Design for Low-Power IoT Systems

Module 1:
Transport Triggering Approach
Jarmo Takala, Tampere University of Technology, Finland

Module 2:
Coarse-Grained Reconfigurable Acceleration Units
Francesca Palumbo, Università degli Studi di Sassari, Italy

Module 3:
Markov Decision Processes and Power/Performance/Thermal (PPT)
Marilyn Wolf, Georgia Tech., USA

Module 4:
Factored MDPs and Application Examples
Shuvra Bhattacharyya, U. Maryland, USA and Tampere U. Technology, Finland

ISCAS 2018 Tutorial, Florence, Italy, May 27, 2018

Design for Low-Power IoT Systems: Transport Triggering Approach

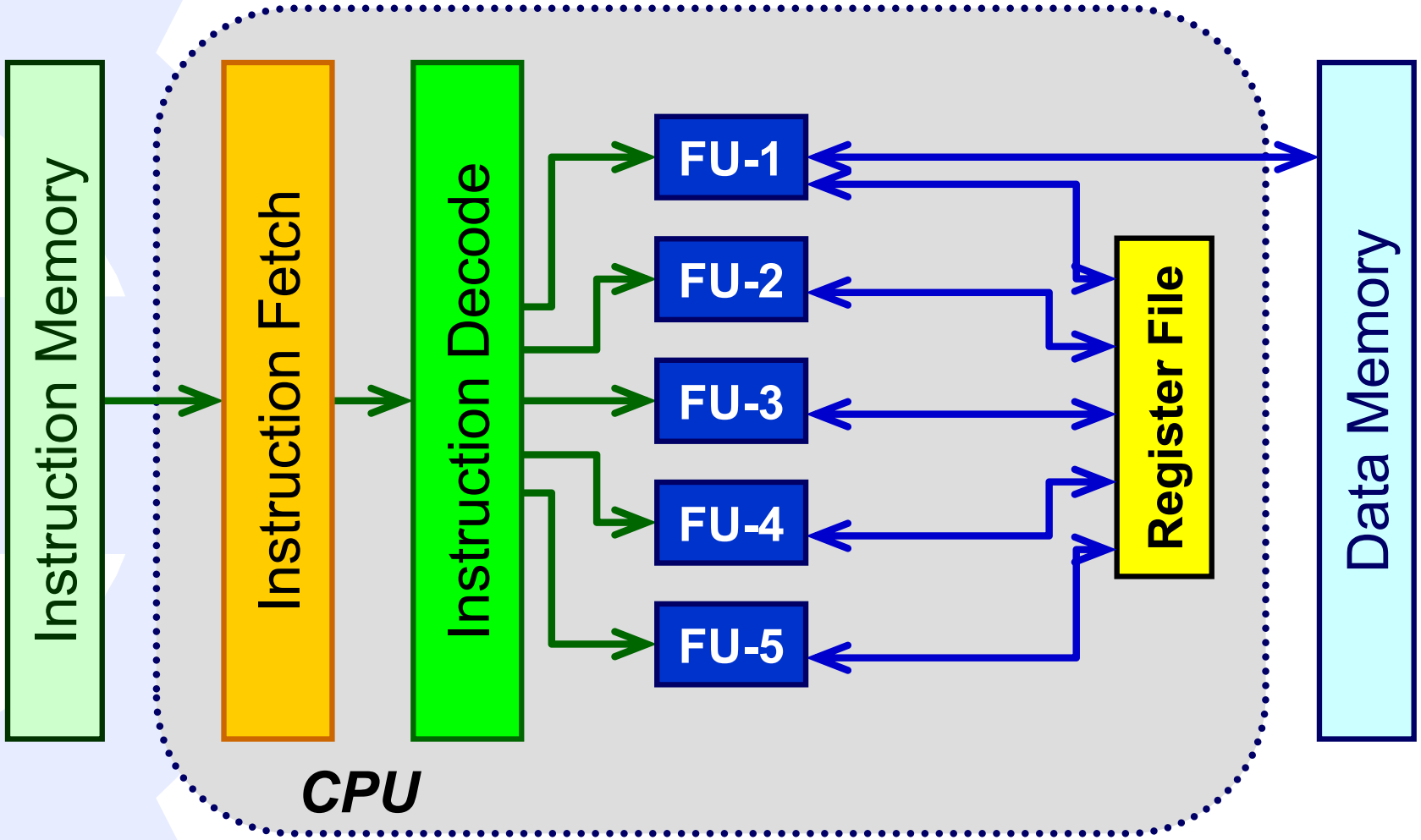
Jarmo Takala, Pekka Jääskeläinen, Timo Viitanen, Heikki Kultala, Joonas Multanen, Fahad Qureshi, Jakub Zadnik, Matias Koskela, and many more

Tampere University of Technology, Finland

Some slides thanks to Heikki Berg, Andrea Cilio, and Henk Corporaal

Transport Triggering

General ILP Architecture (VLIW)



Pipeline

Ifetch	Decode	R-read	Execute	R-write

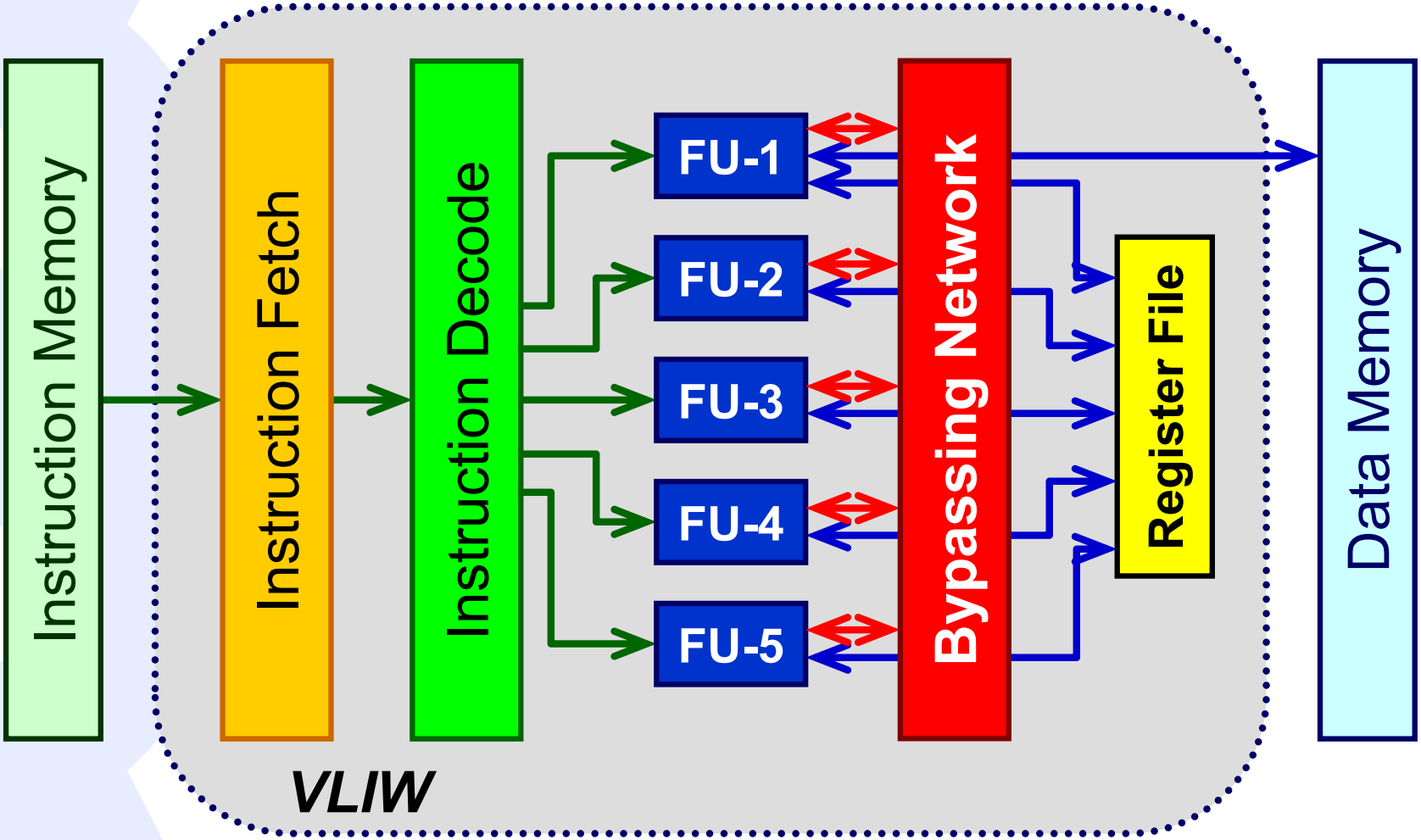
VLIW Architectures

- Strong points:
  - Scalable (add more FUs)
  - Flexible (an FU can be almost anything)
- Disadvantage:
  - Two-operand instructions need three register file accesses
    - With *N* FUs, 3*N* ports to register file are needed
  - Register file complexity O(*N*)

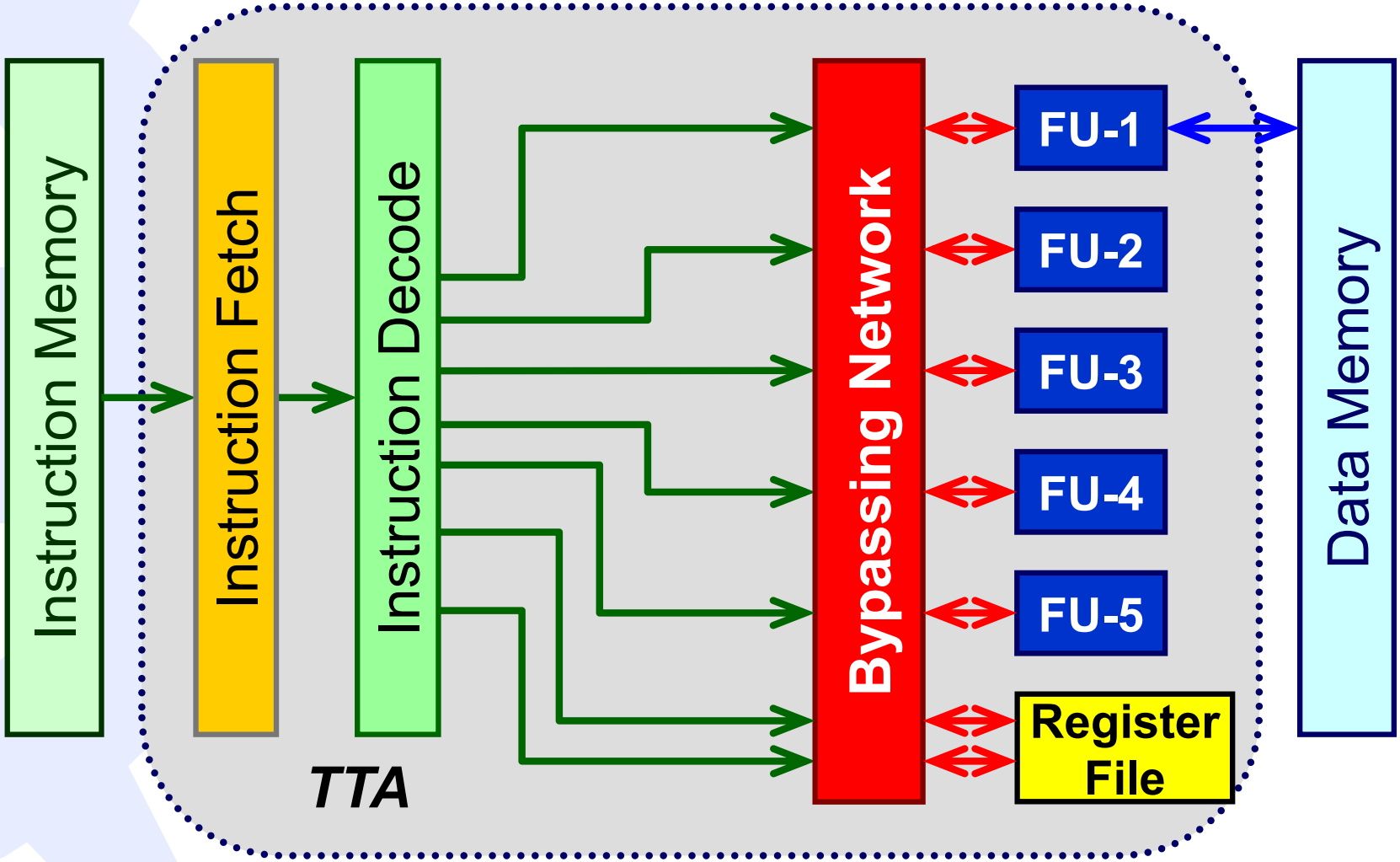
Reducing RF Complexity

- Fever source operands
  - All instructions will not use two operands
- No result
  - All instructions will not produce a result to RF
- Bypassing
  - Writeback to RF and operand read from RF avoided
  - Dead result elimination

From VLIW to TTA



Transport Triggered Architecture

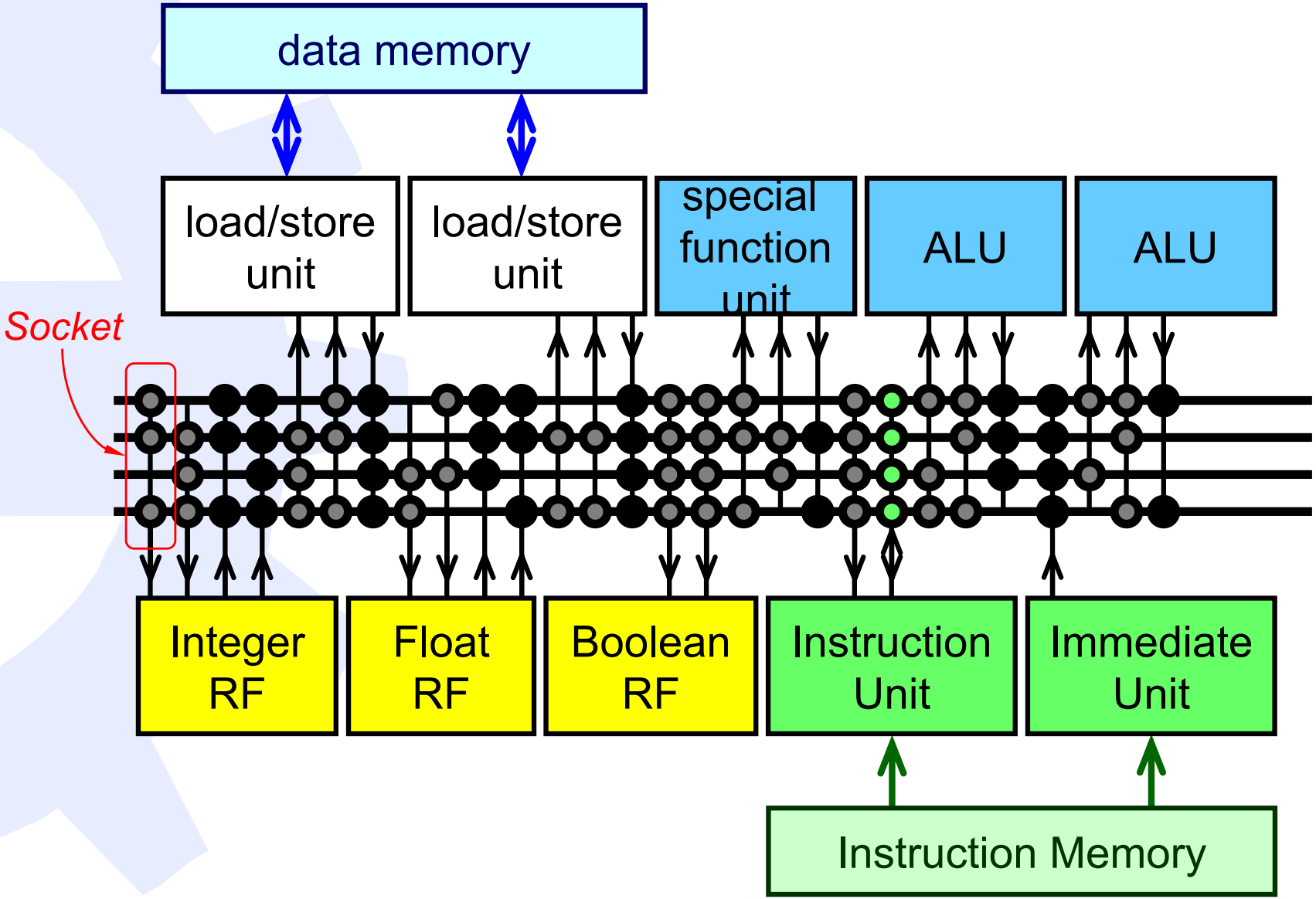


Transport Triggering

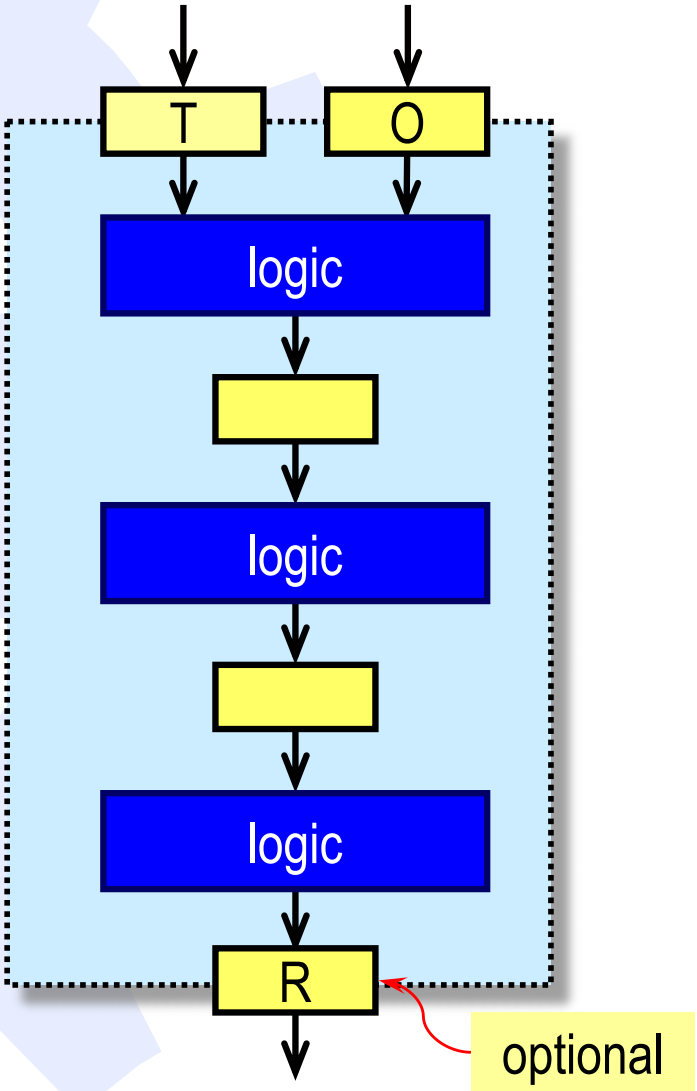
- Mirror the programming paradigm
  - Transports from RF to FU and from FU to RF are specified by instructions
  - Instruction controls the bypassing network

- Tabak&Lipovski, “MOVE Architecture in Digital Controllers,” IEEE J. Solid-State Circuits, vol. SC-15, no. 1, Feb. 1980
- Henk Corporaal, Transport Triggered Architectures: Design and Evaluation, dissertation, TU Delft, Netherlands, 1995
- MOVE tools, Henk Corporaal, TU Delft

TTA Datapath

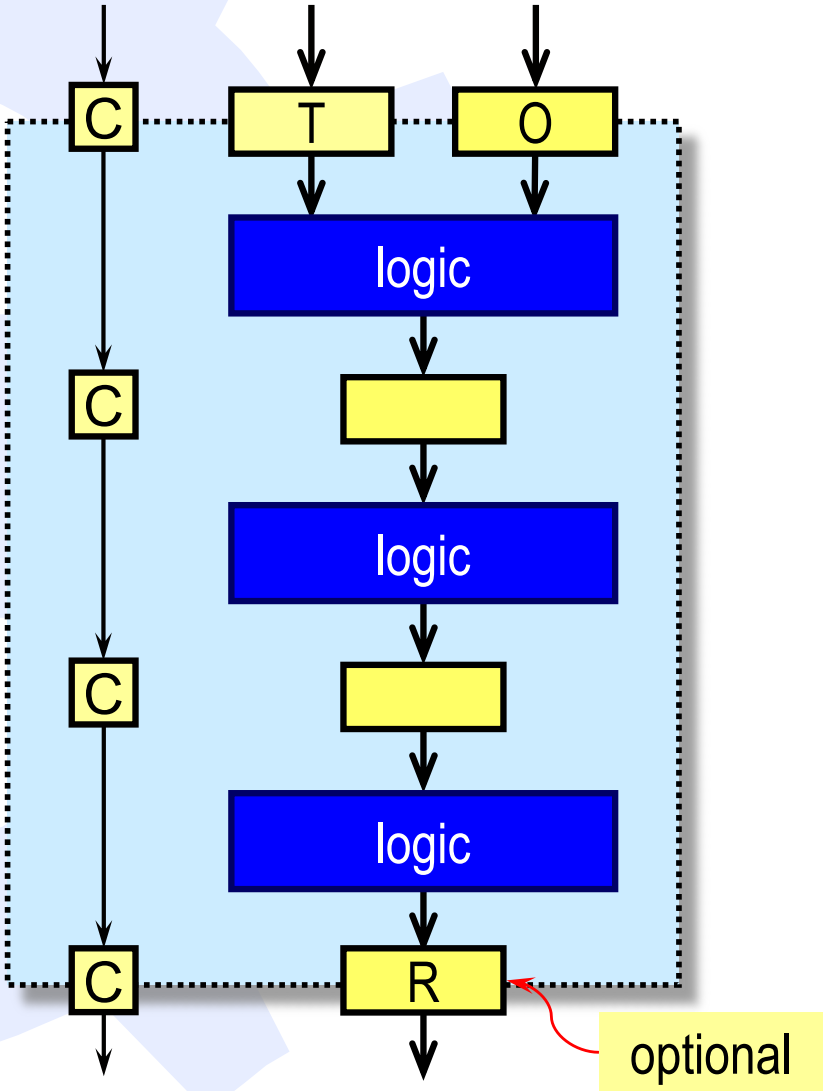


Functional Units



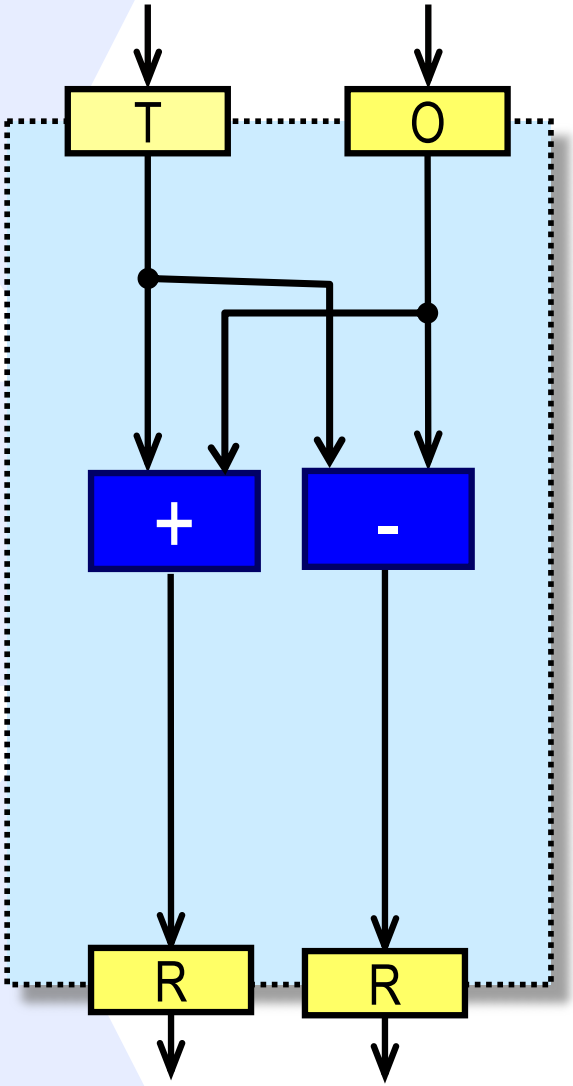
- Operand executed as an side effect on operand transport
- Operands written to operand registers (O)
- Operation executed when operand written to trigger register (T)

Semi Virtual-Time Latching (SVTL)

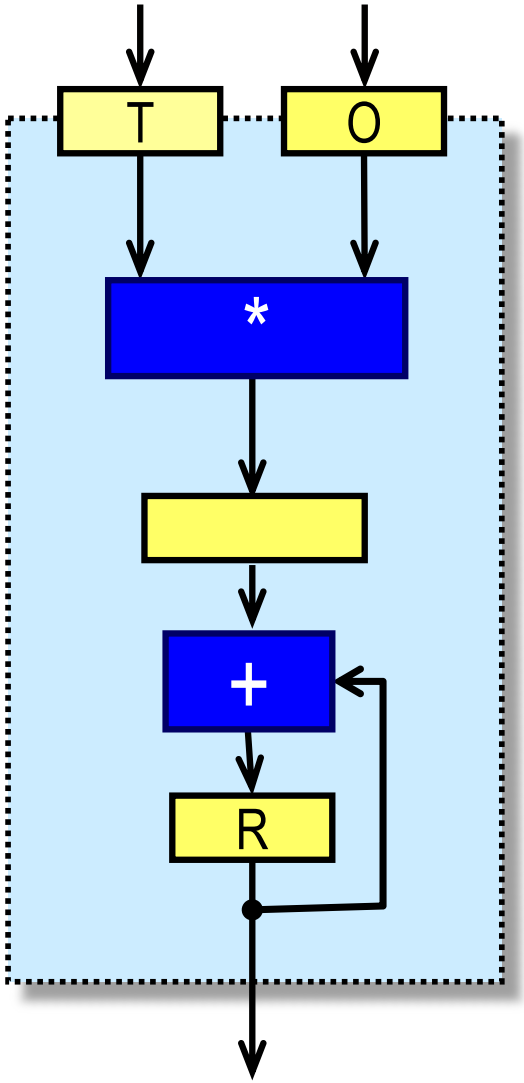


- Only trigger move starts a new operation
- Pipeline synchronized with control bits (C)
- Adds scheduling freedom
  - Result can stay in the output for several cycles
  - However, new result will overwrite the old one
  - Unnecessary results automatically flushed

Functional Units



Multiple outputs



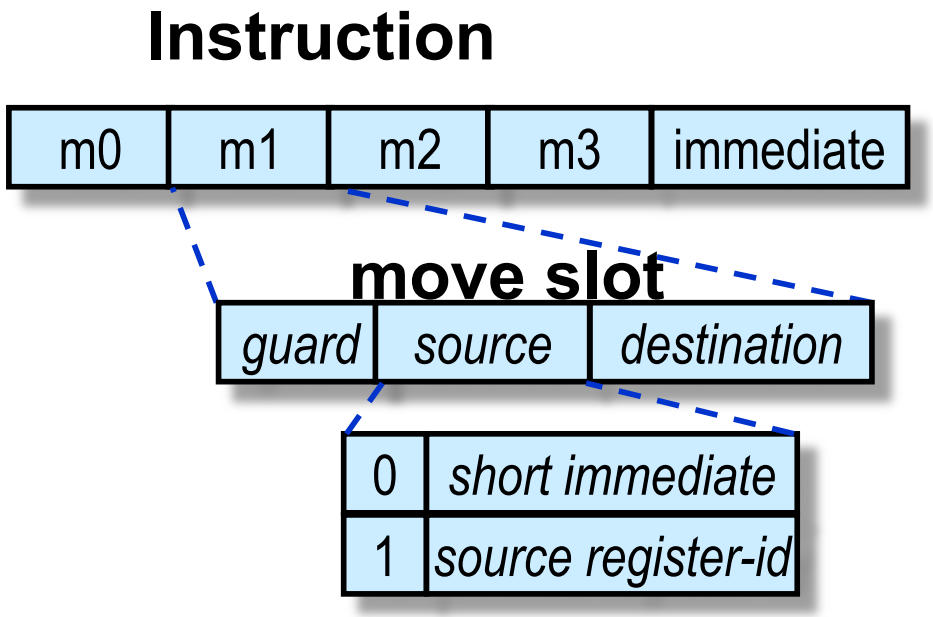
FU can have state

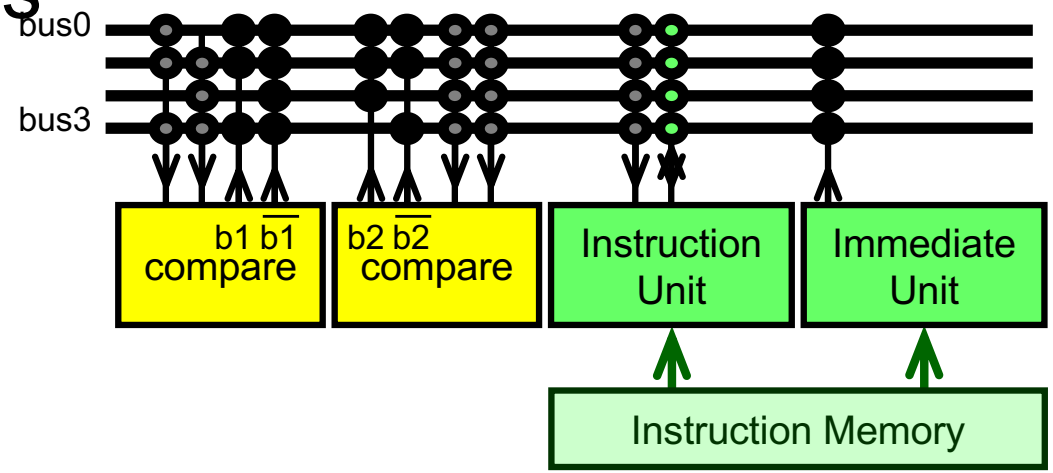
Instructions

- Only one instruction: move
  - e.g., r1->mul.o
    - **move contents of register 1 in RF into operand register of FU "mul"**
  - mul.r -> add.t
    - **move result of FU "mul" into trigger register of FU "add"**
- Branching simply by moving the branch address to PC
  - 0x2267 -> jump.t
    - trigger register of jump FU

Instruction Format

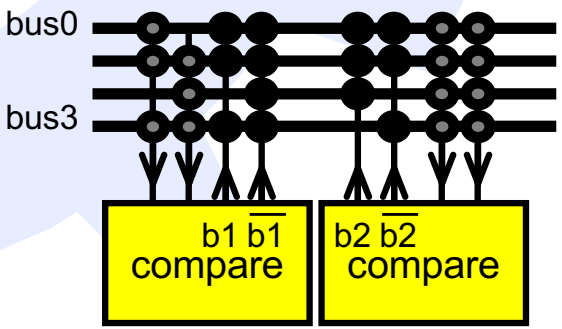
- Instruction contains a slot for each bus
- Separate fields for long immediates
- Short immediates can be packed into source field

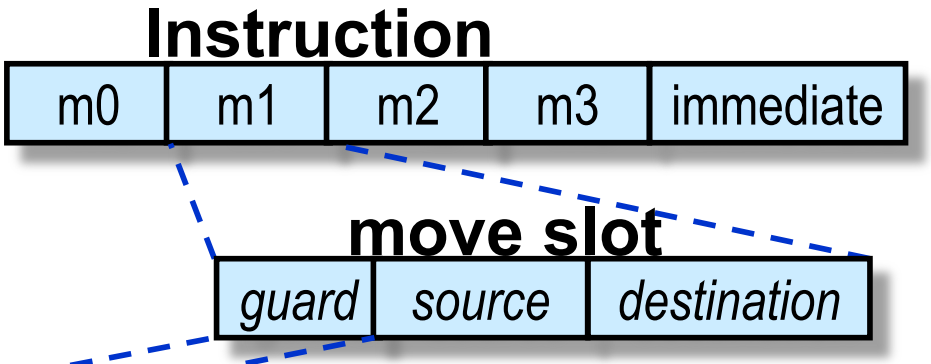




Instruction Format

- Guard allows conditional (guarded) execution
- Guard signals obtained from compare units





guard	*Boolean*	*Assembler notation*
0	b1	b1: src->dst
1	$\overline{b1}$	!b1: src->dst
2	b2	b2: src->dst
3	$\overline{b2}$	!b2: src->dst
4	b1^b2	b1.b2: src->dst
5	$\overline{b1}$^b2	!b1.b2: src->dst
6	b1^$\overline{b2}$	b1.!b2: src->dst
7	TRUE	src->dst

TTA Characteristics: SW

- Traditional operation-triggered instruction:

```
mul r1,r2,r3;
```

- Transport-triggered instruction:

```
r1→mul.o;
r2→mul.t;
mul.r→r3;
```

or

```
r1→mul.o; r2→mul.t;
mul.r→r3; ;
```

or

```
r1→mul.o, . . .;
. . . , . . . ;
. . . , r2→mul.t;
. . . , . . . ;
mul.r→r3;
```

- Reminds dataflow and time-stationary coding

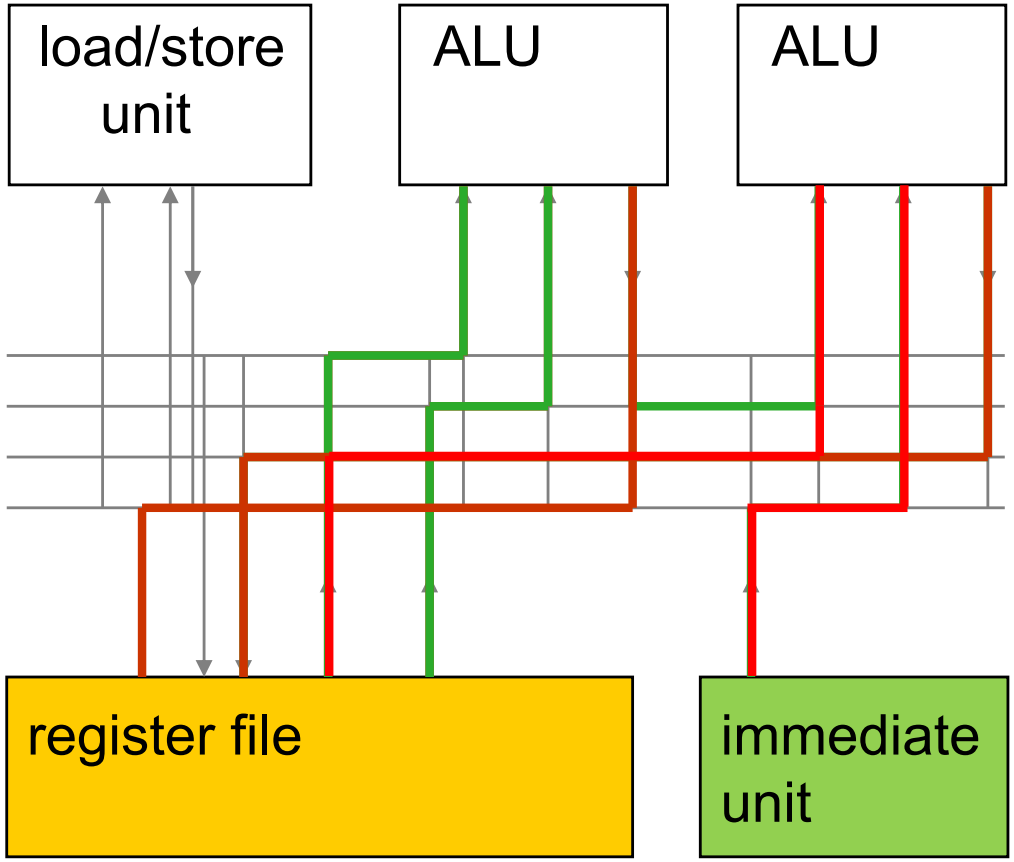
Scheduling example

VLIW

```
add r3,r1,r2

sub r4,r1,99
```

TTA

```
r1 -> add.o1,     r2 -> add.o2

add.r -> sub.o1, 99 -> sub.o2

sub.r -> r4
```



TTA Specific Optimizations

- software bypassing
  - Bypassing can eliminate the need of RF access

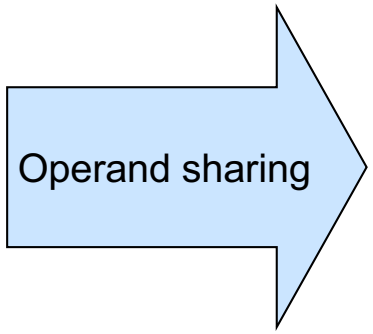
Example:	*r1 → add.o, r2 → add.t;* *add.r → **r3**;* ***r3** → sub.o, r4 → sub.t* *sub.r → r5;*

Translates to:	*r1 → add.o, r2 → add.t;* ***add.r** → **sub.o**, r4 → sub.t;* *sub.r → r5;*

TTA Specific Optimizations

- Operand sharing:
  - IF next operation uses the same operand, do not move it again
  - Value is already stored in the function unit operand register

R1 → add.o ; 32 → add.t;
add.res → R3
R1 → add.o ; 64 → add.t;
add.res → R4



R1 → add.o ; 32 → add.t;
add.res → R3
64 → add.t;
add.res → R4

TTA Co-Design Environment – TCE Framework

TCE in Action

- From high-level language to ASIP

1 Compile initial source code for initial architecture

2 Simulate program image on initial architecture

3 Analyze program execution

4 Change program and/or processor design

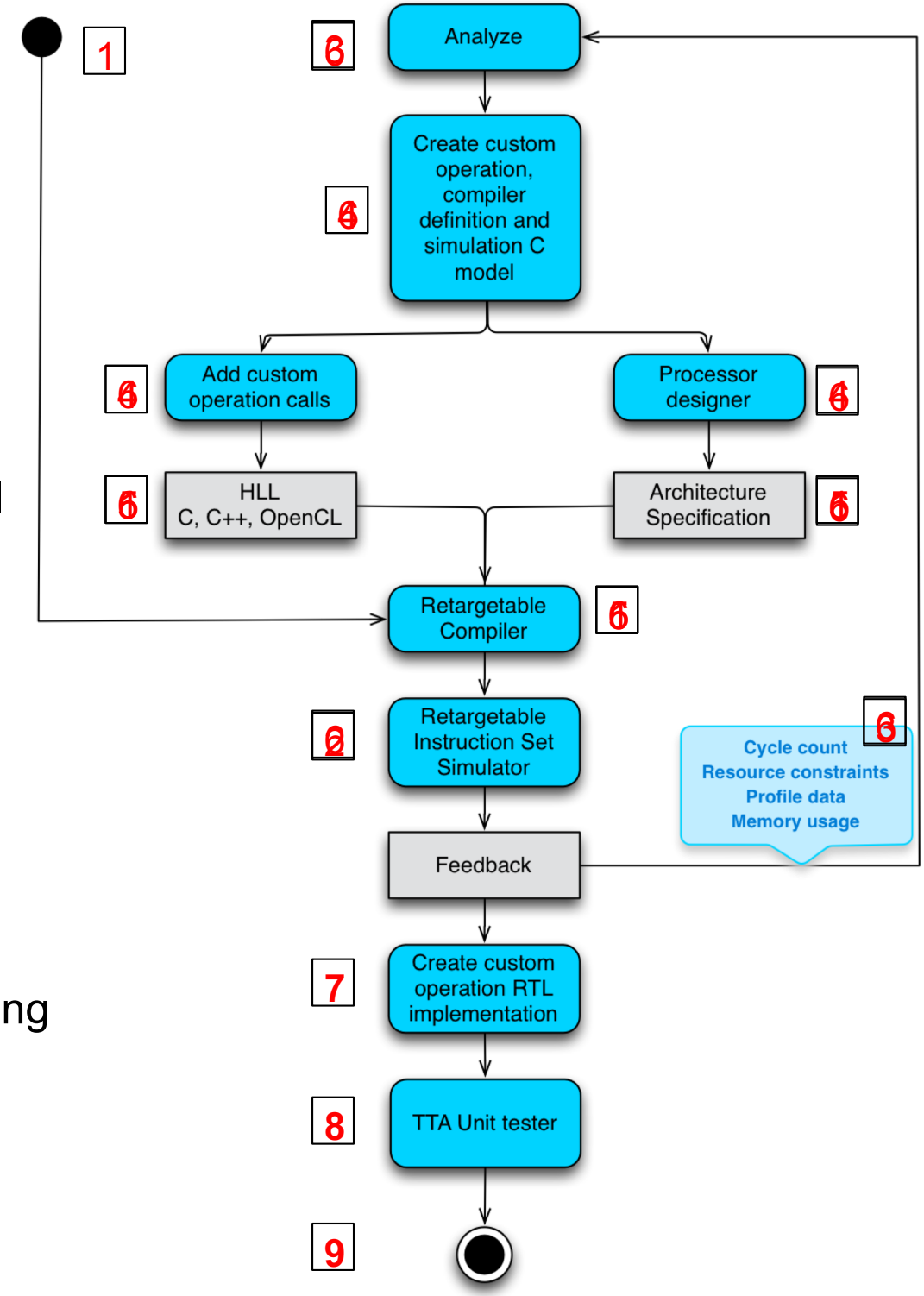
5 Recompile the source code for modified architecture

6 Iterate

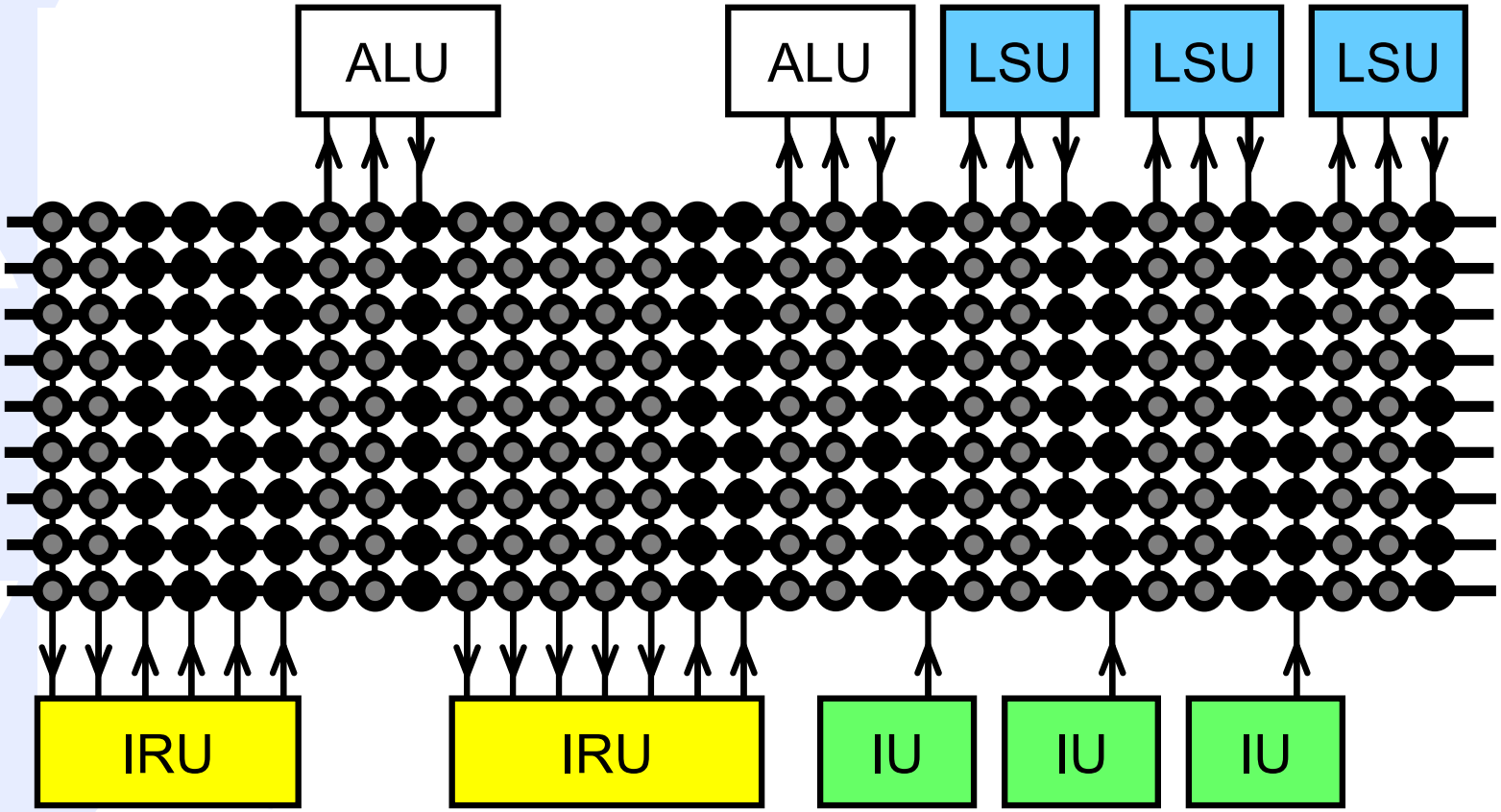
7 Finalize the design by implementing custom operations

8 Test custom operations

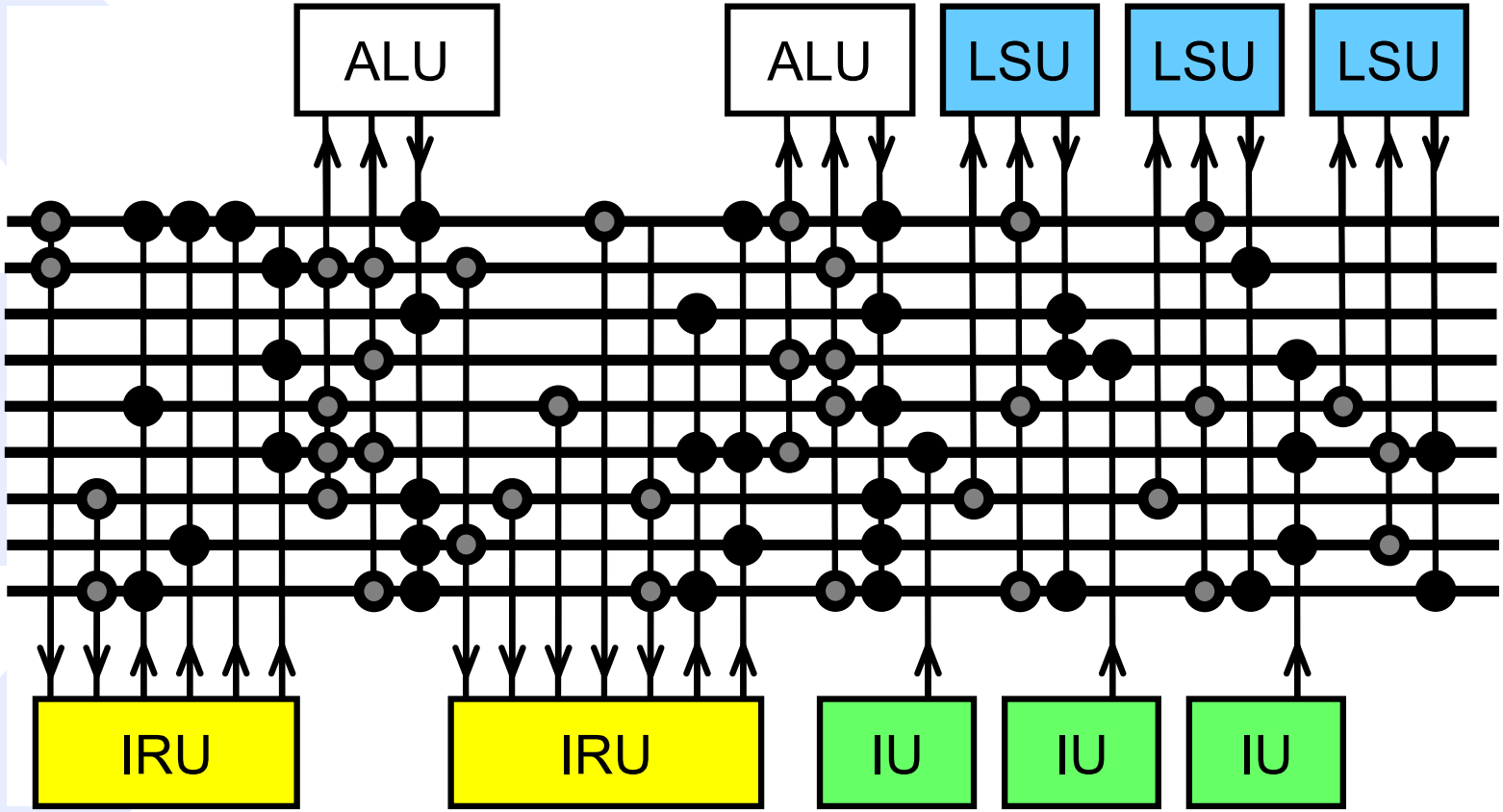
9 Export design for FPGA or SoC



Connectivity Optimization

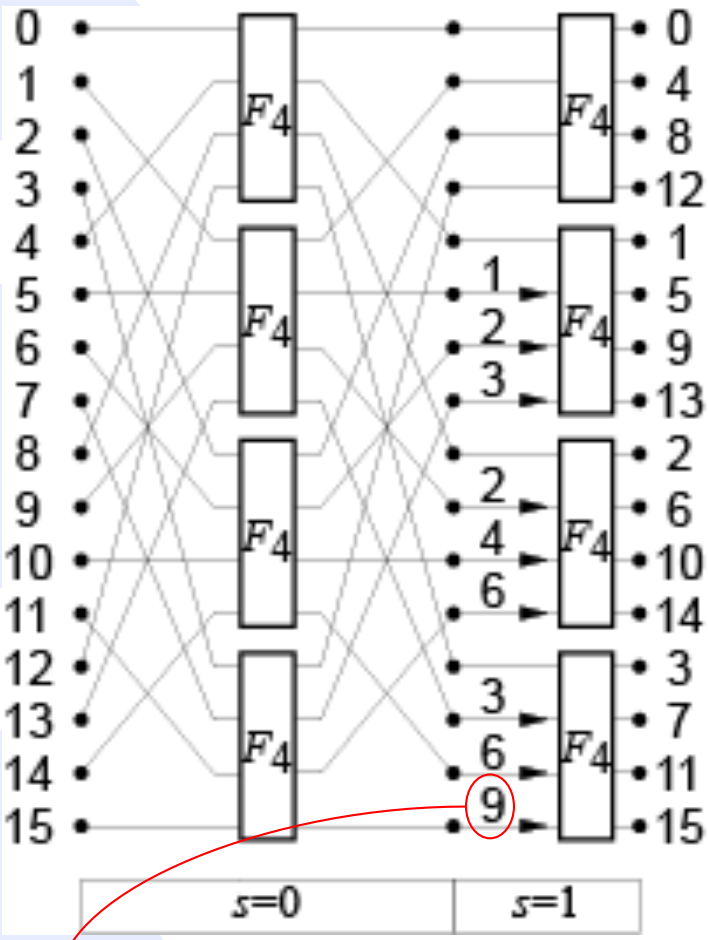


Connectivity Optimization

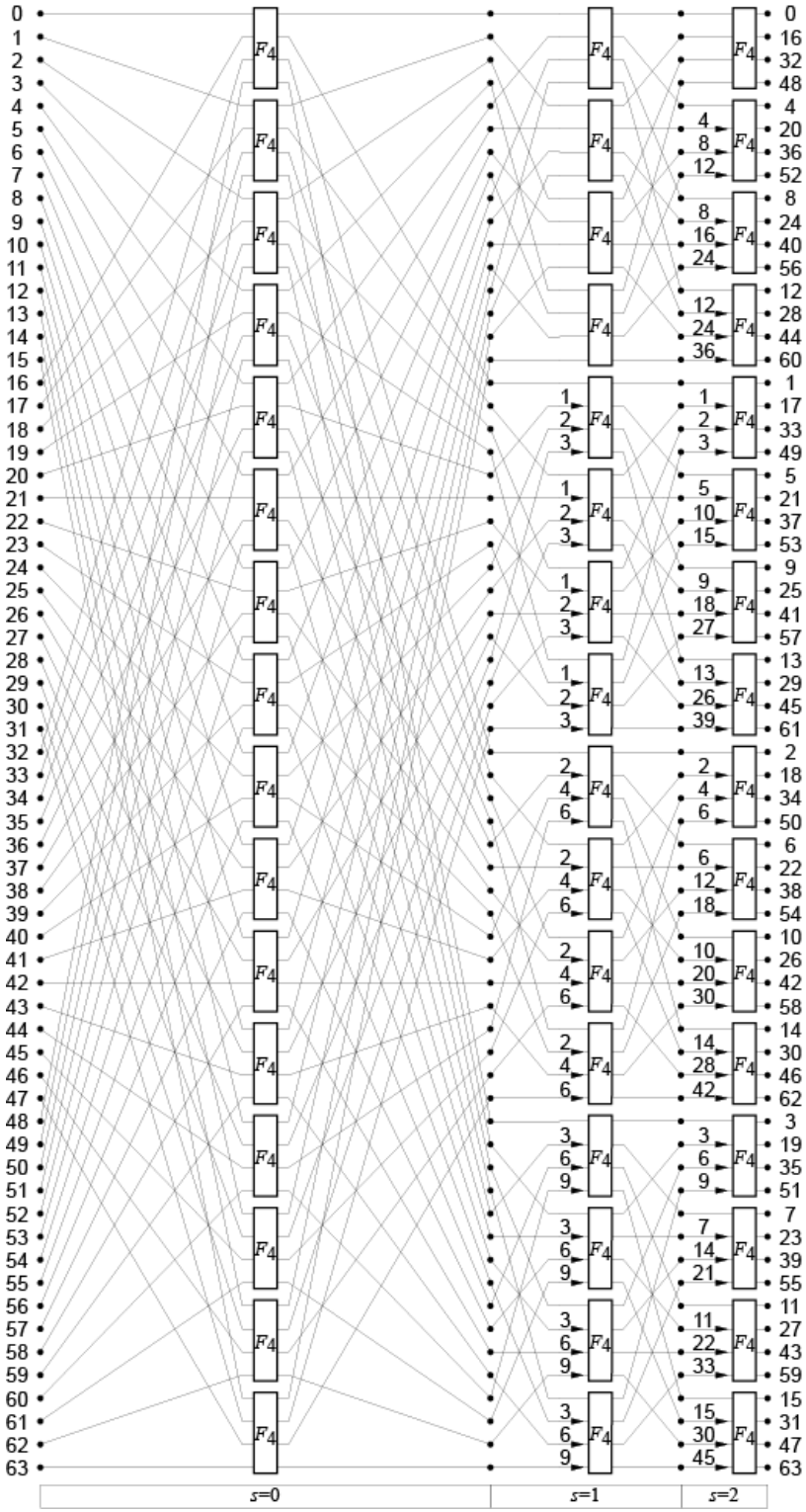


TTA for Fast Fourier Transform

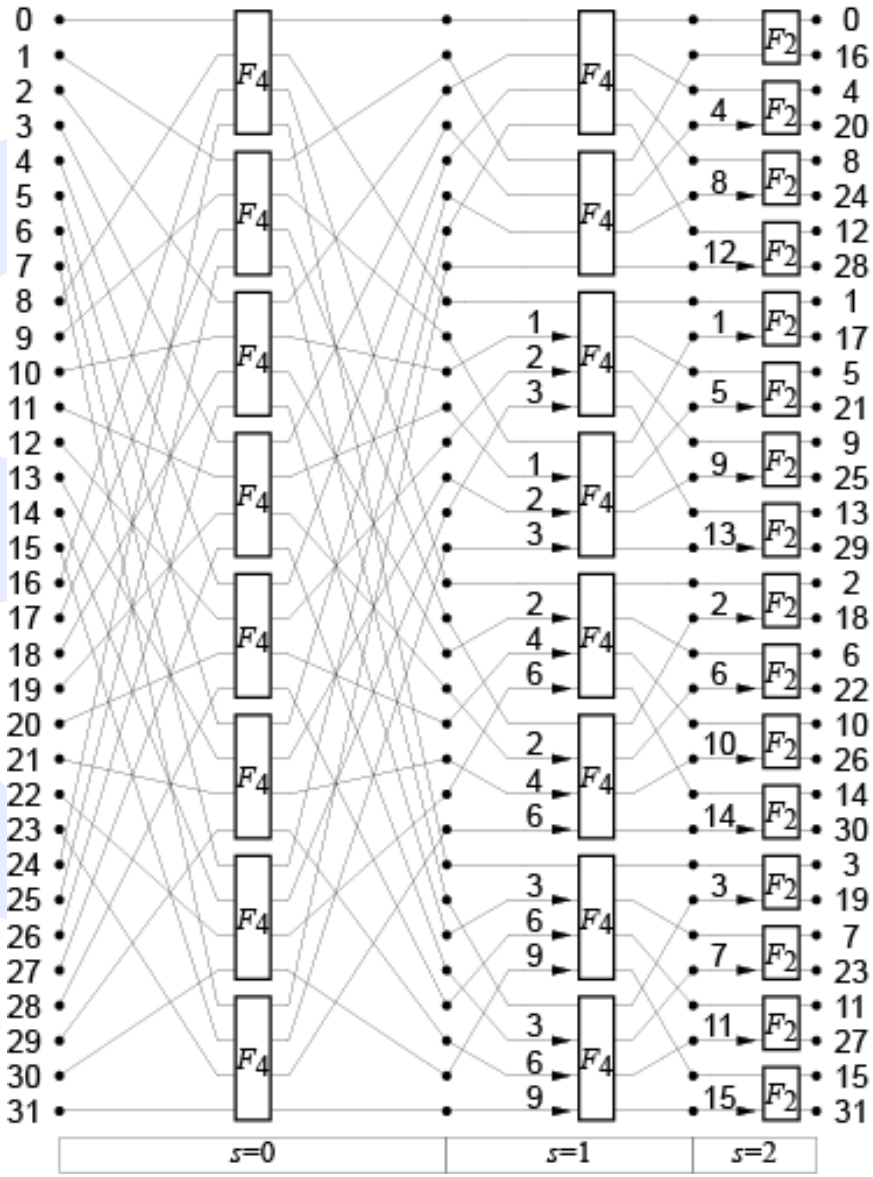
Radix-4 DIT FFT



$k : W_N^k$ $N =$ sequence length



Mixed Radix (4/2) DIT FFT



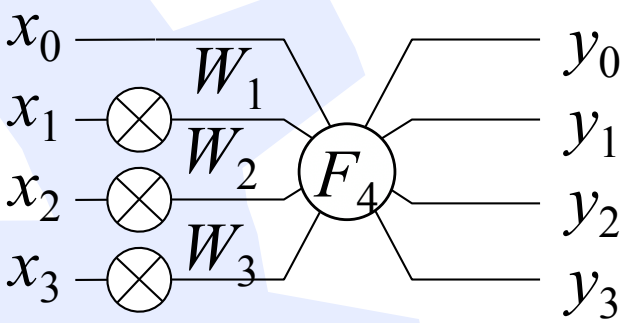
Complex Datatype

- In a 32-bit machine, use 16 bits for real and imaginary parts:

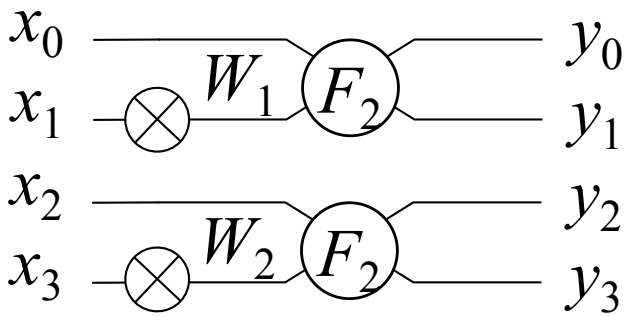
```
typedef short Scalar;
typedef union {
        int word;
        struct {
                short int imag;
                short int real;
        } cplx;
} Complex;
```

Radix-4 and Radix-2 DIT Butterfly

Radix-4 Butterfly



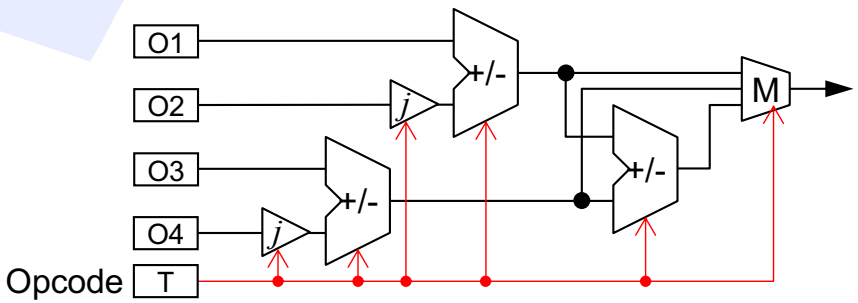
Two Radix-2 Butterflies



$$\begin{cases} y_0 = x_0 + W_1x_1 + W_2x_2 + W_3x_3 \\ y_1 = x_0 - jW_1x_1 - W_2x_2 + jW_3x_3 \\ y_2 = x_0 - W_1x_1 + W_2x_2 - W_3x_3 \\ y_3 = x_0 + jW_1x_1 - W_2x_2 - jW_3x_3 \end{cases} \Leftrightarrow \begin{cases} y_0 = (x_0 + W_1x_1) + (W_2x_2 + W_3x_3) \\ y_1 = (x_0 - jW_1x_1) - (W_2x_2 - jW_3x_3) \\ y_2 = (x_0 - W_1x_1) + (W_2x_2 - W_3x_3) \\ y_3 = (x_0 + jW_1x_1) - (W_2x_2 + jW_3x_3) \end{cases}$$

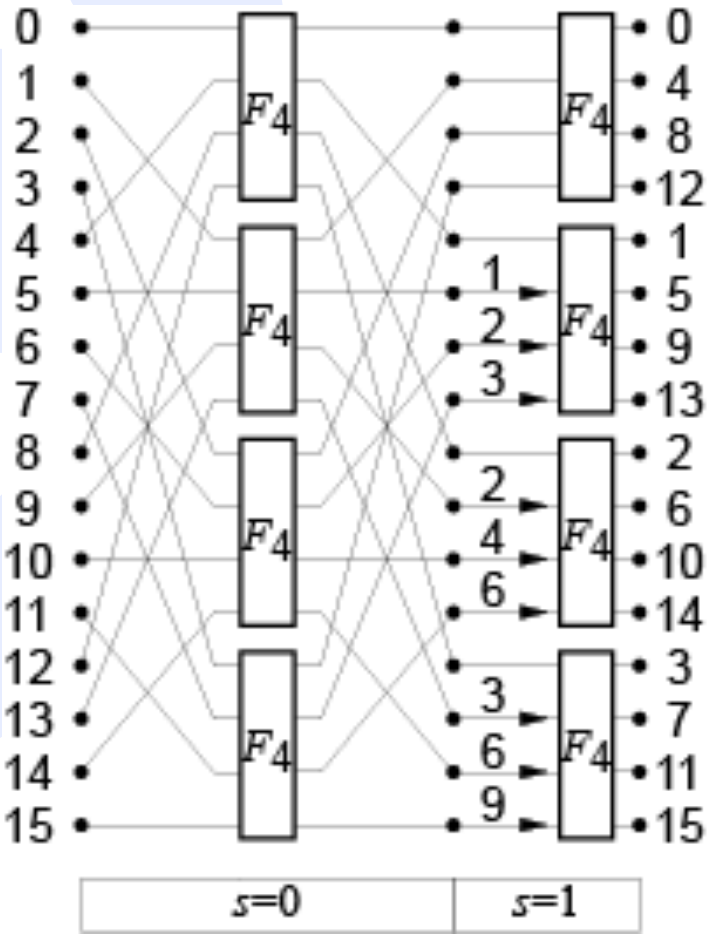
$$\begin{cases} y_0 = x_0 + W_1x_1 \\ y_1 = x_0 - W_1x_1 \\ y_2 = x_2 + W_2x_3 \\ y_3 = x_2 - W_2x_3 \end{cases}$$

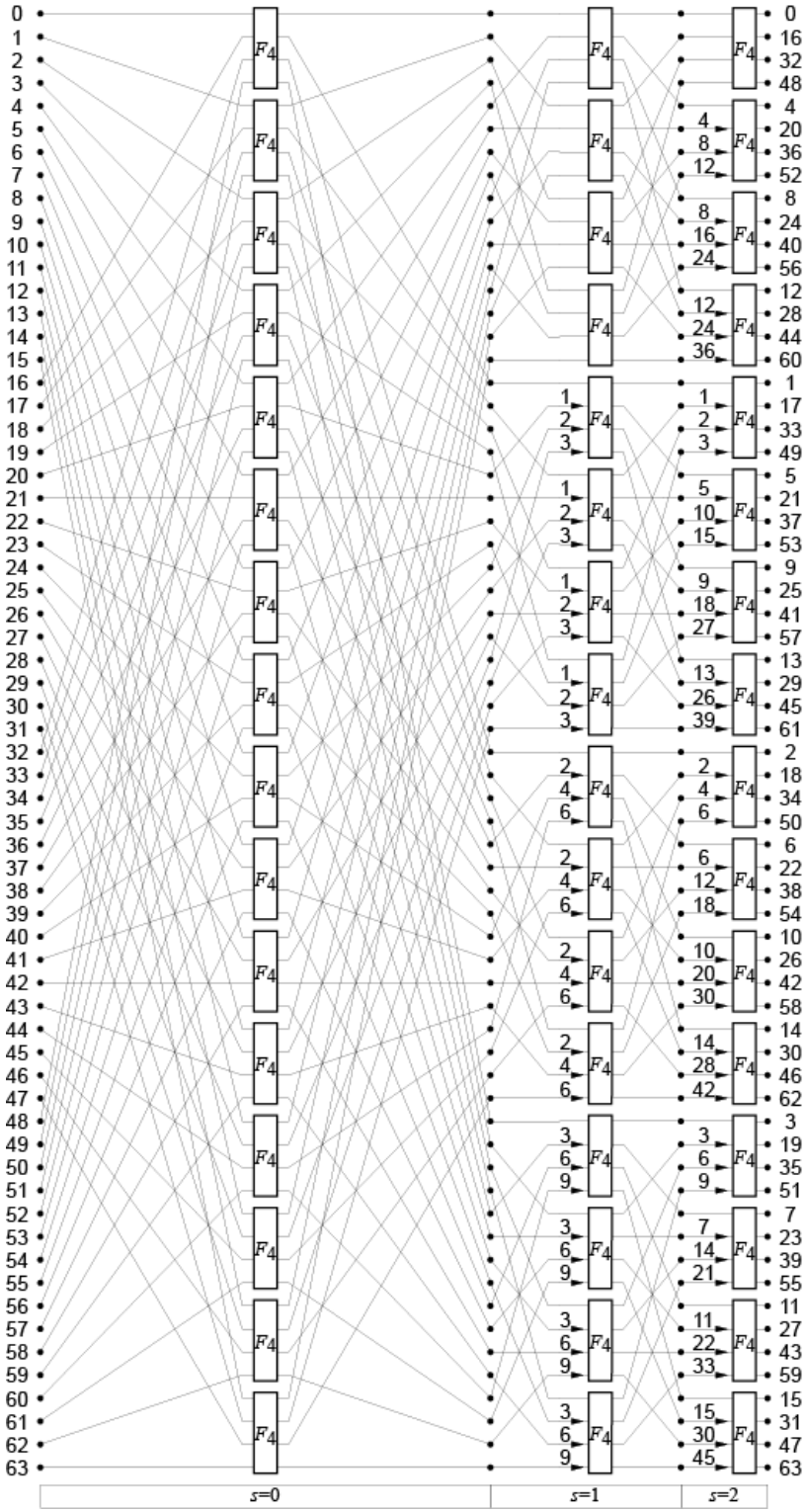
4-Input Complex Adder with Opcode



Operand Access

- Operands accessed in permuted order
  - Stride permutation





Operand Access (Radix-4)

N=64, *s*=0			
Linear idx		Operand idx	
0	000000	0	000000
1	000001	16	010000
2	000010	32	100000
3	000011	48	110000
4	000100	1	000001
5	000101	17	010001
6	000110	33	100001
7	000111	49	110001
8	001000	2	000010
9	001001	18	010010
10	001010	34	100010
11	001011	50	110010
12	001100	3	000011
13	001101	19	010011
14	001110	35	100011
15	001111	51	110011

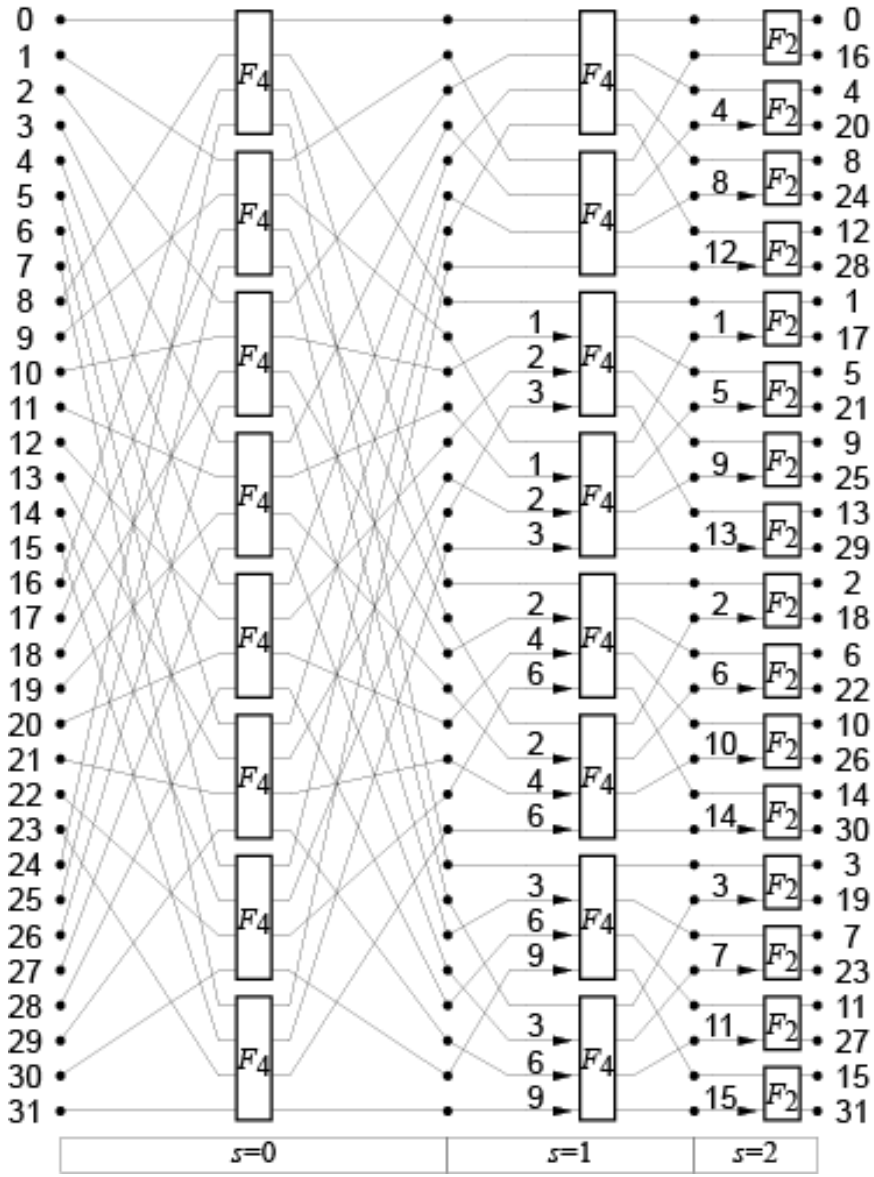
N=64, *s*=1			
Linear idx		Operand idx	
0	000000	0	000000
1	000001	4	000100
2	000010	8	001000
3	000011	12	001100
4	000100	1	000001
5	000101	5	000101
6	000110	9	001001
7	000111	13	001101
8	001000	2	000010
9	001001	6	000110
10	001010	10	001010
11	001011	14	001110
12	001100	3	000011
13	001101	7	000111
14	001110	11	001011
15	001111	15	001111

Operand Access (Radix-4)

- Rotate (n-2s)-bit field in LSB of the linear index two bits to the right; $n = 2k$

linear	a_{n-1}	a_{n-2}	a_{n-3}	a_{n-4}	a_{n-5}	. . .	a_6	a_5	a_4	a_3	a_2	a_1	a_0
$s = 0$	a_1	a_0	a_{n-1}	a_{n-2}	a_{n-3}	. . .	a_8	a_7	a_6	a_5	a_4	a_3	a_2
$s = 1$	a_{n-1}	a_{n-2}	a_1	a_0	a_{n-3}	. . .	a_8	a_7	a_6	a_5	a_4	a_3	a_2
$s = n-3$	a_{n-1}	a_{n-2}	a_{n-3}	a_{n-4}	a_{n-5}	. . .	a_6	a_1	a_0	a_5	a_4	a_3	a_2
$s = n-2$	a_{n-1}	a_{n-2}	a_{n-3}	a_{n-4}	a_{n-5}	. . .	a_6	a_5	a_4	a_1	a_0	a_3	a_2
$s = n-1$	a_{n-1}	a_{n-2}	a_{n-3}	a_{n-4}	a_{n-5}	. . .	a_6	a_5	a_4	a_3	a_2	a_1	a_0

Operand Access (Mixed Radix)



Operand Access (Mixed Radix)

N=32, *s*=0			
Linear idx		Operand idx	
0	00000	0	00000
1	00001	8	01000
2	00010	16	10000
3	00011	24	11000
4	00101	1	00001
5	00110	9	01001
6	00110	17	10001
7	00111	25	11001
8	01000	2	00010
9	01001	10	01010
10	01010	18	10010
11	01011	26	11010
12	01100	3	00011
13	01101	11	01011
14	01110	19	10011
15	01111	27	11011

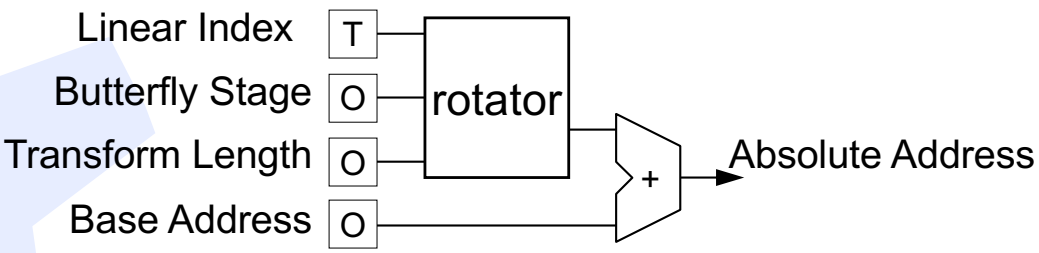
N=32, *s*=1			
Linear idx		Operand idx	
0	00000	0	00000
1	00001	2	00010
2	00010	4	00100
3	00011	6	00110
4	00100	1	00001
5	00101	3	00011
6	00110	5	00101
7	00111	7	00111
8	01000	8	01000
9	01001	10	01010
10	01010	12	01100
11	01011	14	01110
12	01100	9	01001
13	01101	11	01011
14	01110	13	01101
15	01111	15	01111

Operand Access (Mixed Radix)

- Rotate (n-2s)-bit field in LSB of the linear index two bits to the right; $n = 2k+1$

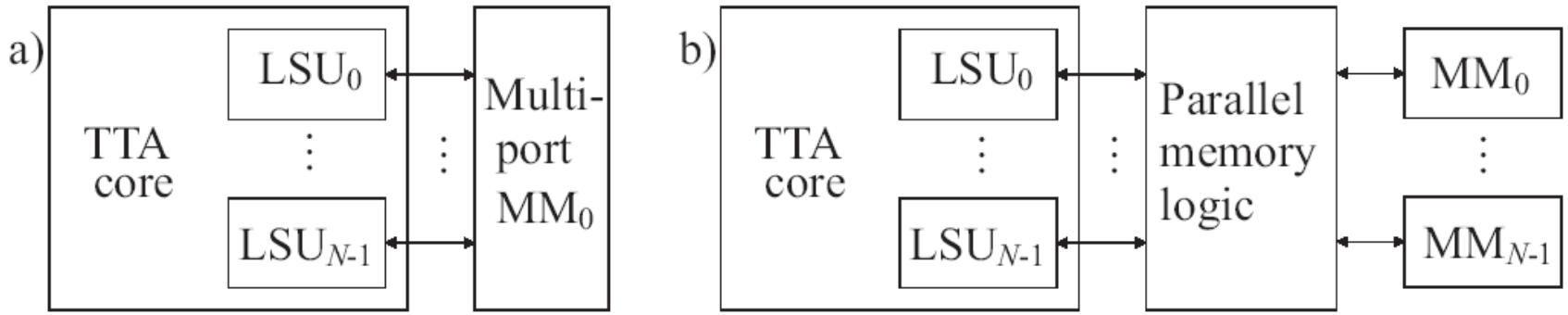
linear	a_{n-1}	a_{n-2}	a_{n-3}	a_{n-4}	a_{n-5}	. . .	a_6	a_5	a_4	a_3	a_2	a_1	a_0
$s = 0$	a_1	a_0	a_{n-1}	a_{n-2}	a_{n-3}	. . .	a_8	a_7	a_6	a_5	a_4	a_3	a_2
$s = 1$	a_{n-1}	a_{n-2}	a_1	a_0	a_{n-3}	. . .	a_8	a_7	a_6	a_5	a_4	a_3	a_2
$s = n-3$	a_{n-1}	a_{n-2}	a_{n-3}	a_{n-4}	a_{n-5}	. . .	a_6	a_5	a_1	a_0	a_4	a_3	a_2
$s = n-2$	a_{n-1}	a_{n-2}	a_{n-3}	a_{n-4}	a_{n-5}	. . .	a_6	a_5	a_4	a_3	a_1	a_0	a_2
$s = n-1$	a_{n-1}	a_{n-2}	a_{n-3}	a_{n-4}	a_{n-5}	. . .	a_6	a_5	a_4	a_3	a_2	a_1	a_0

Operand Address Generator



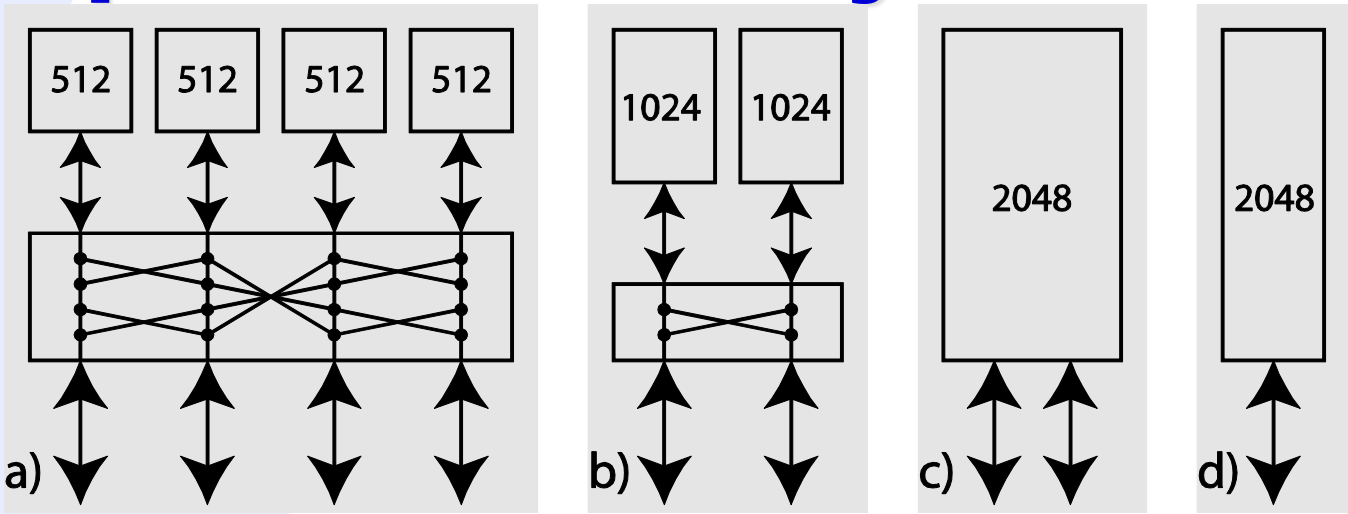
- Reduces significantly the address computation overhead
  - Address computation with basic arithmetic operations requires at least 6 operations

Multi-Port Memory vs. Parallel Memory



Different data memory configurations: a) multi-port memory and b) single-port memories with parallel memory logic. LSU_k Load-store unit. MM_k Memory module.

Comparison: Memory



		Area [kgate]	Power [mW]	Energy/ 1kFFT [μJ]	Cycles/ 1kFFT
c)	Memory	102.4	27.2		5208
	TTA Core	33.1	43.6		
	TOTAL	135	71	1.48	
b)	Memory	29.7	15.7		5208
	TTA Core	33.1	43.6		
	TOTAL	62	59	1.24	
a)	Memory	37.0	29.0		2648
	TTA Core	66.2	88.0		
	TOTAL	103	117	1.24	

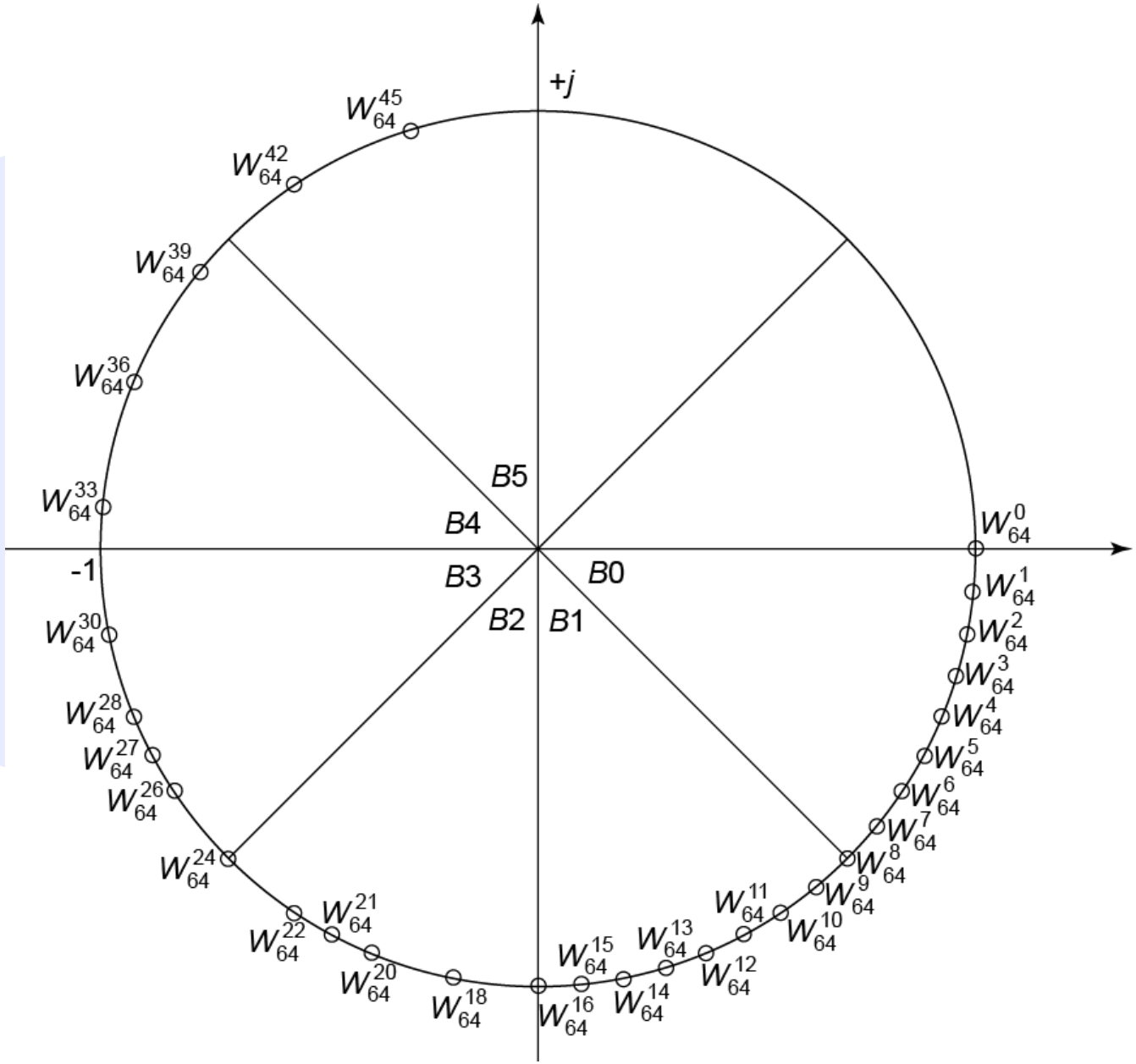
Twiddle Factors

- Complex-valued twiddle factors

$$W_N^k = e^{\frac{-j2\pi k}{N}}, 0 \le k \le \frac{N}{2} - 1$$

- In radix-4 FFT, $3/4N \log_4(N)$ non-trivial complex-valued multiplications are performed
- If coefficients are stored to memory, large area is needed indicating high power consumption
- Method for reducing twiddle factor memory storage in radix-2 FFT proposed in
  - Hasan and Arslan, “FFT coefficient memory reduction technique for OFDM applications, ” in *Proc. IEEE ICASSP*, 2002
  - Twiddle factors from unit circle divided into three sectors
  - Factors for *N*-point FFT can be formed by storing *N*/8+1 complex-valued factors

Twiddle Factors in Radix-4 FFT-64



Memory Reduction For Radix-4 and Mixed Radix FFT

- For up to *N*-point FFT only *N*/8+1 complex- valued factors need to be stored

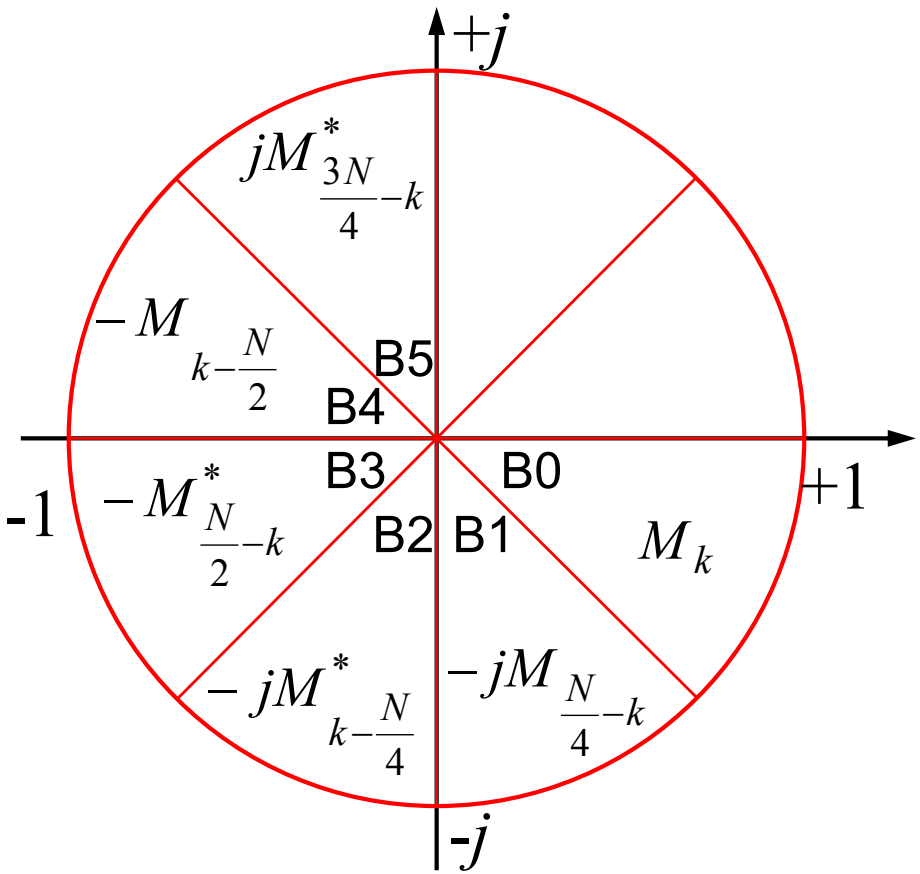
B0	B1	B2	B3	B4	B5
W_{64}^{0} (1.00, 0.00)	W_{64}^{16} (0.00,-1.00)				
W_{64}^{1} (1.00,-0.10)	W_{64}^{15} (0.10,-1.00)			W_{64}^{33} (-1.00, 0.10)	
W_{64}^{2} (0.98,-0.20)	W_{64}^{14} (0.20,-0.98)	W_{64}^{18} (-0.20,-0.98)	W_{64}^{30} (-0.98,-0.20)		
W_{64}^{3} (0.96,-0.29)	W_{64}^{13} (0.29,-0.96)				W_{64}^{45} (-0.29, 0.96)
W_{64}^{4} (0.92,-0.38)	W_{64}^{12} (0.38,-0.92)	W_{64}^{20} (-0.38,-0.92)	W_{64}^{28} (-0.92,-0.38)	W_{64}^{36} (-0.92, 0.38)	
W_{64}^{5} (0.88,-0.47)	W_{64}^{11} (0.47,-0.88)	W_{64}^{21} (-0.47,-0.88)	W_{64}^{27} (-0.88,-0.47)		
W_{64}^{6} (0.83,-0.56)	W_{64}^{10} (0.56,-0.83)	W_{64}^{22} (-0.56,-0.83)	W_{64}^{26} (-0.83,-0.56)		W_{64}^{42} (-0.56,0.83)
W_{64}^{7} (0.77,-0.63)	W_{64}^{9} (0.63,-0.77)			W_{64}^{39} (-0.77, 0.63)	
W_{64}^{8} (0.71,-0.71)		W_{64}^{24} (-0.71,-0.71)			

radix-4 FFT-64 Mixed-radix FFT-32 radix-4 FFT-16

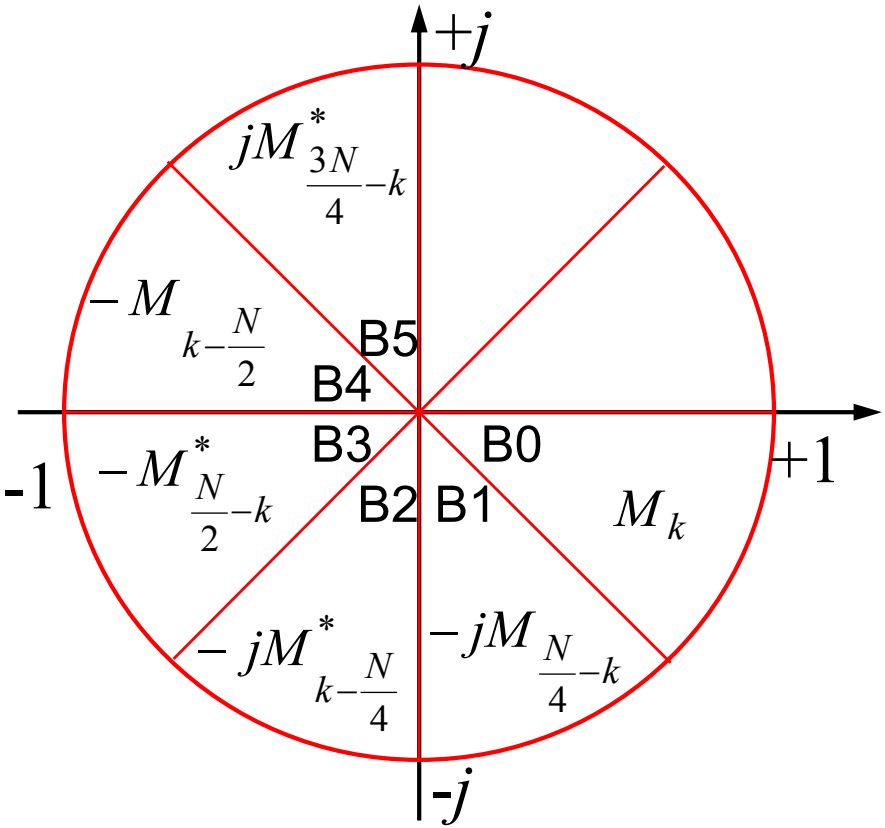
Memory Reduction

- Store $N/8+1$ twiddle factors from sector $B0$ to table $\boldsymbol{M}$
  - $\boldsymbol{M} = [M_0, M_1, \ldots, M_{N/8}]^{\mathrm{T}} \mid M_k = W^k{}_N$
  - then

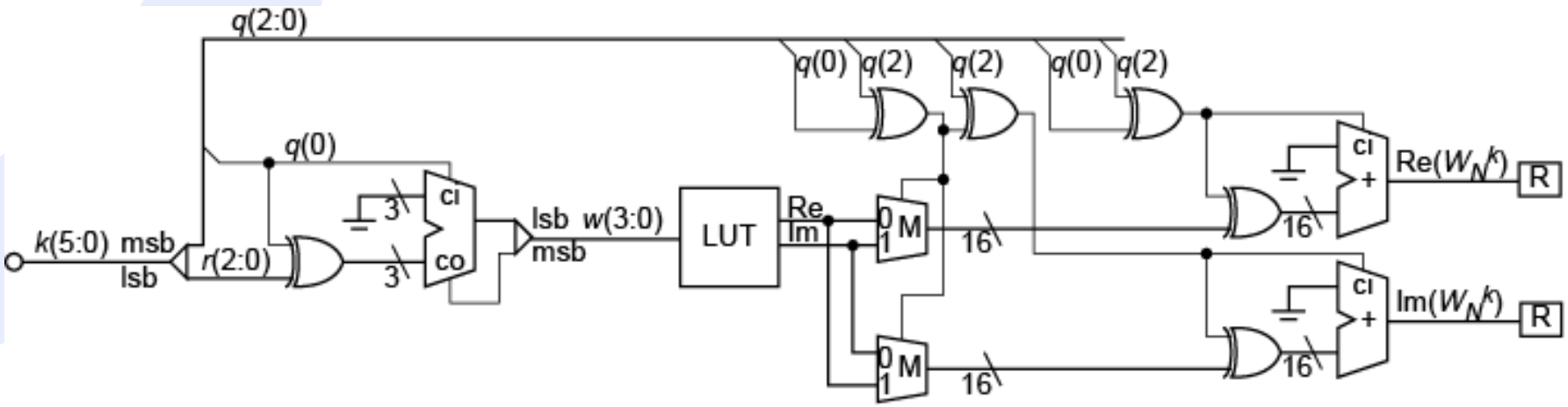
$$W_N^k = \begin{cases} M_k & , k \le \frac{N}{8} \\ -jM_{\frac{N}{4}-k} & , \frac{N}{8} < k \le \frac{N}{4} \\ -jM^*_{k-\frac{N}{4}} & , \frac{N}{4} < k \le \frac{3N}{8} \\ -M^*_{\frac{N}{2}-k} & , \frac{3N}{8} < k \le \frac{N}{2} \\ -M_{k-\frac{N}{2}} & , \frac{N}{2} < k \le \frac{5N}{8} \\ jM^*_{\frac{3N}{4}-k} & , \frac{5N}{8} < k \end{cases}$$



Implementation

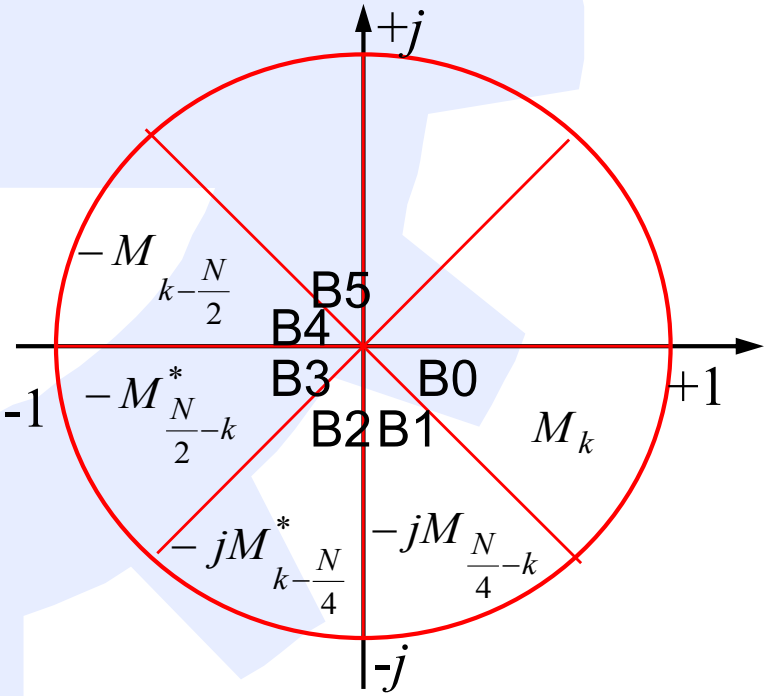


- Twiddle factor generator for FFT-64



Example

$$W_{64}^{27}$$



27: 011011

MSB	**LSB**
011	**1011**
011	**0101**

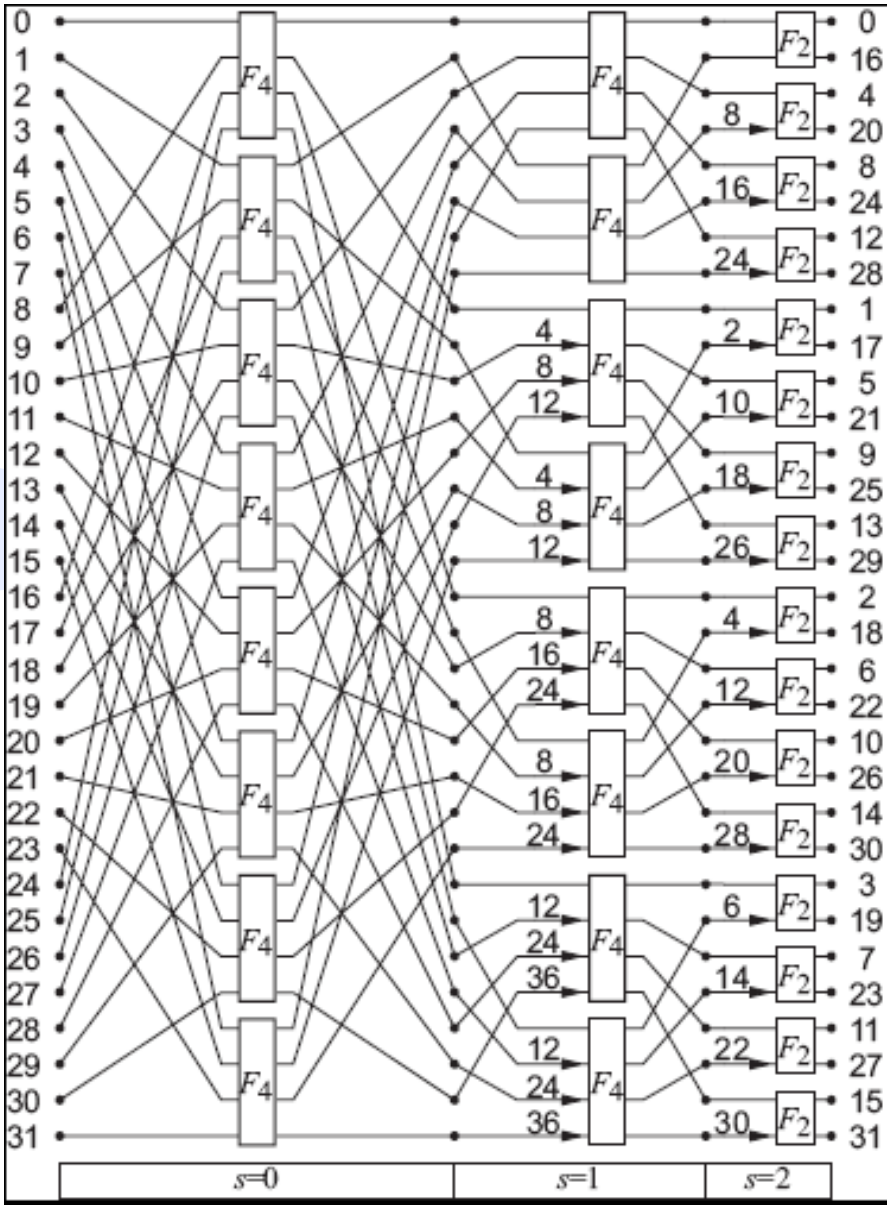
0.88, -0.47

011 => -M*

-0.88, -0.47

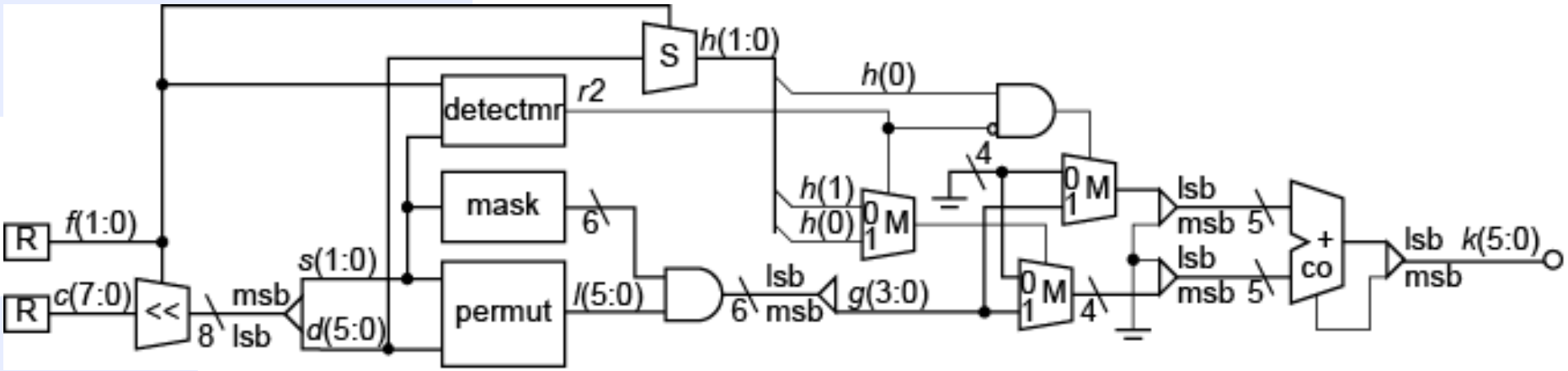
M
W_{64}^{0} (1.00, 0.00)
W_{64}^{1} (1.00,-0.10)
W_{64}^{2} (0.98,-0.20)
W_{64}^{3} (0.96,-0.29)
W_{64}^{4} (0.92,-0.38)
W_{64}^{5} (0.88,-0.47)
W_{64}^{6} (0.83,-0.56)
W_{64}^{7} (0.77,-0.63)
W_{64}^{8} (0.71,-0.71)

Twiddle Factor Exponents



Twiddle factor exponents

- Exponent *k* can be obtained from butterfly index in bit-level by rotating the most significant bits of the index
- E.g., exponent generation for FFT-64



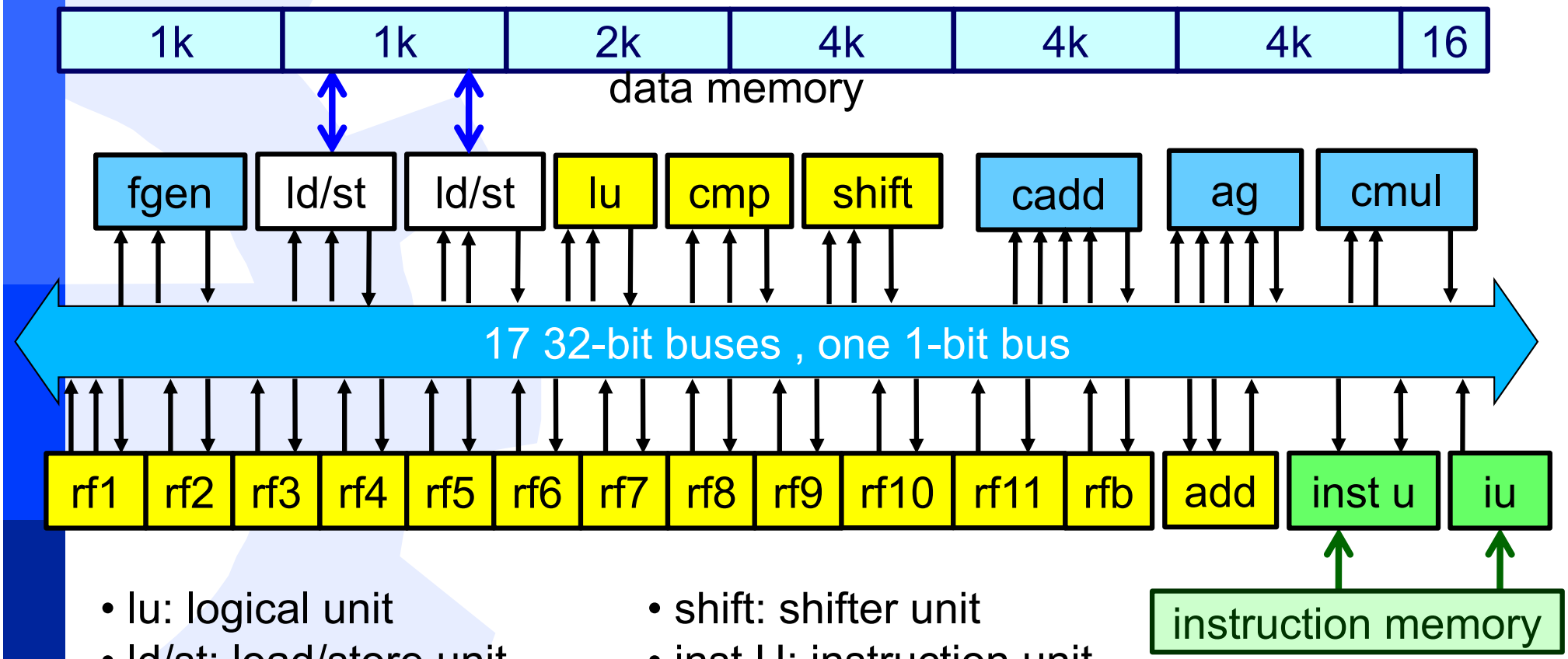
Analysis

- Implementation supporting up to 16k FFTs
- Synthesized to 130nm ASIC technology

	pipelined@250MHz		non-pipelined@140MHz		ROM table [5]@250MHz	
	Power [mW]	Area [kgates]	Power [mW]	Area [kgates]	Power [mW]	Area [kgates]
LUT	1.50	12	2.24	15.8	43.00	20.5
Pipeline	0.95	0.3				
Total	3.70	14.3	4.11	18.4	43.00	20.5

- Compared recursive sine function generator or CORDIC
  - Larger area ~15%
  - But 50% lower power consumption

TTA for FFT Block Diagram



- lu: logical unit
- ld/st: load/store unit
- cmp: comparator unit
- iu: immediate unit
- cadd: complex adder
- rf b: boolean register
- shift: shifter unit
- inst U: instruction unit
- rf: register file
- ag: data address generator
- fgen: coefficient generator
- cmul: complex multiplier

Mixed-radix FFT Code Structure

```
main() {
    initialization(); /* 9-42 instr.*/
    for (stage=0; stage<[log4 N]; stage++) {
        prologue(); /* 14 instr. */
        for( k=0; k<(N-14)/12; k++)
            kernel(); /* 12 instr. */
        if(r2flag == 1)
            oddEpilogue(); /* 18 instr. */
        else
            evenEpilogue(); /* 14 instr. */
```

Radix-4 1K FFT

```
main() {
    prologue; /* 14 instructions */
    for (index=0; index< (log4(N)*N)/16 -1; index++)
        kernel(); /* 16 instructions */
    epilogue; /* 16 instructions */
}
```

- Single loop implementation: 5208 cycles
  - Theoretical memory constrained lower bound: 5121 cycles
    - = $1k*\log_4(1k)$ memory accesses to dual-port memory
- Nested loop implementation: 5234 cycles

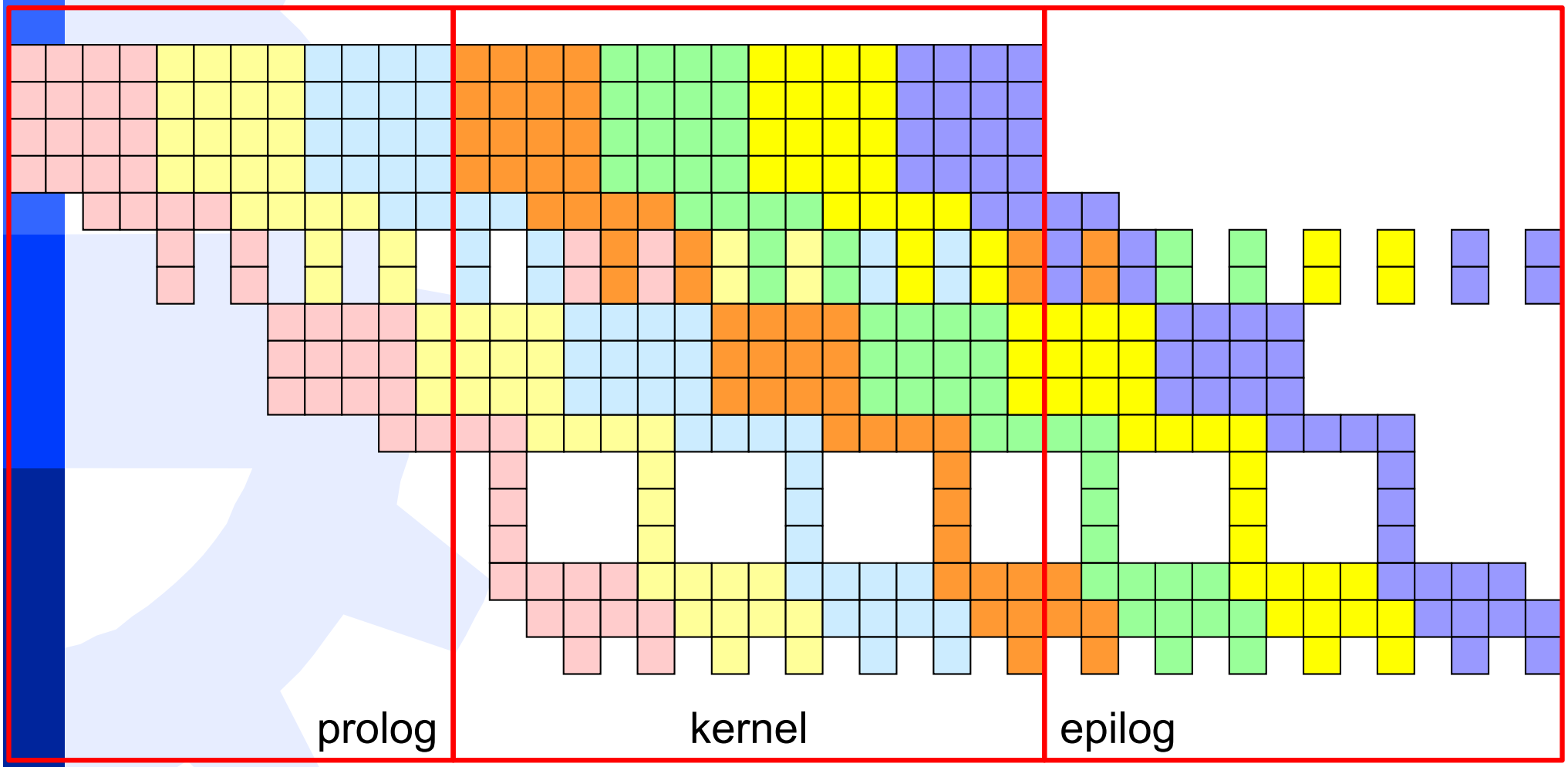
Radix-4 Butterfly Schedule

Index: ADD.r → ADD.o
+1→ADD.t
ADD.r → ag.t
ADD.r → fgen.t
ag.o → RF
ag.o → ld1.t
RF → ld2.t
ld1.r → CMUL.o
ld2.r → RF
fgen.r → CMUL.t
RF→st1.o
RF→st2.o

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
bus#1	i0	i1	i2	i3													
bus#2	+1	+1	+1	+1													
bus#3	i0	i1	i2	i3													
bus#4	i0	i1	i2	i3													
bus#5			ax0	ax1	ax2	ax3											
bus#6				ax1		ax3									ax1		ax3
bus #7				ax0		ax2									ax0		ax2
bus #8							x0	x1	x2	x3							
bus #9							x1		x3								
bus #10							W0	W1	W2	W3							
bus #11										P0	P1	P2	P3				
bus #12													P0				
bus #13													P1				
bus #14													P2				
bus #15													1|0	3|2	5|4	7|6	
bus #16														y0	y1	y2	y3
bus #17															y0		y2

RF → CADD.o
CMUL.r → RF
opcode → CADD.t
CADD.r → st1.t
RF→st2.t

Software Pipelining



TTA for 1k FFT

- Supports only 1k FFT
  - Twiddle factor generator and address generators limited to 1k FFT
- Single loop SW implementation
- 1k FFT 5208 cycles (20.9µs@250MHz)
- Synthesized on 130 nm technology
  - Max. Clock 250 MHz
  - Area 135 kgate (core 33 kgate)
  - Power 59 mW (core 43 mW) (1.5 V, 250MHz)
    - ~5mW (1.0V, 50 MHz)

Case 1: 1k FFTs / mJ

	tech [nm]	V_{CC} [V]	f_{clk} [MHz]	t_{FFT} [µs]	FFT / mJ
1k TTA	130	1.5	250	21	809
	130	1.0	50	105	1920
TI C6416	130	1.2	720	8.3	100
	130	1.2	600	10	167
	130	1.2	300	22	250
Intel P4	130	1.2	3000	24	0.8
SA-110	350	2.0	74	425	60
Imagine	150	1.5	232	160	16
Zhao	180		20	282	43

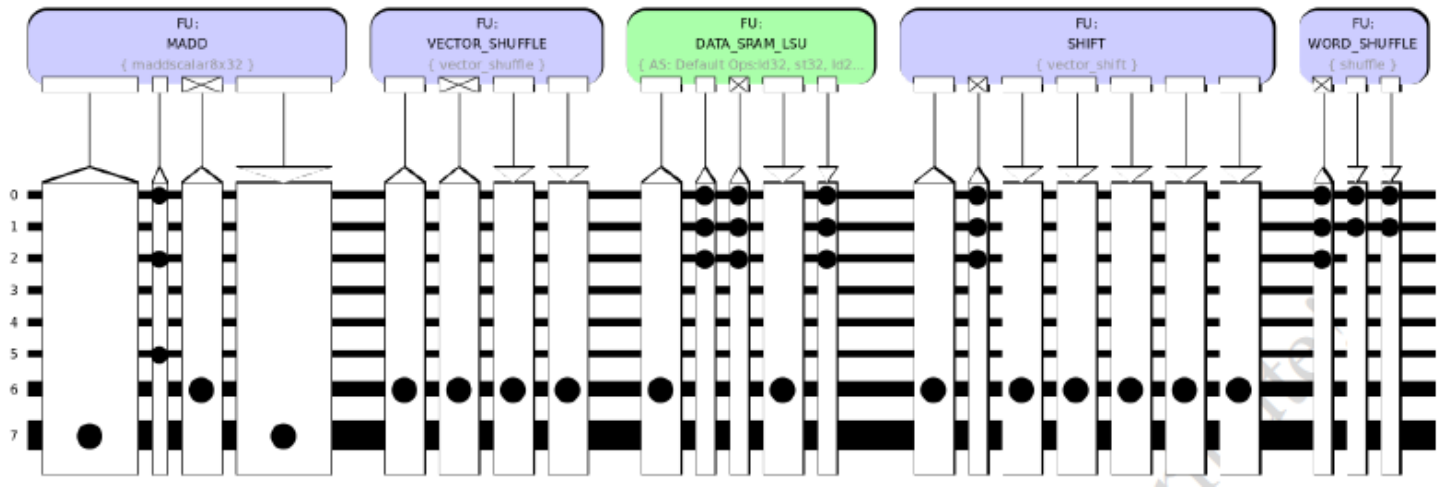
	tech [nm]	V_{CC} [V]	f_{clk} [MHz]	t_{FFT} [µs]	FFT / mJ
Spiffee	600	1.1	16	330	319
	600	2.5	128	21	67
	600	3.3	173	105	39
Stratix	130	1.3	275	4.7	241
	130	1.3	133	9.7	173
	130	1.3	100	12.9	149
MIT FFT	180	0.35	0.01	250000	6452
	180	0.9	6	430	1428
Lin	350	3.3	45	23	93
	350	2.3	18	57	133

Examples

AivoTTA: Custom DSP for CNN-Based Object Recognition

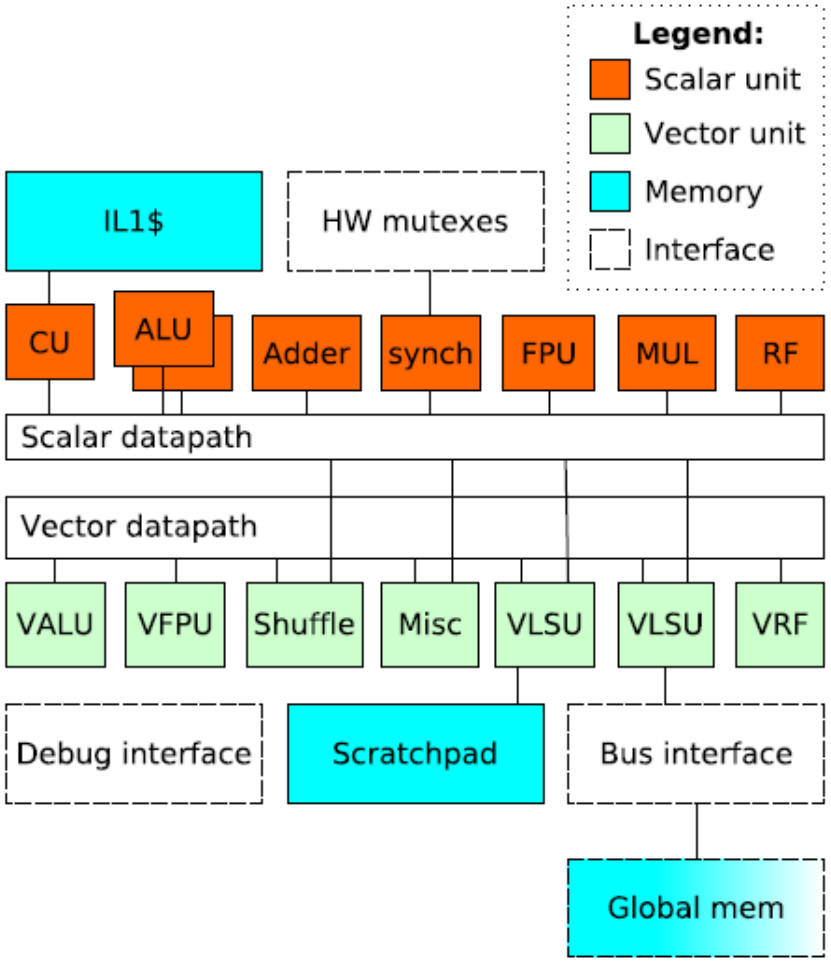
- traffic sign and face detection networks
- Special integer vector MADD unit for convolutions
- Four register files with minimal ports:
  Only 1 rd/wr port each!
  - 8x1024b, 8x256b, 8x32b, 16x32b

- Loop buffer to reduce imem accesses + improve ifetch power
- C and OpenCL C supported
- 28 nm FDSOI (after place'n route):
  - 11 mW 16 GOPS @ 400 MHz
  - 116 mW 57 GOPS @ 1.4 GHz
- **World class power-performance for a compiler-programmed accelerator**



LordCore: High Performance Low Power Wide-SIMD Floating-Point SDR Multicore

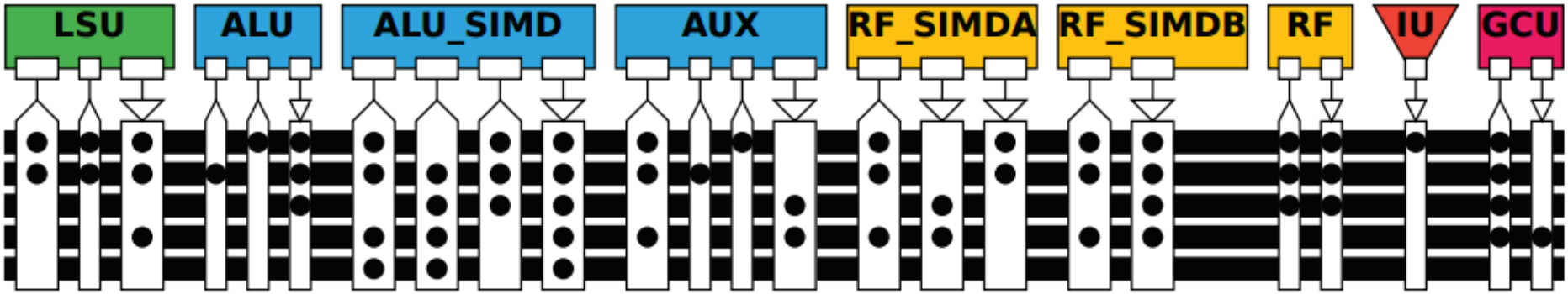
- 32-element FP16 VFUs
- Quite generic design, only a few special instructions
- OpenCL C programmed
- Quad core: 28 nm FDSOI power analysis: 280 mW at 900 MHz, 237 GFLOPS (846 GFLOPS / W)
- Approx. 18% energy savings from the TTA programming model
  - **Three orders of magnitude more power efficient than GPU designs**
  - **Closer to fixed-function HW power-efficiency**



Leibniz Universität Hannover:

Customized High Performance Low Power Processor for Binaural Speaker Localization

- Custom DSP targeted to hearing aid devices with support for advanced algorithms:
  - Very low power consumption, high computational performance, small form factor
- 32 x int32 SIMD (1024b) datapath
- Synthesized on 28 nm FDSOI
- 12 mW at 50 MHz, 1V
- **2-split SIMD RF, 1 write port each**
  - **Only 10.5% of total power thanks to software bypassing and DRE**
- Published in IEEE International Conference on Electronics, Circuits and Systems (ICECS) 2016



MIT-licensed tools: TTA-based Co-Design Environment (TCE)

http://tce.cs.tut.fi