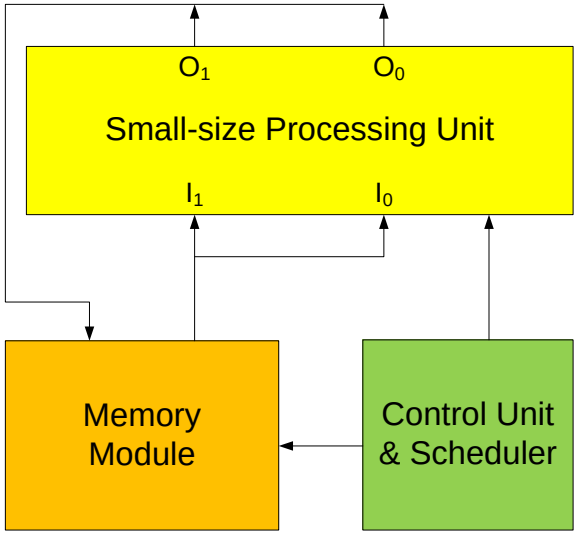
(a_n, b_n) = (15,15)

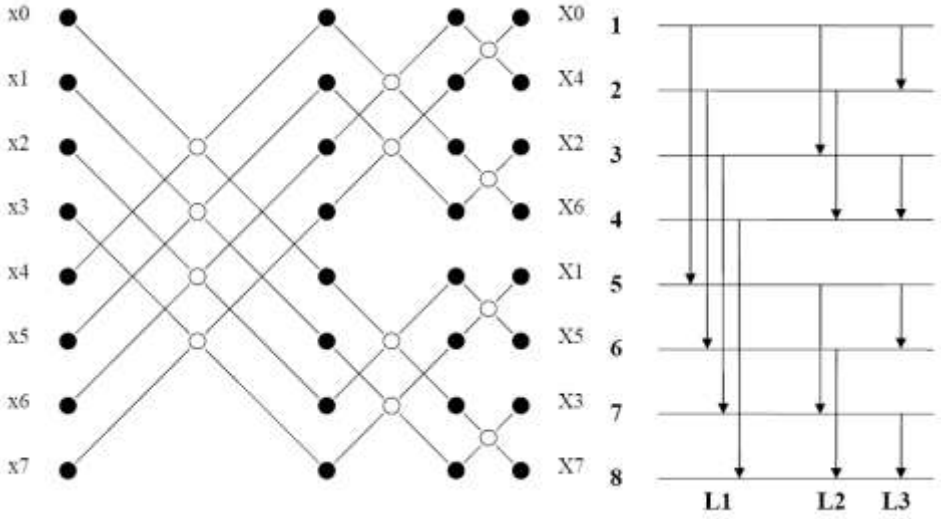
Pseudo Code for Generating the Schedule

```
for i3, j3 = 0 to 8 step 8 do
    for i2, j2 = 0 to 4 step 4 do
        for i1, j1 = 0 to 2 step 2 do
            for i0, j0 = 0 to 1 step 1 do
                an = i0 + i1 + i2 + i3;
                bn = j0 + j1 + j2 + j3;
```

Large-number-of-operands FFT: a case of transformation operations

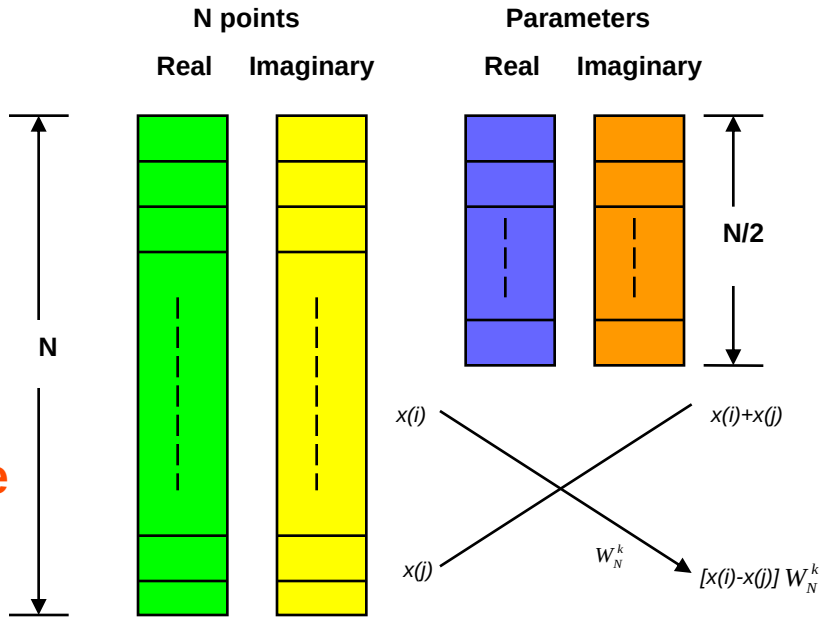
- A small-size processing unit with fixed I/O
- Input, intermediate results, and output are stored in memory
- No merging step
- A greedy method
  - **Generate the schedule level by level**
  - **Use line representation**





Large-number-of-operands FFT *(cont.)*

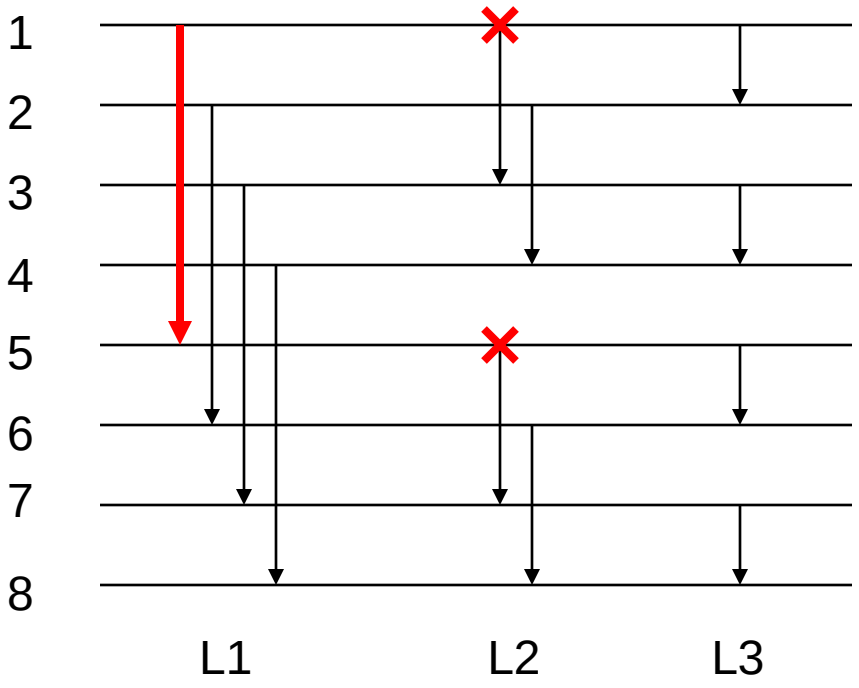
- Memory requirement for optimal performance
  - **Four separate memory modules for storing *N* points and *N*/2 twiddle factors**
  - **Separate reading and writing ports for accessing each memory module**



- Using a *k*-point FFT unit to perform *N*-point FFT
  - **Normal mode: performing *k*-point FFT operations**
    - **Used in the operations of last *log(k)* levels**
  - **Augmented mode: performing butterfly operations between *k* points**
    - **Used in the operations of first (*log(N)-log(k)*) levels**

Large-number-of-operands FFT *(cont.)*

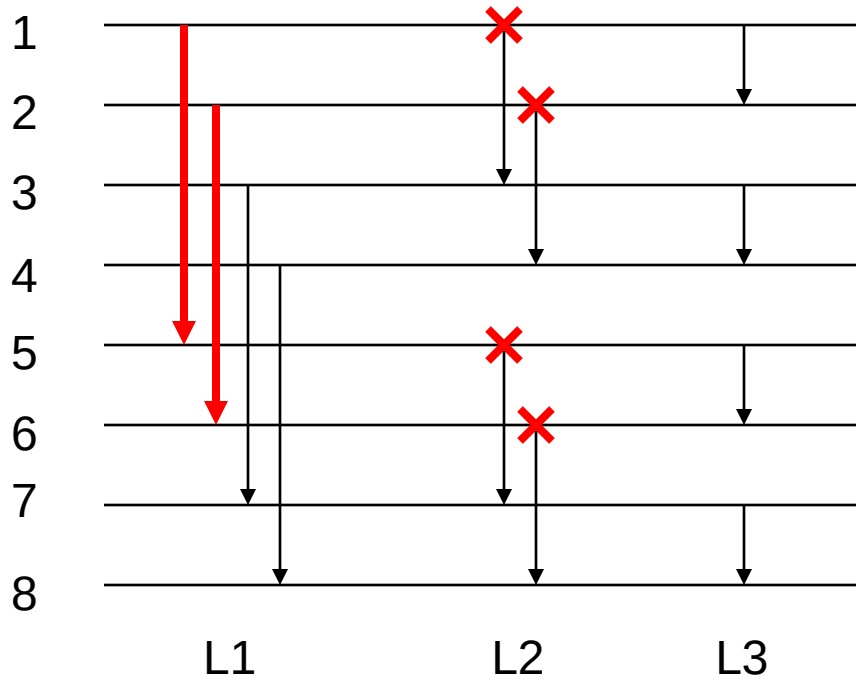
- Using greedy method to generate the schedule level by level



(1,5)

Large-number-of-operands FFT *(cont.)*

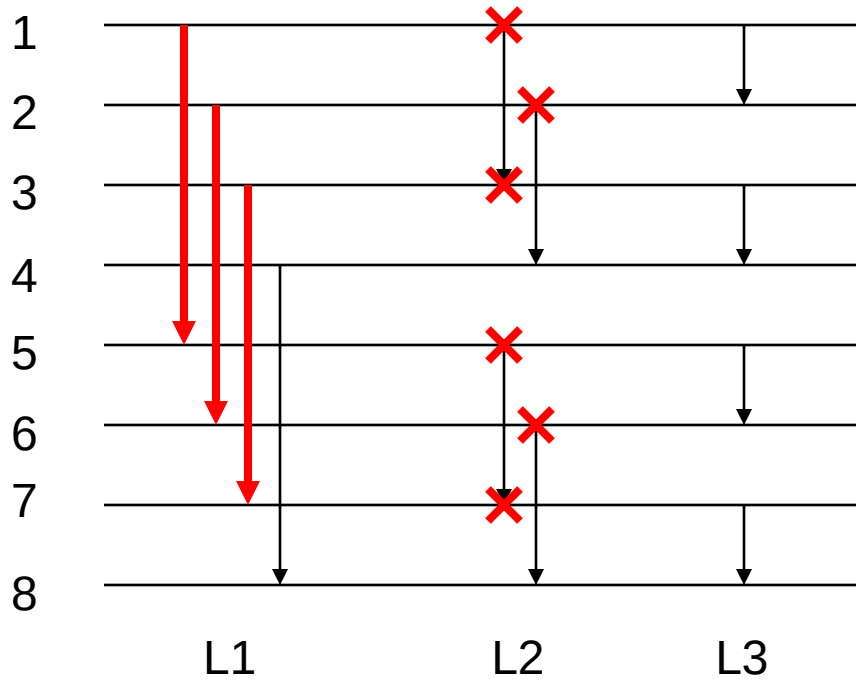
- Using greedy method to generate the schedule level by level



(1,5), (2,6)

Large-number-of-operands FFT *(cont.)*

- Using greedy method to generate the schedule level by level

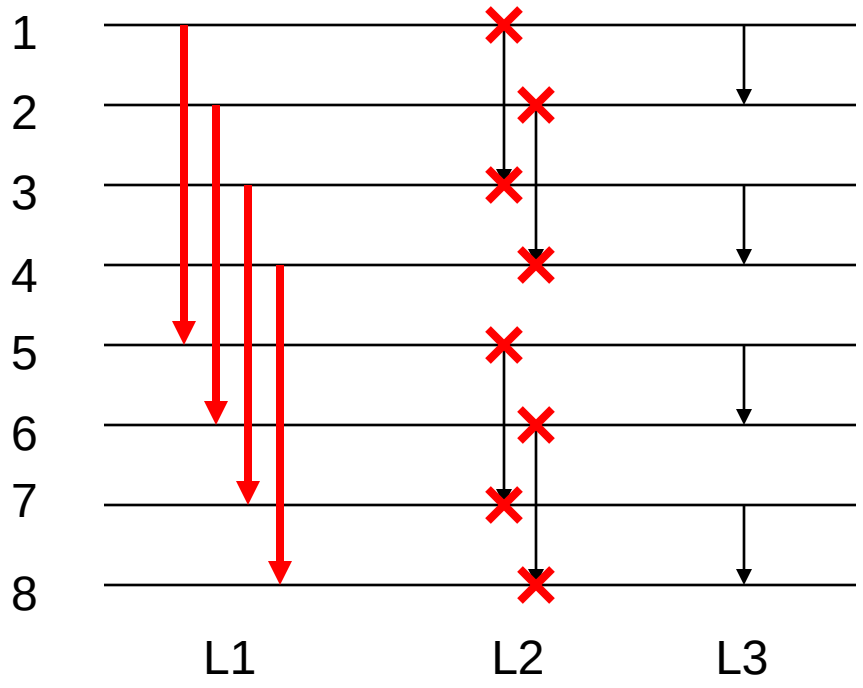


(1,5), (2,6), (3,7)

(1,3), (5,7)

Large-number-of-operands FFT *(cont.)*

- Using greedy method to generate the schedule level by level

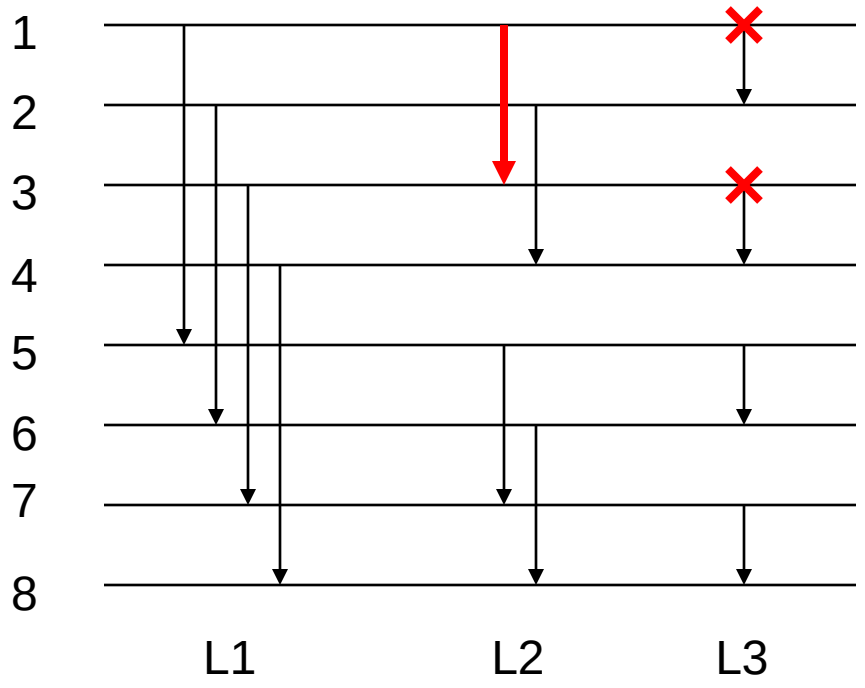


(1,5), (2,6), (3,7), (4,8),

(1,3), (5,7), (2,4), (6,8),

Large-number-of-operands FFT *(cont.)*

- Using greedy method to generate the schedule level by level

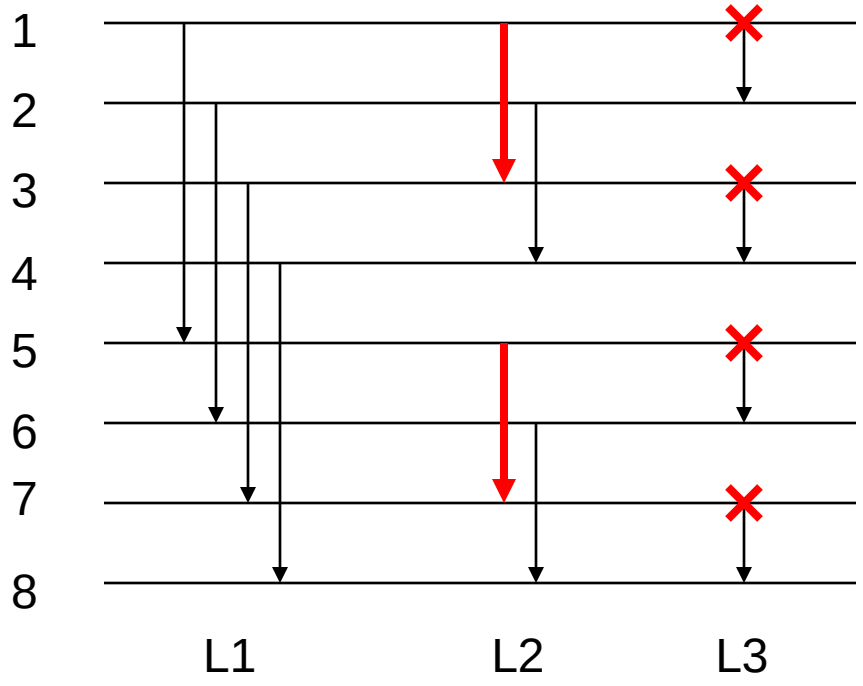


(1,5), (2,6), (3,7), (4,8),

(1,3), (5,7), (2,4), (6,8),

Large-number-of-operands FFT *(cont.)*

- Using greedy method to generate the schedule level by level

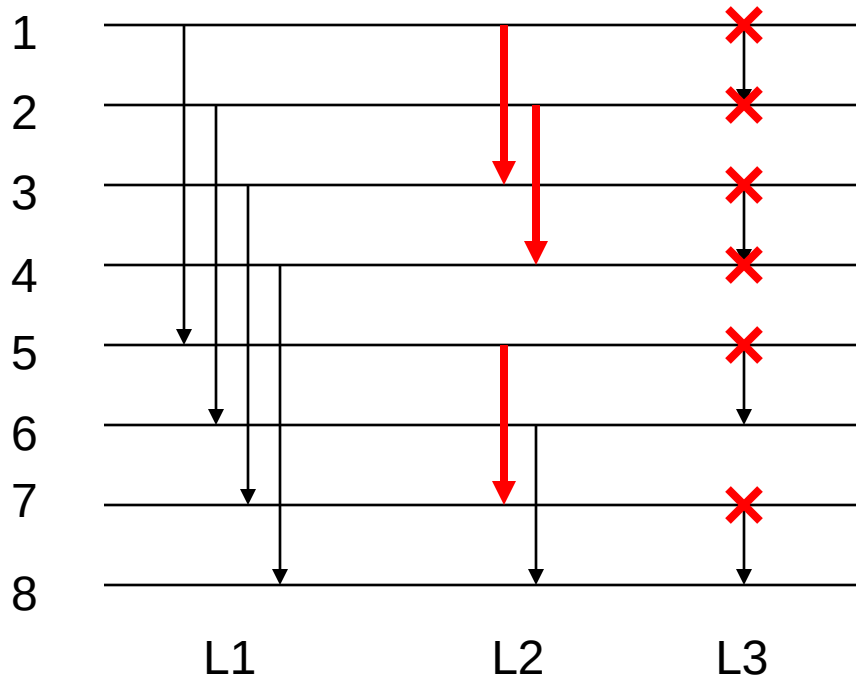


(1,5), (2,6), (3,7), (4,8),

(1,3), (5,7), (2,4), (6,8),

Large-number-of-operands FFT *(cont.)*

- Using greedy method to generate the schedule level by level



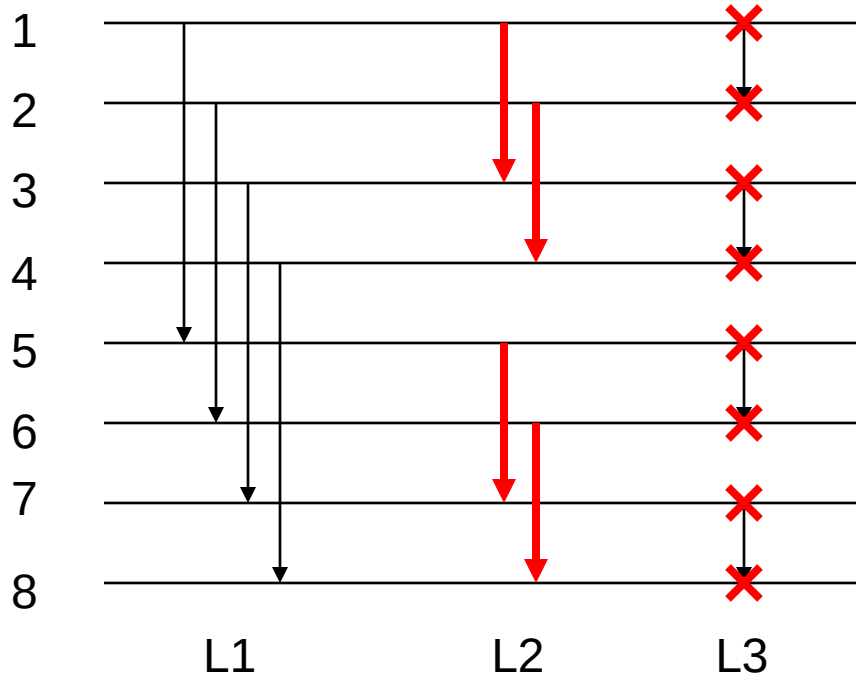
(1,5), (2,6), (3,7), (4,8),

(1,3), (5,7), (2,4), (6,8),

(1,2), (3,4),

Large-number-of-operands FFT *(cont.)*

- Using greedy method to generate the schedule level by level



(1,5), (2,6), (3,7), (4,8),

(1,3), (5,7), (2,4), (6,8),

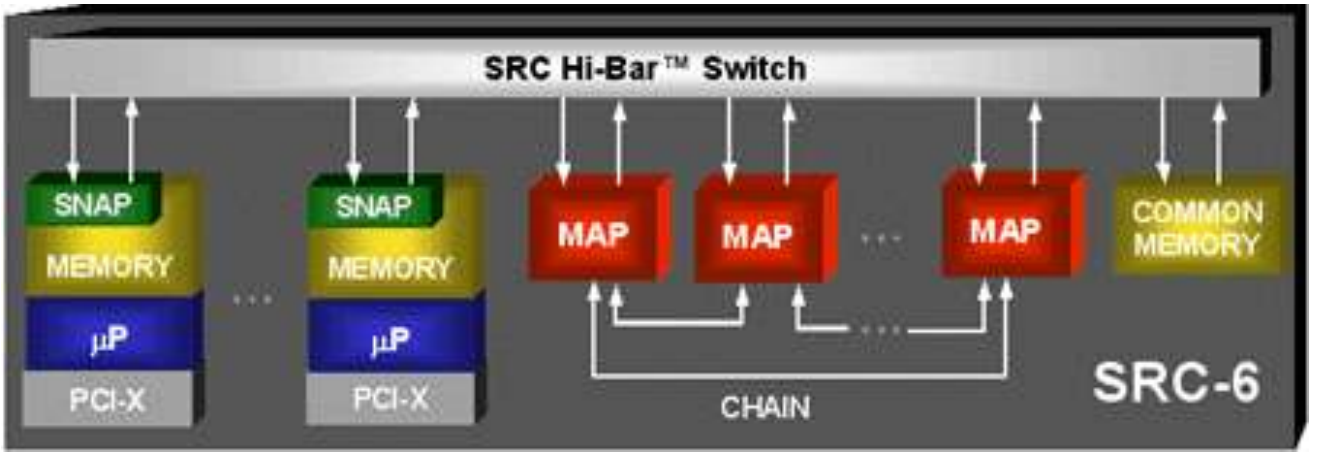
(1,2), (3,4), (5,6), (7,8),

Outline

- Background
  - **Motivation**
  - **Challenges and Approach**
- A generic architecture and case studies
  - **Large-size-operand functions**
    - **Multiplication**
  - **Large-number-of-operands functions**
    - **FFT**
- Experimental results on SRC-6
- Conclusions

Implementation results on SRC-6

- Test bed
  - **Hi-Bar Switch connecting µP board and MAP (FPGA) board**

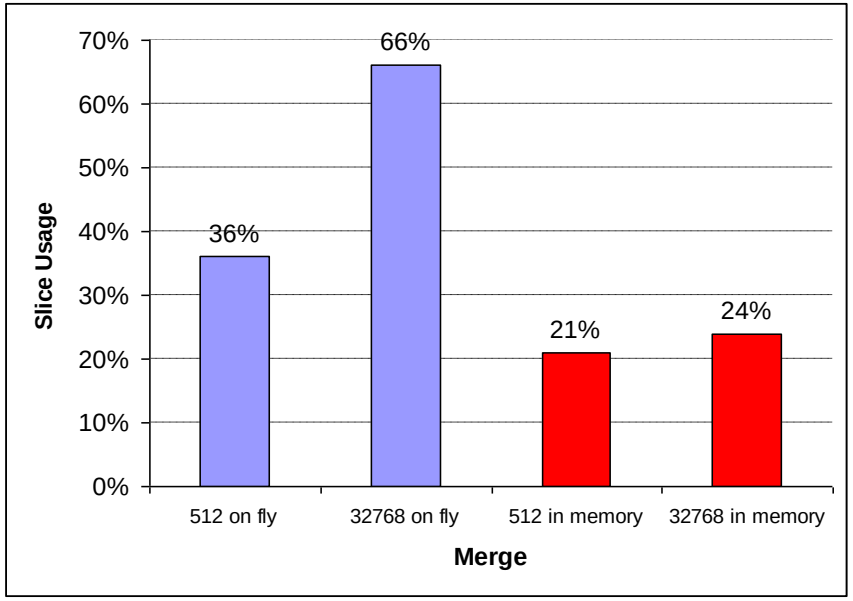


  - **Each MAP board consists of two Xilinx V2-6000 FPGA devices sharing 6 SRAM memory modules**
  - **Compute rate is 100MHz**
- Experiments
  - **32,768bitx32,768bit multiplier**
    - **Merge on the fly (MoF)**
    - **Merge in memory (MiM)**
  - **524,288-point FFT**

	Multiplication				FFT
	512x512		32,768x32,768		524,288 -point
	MoF	MiM	MoF	MiM	
Slices	12,284 (36%)	7,190 (21%)	22,465 (66%)	8,161 (24%)	12,630 (37%)

Implementation results on SRC-6 *(cont.)*

Large-operand Multiplication Performance



Operand Precision (bits)	SRC-6		µP Xeon 2.8GHz (µs)	Speedup
	MoF (µs)	MiM (µs)		
512	2.56	329.52	35	13
1,024	10.24	1,318.08	130	12
2,048	40.96	5,272.32	500	12
4,096	163.84	21,089.28	2,000	12
8,192	655.36	84,357.12	8,000	12
16,384	2,621.44	337,428.48	32,800	12
32,768	10,485.76	1,349,713.92	135,550	12

- **MoF**
  - **Highly scalable**
    - Upper Bounded by available resources
  - **High performance**
- **MiM**
  - **Highly scalable**
  - **Low performance**

Conclusions

- Large-size functions on reconfigurable computers
  - **Large-size-operand functions**
  - **Large-number-of-operands functions**
- Divide-and-Conquer approach
- A generic architecture
  - **Small-operand unit**
  - **Scheduler**
  - **Merger**
- Two case studies on SRC-6
  - **Reduction operations**
    - **32,768bit×32,768bit multiplier**
  - **Transformation operations**
    - **524,288-point FFT**
  - **Tradeoff between performance and resources**
    - **Merge on the fly**
    - **Merge in memory**