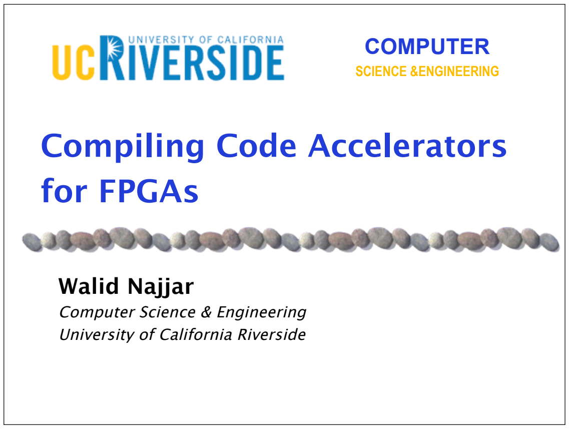
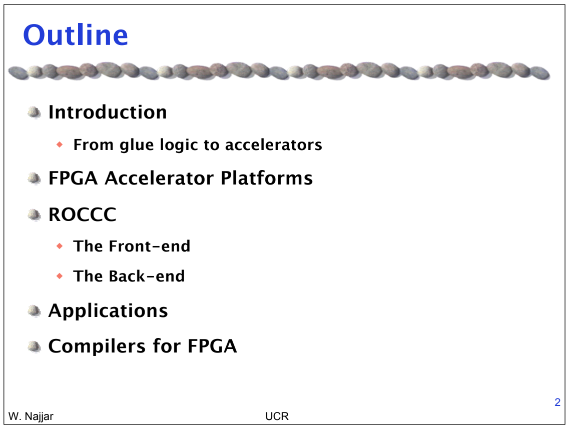


UC RIVERSIDE UNIVERSITY OF CALIFORNIA

COMPUTER
SCIENCE &ENGINEERING

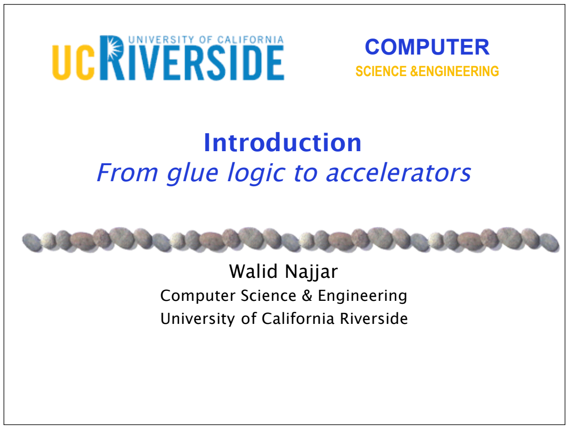
Compiling Code Accelerators for FPGAs

Walid Najjar
Computer Science & Engineering
University of California Riverside



Outline

- Introduction
  - From glue logic to accelerators
- FPGA Accelerator Platforms
- ROCCC
  - The Front-end
  - The Back-end
- Applications
- Compilers for FPGA

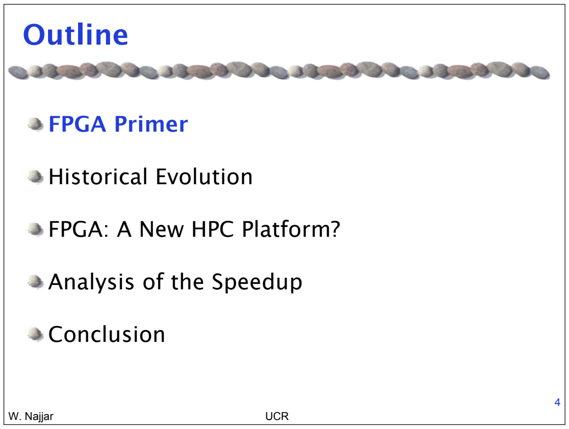


UC RIVERSIDE UNIVERSITY OF CALIFORNIA

COMPUTER
SCIENCE &ENGINEERING

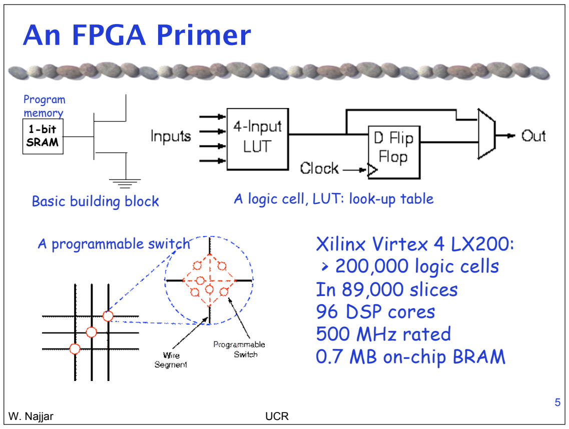
Introduction
From glue logic to accelerators

Walid Najjar
Computer Science & Engineering
University of California Riverside



Outline

- **FPGA Primer**
- Historical Evolution
- FPGA: A New HPC Platform?
- Analysis of the Speedup
- Conclusion



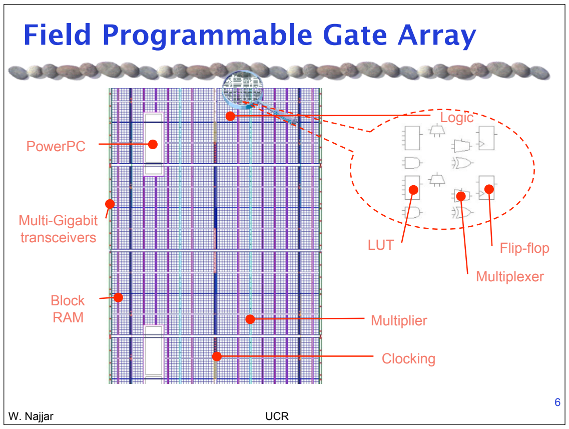
An FPGA Primer

Basic building block

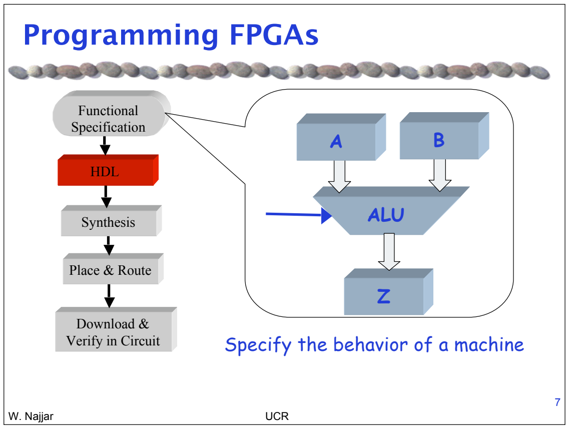
A logic cell, LUT: look-up table

A programmable switch

Xilinx Virtex 4 LX200:
- > 200,000 logic cells
- In 89,000 slices
- 96 DSP cores
- 500 MHz rated
- 0.7 MB on-chip BRAM

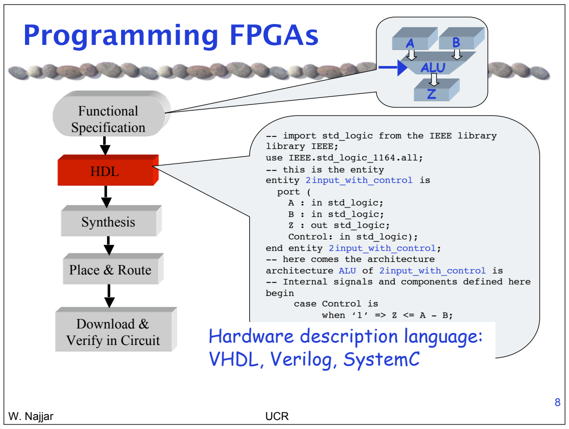


Field Programmable Gate Array



Programming FPGAs

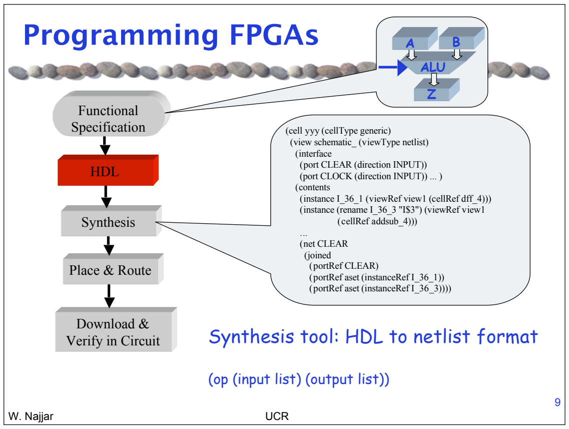
Specify the behavior of a machine



Programming FPGAs

```
-- import std_logic from the IEEE library
library IEEE;
use IEEE.std_logic_1164.all;
-- this is the entity
entity 2input_with_control is
  port (
    A : in std_logic;
    B : in std_logic;
    Z : out std_logic;
    Control: in std_logic);
end entity 2input_with_control;
-- here comes the architecture
architecture ALU of 2input_with_control is
-- Internal signals and components defined here
begin
     case Control is
          when ‘1’ => Z <= A - B;
```

Hardware description language:
VHDL, Verilog, SystemC

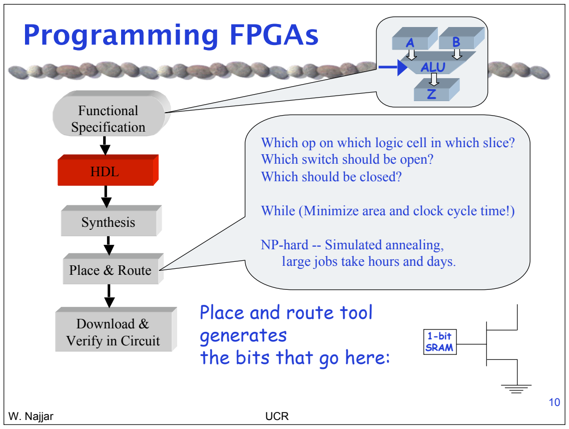


Programming FPGAs

```
(cell yyy (cellType generic)
 (view schematic_ (viewType netlist)
  (interface
   (port CLEAR (direction INPUT))
   (port CLOCK (direction INPUT)) ... )
  (contents
   (instance I_36_1 (viewRef view1 (cellRef dff_4)))
   (instance (rename I_36_3 "I$3") (viewRef view1
               (cellRef addsub_4)))
   ...
   (net CLEAR
    (joined
     (portRef CLEAR)
     (portRef aset (instanceRef I_36_1))
     (portRef aset (instanceRef I_36_3))))
```

Synthesis tool: HDL to netlist format

(op (input list) (output list))



Programming FPGAs

Functional Specification

HDL

Synthesis

Place & Route

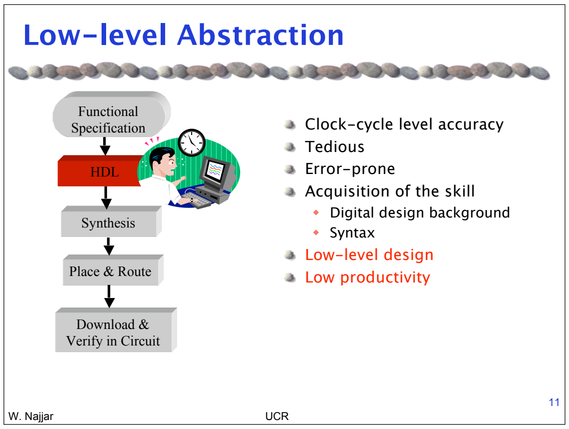
Download & Verify in Circuit

Which op on which logic cell in which slice?
Which switch should be open?
Which should be closed?

While (Minimize area and clock cycle time!)

NP-hard -- Simulated annealing,
large jobs take hours and days.

Place and route tool generates the bits that go here:



Low-level Abstraction

Functional Specification

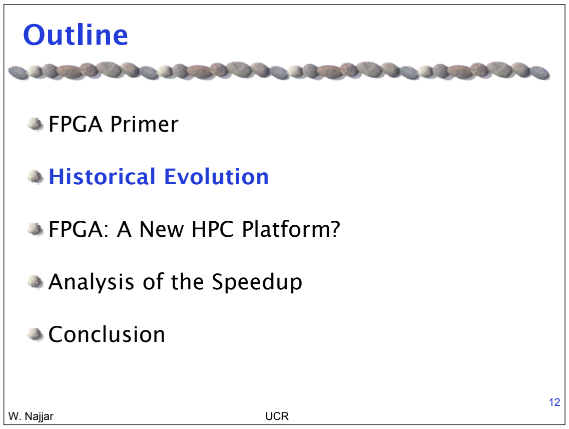
HDL

Synthesis

Place & Route

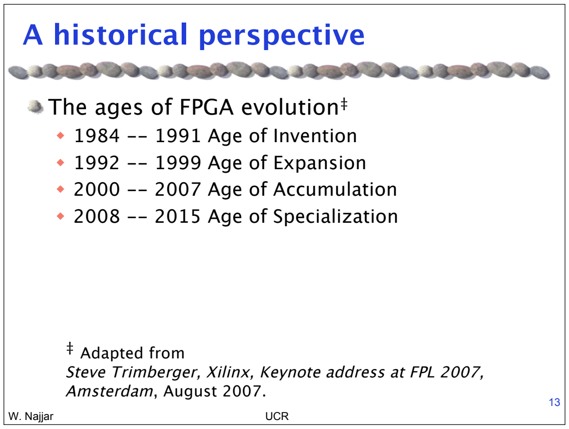
Download & Verify in Circuit

- Clock-cycle level accuracy
- Tedious
- Error-prone
- Acquisition of the skill
  - Digital design background
  - Syntax
- Low-level design
- Low productivity



Outline

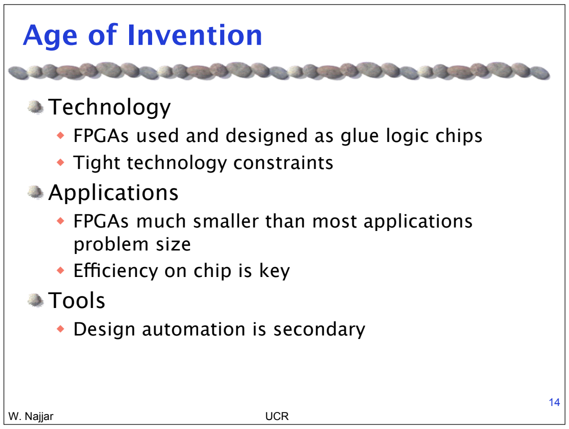
- FPGA Primer
- **Historical Evolution**
- FPGA: A New HPC Platform?
- Analysis of the Speedup
- Conclusion



A historical perspective

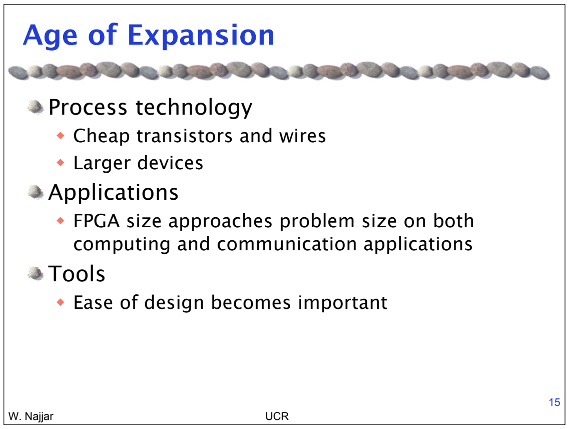
- The ages of FPGA evolution‡
  - 1984 -- 1991 Age of Invention
  - 1992 -- 1999 Age of Expansion
  - 2000 -- 2007 Age of Accumulation
  - 2008 -- 2015 Age of Specialization

‡ Adapted from
Steve Trimberger, Xilinx, Keynote address at FPL 2007, Amsterdam, August 2007.



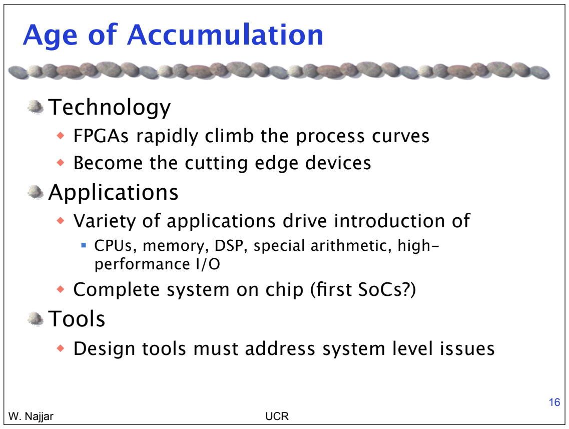
Age of Invention

- Technology
  - FPGAs used and designed as glue logic chips
  - Tight technology constraints
- Applications
  - FPGAs much smaller than most applications problem size
  - Efficiency on chip is key
- Tools
  - Design automation is secondary



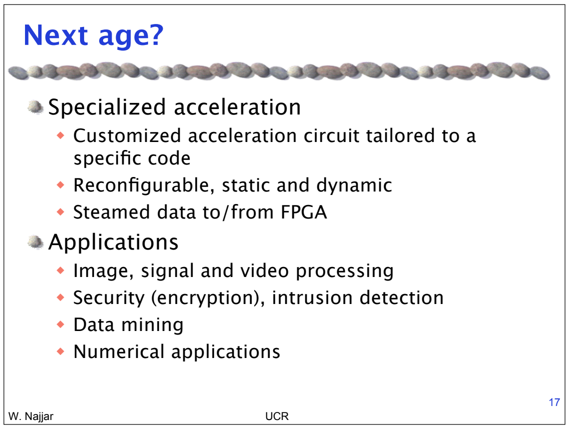
Age of Expansion

- Process technology
  - Cheap transistors and wires
  - Larger devices
- Applications
  - FPGA size approaches problem size on both computing and communication applications
- Tools
  - Ease of design becomes important



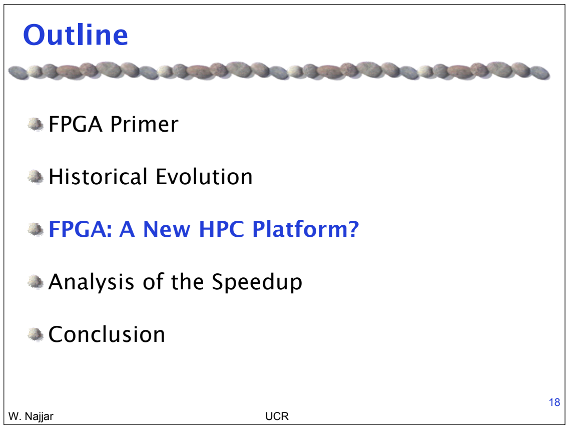
Age of Accumulation

- Technology
  - FPGAs rapidly climb the process curves
  - Become the cutting edge devices
- Applications
  - Variety of applications drive introduction of
    - CPUs, memory, DSP, special arithmetic, high-performance I/O
  - Complete system on chip (first SoCs?)
- Tools
  - Design tools must address system level issues



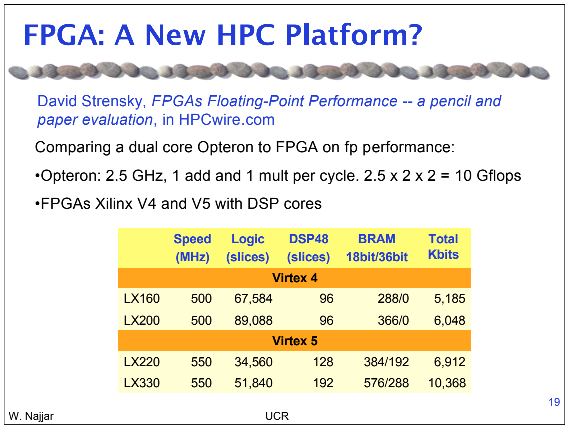
Next age?

- Specialized acceleration
  - Customized acceleration circuit tailored to a specific code
  - Reconfigurable, static and dynamic
  - Steamed data to/from FPGA
- Applications
  - Image, signal and video processing
  - Security (encryption), intrusion detection
  - Data mining
  - Numerical applications



Outline

- FPGA Primer
- Historical Evolution
- **FPGA: A New HPC Platform?**
- Analysis of the Speedup
- Conclusion



FPGA: A New HPC Platform?

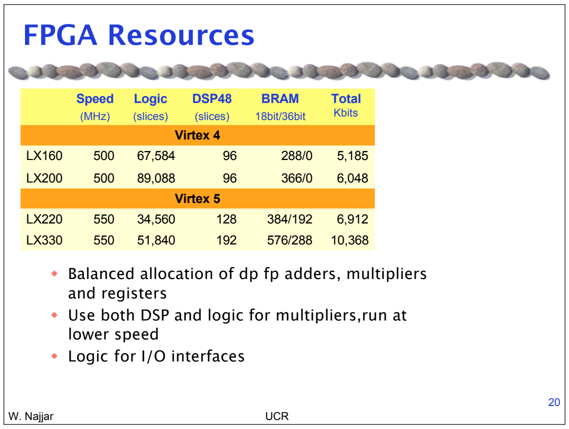
David Strensky, *FPGAs Floating-Point Performance -- a pencil and paper evaluation*, in HPCwire.com

Comparing a dual core Opteron to FPGA on fp performance:

•Opteron: 2.5 GHz, 1 add and 1 mult per cycle. 2.5 x 2 x 2 = 10 Gflops

•FPGAs Xilinx V4 and V5 with DSP cores

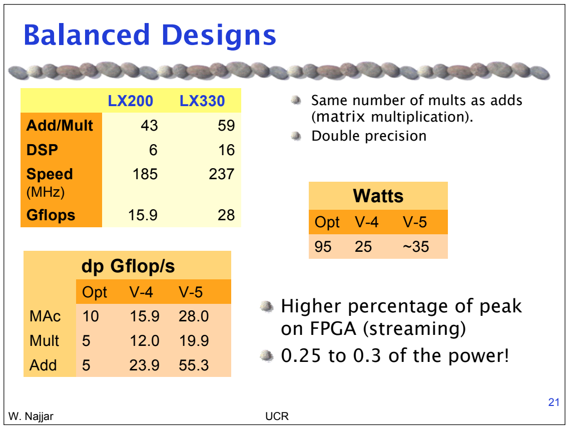
	Speed (MHz)	Logic (slices)	DSP48 (slices)	BRAM 18bit/36bit	Total Kbits
Virtex 4					
LX160	500	67,584	96	288/0	5,185
LX200	500	89,088	96	366/0	6,048
Virtex 5					
LX220	550	34,560	128	384/192	6,912
LX330	550	51,840	192	576/288	10,368



FPGA Resources

	Speed (MHz)	Logic (slices)	DSP48 (slices)	BRAM 18bit/36bit	Total Kbits
Virtex 4					
LX160	500	67,584	96	288/0	5,185
LX200	500	89,088	96	366/0	6,048
Virtex 5					
LX220	550	34,560	128	384/192	6,912
LX330	550	51,840	192	576/288	10,368

- Balanced allocation of dp fp adders, multipliers and registers
- Use both DSP and logic for multipliers,run at lower speed
- Logic for I/O interfaces



Balanced Designs

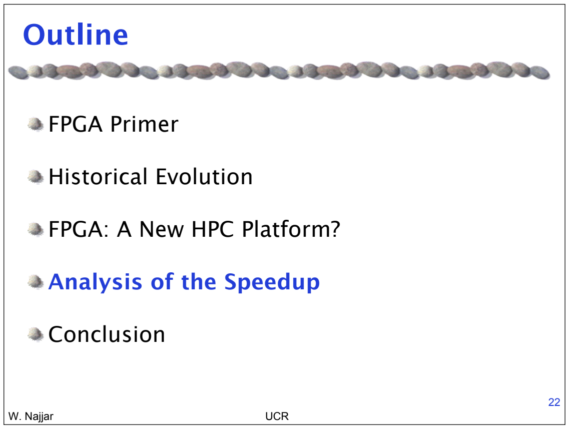
	LX200	LX330
Add/Mult	43	59
DSP	6	16
Speed (MHz)	185	237
Gflops	15.9	28

- Same number of mults as adds (matrix multiplication).
- Double precision

Watts		
Opt	V-4	V-5
95	25	~35

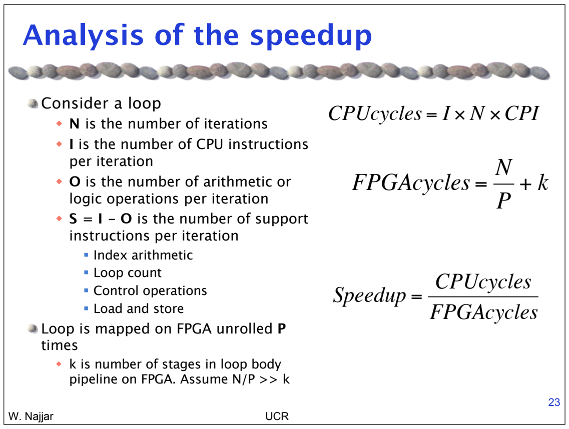
	dp Gflop/s		
	Opt	V-4	V-5
MAc	10	15.9	28.0
Mult	5	12.0	19.9
Add	5	23.9	55.3

- Higher percentage of peak on FPGA (streaming)
- 0.25 to 0.3 of the power!



Outline

- FPGA Primer
- Historical Evolution
- FPGA: A New HPC Platform?
- **Analysis of the Speedup**
- Conclusion



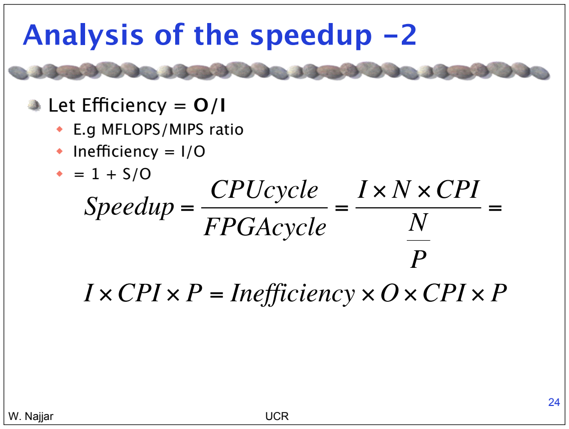
Analysis of the speedup

- Consider a loop
  - **N** is the number of iterations
  - **I** is the number of CPU instructions per iteration
  - **O** is the number of arithmetic or logic operations per iteration
  - **S** = **I** – **O** is the number of support instructions per iteration
    - Index arithmetic
    - Loop count
    - Control operations
    - Load and store
- Loop is mapped on FPGA unrolled **P** times
  - k is number of stages in loop body pipeline on FPGA. Assume N/P >> k

$$CPUcycles = I \times N \times CPI$$

$$FPGAcycles = \frac{N}{P} + k$$

$$Speedup = \frac{CPUcycles}{FPGAcycles}$$

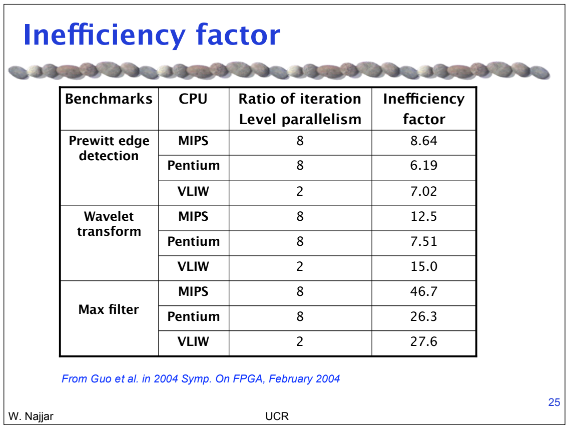


Analysis of the speedup -2

- Let Efficiency = **O/I**
  - E.g MFLOPS/MIPS ratio
  - Inefficiency = I/O
  - = 1 + S/O

$$Speedup = \frac{CPUcycle}{FPGAcycle} = \frac{I \times N \times CPI}{\frac{N}{P}} =$$

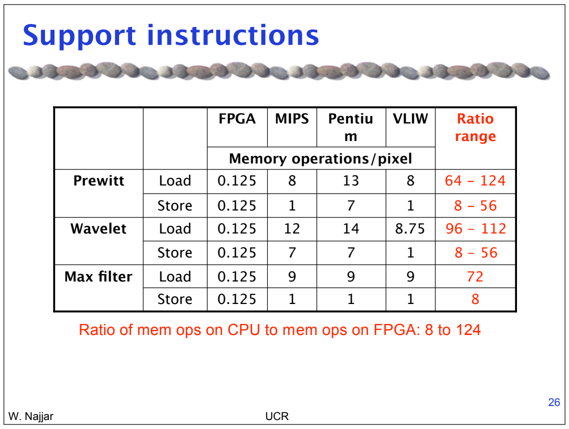
$$I \times CPI \times P = Inefficiency \times O \times CPI \times P$$



Inefficiency factor

Benchmarks	CPU	Ratio of iteration Level parallelism	Inefficiency factor
Prewitt edge detection	MIPS	8	8.64
	Pentium	8	6.19
	VLIW	2	7.02
Wavelet transform	MIPS	8	12.5
	Pentium	8	7.51
	VLIW	2	15.0
Max filter	MIPS	8	46.7
	Pentium	8	26.3
	VLIW	2	27.6

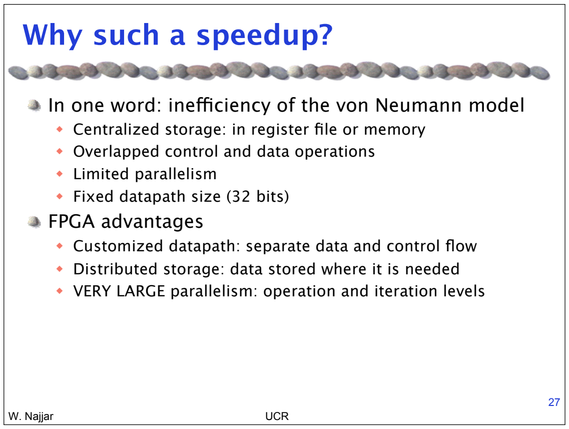
From Guo et al. in 2004 Symp. On FPGA, February 2004



Support instructions

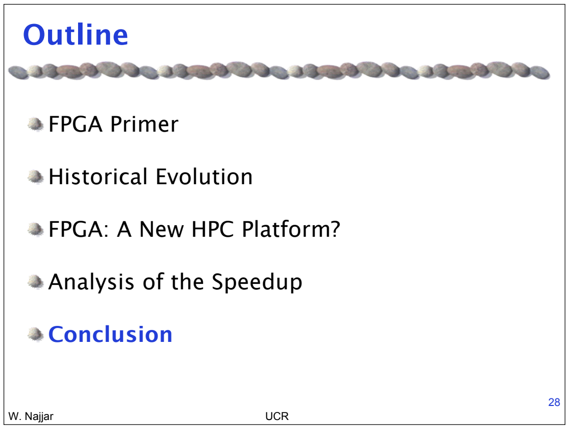
		FPGA	MIPS	Pentiu m	VLIW	Ratio range
		Memory operations/pixel				
Prewitt	Load	0.125	8	13	8	64 – 124
	Store	0.125	1	7	1	8 – 56
Wavelet	Load	0.125	12	14	8.75	96 – 112
	Store	0.125	7	7	1	8 – 56
Max filter	Load	0.125	9	9	9	72
	Store	0.125	1	1	1	8

Ratio of mem ops on CPU to mem ops on FPGA: 8 to 124



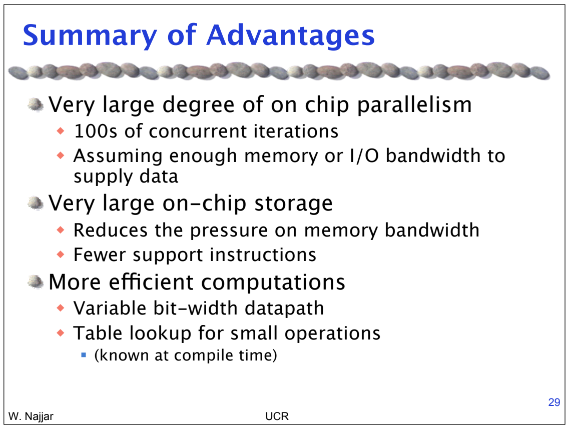
Why such a speedup?

- In one word: inefficiency of the von Neumann model
  - Centralized storage: in register file or memory
  - Overlapped control and data operations
  - Limited parallelism
  - Fixed datapath size (32 bits)
- FPGA advantages
  - Customized datapath: separate data and control flow
  - Distributed storage: data stored where it is needed
  - VERY LARGE parallelism: operation and iteration levels



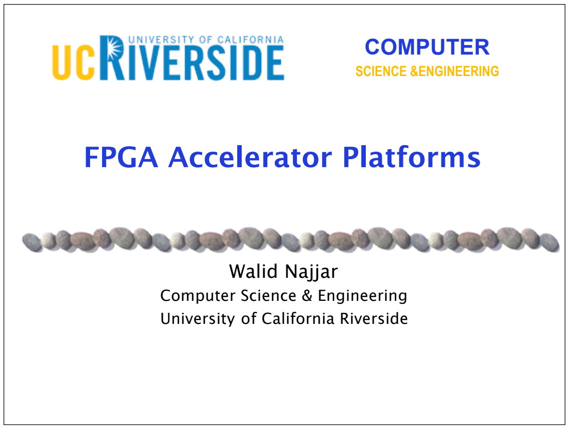
Outline

- FPGA Primer
- Historical Evolution
- FPGA: A New HPC Platform?
- Analysis of the Speedup
- **Conclusion**



Summary of Advantages

- Very large degree of on chip parallelism
  - 100s of concurrent iterations
  - Assuming enough memory or I/O bandwidth to supply data
- Very large on-chip storage
  - Reduces the pressure on memory bandwidth
  - Fewer support instructions
- More efficient computations
  - Variable bit-width datapath
  - Table lookup for small operations
    - (known at compile time)

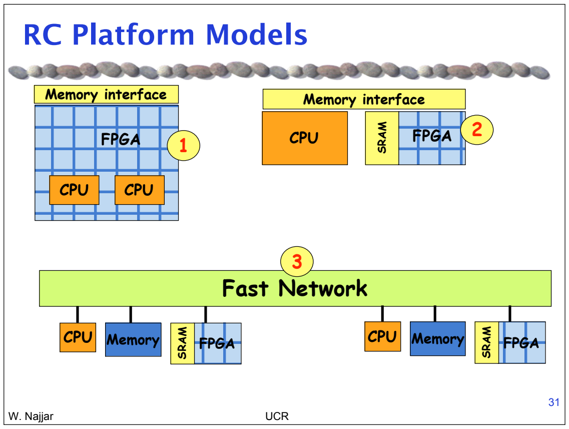


UC RIVERSIDE UNIVERSITY OF CALIFORNIA

COMPUTER SCIENCE &ENGINEERING

FPGA Accelerator Platforms

Walid Najjar
Computer Science & Engineering
University of California Riverside



RC Platform Models

Memory interface

FPGA

1

CPU

CPU

Memory interface

CPU

SRAM

FPGA

2

3

Fast Network

CPU

Memory

SRAM

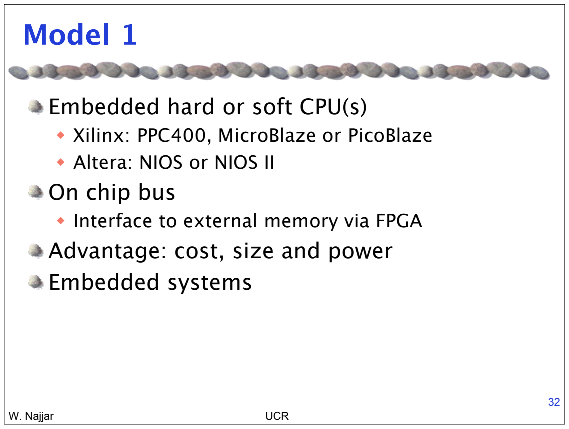
FPGA

CPU

Memory

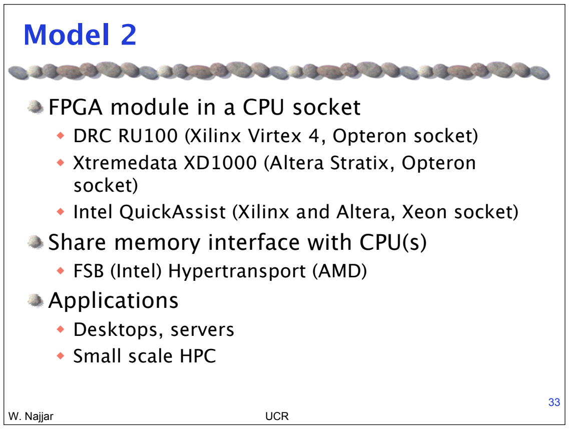
SRAM

FPGA



Model 1

- Embedded hard or soft CPU(s)
  - Xilinx: PPC400, MicroBlaze or PicoBlaze
  - Altera: NIOS or NIOS II
- On chip bus
  - Interface to external memory via FPGA
- Advantage: cost, size and power
- Embedded systems



Model 2

- FPGA module in a CPU socket
  - DRC RU100 (Xilinx Virtex 4, Opteron socket)
  - Xtremedata XD1000 (Altera Stratix, Opteron socket)
  - Intel QuickAssist (Xilinx and Altera, Xeon socket)
- Share memory interface with CPU(s)
  - FSB (Intel) Hypertransport (AMD)
- Applications
  - Desktops, servers
  - Small scale HPC

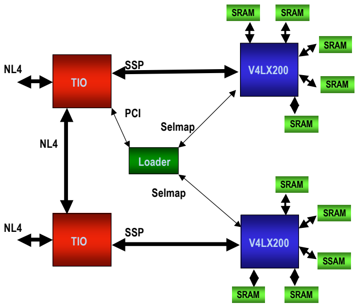
Model 3

- Medium to high-end HPC systems
  - Very fast network
  - Large number of FPGAs
  - Very large memory
- Examples
  - Cray XD1 (defunct)
  - SGI Altix 4700 with RASC blade
  - SRC (new SRC 7)
  - Cray XT3 with DRC modules (future)

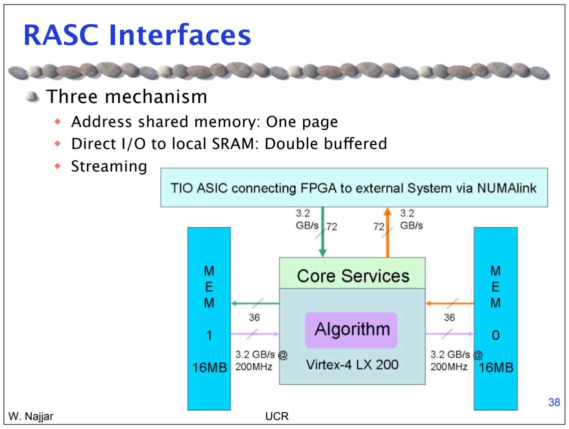
Accelerator Platforms

- SGI Altix 4700
  - Shared memory machine, fast interconnect: 12.8 GB/sec
  - Itanium 2, 1.6 GHz
  - RASC RC100 Blade: 2 Virtex 4 LX200
  - Memory size independent of number of CPUs
- Xtremedata XD1000
  - Altera Stratix II drop-in for AMD Opteron
  - Integrated interface to Hypertransport
    - 16 bits @ 800 M transfers/sec
  - Memory interface
    - 128 bits DDR-333up to 4 x 4 GB ECC
    - Flash memory
      - For FPGA configuration or data

SGI® RASC™ RC100 Blade

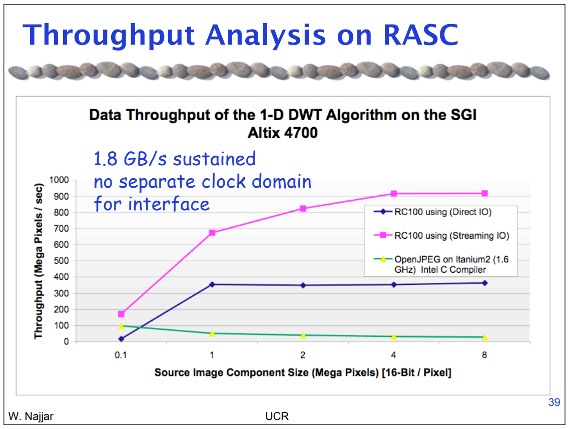


SGI® RASC™ RC100 Blade



RASC Interfaces

- Three mechanism
  - Address shared memory: One page
  - Direct I/O to local SRAM: Double buffered
  - Streaming



Throughput Analysis on RASC

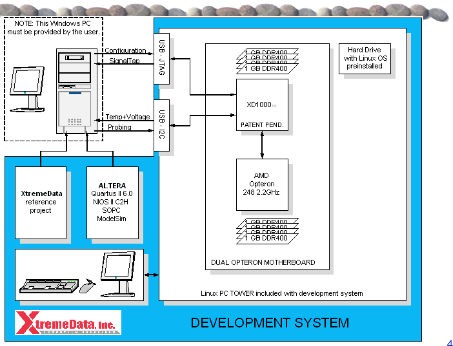
Data Throughput of the 1-D DWT Algorithm on the SGI Altix 4700

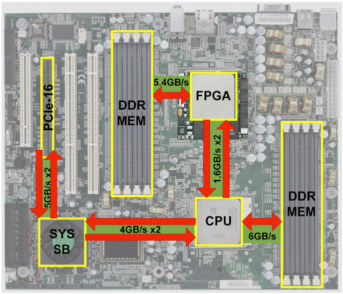
1.8 GB/s sustained
no separate clock domain
for interface

XD 1000 FPGA Co-processor

- Drop-in an Opteron 940 socket
  - Altera Stratix II FPGA
  - HyperTransport
    - Multiple interfaces
    - 16 bits @ 800 M transfers/sec
  - SRAM 4 MB ZBT
- Memory interface
  - 128 bits DDR-333up to 4 x 4 GB ECC
  - Flash memory
    - For FPGA configuration or data

Xtremedata XD1000





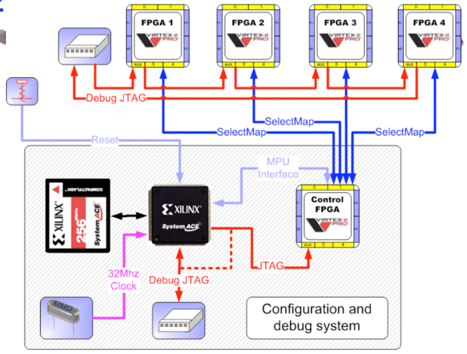
XD 1000

XD 1000 (drop-in)

Standalone Platforms

- A board with many FPGAs
- BEE2 and BEE3
  - Berkeley Emulation Engine, developed at BWRC
  - BEE3 is the platform for RAMP
    - Objective: emulation/simulation of multicore architecture design
- Many other manufacturers, examples:
  - Nallatech http://www.nallatech.com/
  - Dini http://www.dinigroup.com/

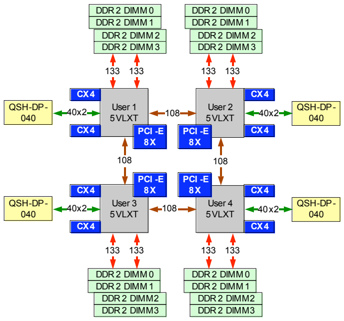
BEE2

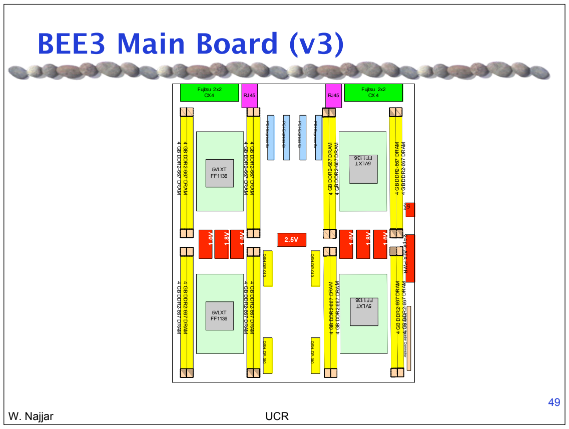


BEE3 Highlights

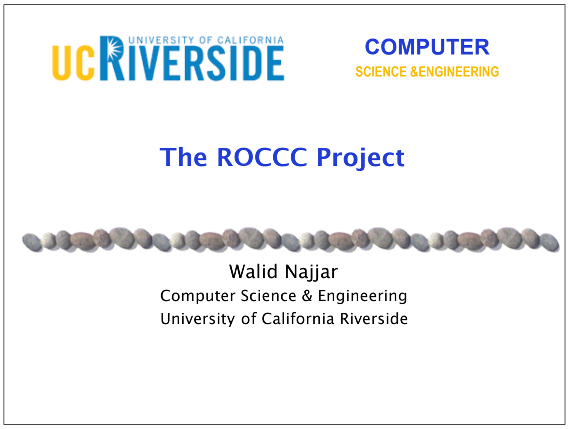
- 4 Xilinx Virtex 5
  - V5 is a major improvement (65nm)
    - 6-input LUT (64 bit DP RAM)
    - Better Block RAMs
    - Improved interconnect
    - Better signal integrity
- 8 Infiniband/CX4 channels
- 4 x8 PCI Express low profile slots

BEE3 Main Board





BEE3 Main Board (v3)

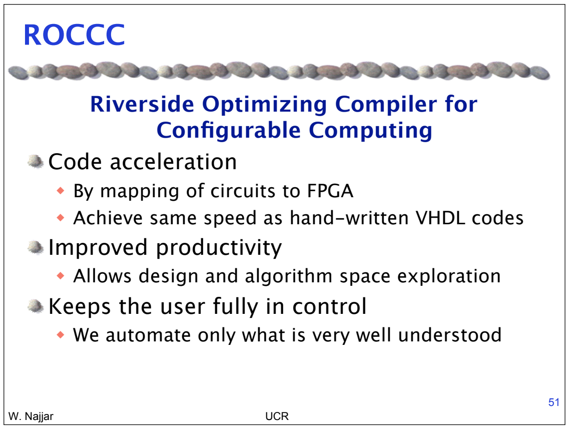


UC RIVERSIDE UNIVERSITY OF CALIFORNIA

COMPUTER
SCIENCE &ENGINEERING

The ROCCC Project

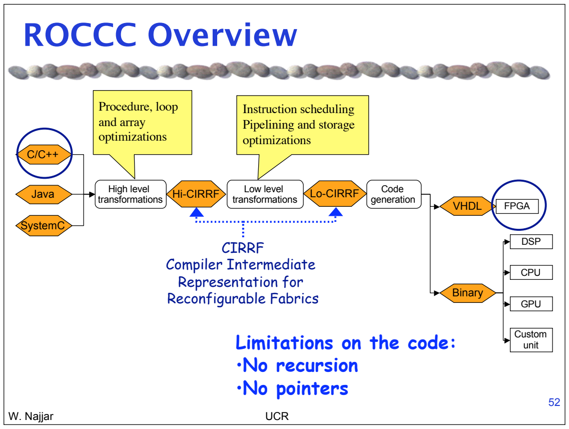
Walid Najjar
Computer Science & Engineering
University of California Riverside



ROCCC

Riverside Optimizing Compiler for Configurable Computing

- Code acceleration
  - By mapping of circuits to FPGA
  - Achieve same speed as hand-written VHDL codes
- Improved productivity
  - Allows design and algorithm space exploration
- Keeps the user fully in control
  - We automate only what is very well understood

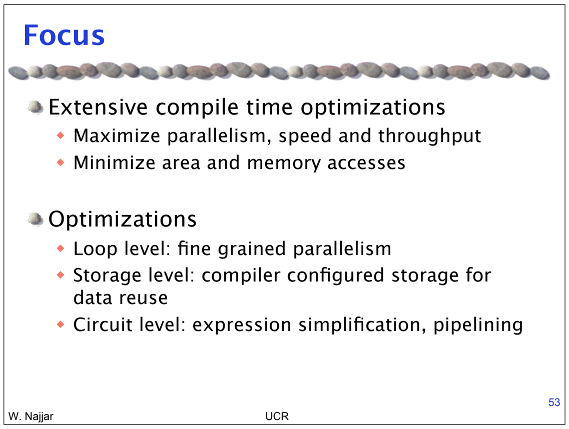


ROCCC Overview

Procedure, loop and array optimizations
Instruction scheduling Pipelining and storage optimizations
C/C++
Java
SystemC
High level transformations
Hi-CIRRF
Low level transformations
Lo-CIRRF
Code generation
VHDL
FPGA
DSP
CPU
Binary
GPU
Custom unit

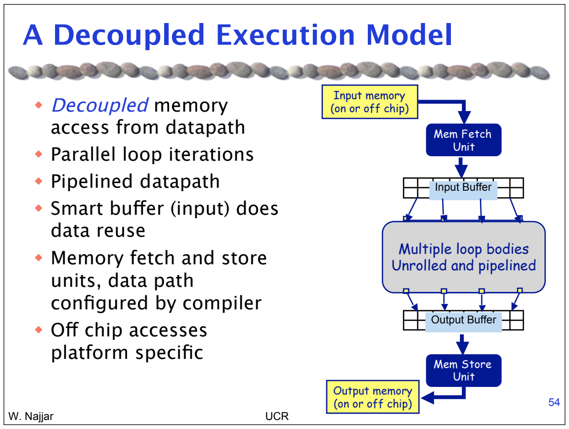
CIRRF
Compiler Intermediate
Representation for
Reconfigurable Fabrics

Limitations on the code:
·No recursion
·No pointers



Focus

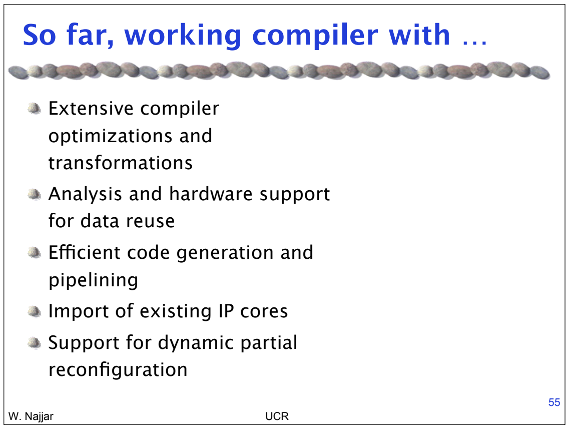
- Extensive compile time optimizations
  - Maximize parallelism, speed and throughput
  - Minimize area and memory accesses
- Optimizations
  - Loop level: fine grained parallelism
  - Storage level: compiler configured storage for data reuse
  - Circuit level: expression simplification, pipelining



A Decoupled Execution Model

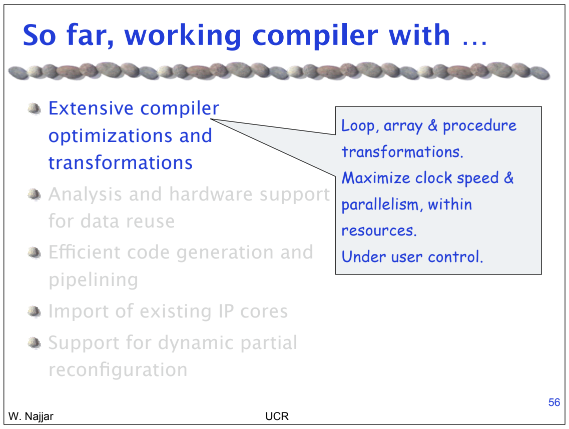
- *Decoupled* memory access from datapath
- Parallel loop iterations
- Pipelined datapath
- Smart buffer (input) does data reuse
- Memory fetch and store units, data path configured by compiler
- Off chip accesses platform specific

Input memory (on or off chip)
Mem Fetch Unit
Input Buffer
Multiple loop bodies Unrolled and pipelined
Output Buffer
Mem Store Unit
Output memory (on or off chip)



So far, working compiler with …

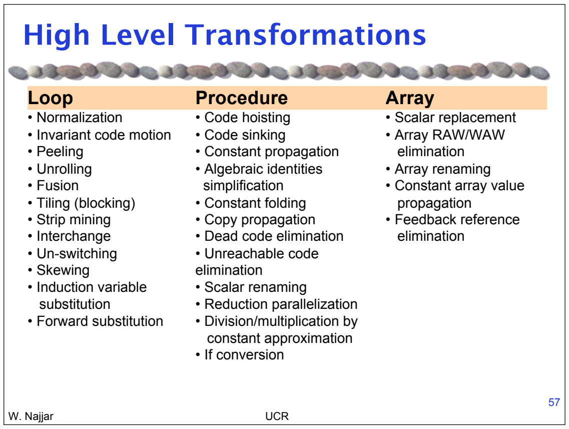
- Extensive compiler optimizations and transformations
- Analysis and hardware support for data reuse
- Efficient code generation and pipelining
- Import of existing IP cores
- Support for dynamic partial reconfiguration



So far, working compiler with …

- Extensive compiler optimizations and transformations
- Analysis and hardware support for data reuse
- Efficient code generation and pipelining
- Import of existing IP cores
- Support for dynamic partial reconfiguration

Loop, array & procedure transformations.
Maximize clock speed & parallelism, within resources.
Under user control.

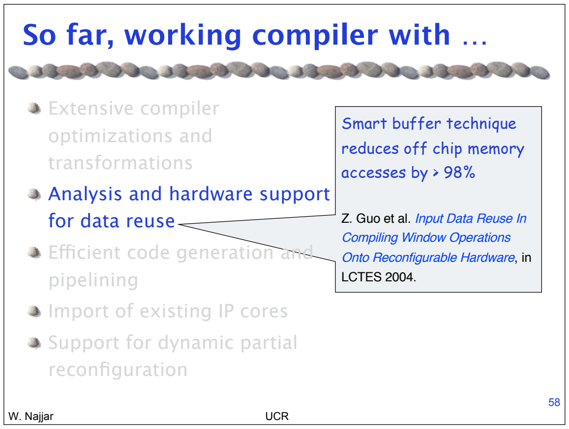


High Level Transformations

Loop
- Normalization
- Invariant code motion
- Peeling
- Unrolling
- Fusion
- Tiling (blocking)
- Strip mining
- Interchange
- Un-switching
- Skewing
- Induction variable substitution
- Forward substitution

Procedure
- Code hoisting
- Code sinking
- Constant propagation
- Algebraic identities simplification
- Constant folding
- Copy propagation
- Dead code elimination
- Unreachable code elimination
- Scalar renaming
- Reduction parallelization
- Division/multiplication by constant approximation
- If conversion

Array
- Scalar replacement
- Array RAW/WAW elimination
- Array renaming
- Constant array value propagation
- Feedback reference elimination

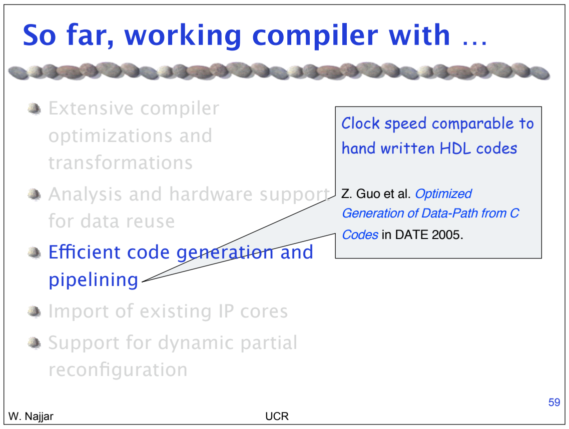


So far, working compiler with …

- Extensive compiler optimizations and transformations
- Analysis and hardware support for data reuse
- Efficient code generation and pipelining
- Import of existing IP cores
- Support for dynamic partial reconfiguration

Smart buffer technique reduces off chip memory accesses by > 98%

Z. Guo et al. *Input Data Reuse In Compiling Window Operations Onto Reconfigurable Hardware*, in LCTES 2004.

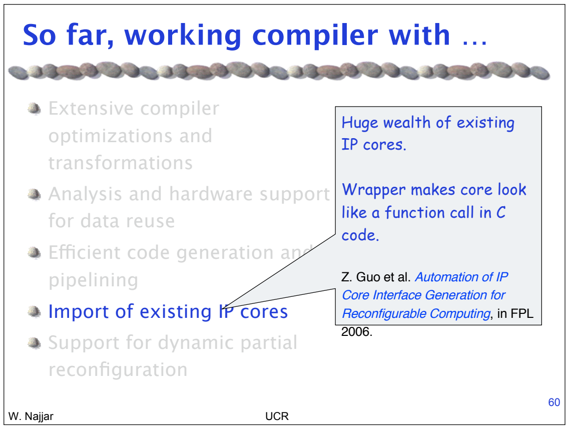


So far, working compiler with …

- Extensive compiler optimizations and transformations
- Analysis and hardware support for data reuse
- Efficient code generation and pipelining
- Import of existing IP cores
- Support for dynamic partial reconfiguration

Clock speed comparable to hand written HDL codes

Z. Guo et al. *Optimized Generation of Data-Path from C Codes* in DATE 2005.



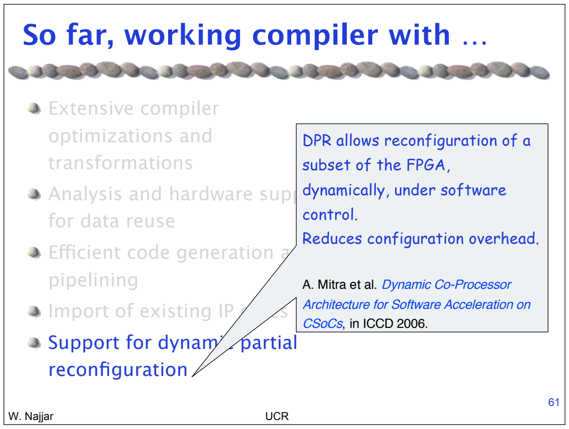
So far, working compiler with …

- Extensive compiler optimizations and transformations
- Analysis and hardware support for data reuse
- Efficient code generation and pipelining
- Import of existing IP cores
- Support for dynamic partial reconfiguration

Huge wealth of existing IP cores.

Wrapper makes core look like a function call in C code.

Z. Guo et al. *Automation of IP Core Interface Generation for Reconfigurable Computing*, in FPL 2006.

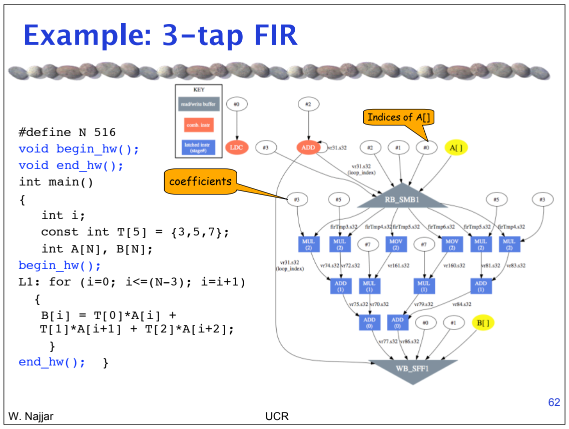


So far, working compiler with …

- Extensive compiler optimizations and transformations
- Analysis and hardware support for data reuse
- Efficient code generation and pipelining
- Import of existing IP cores
- Support for dynamic partial reconfiguration

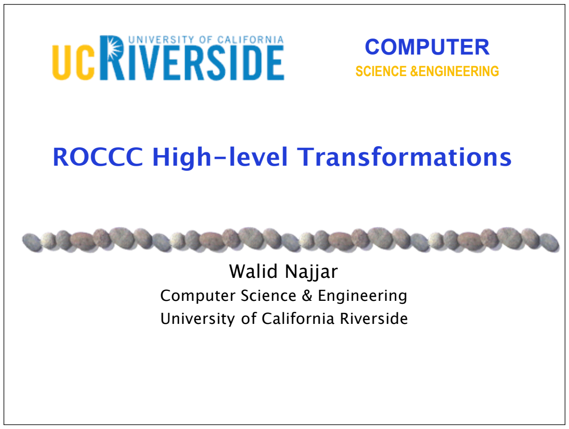
DPR allows reconfiguration of a subset of the FPGA, dynamically, under software control.
Reduces configuration overhead.

A. Mitra et al. *Dynamic Co-Processor Architecture for Software Acceleration on CSoCs*, in ICCD 2006.



Example: 3-tap FIR

```
#define N 516
void begin_hw();
void end_hw();
int main()
{
   int i;
   const int T[5] = {3,5,7};
   int A[N], B[N];
begin_hw();
L1: for (i=0; i<=(N-3); i=i+1)
  {
   B[i] = T[0]*A[i] +
   T[1]*A[i+1] + T[2]*A[i+2];
    }
end_hw();  }
```

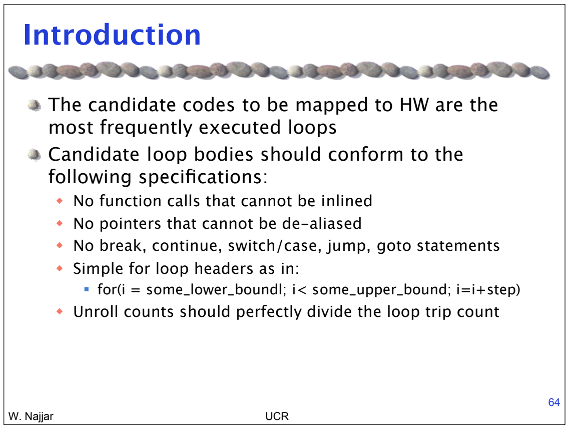


UC RIVERSIDE UNIVERSITY OF CALIFORNIA

COMPUTER
SCIENCE &ENGINEERING

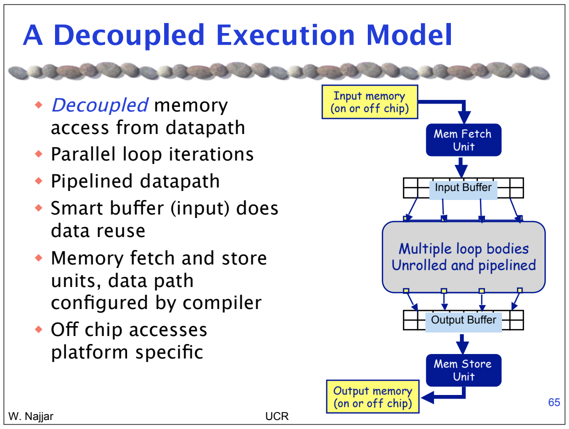
ROCCC High-level Transformations

Walid Najjar
Computer Science & Engineering
University of California Riverside



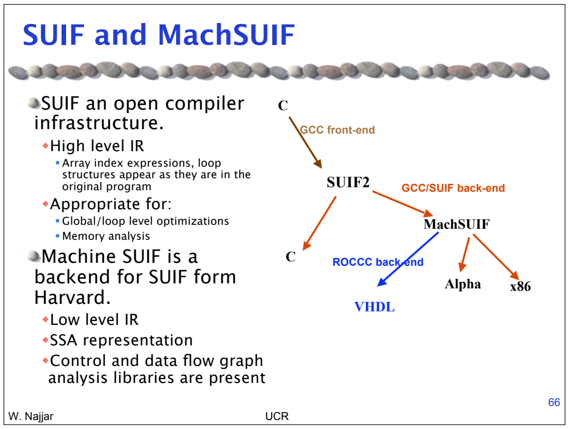
Introduction

- The candidate codes to be mapped to HW are the most frequently executed loops
- Candidate loop bodies should conform to the following specifications:
  - No function calls that cannot be inlined
  - No pointers that cannot be de-aliased
  - No break, continue, switch/case, jump, goto statements
  - Simple for loop headers as in:
    - for(i = some_lower_boundl; i< some_upper_bound; i=i+step)
  - Unroll counts should perfectly divide the loop trip count



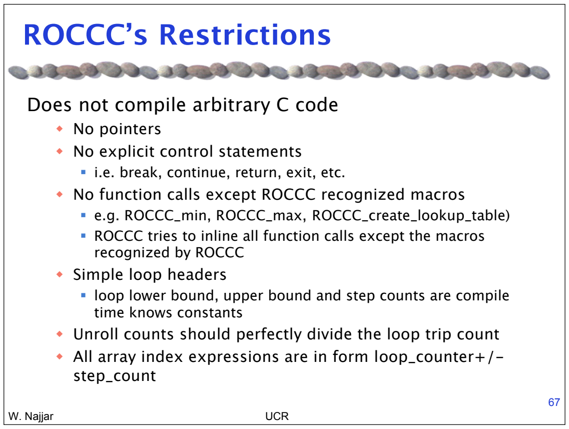
A Decoupled Execution Model

- *Decoupled* memory access from datapath
- Parallel loop iterations
- Pipelined datapath
- Smart buffer (input) does data reuse
- Memory fetch and store units, data path configured by compiler
- Off chip accesses platform specific



SUIF and MachSUIF

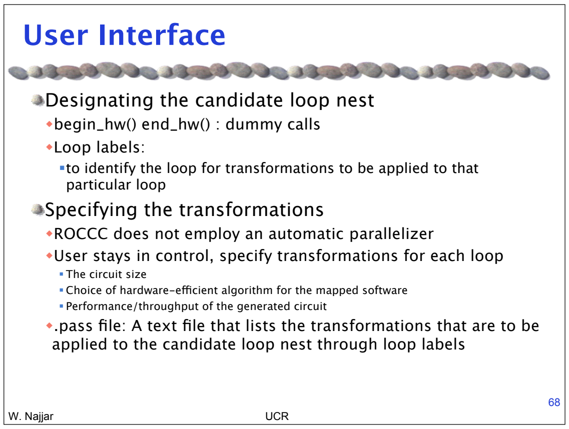
- SUIF an open compiler infrastructure.
  - High level IR
    - Array index expressions, loop structures appear as they are in the original program
  - Appropriate for:
    - Global/loop level optimizations
    - Memory analysis
- Machine SUIF is a backend for SUIF form Harvard.
  - Low level IR
  - SSA representation
  - Control and data flow graph analysis libraries are present



ROCCC's Restrictions

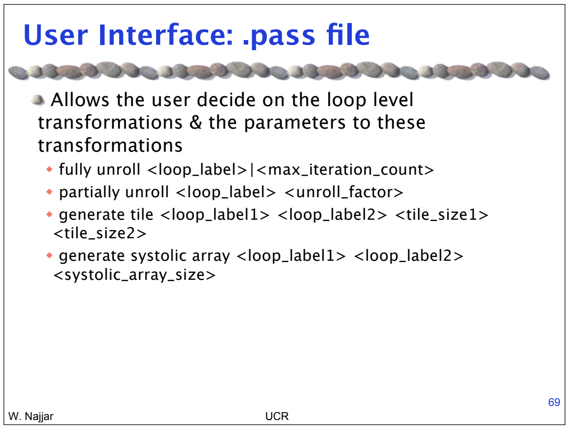
Does not compile arbitrary C code

- No pointers
- No explicit control statements
  - i.e. break, continue, return, exit, etc.
- No function calls except ROCCC recognized macros
  - e.g. ROCCC_min, ROCCC_max, ROCCC_create_lookup_table)
  - ROCCC tries to inline all function calls except the macros recognized by ROCCC
- Simple loop headers
  - loop lower bound, upper bound and step counts are compile time knows constants
- Unroll counts should perfectly divide the loop trip count
- All array index expressions are in form loop_counter+/-step_count



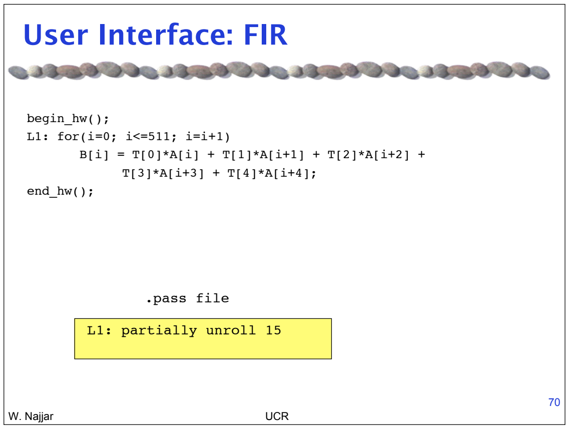
User Interface

- Designating the candidate loop nest
  - begin_hw() end_hw() : dummy calls
  - Loop labels:
    - to identify the loop for transformations to be applied to that particular loop
- Specifying the transformations
  - ROCCC does not employ an automatic parallelizer
  - User stays in control, specify transformations for each loop
    - The circuit size
    - Choice of hardware-efficient algorithm for the mapped software
    - Performance/throughput of the generated circuit
  - .pass file: A text file that lists the transformations that are to be applied to the candidate loop nest through loop labels



User Interface: .pass file

- Allows the user decide on the loop level transformations & the parameters to these transformations
  - fully unroll <loop_label>|<max_iteration_count>
  - partially unroll <loop_label> <unroll_factor>
  - generate tile <loop_label1> <loop_label2> <tile_size1> <tile_size2>
  - generate systolic array <loop_label1> <loop_label2> <systolic_array_size>

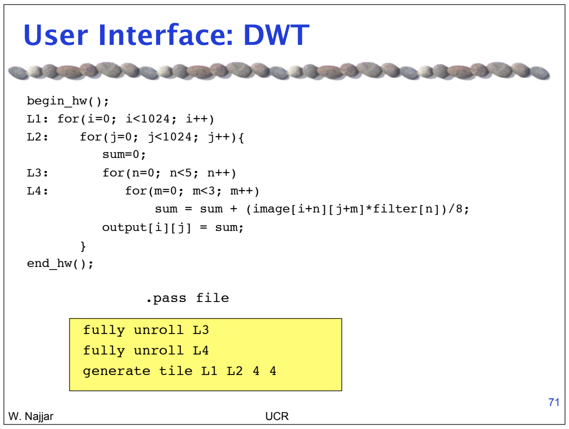


User Interface: FIR

```
begin_hw();
L1: for(i=0; i<=511; i=i+1)
       B[i] = T[0]*A[i] + T[1]*A[i+1] + T[2]*A[i+2] +
             T[3]*A[i+3] + T[4]*A[i+4];
end_hw();
```

.pass file

```
L1: partially unroll 15
```

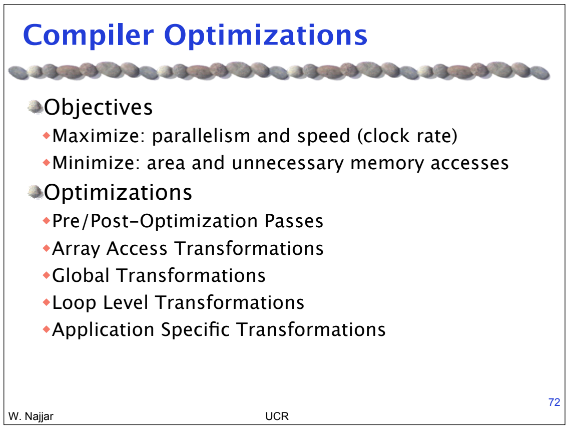


User Interface: DWT

```
begin_hw();
L1: for(i=0; i<1024; i++)
L2:     for(j=0; j<1024; j++){
           sum=0;
L3:        for(n=0; n<5; n++)
L4:           for(m=0; m<3; m++)
                  sum = sum + (image[i+n][j+m]*filter[n])/8;
           output[i][j] = sum;
        }
end_hw();
```

.pass file

```
fully unroll L3
fully unroll L4
generate tile L1 L2 4 4
```



Compiler Optimizations

- Objectives
  - Maximize: parallelism and speed (clock rate)
  - Minimize: area and unnecessary memory accesses
- Optimizations
  - Pre/Post-Optimization Passes
  - Array Access Transformations
  - Global Transformations
  - Loop Level Transformations
  - Application Specific Transformations

Compiler Transformations

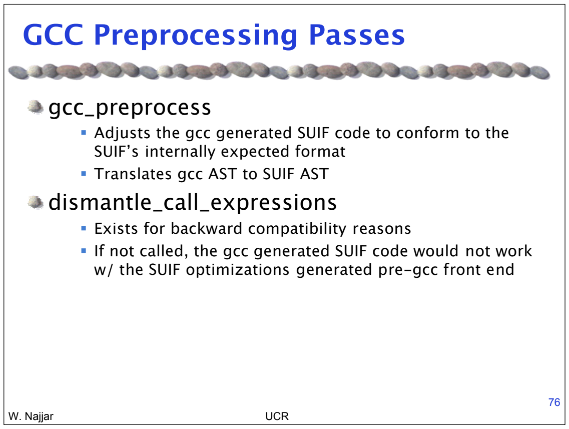
- Pre-Optimization Passes
  - Switch Case to If Statements
  - Do While to While Statement
  - Inlining Pass
  - Control Flow Analysis
  - Data Flow Analysis
  - Use/Def & Def/Use Chain Builder
  - Lookup Table Expansion
  - DFA Expansion
- Array Access Transformations
  - Constant Propagation of "const" qualified Arrays
  - RAW/WAW Elimination
  - Scalar Replacement
  - Feedback Array Access Elimination
  - Array Renaming
  - Systolic Array Generation

Compiler Transformations

- Global Transformations
  - Constant Propagation, Folding and Elimination of Algebraic Identities
  - Code Hoisting &Sinking
  - Copy & Reverse Copy Propagation
  - Common Subexpression Elimination
  - Dead & Unreachable Code Elimination
  - Division & Multiplication By Constants Approximation
  - If Conversion
  - Reduction Parallelization
  - Scalar Renaming
- Loop Level Transformations
  - Partial & Full Loop Unrolling
  - Loop Fusing
  - Loop Interchanging
  - Loop Invariant Code Motion
  - Loop Peeling
  - Loop Normalization
  - Loop Tiling / Strip Mining
  - Loop Unswitching

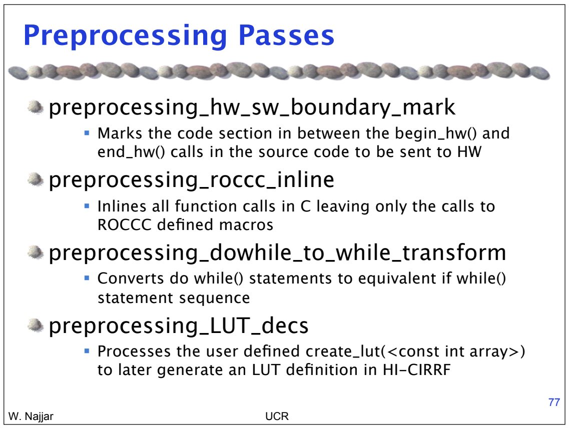
Phase Order of Transformations

1) Preprocessing Passes
2) Global & Loop Level Transformations
   - To simplify and bring the array index expressions to a standard format
3) Array Transformations
4) Global Transformations
5) Output (Hi-CIRRF) Generation



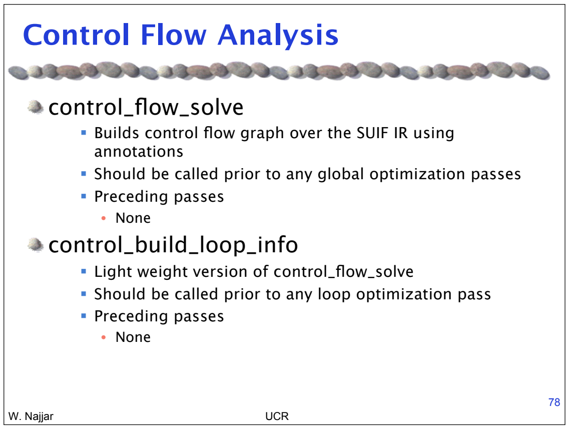
GCC Preprocessing Passes

- gcc_preprocess
  - Adjusts the gcc generated SUIF code to conform to the SUIF's internally expected format
  - Translates gcc AST to SUIF AST
- dismantle_call_expressions
  - Exists for backward compatibility reasons
  - If not called, the gcc generated SUIF code would not work w/ the SUIF optimizations generated pre-gcc front end



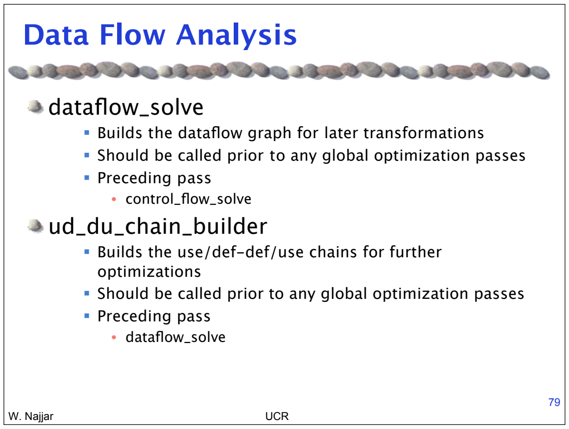
Preprocessing Passes

- preprocessing_hw_sw_boundary_mark
  - Marks the code section in between the begin_hw() and end_hw() calls in the source code to be sent to HW
- preprocessing_roccc_inline
  - Inlines all function calls in C leaving only the calls to ROCCC defined macros
- preprocessing_dowhile_to_while_transform
  - Converts do while() statements to equivalent if while() statement sequence
- preprocessing_LUT_decs
  - Processes the user defined create_lut(<const int array>) to later generate an LUT definition in HI-CIRRF



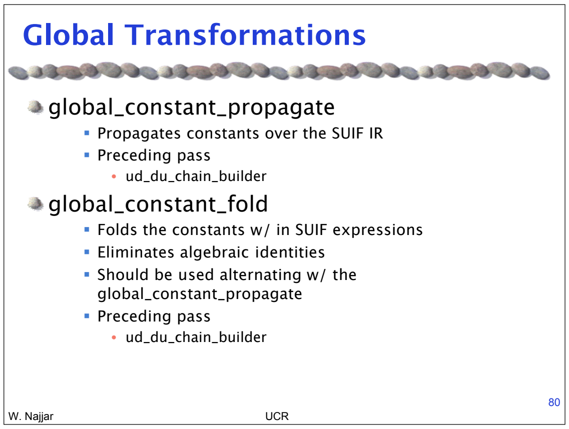
Control Flow Analysis

- control_flow_solve
  - Builds control flow graph over the SUIF IR using annotations
  - Should be called prior to any global optimization passes
  - Preceding passes
    - None
- control_build_loop_info
  - Light weight version of control_flow_solve
  - Should be called prior to any loop optimization pass
  - Preceding passes
    - None



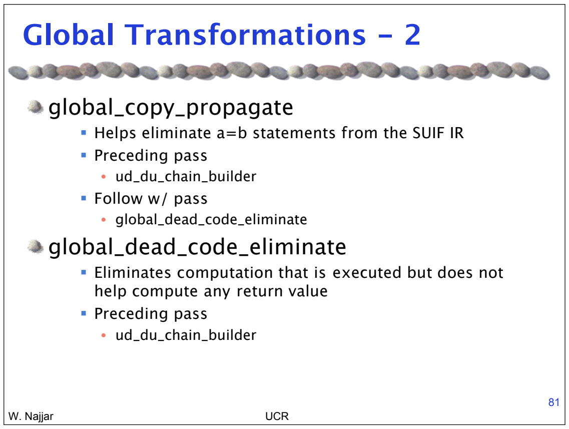
Data Flow Analysis

- dataflow_solve
  - Builds the dataflow graph for later transformations
  - Should be called prior to any global optimization passes
  - Preceding pass
    - control_flow_solve
- ud_du_chain_builder
  - Builds the use/def-def/use chains for further optimizations
  - Should be called prior to any global optimization passes
  - Preceding pass
    - dataflow_solve



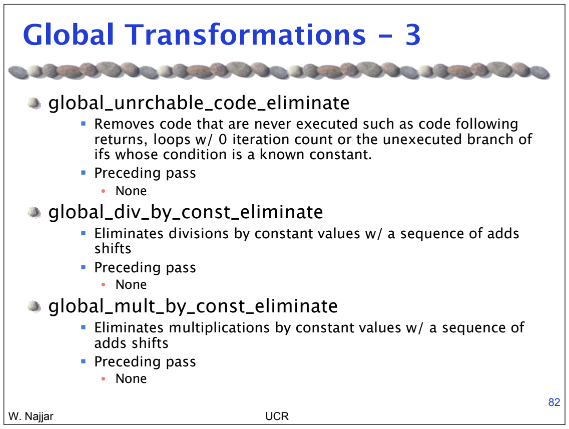
Global Transformations

- global_constant_propagate
  - Propagates constants over the SUIF IR
  - Preceding pass
    - ud_du_chain_builder
- global_constant_fold
  - Folds the constants w/ in SUIF expressions
  - Eliminates algebraic identities
  - Should be used alternating w/ the global_constant_propagate
  - Preceding pass
    - ud_du_chain_builder



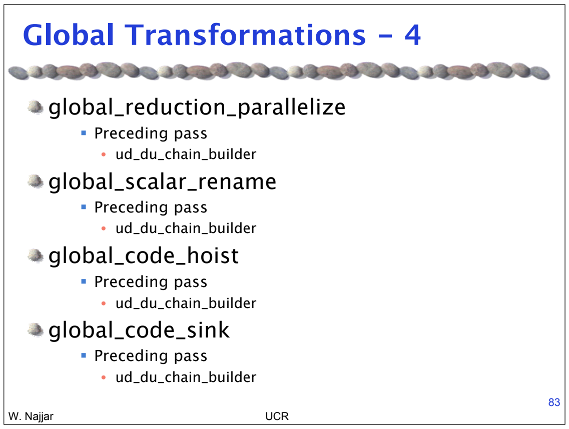
Global Transformations – 2

- global_copy_propagate
  - Helps eliminate a=b statements from the SUIF IR
  - Preceding pass
    - ud_du_chain_builder
  - Follow w/ pass
    - global_dead_code_eliminate
- global_dead_code_eliminate
  - Eliminates computation that is executed but does not help compute any return value
  - Preceding pass
    - ud_du_chain_builder



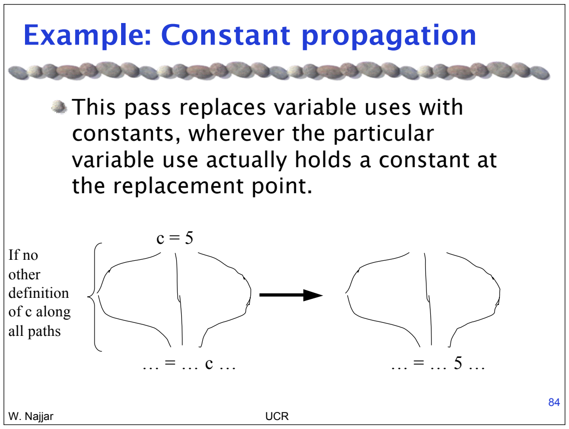
Global Transformations – 3

- global_unrchable_code_eliminate
  - Removes code that are never executed such as code following returns, loops w/ 0 iteration count or the unexecuted branch of ifs whose condition is a known constant.
  - Preceding pass
    - None
- global_div_by_const_eliminate
  - Eliminates divisions by constant values w/ a sequence of adds shifts
  - Preceding pass
    - None
- global_mult_by_const_eliminate
  - Eliminates multiplications by constant values w/ a sequence of adds shifts
  - Preceding pass
    - None



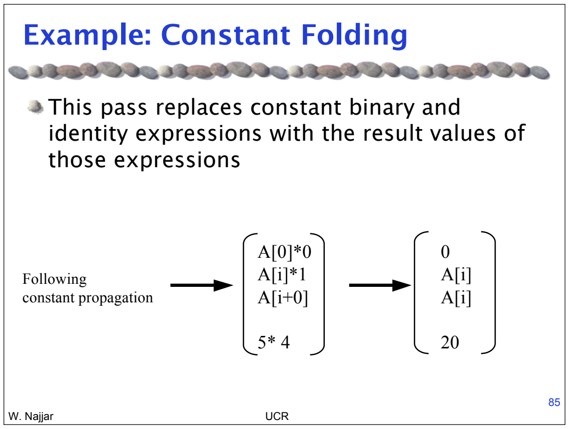
Global Transformations – 4

- global_reduction_parallelize
  - Preceding pass
    - ud_du_chain_builder
- global_scalar_rename
  - Preceding pass
    - ud_du_chain_builder
- global_code_hoist
  - Preceding pass
    - ud_du_chain_builder
- global_code_sink
  - Preceding pass
    - ud_du_chain_builder



Example: Constant propagation

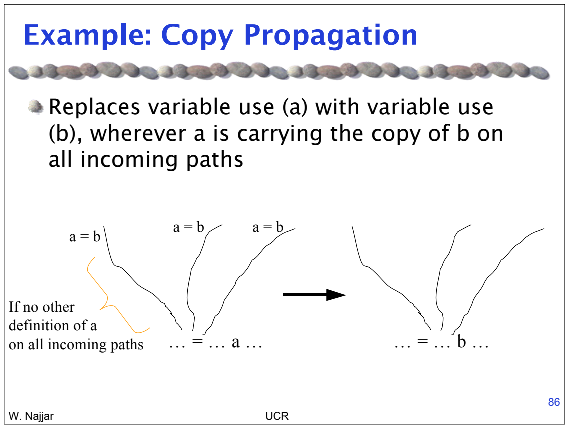
- This pass replaces variable uses with constants, wherever the particular variable use actually holds a constant at the replacement point.



Example: Constant Folding

- This pass replaces constant binary and identity expressions with the result values of those expressions

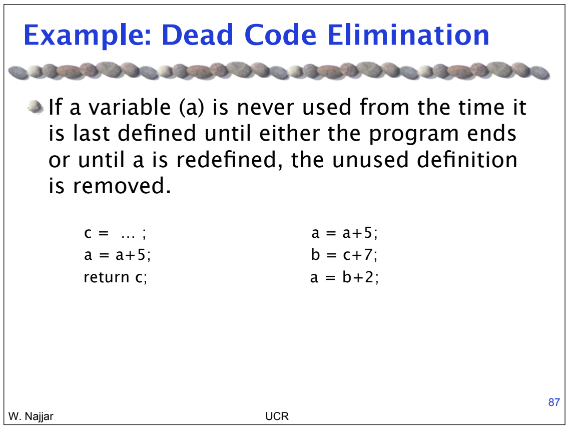
Following
constant propagation
A[0]*0
A[i]*1
A[i+0]
5* 4
0
A[i]
A[i]
20



Example: Copy Propagation

- Replaces variable use (a) with variable use (b), wherever a is carrying the copy of b on all incoming paths

a = b
a = b
a = b
If no other
definition of a
on all incoming paths
… = … a …
… = … b …



Example: Dead Code Elimination

- If a variable (a) is never used from the time it is last defined until either the program ends or until a is redefined, the unused definition is removed.

```
c =  … ;
a = a+5;
return c;
```

```
a = a+5;
b = c+7;
a = b+2;
```

Example: Invariant Code Motion

- Moves any computation that computes the same value at every iteration outside of the loop.

```
for i=1 to N do
    A[i] = A[i] + b*8;
```

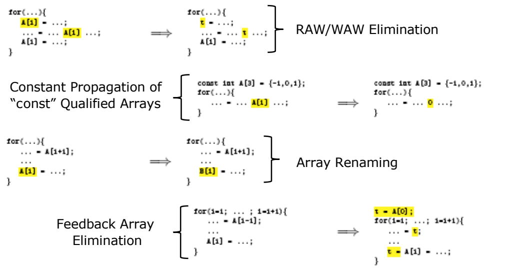
→

```
temp = b*8;
for i=1 to N do
    A[i] = A[i] + temp;
```

Array Transformations

- array_constant_propagate
  - Preceding pass
    - None
- array_rename
  - Preceding pass
    - dataDep_preprocess
- array_raw_eliminate
  - Preceding pass
    - None
- array_scalar_eliminate
  - Preceding pass
    - array_raw_eliminate
- array_feedback_eliminate
  - Parameters: 1-enabled 0-not enabled
  - Preceding pass
    - dataDep_preprocess

Array Access Transformations



RAW/WAW Elimination

```
for(...){
   A[i] = ...;
   ... = ... A[i] ...;
   A[i] = ...;
}
```

⟹

```
for(...){
   t = ...;
   ... = ... t ...;
   A[i] = ...;
}
```

Constant Propagation of "const" Qualified Arrays

```
const int A[3] = {-1,0,1};
for(...){
   ... = ... A[1] ...;
}
```

⟹

```
const int A[3] = {-1,0,1};
for(...){
   ... = ... 0 ...;
}
```

Array Renaming

```
for(...){
   ... = A[i+1];
   ...
   A[i] = ...;
}
```

⟹

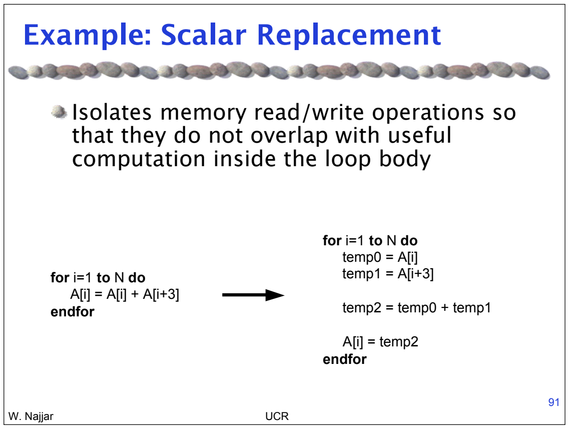
```
for(...){
   ... = A[i+1];
   ...
   B[i] = ...;
}
```

Feedback Array Elimination

```
for(i=1; ... ; i=i+1){
   ... = A[i-1];
   ...
   A[i] = ...;
}
```

⟹

```
t = A[0];
for(i=1; ...; i=i+1){
   ... = t;
   ...
   t = A[i] = ...;
}
```



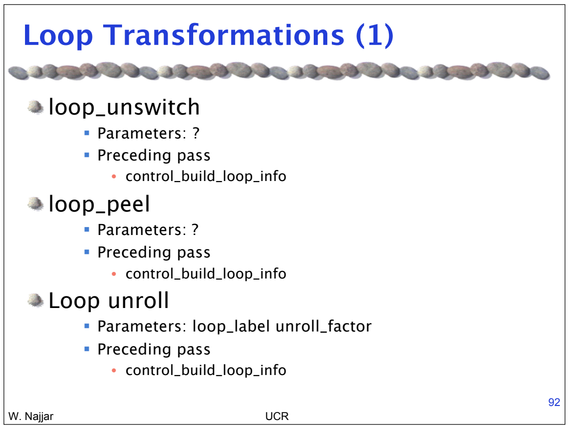
Example: Scalar Replacement

- Isolates memory read/write operations so that they do not overlap with useful computation inside the loop body

```
for i=1 to N do
    A[i] = A[i] + A[i+3]
endfor
```

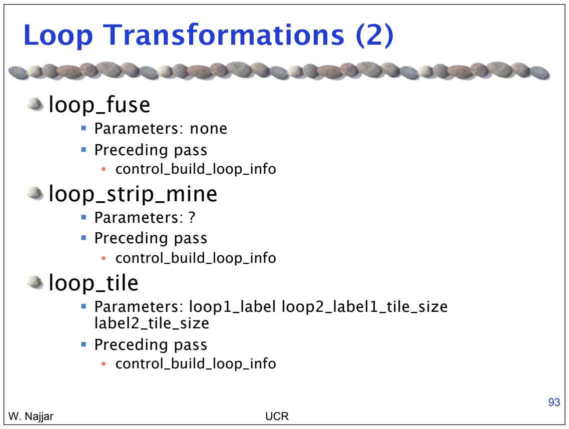
→

```
for i=1 to N do
    temp0 = A[i]
    temp1 = A[i+3]

    temp2 = temp0 + temp1

    A[i] = temp2
endfor
```



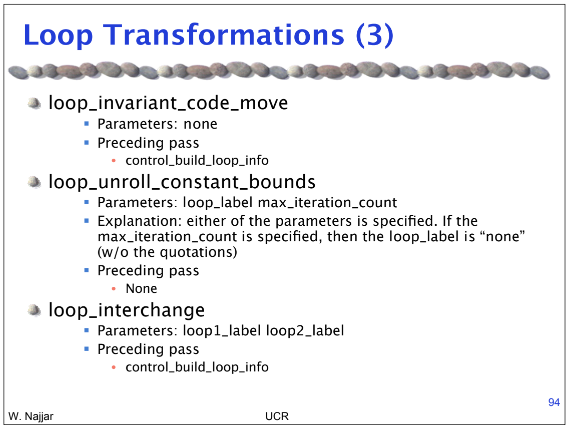
Loop Transformations (1)

- loop_unswitch
  - Parameters: ?
  - Preceding pass
    - control_build_loop_info
- loop_peel
  - Parameters: ?
  - Preceding pass
    - control_build_loop_info
- Loop unroll
  - Parameters: loop_label unroll_factor
  - Preceding pass
    - control_build_loop_info



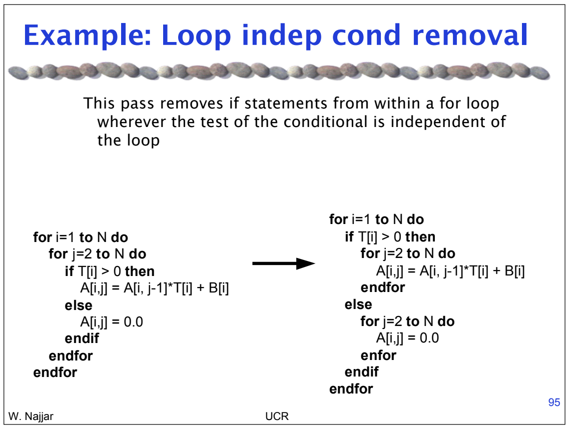
Loop Transformations (2)

- loop_fuse
  - Parameters: none
  - Preceding pass
    - control_build_loop_info
- loop_strip_mine
  - Parameters: ?
  - Preceding pass
    - control_build_loop_info
- loop_tile
  - Parameters: loop1_label loop2_label1_tile_size label2_tile_size
  - Preceding pass
    - control_build_loop_info



Loop Transformations (3)

- loop_invariant_code_move
  - Parameters: none
  - Preceding pass
    - control_build_loop_info
- loop_unroll_constant_bounds
  - Parameters: loop_label max_iteration_count
  - Explanation: either of the parameters is specified. If the max_iteration_count is specified, then the loop_label is “none” (w/o the quotations)
  - Preceding pass
    - None
- loop_interchange
  - Parameters: loop1_label loop2_label
  - Preceding pass
    - control_build_loop_info



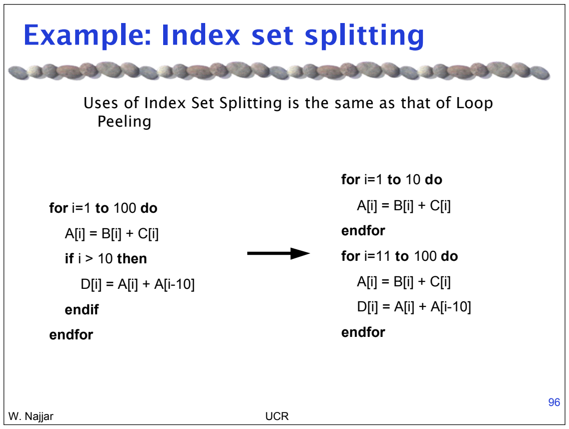
Example: Loop indep cond removal

This pass removes if statements from within a for loop wherever the test of the conditional is independent of the loop

```
for i=1 to N do
   for j=2 to N do
      if T[i] > 0 then
         A[i,j] = A[i, j-1]*T[i] + B[i]
      else
         A[i,j] = 0.0
      endif
   endfor
endfor
```

→

```
for i=1 to N do
   if T[i] > 0 then
      for j=2 to N do
         A[i,j] = A[i, j-1]*T[i] + B[i]
      endfor
   else
      for j=2 to N do
         A[i,j] = 0.0
      enfor
   endif
endfor
```



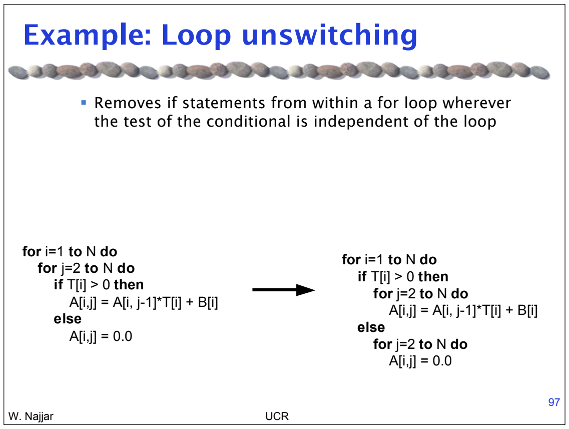
Example: Index set splitting

Uses of Index Set Splitting is the same as that of Loop Peeling

```
for i=1 to 100 do
   A[i] = B[i] + C[i]
   if i > 10 then
      D[i] = A[i] + A[i-10]
   endif
endfor
```

→

```
for i=1 to 10 do
   A[i] = B[i] + C[i]
endfor
for i=11 to 100 do
   A[i] = B[i] + C[i]
   D[i] = A[i] + A[i-10]
endfor
```



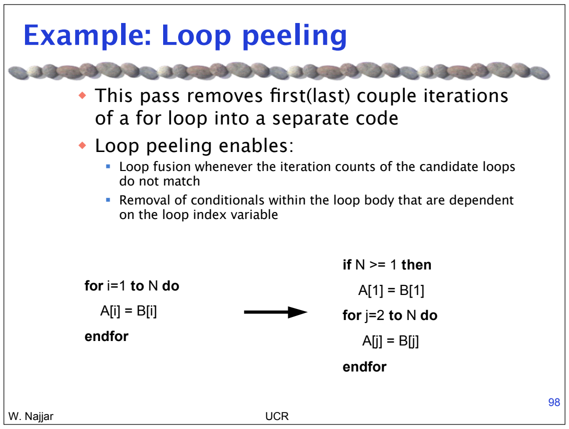
Example: Loop unswitching

- Removes if statements from within a for loop wherever the test of the conditional is independent of the loop

```
for i=1 to N do
  for j=2 to N do
    if T[i] > 0 then
      A[i,j] = A[i, j-1]*T[i] + B[i]
    else
      A[i,j] = 0.0
```

→

```
for i=1 to N do
  if T[i] > 0 then
    for j=2 to N do
      A[i,j] = A[i, j-1]*T[i] + B[i]
  else
    for j=2 to N do
      A[i,j] = 0.0
```



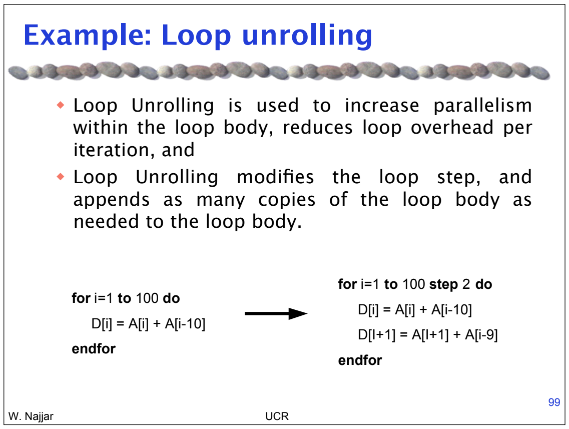
Example: Loop peeling

- This pass removes first(last) couple iterations of a for loop into a separate code
- Loop peeling enables:
  - Loop fusion whenever the iteration counts of the candidate loops do not match
  - Removal of conditionals within the loop body that are dependent on the loop index variable

```
for i=1 to N do
  A[i] = B[i]
endfor
```

→

```
if N >= 1 then
  A[1] = B[1]
for j=2 to N do
  A[j] = B[j]
endfor
```



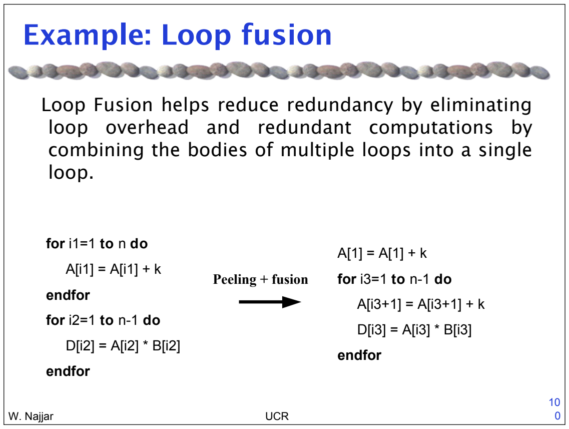
Example: Loop unrolling

- Loop Unrolling is used to increase parallelism within the loop body, reduces loop overhead per iteration, and
- Loop Unrolling modifies the loop step, and appends as many copies of the loop body as needed to the loop body.

```
for i=1 to 100 do
  D[i] = A[i] + A[i-10]
endfor
```

→

```
for i=1 to 100 step 2 do
  D[i] = A[i] + A[i-10]
  D[I+1] = A[I+1] + A[i-9]
endfor
```



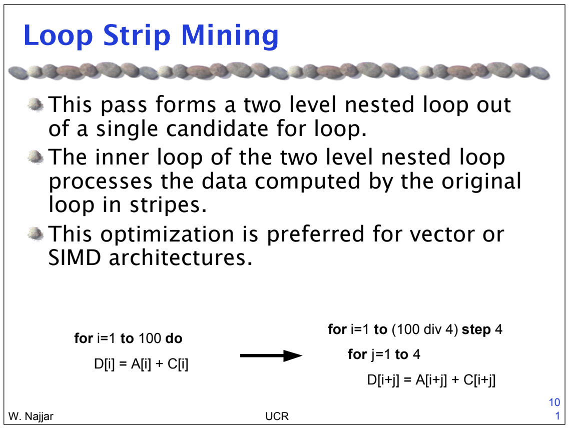
Example: Loop fusion

Loop Fusion helps reduce redundancy by eliminating loop overhead and redundant computations by combining the bodies of multiple loops into a single loop.

```
for i1=1 to n do
    A[i1] = A[i1] + k
endfor
for i2=1 to n-1 do
    D[i2] = A[i2] * B[i2]
endfor
```

Peeling + fusion →

```
A[1] = A[1] + k
for i3=1 to n-1 do
    A[i3+1] = A[i3+1] + k
    D[i3] = A[i3] * B[i3]
endfor
```



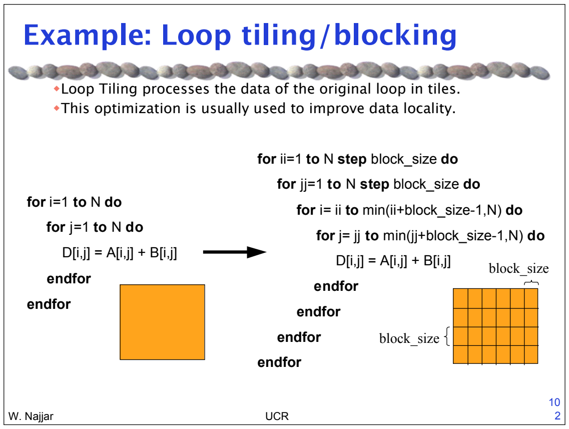
Loop Strip Mining

- This pass forms a two level nested loop out of a single candidate for loop.
- The inner loop of the two level nested loop processes the data computed by the original loop in stripes.
- This optimization is preferred for vector or SIMD architectures.

```
for i=1 to 100 do
    D[i] = A[i] + C[i]
```

→

```
for i=1 to (100 div 4) step 4
    for j=1 to 4
        D[i+j] = A[i+j] + C[i+j]
```



Example: Loop tiling/blocking

- Loop Tiling processes the data of the original loop in tiles.
- This optimization is usually used to improve data locality.

```
for i=1 to N do
    for j=1 to N do
        D[i,j] = A[i,j] + B[i,j]
    endfor
endfor
```

→

```
for ii=1 to N step block_size do
    for jj=1 to N step block_size do
        for i= ii to min(ii+block_size-1,N) do
            for j= jj to min(jj+block_size-1,N) do
                D[i,j] = A[i,j] + B[i,j]
            endfor
        endfor
    endfor
endfor
```

block_size
block_size

Output Generation

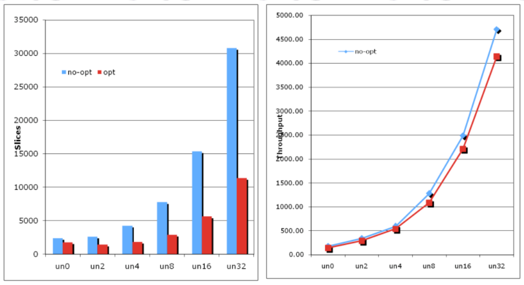
- Several passes help generate the output and the sequence of these passes should stay the same as given in the gen-datapath under the utils directory
- The only exception is for systolic array generation. To enable call:
    - array_feedback_eliminate 1

  To disable:
    - array_feedback_eliminate 0

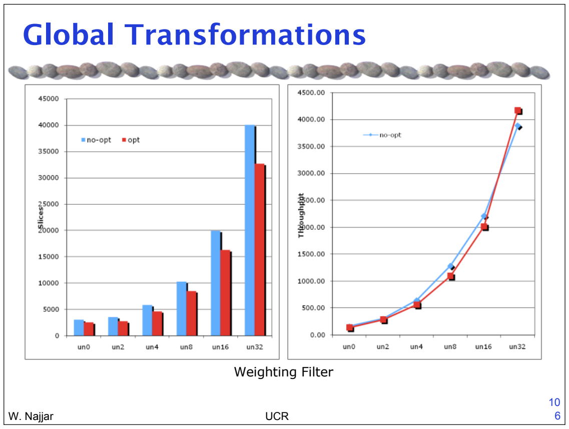
Effects of Transformations

- On area
- Using a set of small and compact cores
    - Limited opportunities, more meaningful results
    - Codes
        - Moving filter
        - Weighing filter
        - 5-tap FIR
        - Bit count
        - Discrete wavelet transform

Global Transformations



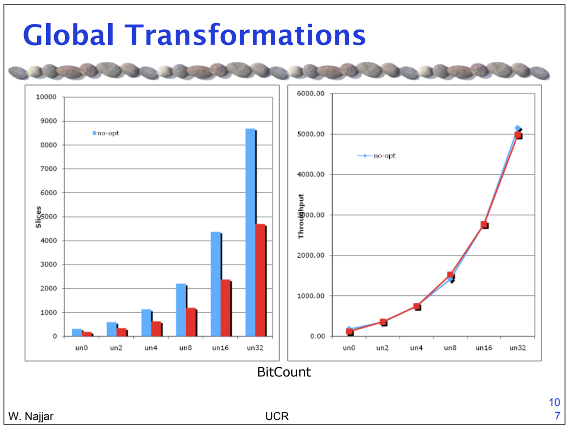
Moving Filter



Global Transformations

Weighting Filter

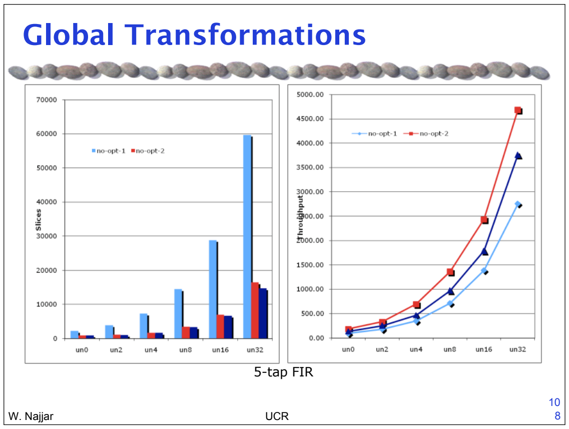
W. Najjar UCR 106



Global Transformations

BitCount

W. Najjar UCR 107

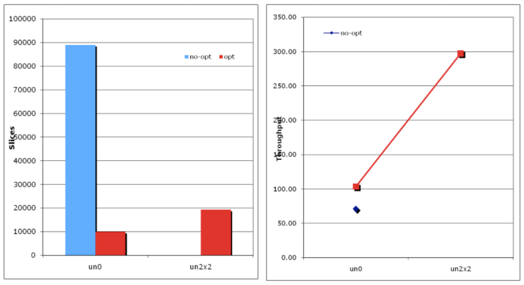


Global Transformations

5-tap FIR

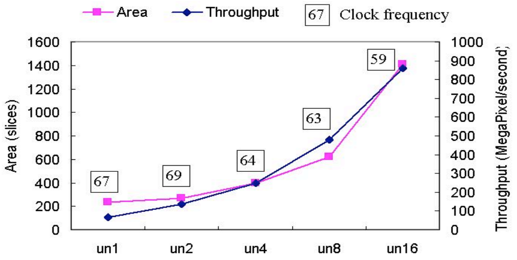
W. Najjar UCR 108

Global Transformations



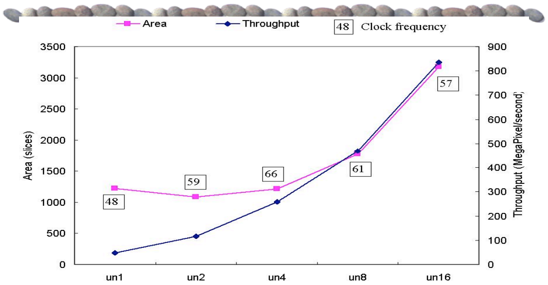
DWT

Loop Transformations – Unrolling

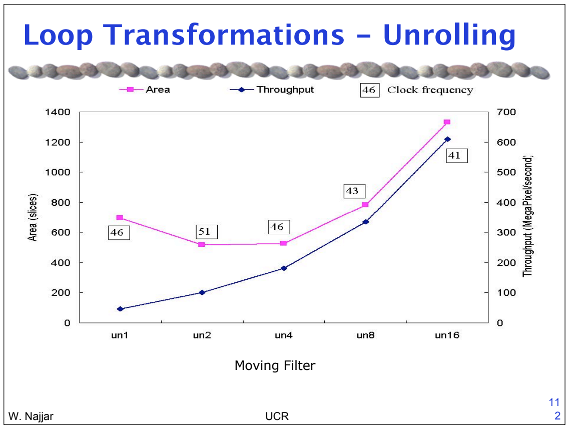


5- tap FIR

Loop Transformations – Unrolling

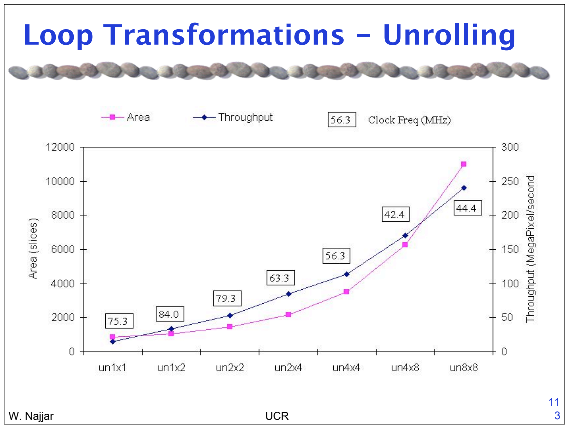


15- tap FIR



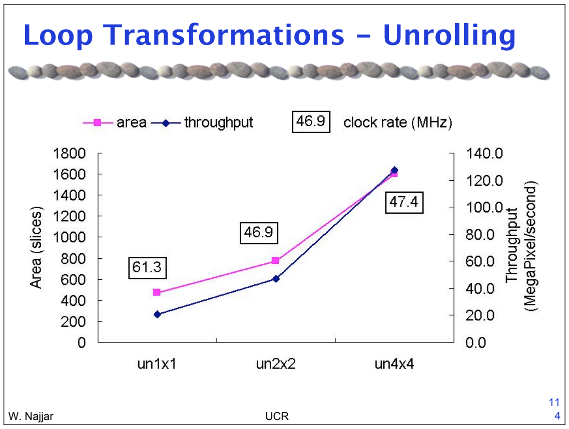
Loop Transformations – Unrolling

W. Najjar UCR 112



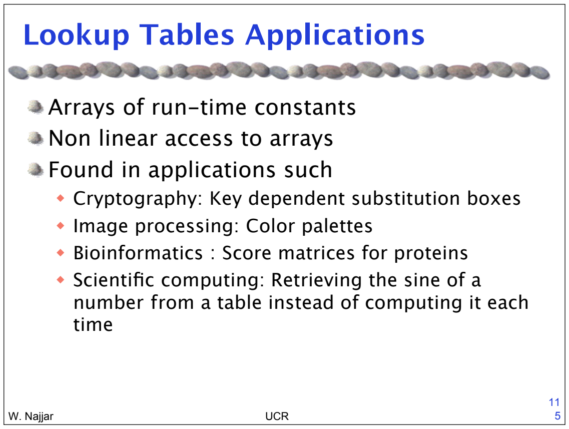
Loop Transformations – Unrolling

W. Najjar UCR 113



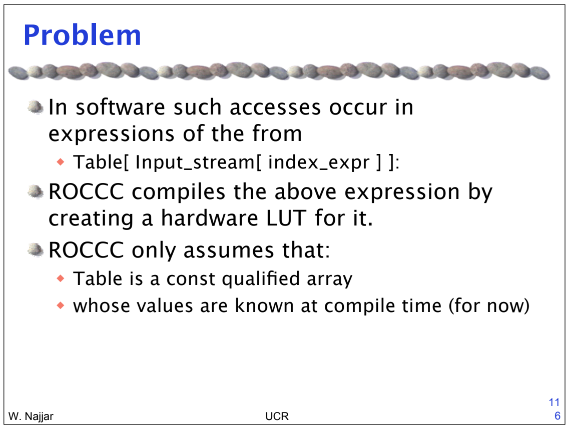
Loop Transformations – Unrolling

W. Najjar UCR 114



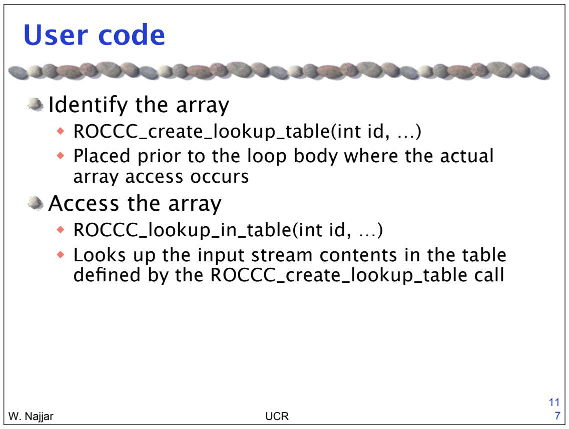
Lookup Tables Applications

- Arrays of run-time constants
- Non linear access to arrays
- Found in applications such
  - Cryptography: Key dependent substitution boxes
  - Image processing: Color palettes
  - Bioinformatics : Score matrices for proteins
  - Scientific computing: Retrieving the sine of a number from a table instead of computing it each time



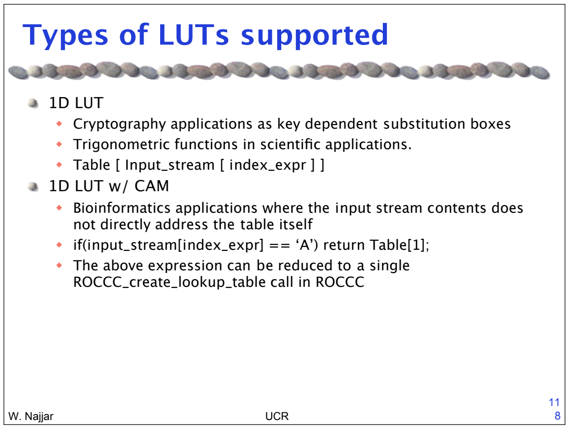
Problem

- In software such accesses occur in expressions of the from
  - Table[Input_stream[index_expr]]:
- ROCCC compiles the above expression by creating a hardware LUT for it.
- ROCCC only assumes that:
  - Table is a const qualified array
  - whose values are known at compile time (for now)



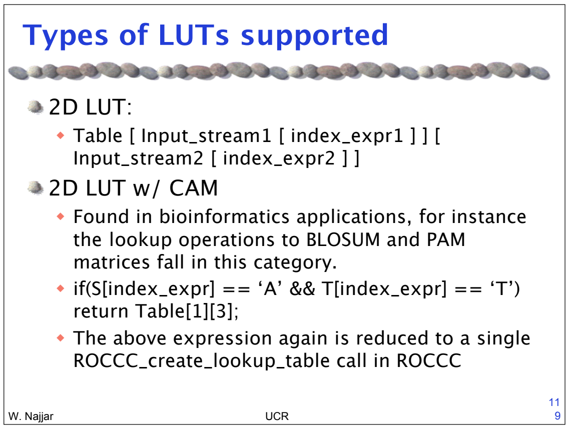
User code

- Identify the array
  - ROCCC_create_lookup_table(int id, …)
  - Placed prior to the loop body where the actual array access occurs
- Access the array
  - ROCCC_lookup_in_table(int id, …)
  - Looks up the input stream contents in the table defined by the ROCCC_create_lookup_table call



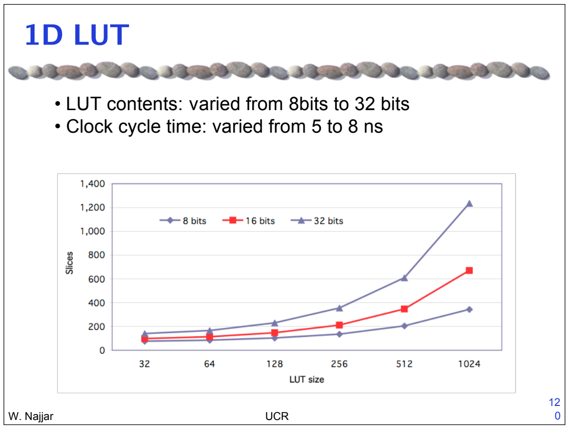
Types of LUTs supported

- 1D LUT
  - Cryptography applications as key dependent substitution boxes
  - Trigonometric functions in scientific applications.
  - Table [Input_stream [index_expr]]
- 1D LUT w/ CAM
  - Bioinformatics applications where the input stream contents does not directly address the table itself
  - if(input_stream[index_expr] == ‘A’) return Table[1];
  - The above expression can be reduced to a single ROCCC_create_lookup_table call in ROCCC



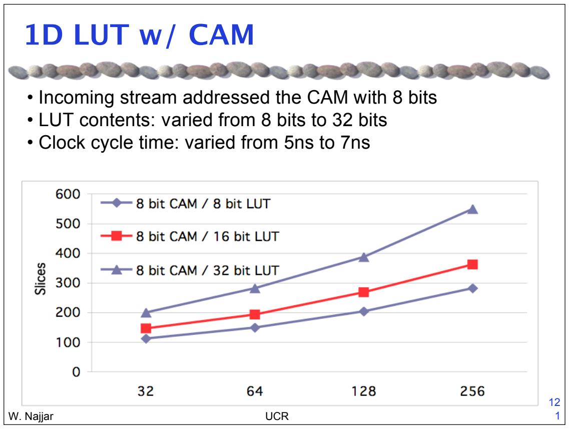
Types of LUTs supported

- 2D LUT:
  - Table [Input_stream1 [index_expr1]] [Input_stream2 [index_expr2]]
- 2D LUT w/ CAM
  - Found in bioinformatics applications, for instance the lookup operations to BLOSUM and PAM matrices fall in this category.
  - if(S[index_expr] == ‘A’ && T[index_expr] == ‘T’) return Table[1][3];
  - The above expression again is reduced to a single ROCCC_create_lookup_table call in ROCCC



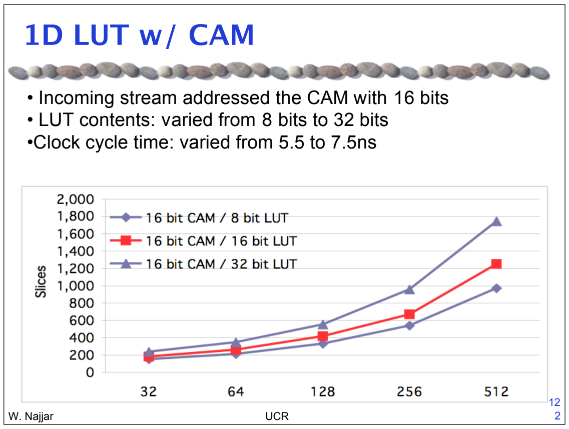
1D LUT

• LUT contents: varied from 8bits to 32 bits
• Clock cycle time: varied from 5 to 8 ns



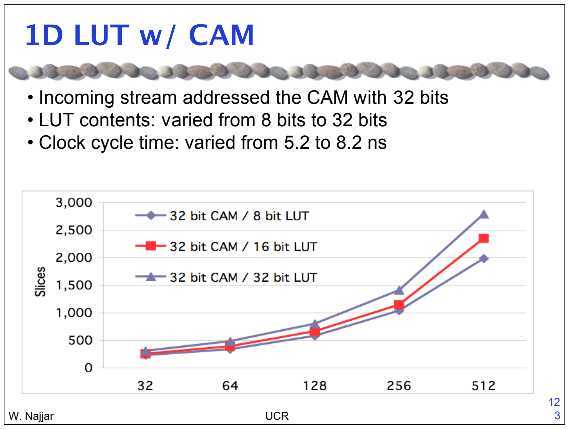
1D LUT w/ CAM

- Incoming stream addressed the CAM with 8 bits
- LUT contents: varied from 8 bits to 32 bits
- Clock cycle time: varied from 5ns to 7ns



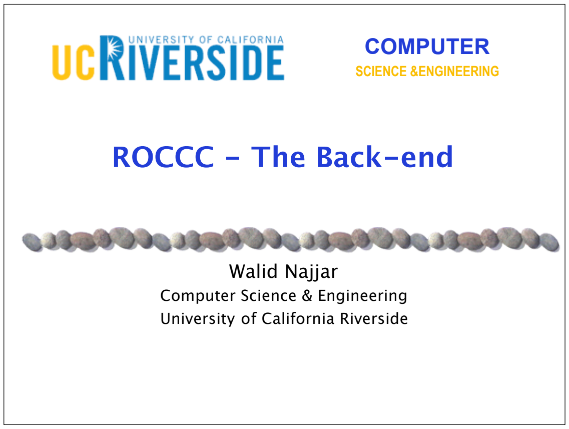
1D LUT w/ CAM

- Incoming stream addressed the CAM with 16 bits
- LUT contents: varied from 8 bits to 32 bits
- •Clock cycle time: varied from 5.5 to 7.5ns



1D LUT w/ CAM

- Incoming stream addressed the CAM with 32 bits
- LUT contents: varied from 8 bits to 32 bits
- Clock cycle time: varied from 5.2 to 8.2 ns

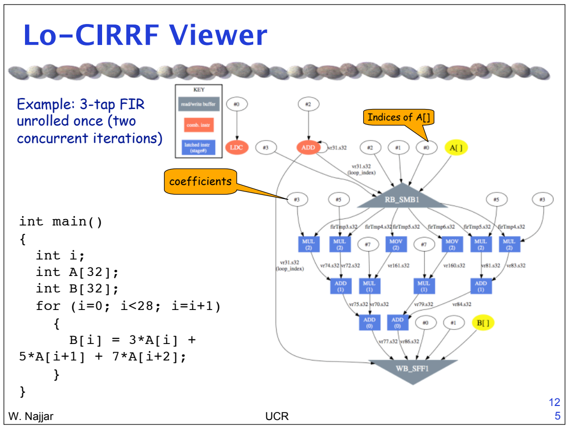


UC RIVERSIDE UNIVERSITY OF CALIFORNIA

COMPUTER
SCIENCE &ENGINEERING

ROCCC – The Back-end

Walid Najjar
Computer Science & Engineering
University of California Riverside



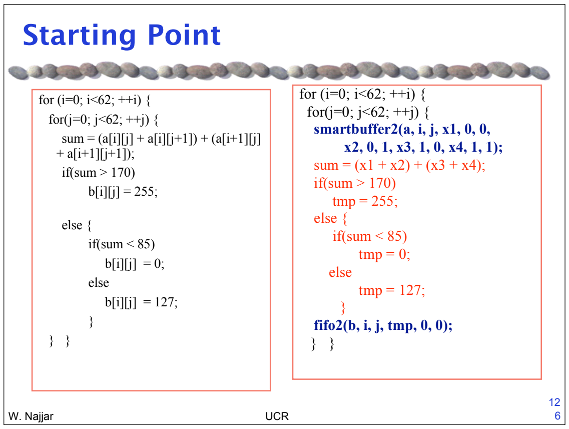
Lo-CIRRF Viewer

Example: 3-tap FIR unrolled once (two concurrent iterations)

```
int main()
{
  int i;
  int A[32];
  int B[32];
  for (i=0; i<28; i=i+1)
    {
      B[i] = 3*A[i] +
5*A[i+1] + 7*A[i+2];
    }
}
```

Indices of A[]

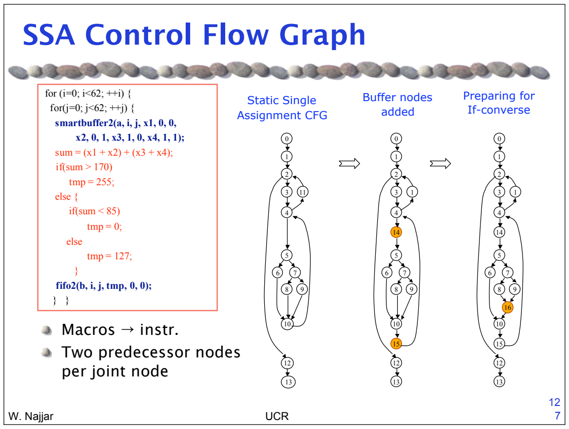
coefficients



Starting Point

```
for (i=0; i<62; ++i) {
  for(j=0; j<62; ++j) {
     sum = (a[i][j] + a[i][j+1]) + (a[i+1][j]
   + a[i+1][j+1]);
     if(sum > 170)
          b[i][j] = 255;

     else {
          if(sum < 85)
              b[i][j]  = 0;
          else
              b[i][j]  = 127;
          }
  }  }
```

```
for (i=0; i<62; ++i) {
  for(j=0; j<62; ++j) {
    smartbuffer2(a, i, j, x1, 0, 0,
           x2, 0, 1, x3, 1, 0, x4, 1, 1);
    sum = (x1 + x2) + (x3 + x4);
    if(sum > 170)
        tmp = 255;
    else {
        if(sum < 85)
             tmp = 0;
       else
             tmp = 127;
         }
    fifo2(b, i, j, tmp, 0, 0);
  }  }
```

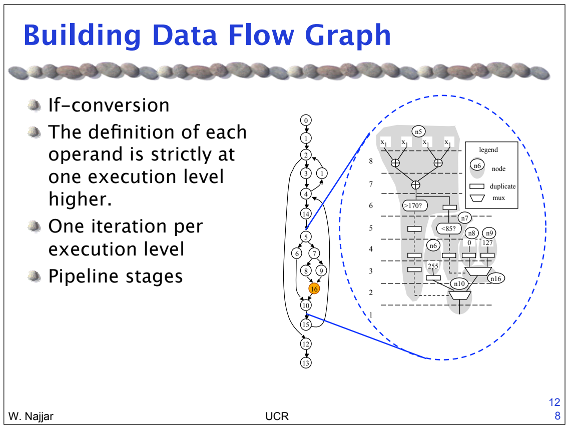


SSA Control Flow Graph

```
for (i=0; i<62; ++i) {
 for(j=0; j<62; ++j) {
  smartbuffer2(a, i, j, x1, 0, 0,
       x2, 0, 1, x3, 1, 0, x4, 1, 1);
  sum = (x1 + x2) + (x3 + x4);
  if(sum > 170)
      tmp = 255;
  else {
      if(sum < 85)
           tmp = 0;
     else
           tmp = 127;
        }
  fifo2(b, i, j, tmp, 0, 0);
 }  }
```

- Macros → instr.
- Two predecessor nodes per joint node

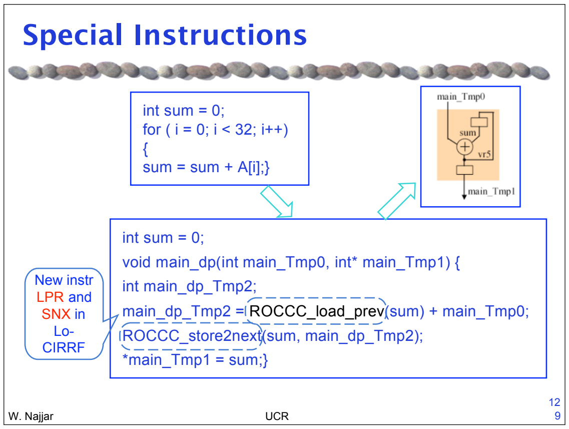
W. Najjar UCR 127



Building Data Flow Graph

- If-conversion
- The definition of each operand is strictly at one execution level higher.
- One iteration per execution level
- Pipeline stages

W. Najjar UCR 128



Special Instructions

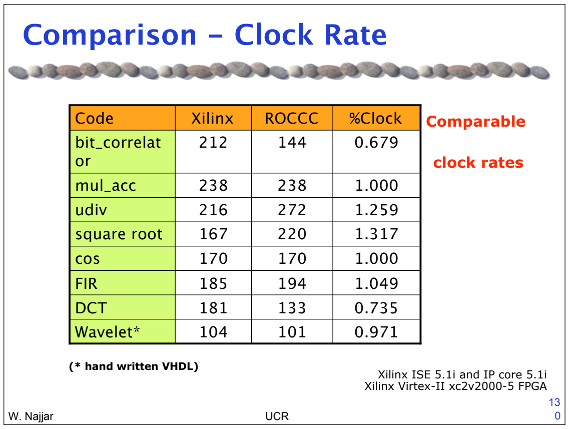
```
int sum = 0;
for ( i = 0; i < 32; i++)
{
sum = sum + A[i];}
```

```
int sum = 0;
void main_dp(int main_Tmp0, int* main_Tmp1) {
int main_dp_Tmp2;
main_dp_Tmp2 = ROCCC_load_prev(sum) + main_Tmp0;
ROCCC_store2next(sum, main_dp_Tmp2);
*main_Tmp1 = sum;}
```

New instr LPR and SNX in Lo-CIRRF

main_Tmp0
sum
vr5
main_Tmp1

W. Najjar UCR 129



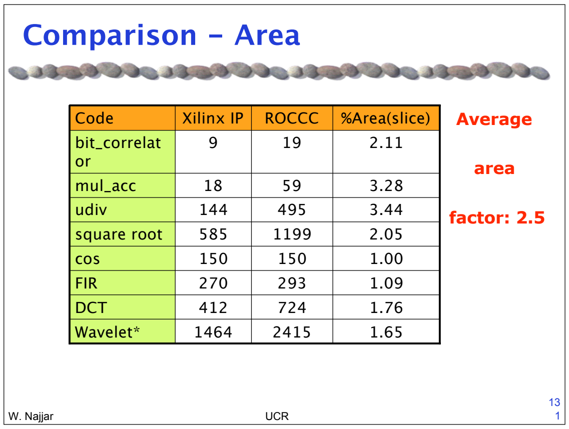
Comparison – Clock Rate

Code	Xilinx	ROCCC	%Clock
bit_correlator	212	144	0.679
mul_acc	238	238	1.000
udiv	216	272	1.259
square root	167	220	1.317
cos	170	170	1.000
FIR	185	194	1.049
DCT	181	133	0.735
Wavelet*	104	101	0.971

Comparable clock rates

(* hand written VHDL)

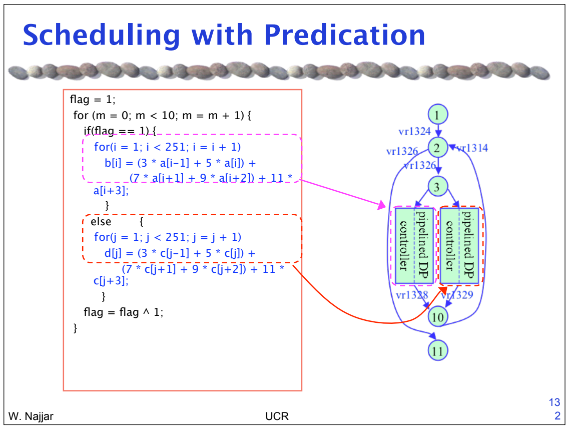
Xilinx ISE 5.1i and IP core 5.1i
Xilinx Virtex-II xc2v2000-5 FPGA



Comparison – Area

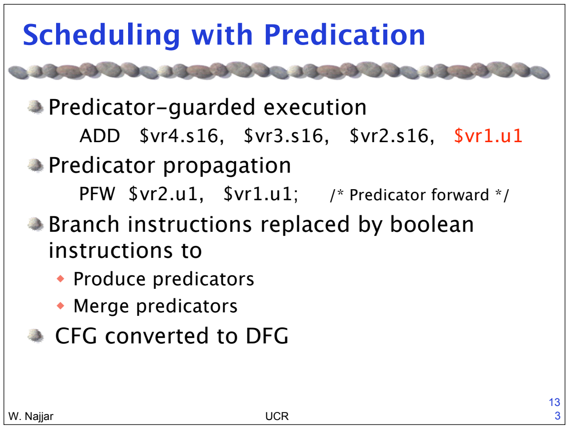
Code	Xilinx IP	ROCCC	%Area(slice)
bit_correlator	9	19	2.11
mul_acc	18	59	3.28
udiv	144	495	3.44
square root	585	1199	2.05
cos	150	150	1.00
FIR	270	293	1.09
DCT	412	724	1.76
Wavelet*	1464	2415	1.65

Average area factor: 2.5



Scheduling with Predication

```
flag = 1;
 for (m = 0; m < 10; m = m + 1) {
   if(flag == 1) {
    for(i = 1; i < 251; i = i + 1)
      b[i] = (3 * a[i-1] + 5 * a[i]) +
             (7 * a[i+1] + 9 * a[i+2]) + 11 *
    a[i+3];
       }
    else       {
    for(j = 1; j < 251; j = j + 1)
      d[j] = (3 * c[j-1] + 5 * c[j]) +
            (7 * c[j+1] + 9 * c[j+2]) + 11 *
    c[j+3];
      }
   flag = flag ^ 1;
 }
```



Scheduling with Predication

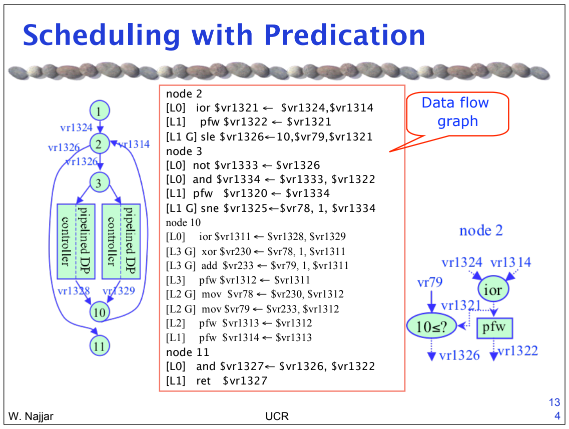
- Predicator-guarded execution

  ADD $vr4.s16, $vr3.s16, $vr2.s16, $vr1.u1

- Predicator propagation

  PFW $vr2.u1, $vr1.u1; /* Predicator forward */

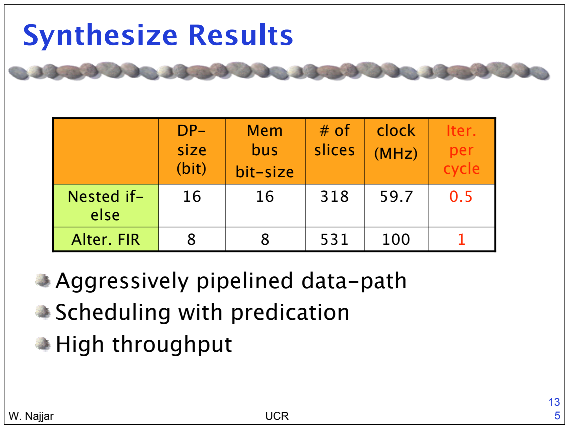
- Branch instructions replaced by boolean instructions to
  - Produce predicators
  - Merge predicators
- CFG converted to DFG



Scheduling with Predication

```
node 2
[L0]   ior $vr1321 ←  $vr1324,$vr1314
[L1]    pfw $vr1322 ← $vr1321
[L1 G] sle $vr1326←10,$vr79,$vr1321
node 3
[L0]  not $vr1333 ← $vr1326
[L0]  and $vr1334 ← $vr1333, $vr1322
[L1]  pfw   $vr1320 ← $vr1334
[L1 G] sne $vr1325←$vr78, 1, $vr1334
node 10
[L0]    ior $vr1311 ← $vr1328, $vr1329
[L3 G]  xor $vr230 ← $vr78, 1, $vr1311
[L3 G]  add  $vr233 ← $vr79, 1, $vr1311
[L3]    pfw $vr1312 ← $vr1311
[L2 G]  mov  $vr78 ← $vr230, $vr1312
[L2 G]  mov $vr79 ← $vr233, $vr1312
[L2]    pfw  $vr1313 ← $vr1312
[L1]    pfw  $vr1314 ← $vr1313
node 11
[L0]   and $vr1327← $vr1326, $vr1322
[L1]   ret   $vr1327
```

Data flow graph



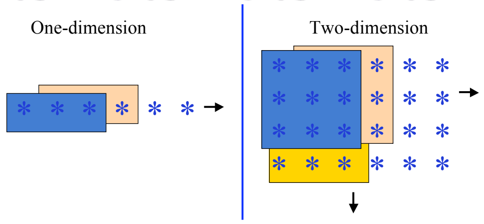
Synthesize Results

	DP-size (bit)	Mem bus bit-size	# of slices	clock (MHz)	Iter. per cycle
Nested if-else	16	16	318	59.7	0.5
Alter. FIR	8	8	531	100	1

- Aggressively pipelined data-path
- Scheduling with predication
- High throughput

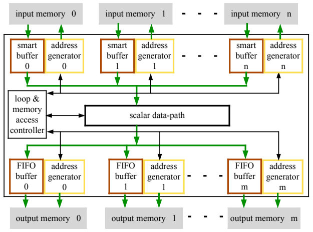
Input data reuse

Input Data Reuse Opportunity

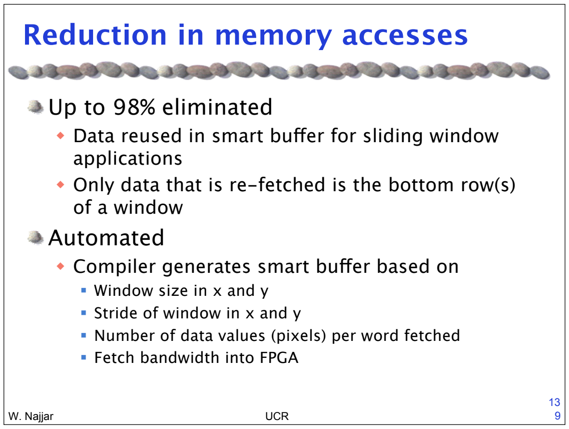


- High memory bandwidth pressure
- Opportunity to perform input data reuse

Smart Buffer Overview

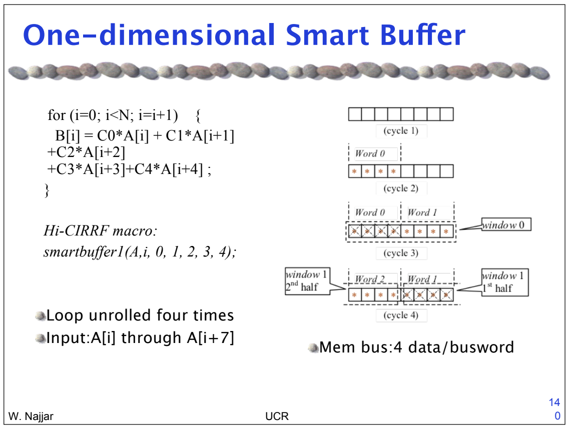


- Input data reuse
- Multiple buffers
  - Synchronization
  - Buffer flash



Reduction in memory accesses

- Up to 98% eliminated
  - Data reused in smart buffer for sliding window applications
  - Only data that is re-fetched is the bottom row(s) of a window
- Automated
  - Compiler generates smart buffer based on
    - Window size in x and y
    - Stride of window in x and y
    - Number of data values (pixels) per word fetched
    - Fetch bandwidth into FPGA

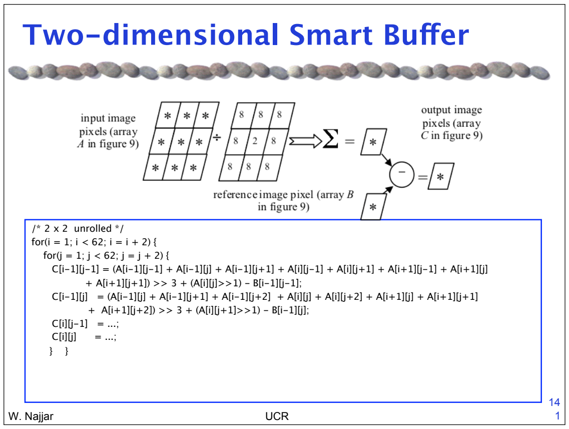


One-dimensional Smart Buffer

```
for (i=0; i<N; i=i+1)    {
  B[i] = C0*A[i] + C1*A[i+1]
+C2*A[i+2]
+C3*A[i+3]+C4*A[i+4] ;
}
```

Hi-CIRRF macro:
smartbuffer1(A,i, 0, 1, 2, 3, 4);

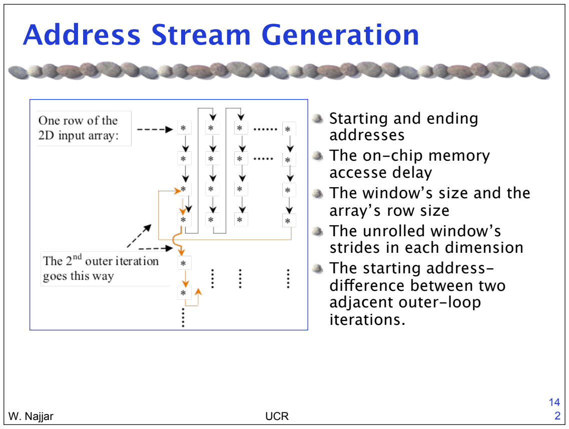
- Loop unrolled four times
- Input:A[i] through A[i+7]
- Mem bus:4 data/busword



Two-dimensional Smart Buffer

input image pixels (array A in figure 9)
* * *
* * *
* * *
÷
8 8 8
8 2 8
8 8 8
Σ =
*
output image pixels (array C in figure 9)
−
= *
reference image pixel (array B in figure 9)
*

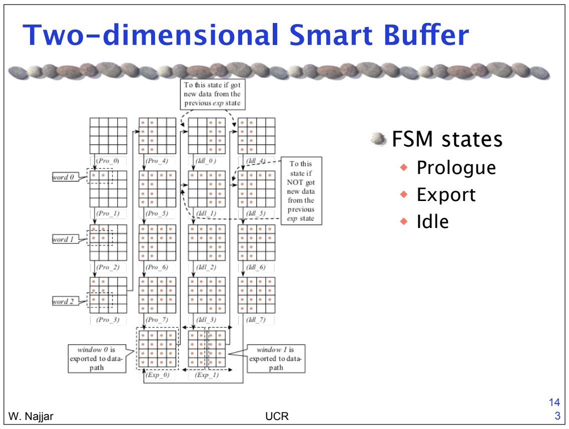
```
/* 2 x 2  unrolled */
for(i = 1; i < 62; i = i + 2) {
   for(j = 1; j < 62; j = j + 2) {
      C[i–1][j–1] = (A[i–1][j–1] + A[i–1][j] + A[i–1][j+1] + A[i][j–1] + A[i][j+1] + A[i+1][j–1] + A[i+1][j]
                 + A[i+1][j+1]) >> 3 + (A[i][j]>>1) – B[i–1][j–1];
      C[i–1][j]   = (A[i–1][j] + A[i–1][j+1] + A[i–1][j+2]  + A[i][j] + A[i][j+2] + A[i+1][j] + A[i+1][j+1]
                  +  A[i+1][j+2]) >> 3 + (A[i][j+1]>>1) – B[i–1][j];
      C[i][j–1]   = ...;
      C[i][j]     = ...;
     }   }
```



Address Stream Generation

One row of the 2D input array:
*
The 2nd outer iteration goes this way

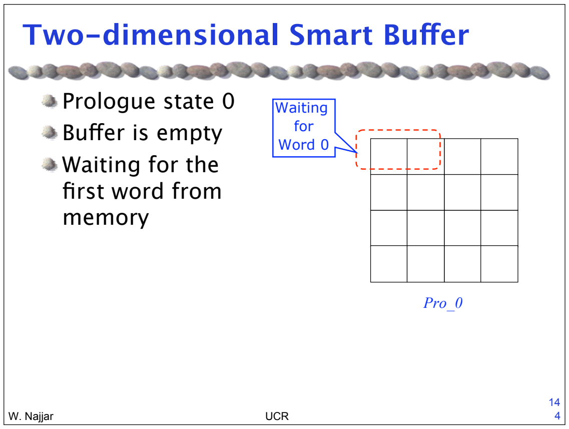
- Starting and ending addresses
- The on-chip memory accesse delay
- The window's size and the array's row size
- The unrolled window's strides in each dimension
- The starting address-difference between two adjacent outer-loop iterations.



Two-dimensional Smart Buffer

To this state if got new data from the previous exp state
(Pro_0)
(Pro_1)
(Pro_2)
(Pro_3)
(Pro_4)
(Pro_5)
(Pro_6)
(Pro_7)
(Idl_0)
(Idl_1)
(Idl_2)
(Idl_3)
(Idl_4)
(Idl_5)
(Idl_6)
(Idl_7)
word 0
word 1
word 2
To this state if NOT got new data from the previous exp state
window 0 is exported to data-path
window 1 is exported to data-path
(Exp_0)
(Exp_1)

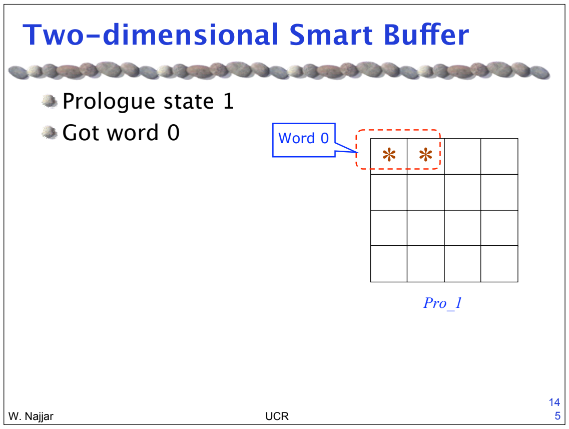
- FSM states
  - Prologue
  - Export
  - Idle



Two-dimensional Smart Buffer

- Prologue state 0
- Buffer is empty
- Waiting for the first word from memory

Waiting for Word 0
Pro_0



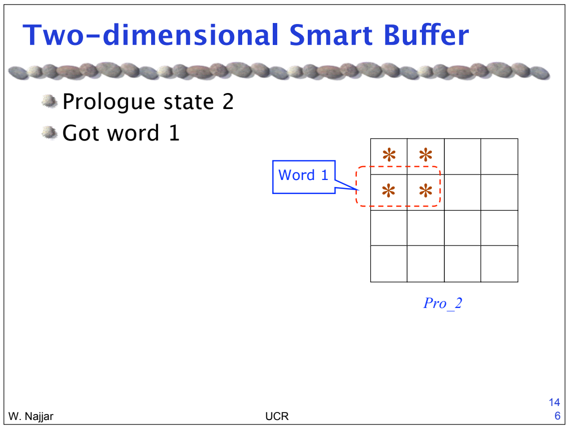
Two-dimensional Smart Buffer

- Prologue state 1
- Got word 0

Word 0

*	*		

Pro_1



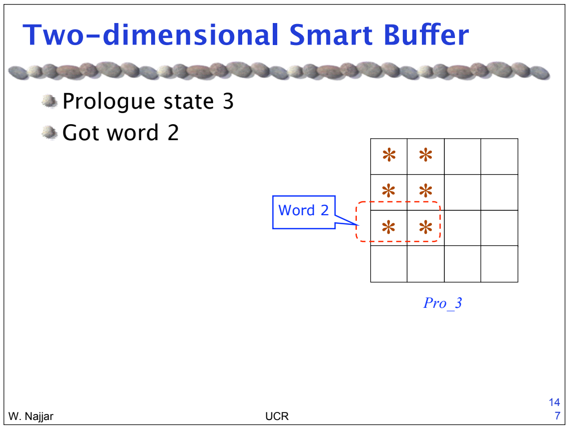
Two-dimensional Smart Buffer

- Prologue state 2
- Got word 1

Word 1

*	*		
*	*		

Pro_2



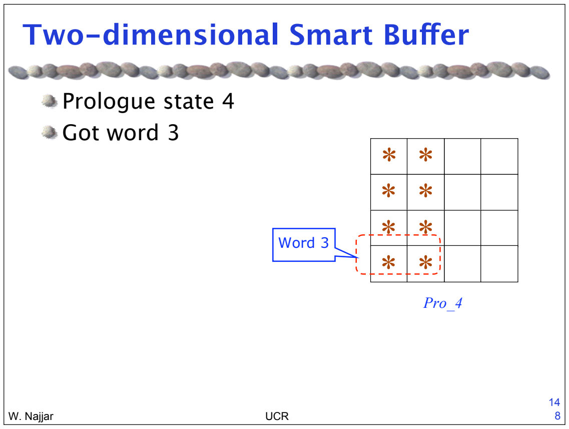
Two-dimensional Smart Buffer

- Prologue state 3
- Got word 2

Word 2

*	*		
*	*		
*	*		

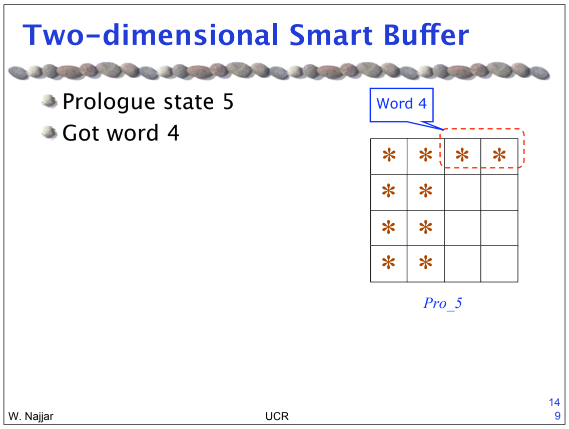
Pro_3



Two-dimensional Smart Buffer

- Prologue state 4
- Got word 3

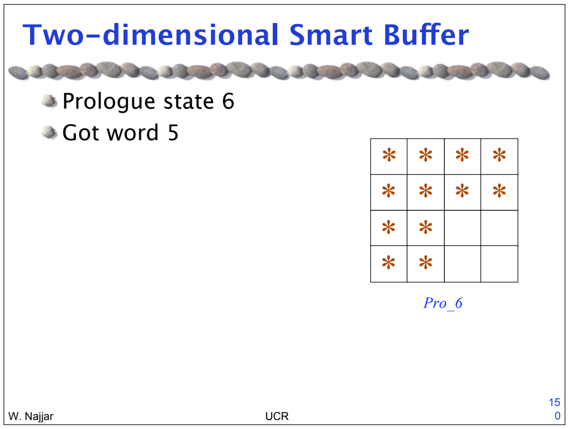
*	*		
*	*		
*	*		
*	*		



Two-dimensional Smart Buffer

- Prologue state 5
- Got word 4

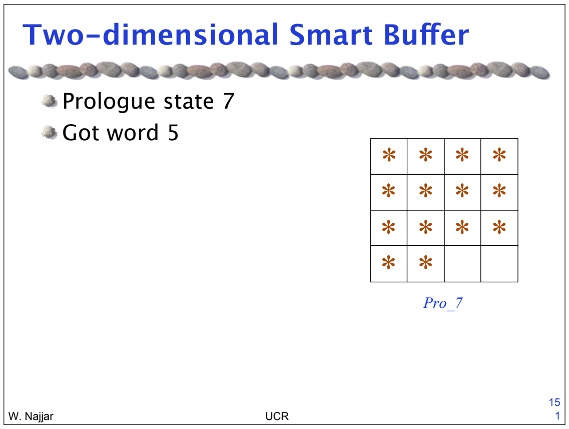
*	*	*	*
*	*		
*	*		
*	*		



Two-dimensional Smart Buffer

- Prologue state 6
- Got word 5

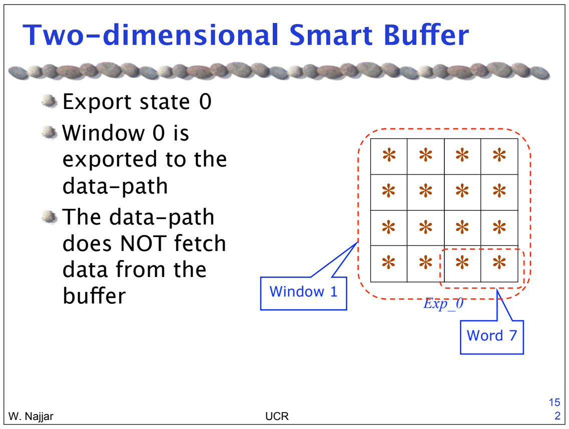
*	*	*	*
*	*	*	*
*	*		
*	*		



Two-dimensional Smart Buffer

- Prologue state 7
- Got word 5

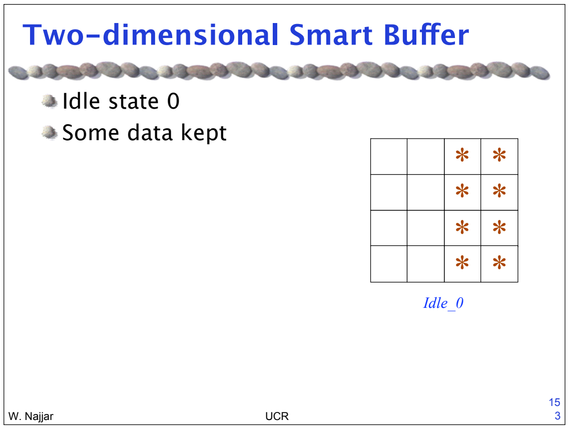
*	*	*	*
*	*	*	*
*	*	*	*
*	*		



Two-dimensional Smart Buffer

- Export state 0
- Window 0 is exported to the data-path
- The data-path does NOT fetch data from the buffer

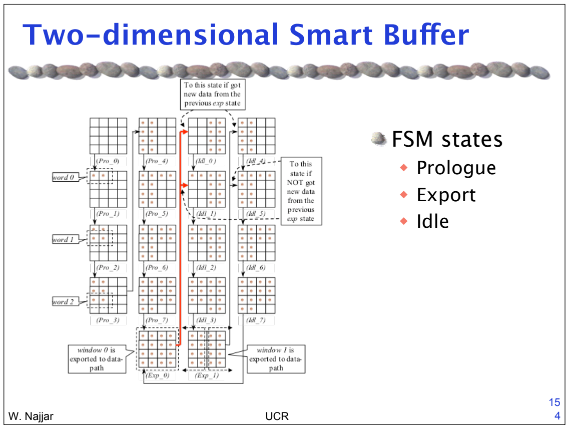
*	*	*	*
*	*	*	*
*	*	*	*
*	*	*	*



Two-dimensional Smart Buffer

- Idle state 0
- Some data kept

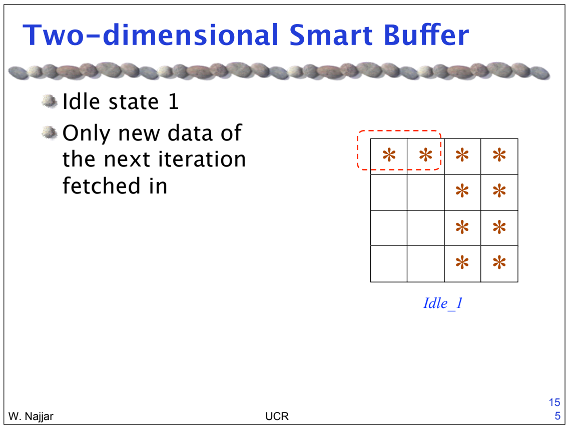
		*	*
		*	*
		*	*
		*	*



Two-dimensional Smart Buffer

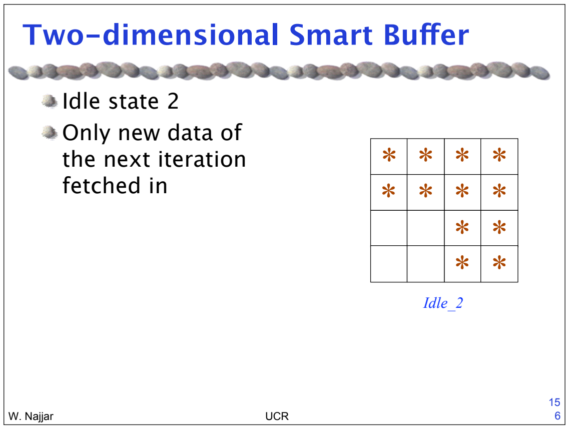
To this state if got new data from the previous *exp* state
(Pro_0)
(Pro_1)
(Pro_2)
(Pro_3)
(Pro_4)
(Pro_5)
(Pro_6)
(Pro_7)
(Idl_0)
(Idl_1)
(Idl_2)
(Idl_3)
(Idl_4)
(Idl_5)
(Idl_6)
(Idl_7)
word 0
word 1
word 2
To this state if NOT got new data from the previous *exp* state
window 0 is exported to data-path
window 1 is exported to data-path
(Exp_0)
(Exp_1)

- FSM states
  - Prologue
  - Export
  - Idle



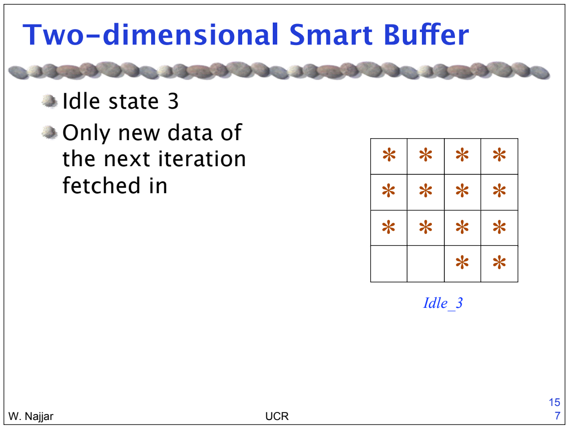
Two-dimensional Smart Buffer

- Idle state 1
- Only new data of the next iteration fetched in



Two-dimensional Smart Buffer

- Idle state 2
- Only new data of the next iteration fetched in

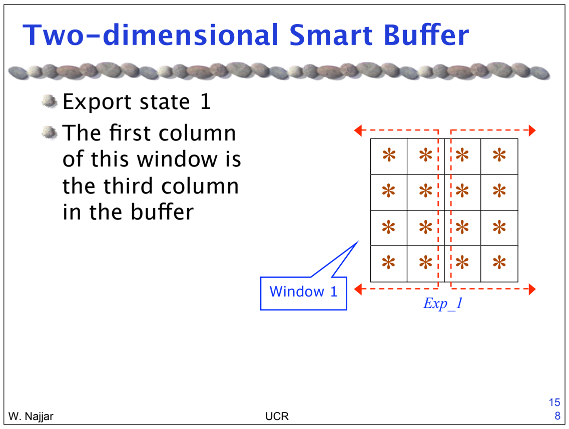


Two-dimensional Smart Buffer

- Idle state 3
- Only new data of the next iteration fetched in

*	*	*	*
*	*	*	*
*	*	*	*
		*	*

Idle_3

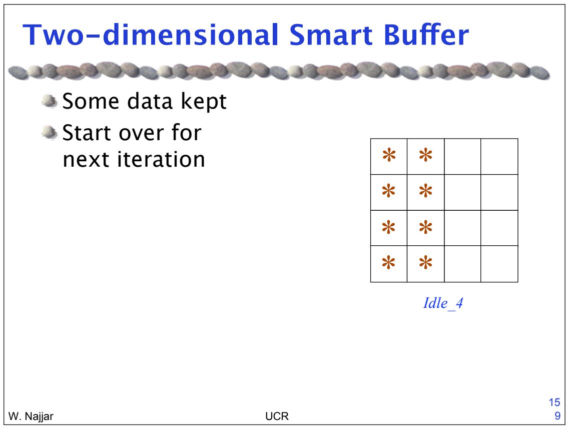


Two-dimensional Smart Buffer

- Export state 1
- The first column of this window is the third column in the buffer

*	*	*	*
*	*	*	*
*	*	*	*
*	*	*	*

Exp_1



Two-dimensional Smart Buffer

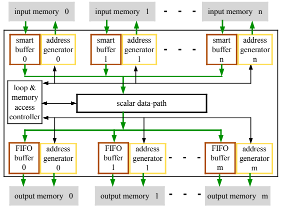
- Some data kept
- Start over for next iteration

*	*		
*	*		
*	*		
*	*		

Idle_4

Other Smart Buffer Features

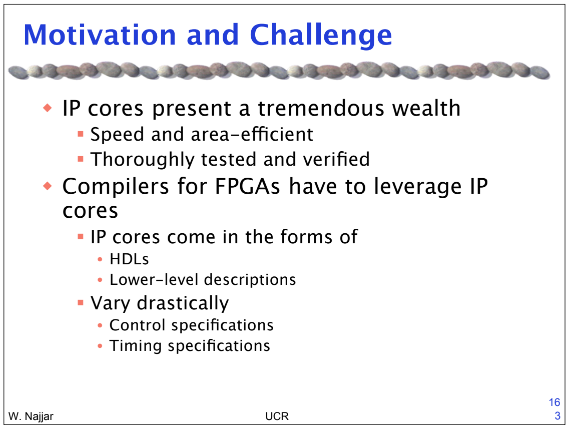
- Multiple input array support
  - Multi-mode and Single-mode
  - Synchronization
  - Buffer flashing
- Input/output balance
  - Extra idle cycles added



Smart Buffer Performance

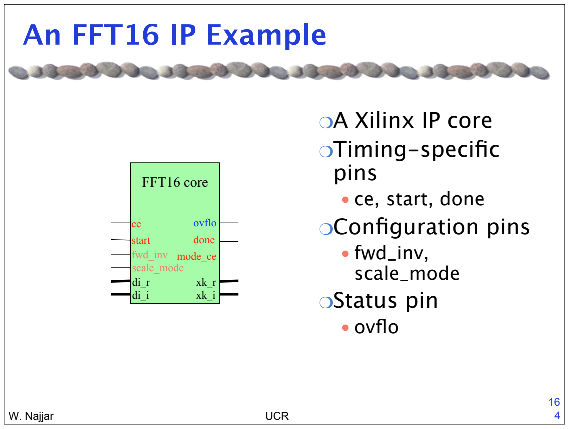
		constant FIR	variable FIR	complex FIR	2D_lowpass filter	motion detection
Input buffer A	Area (slices)	156	159	132	325	327
	# of regs	5	5	6	16	16
	# of states	14	14	8	18	18
	Bus size (bits)	8	8	16	16	16
Input buffer B	Area (slices)		159			150
	# of regs		5			4
	# of states		14			4
	Bus size (bits)		8			16
Output buffer C	Area (slices)	11	11	12	73	73
	# of regs	1	1	2	2	2
	# of states	1	1	2	2	2
	Bus size (bits)	8	8	8	16	16
Data-path	Area (slices)	43	5 mltpl	99	144	164
	Bit size	8	8	8	8	8
Overall area (slices)		210	329	243	542	714
Clock rate (MHz)		94	68	85	69	42
Execution time (cycles)		262	1019	260	5980	5986
Throughput (iteration/cycle)		0.96	0.25	0.48	0.16	0.16

Automation Of IP Core Interface



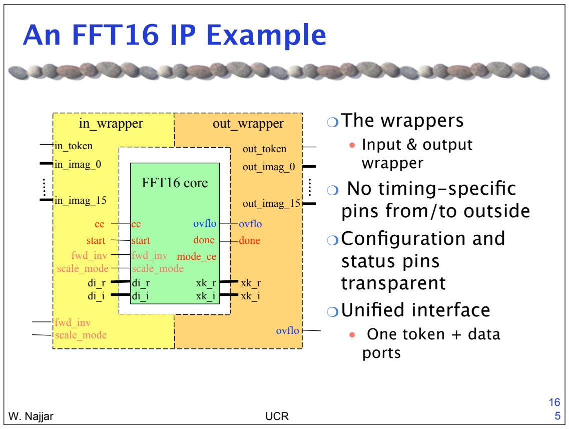
Motivation and Challenge

- IP cores present a tremendous wealth
  - Speed and area-efficient
  - Thoroughly tested and verified
- Compilers for FPGAs have to leverage IP cores
  - IP cores come in the forms of
    - HDLs
    - Lower-level descriptions
  - Vary drastically
    - Control specifications
    - Timing specifications



An FFT16 IP Example

- A Xilinx IP core
- Timing-specific pins
  - ce, start, done
- Configuration pins
  - fwd_inv, scale_mode
- Status pin
  - ovflo



An FFT16 IP Example

- The wrappers
  - Input & output wrapper
- No timing-specific pins from/to outside
- Configuration and status pins transparent
- Unified interface
  - One token + data ports

High-level Abstraction

```
*START = 1;
*CE = 1;
 wait_cycles_for(1);
*START = 0;
 wait_cycles_for(1);
  *DI_R = real_reg_0;
  *DI_I = imag_reg_0;
  ......
  wait_cycles_for(1);
  *DI_R = real_reg_15;
  *DI_I = imag_reg_15;
   wait_cycles_for(69);
    *CE = 0;
```

- Timed C
  - Macros-defined timing
- High-level
  - No cycle-level implementation needed
- A bridge between timing diagrams and HDLs

Wrapper Description

```
void in_fft16  (int in_token,  /*the core's input predicator*/
  int real_0, ... , int real_15,  /*16 real-component inputs*/
  int imag_0, ... , int imag_15,/*16 imaginary-component inputs*/
  int* CE, int* SCALE_MODE, /*pointers are output*/
  int* START, int* FWD_INV, int* DI_R, int* DI_I)
 {
    int real_reg_0, ..., real_reg_15; /*internal registers to*/
    int imag_reg_0, ..., imag_reg_15; /*store the input data*/

    *SCALE_MODE = 1;
    *FWD_INV = 1;

    if(in_token == 1)     {
        wait_cycles_for(1);
        real_reg_0 = real_0;
        ......
        real_reg_15 = real_15;

        imag_reg_0 = imag_0;
       ......
        imag_reg_15 = imag_15;
```

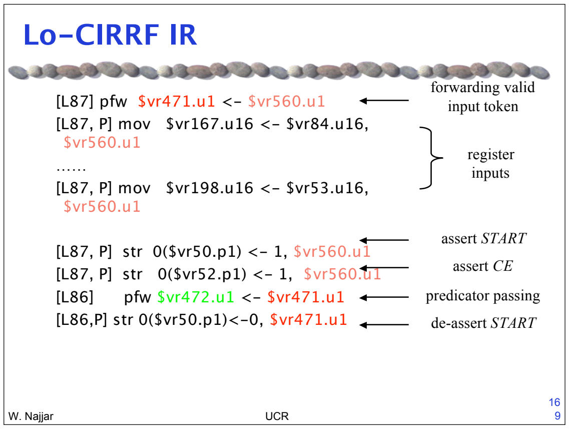
store the 16 pairs of input data into internal registers in this cycle

```
        *START = 1; /*assert start signal in this cycles*/
        *CE = 1; /*assert ce signal in this cycles*/

        wait_cycles_for(1);
        *START = 0; /*de-assert start signal in this cycles*/

         wait_cycles_for(1);
         *DI_R = real_reg_0;
         *DI_I = imag_reg_0;
           ......

         wait_cycles_for(1);
         *DI_R = real_reg_15;
         *DI_I = imag_reg_15;

          wait_cycles_for(69);
          *CE = 0;   /*de-assert ce signal 69 cycles later*/
    }  }
```

export the 16 pairs of data into the core serially in 16 consecutive cycles

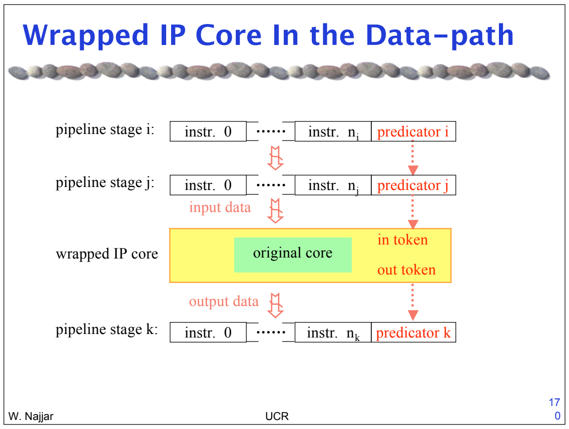
Wrapper Generation

- Macros in C to instructions
  - Wrapper pragma guides the compiler
  - Macros → WCF n   (wait cycles for)
  - CFG → SSA-CFG
  - WCF n → WCF 1 (passing predicator to next pipeline stage)
- CFG to predicated DFG
  - Instructions guarded by predicator
    - except PFW (predicator forward)
  - All instructions between two WCF 1 executed in the same cycle
    - Consist with high-level timing macros
  - WCF 1 → PFW
    - Timing converted to sequential operations

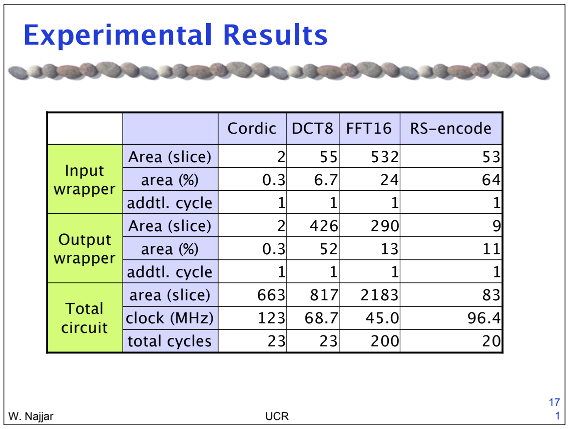


Lo-CIRRF IR

```
[L87] pfw  $vr471.u1 <- $vr560.u1                 forwarding valid
                                                   input token
[L87, P] mov   $vr167.u16 <- $vr84.u16,
 $vr560.u1
......                                             register
                                                   inputs
[L87, P] mov   $vr198.u16 <- $vr53.u16,
 $vr560.u1

                                                   assert START
[L87, P]  str  0($vr50.p1) <- 1, $vr560.u1
[L87, P]  str   0($vr52.p1) <- 1,  $vr560.u1       assert CE
[L86]      pfw $vr472.u1 <- $vr471.u1             predicator passing
[L86,P] str 0($vr50.p1)<-0, $vr471.u1             de-assert START
```

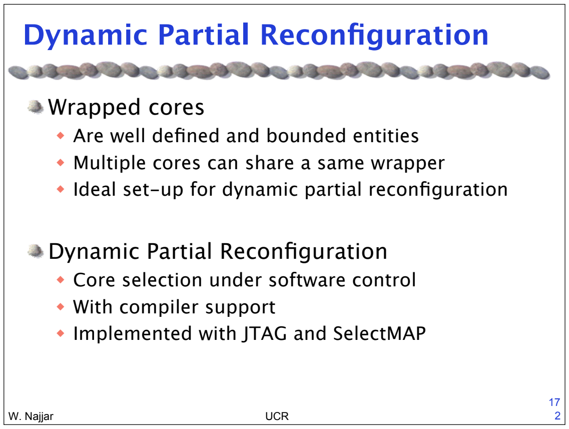


Wrapped IP Core In the Data-path



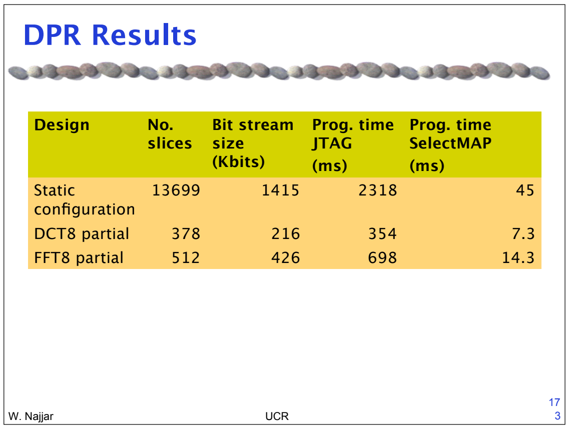
Experimental Results

		Cordic	DCT8	FFT16	RS-encode
Input wrapper	Area (slice)	2	55	532	53
	area (%)	0.3	6.7	24	64
	addtl. cycle	1	1	1	1
Output wrapper	Area (slice)	2	426	290	9
	area (%)	0.3	52	13	11
	addtl. cycle	1	1	1	1
Total circuit	area (slice)	663	817	2183	83
	clock (MHz)	123	68.7	45.0	96.4
	total cycles	23	23	200	20



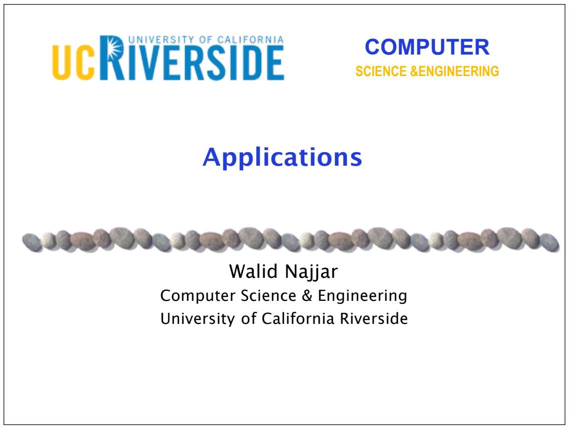
Dynamic Partial Reconfiguration

- Wrapped cores
  - Are well defined and bounded entities
  - Multiple cores can share a same wrapper
  - Ideal set-up for dynamic partial reconfiguration
- Dynamic Partial Reconfiguration
  - Core selection under software control
  - With compiler support
  - Implemented with JTAG and SelectMAP



DPR Results

Design	No. slices	Bit stream size (Kbits)	Prog. time JTAG (ms)	Prog. time SelectMAP (ms)
Static configuration	13699	1415	2318	45
DCT8 partial	378	216	354	7.3
FFT8 partial	512	426	698	14.3

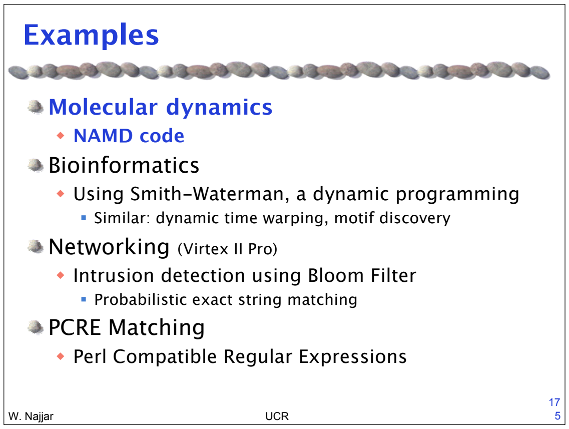


UC RIVERSIDE UNIVERSITY OF CALIFORNIA

COMPUTER SCIENCE &ENGINEERING

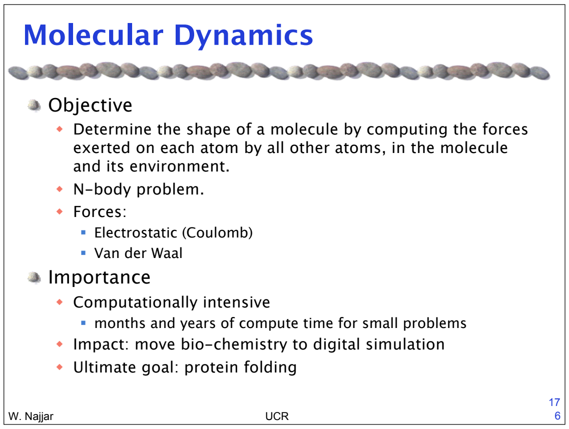
Applications

Walid Najjar
Computer Science & Engineering
University of California Riverside



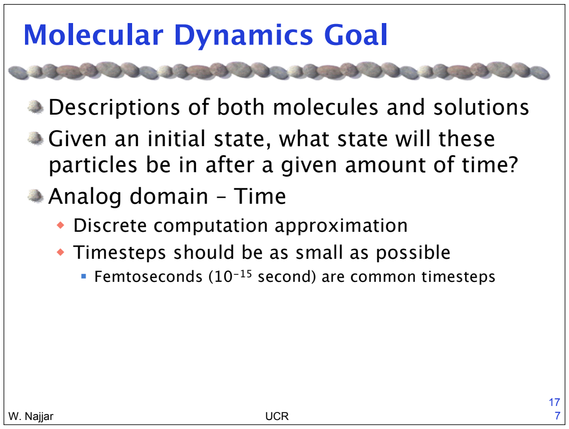
Examples

- Molecular dynamics
  - NAMD code
- Bioinformatics
  - Using Smith-Waterman, a dynamic programming
    - Similar: dynamic time warping, motif discovery
- Networking (Virtex II Pro)
  - Intrusion detection using Bloom Filter
    - Probabilistic exact string matching
- PCRE Matching
  - Perl Compatible Regular Expressions



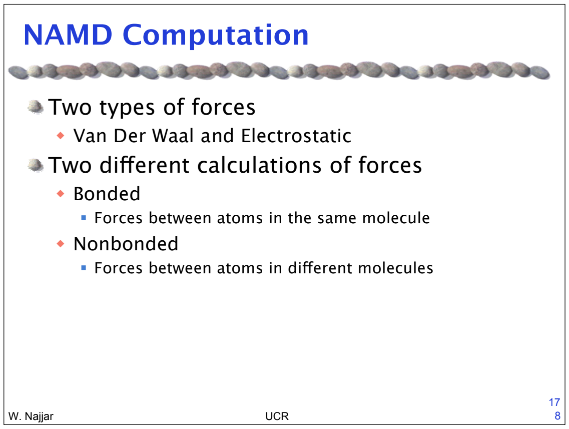
Molecular Dynamics

- Objective
  - Determine the shape of a molecule by computing the forces exerted on each atom by all other atoms, in the molecule and its environment.
  - N-body problem.
  - Forces:
    - Electrostatic (Coulomb)
    - Van der Waal
- Importance
  - Computationally intensive
    - months and years of compute time for small problems
  - Impact: move bio-chemistry to digital simulation
  - Ultimate goal: protein folding



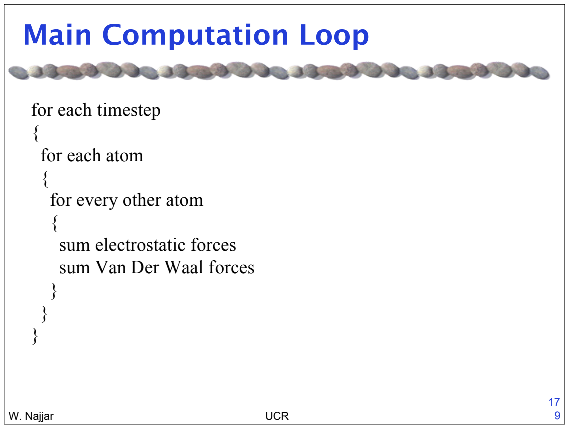
Molecular Dynamics Goal

- Descriptions of both molecules and solutions
- Given an initial state, what state will these particles be in after a given amount of time?
- Analog domain - Time
  - Discrete computation approximation
  - Timesteps should be as small as possible
    - Femtoseconds (10^{-15} second) are common timesteps



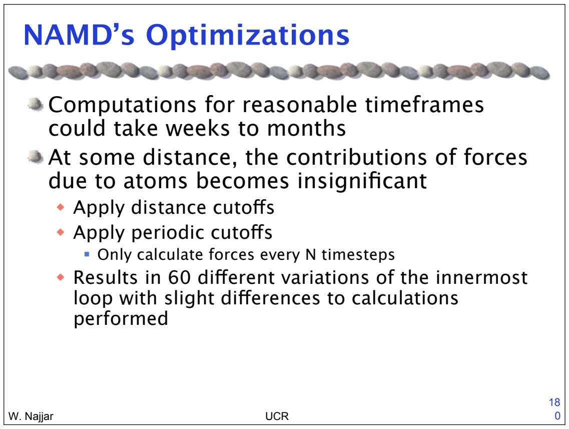
NAMD Computation

- Two types of forces
  - Van Der Waal and Electrostatic
- Two different calculations of forces
  - Bonded
    - Forces between atoms in the same molecule
  - Nonbonded
    - Forces between atoms in different molecules



Main Computation Loop

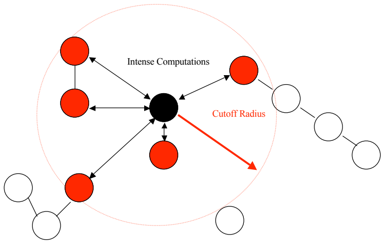
```
for each timestep
{
 for each atom
 {
  for every other atom
  {
   sum electrostatic forces
   sum Van Der Waal forces
  }
 }
}
```



NAMD's Optimizations

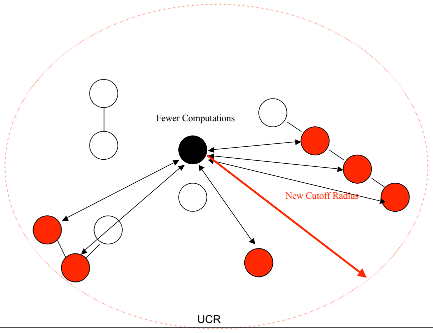
- Computations for reasonable timeframes could take weeks to months
- At some distance, the contributions of forces due to atoms becomes insignificant
  - Apply distance cutoffs
  - Apply periodic cutoffs
    - Only calculate forces every N timesteps
  - Results in 60 different variations of the innermost loop with slight differences to calculations performed

NAMD Execution



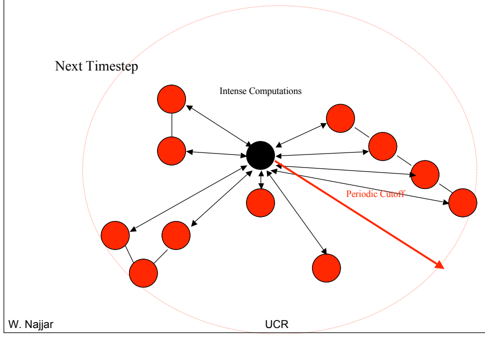
W. Najjar UCR 181

NAMD Execution

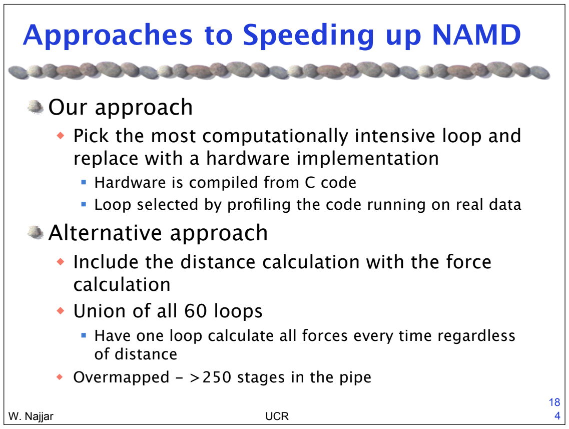


W. Najjar UCR 182

NAMD Execution

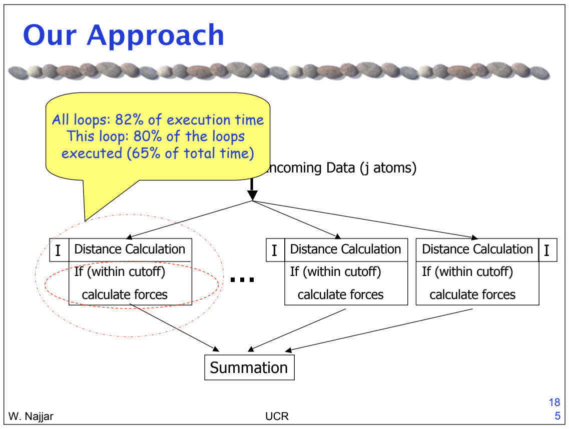


W. Najjar UCR 183



Approaches to Speeding up NAMD

- Our approach
  - Pick the most computationally intensive loop and replace with a hardware implementation
    - Hardware is compiled from C code
    - Loop selected by profiling the code running on real data
- Alternative approach
  - Include the distance calculation with the force calculation
  - Union of all 60 loops
    - Have one loop calculate all forces every time regardless of distance
  - Overmapped – >250 stages in the pipe



Our Approach

All loops: 82% of execution time
This loop: 80% of the loops executed (65% of total time)

ncoming Data (j atoms)

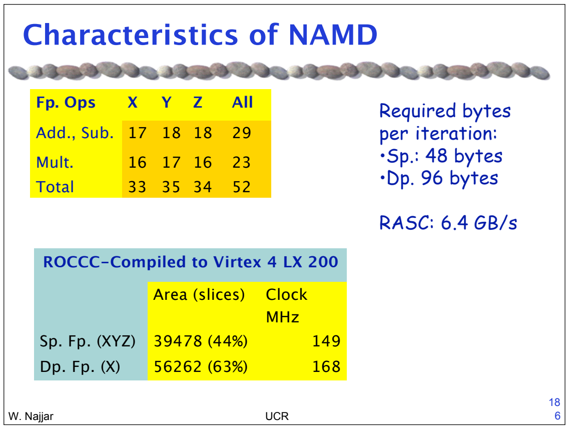
I | Distance Calculation | If (within cutoff) | calculate forces

…

I | Distance Calculation | If (within cutoff) | calculate forces

Distance Calculation | If (within cutoff) | calculate forces | I

Summation



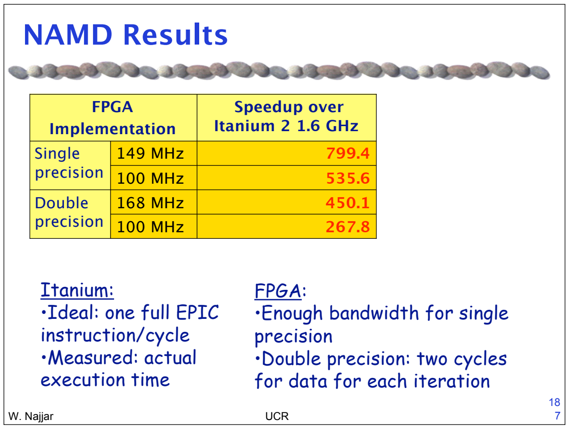
Characteristics of NAMD

Fp. Ops	X	Y	Z	All
Add., Sub.	17	18	18	29
Mult.	16	17	16	23
Total	33	35	34	52

Required bytes per iteration:
•Sp.: 48 bytes
•Dp. 96 bytes

RASC: 6.4 GB/s

ROCCC-Compiled to Virtex 4 LX 200		
	Area (slices)	Clock MHz
Sp. Fp. (XYZ)	39478 (44%)	149
Dp. Fp. (X)	56262 (63%)	168

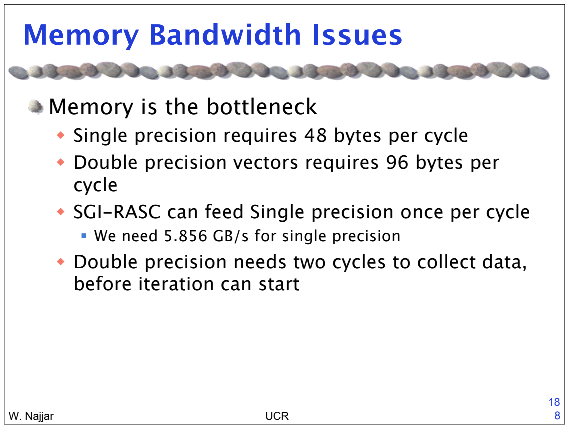


NAMD Results

FPGA Implementation		Speedup over Itanium 2 1.6 GHz
Single precision	149 MHz	799.4
	100 MHz	535.6
Double precision	168 MHz	450.1
	100 MHz	267.8

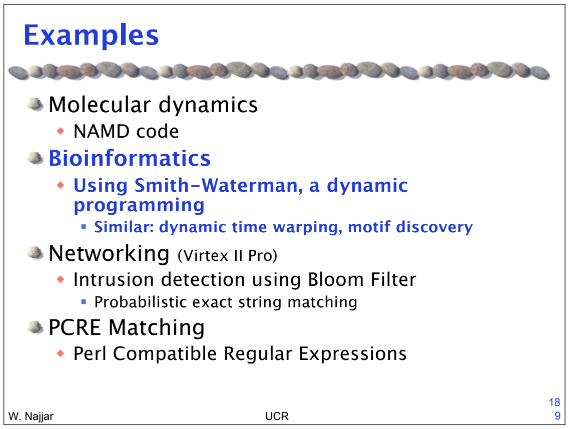
Itanium:
•Ideal: one full EPIC instruction/cycle
•Measured: actual execution time

FPGA:
•Enough bandwidth for single precision
•Double precision: two cycles for data for each iteration



Memory Bandwidth Issues

- Memory is the bottleneck
  - Single precision requires 48 bytes per cycle
  - Double precision vectors requires 96 bytes per cycle
  - SGI-RASC can feed Single precision once per cycle
    - We need 5.856 GB/s for single precision
  - Double precision needs two cycles to collect data, before iteration can start



Examples

- Molecular dynamics
  - NAMD code
- **Bioinformatics**
  - **Using Smith-Waterman, a dynamic programming**
    - **Similar: dynamic time warping, motif discovery**
- Networking (Virtex II Pro)
  - Intrusion detection using Bloom Filter
    - Probabilistic exact string matching
- PCRE Matching
  - Perl Compatible Regular Expressions

Smith Waterman Algorithm

- Dynamic programming string matching algorithm used widely in genetics related research.
- Computes a matching score of two input strings S and T using a 2D matrix.
- Computation of each cell depends on the computed values of three neighboring cells: north, west and northwest.

		S_i
	a	b
T_j	c	d

$$d = \min \begin{cases} \begin{cases} a & \text{if } S_i == T_j \\ a + substitution\_cost & \text{if } S_i \;!{=} T_j \end{cases} \\ b + insertion\_cost \\ c + deletion\_cost \end{cases}$$

Smith-Waterman Code

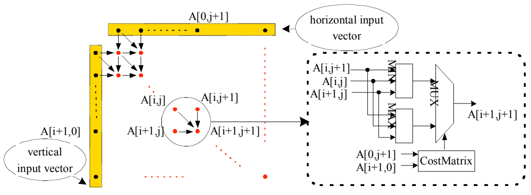
- Dynamic Programming
  - Used in protein modeling, bio-informatics, data mining …
  - A wave-front algorithm with two input strings

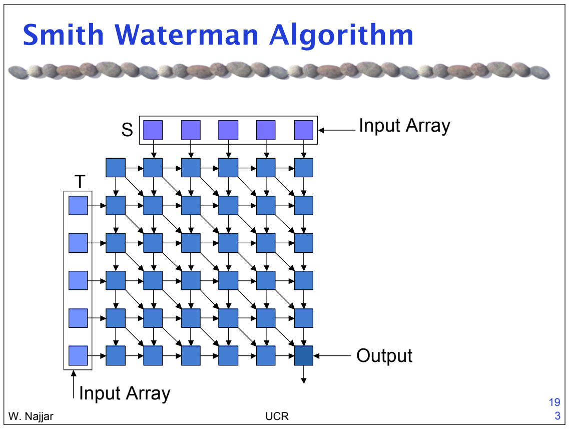
$$A[i,j] = F(A[i,j-1], A[i-1, j-1], A[i-1, j])$$

$$F = CostMatrix(A[i,0],A[0,j])$$

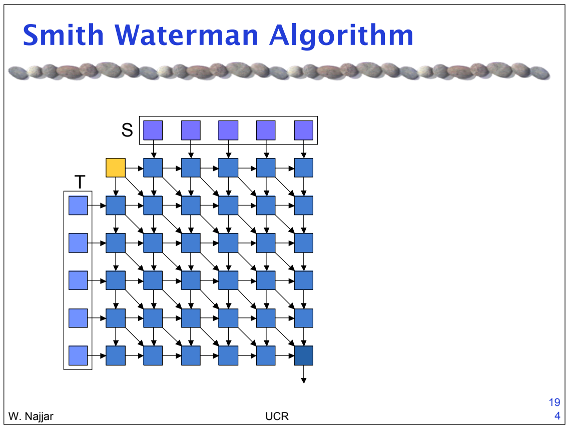
- Our Approach
  - “Chunk” the input strings in fixed sizes *k*
  - Build a *k x k* template hardware by compiling two nested loops (k each) and fully unrolling both.
  - Host strip mines the two outer loops over this template.

S-W View

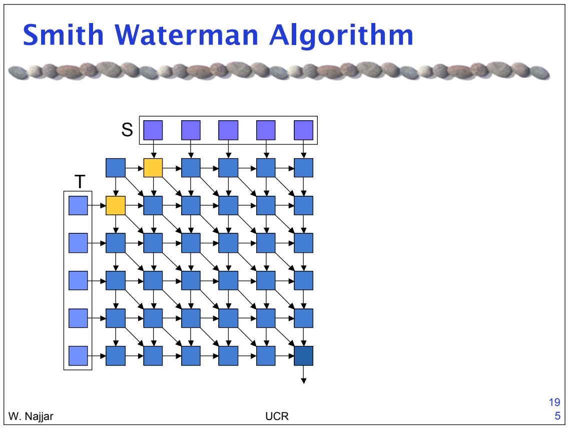




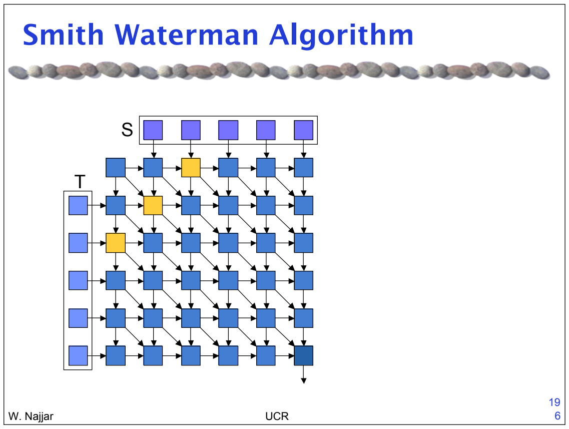
Smith Waterman Algorithm



Smith Waterman Algorithm

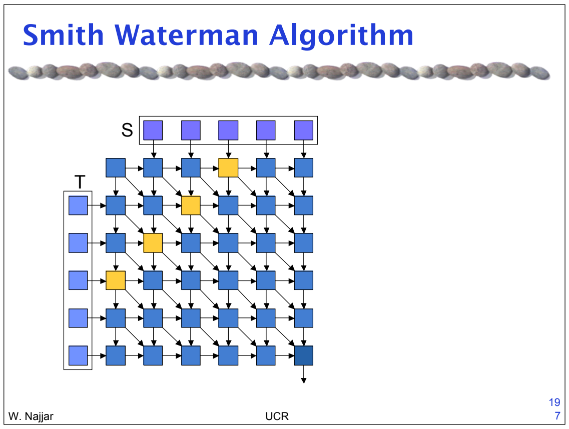


Smith Waterman Algorithm



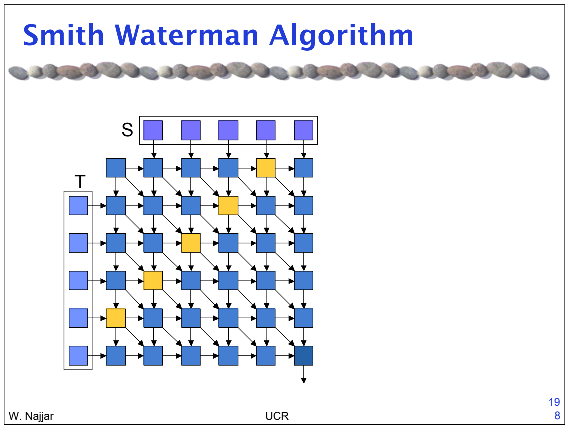
Smith Waterman Algorithm

W. Najjar UCR 19 6



Smith Waterman Algorithm

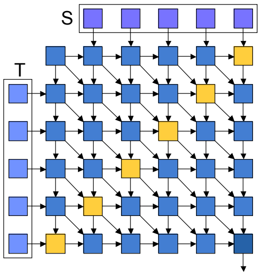
W. Najjar UCR 19 7



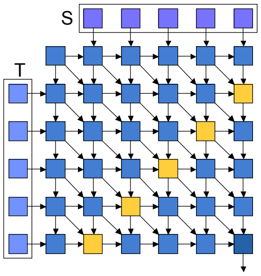
Smith Waterman Algorithm

W. Najjar UCR 19 8

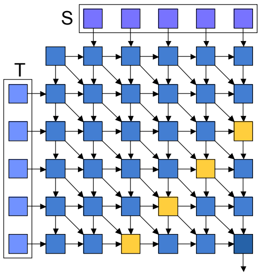
Smith Waterman Algorithm

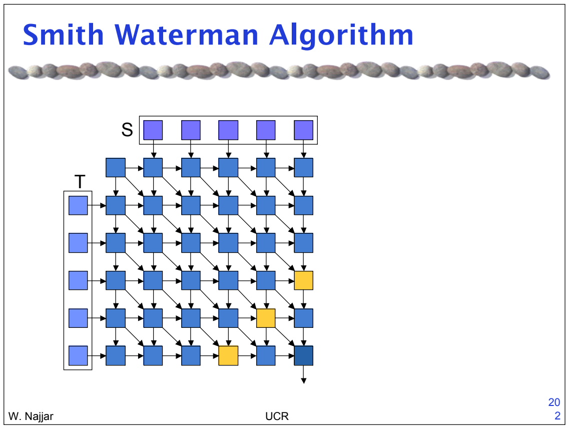


Smith Waterman Algorithm



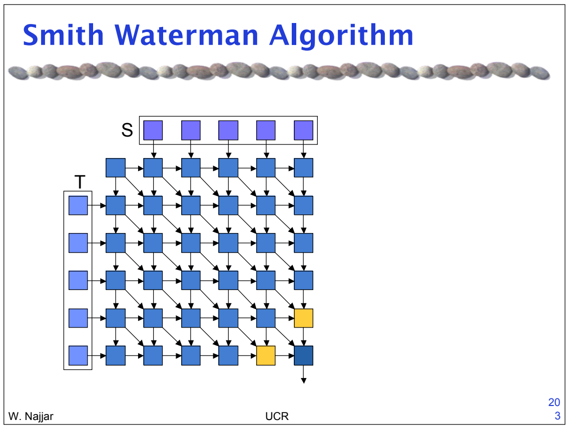
Smith Waterman Algorithm





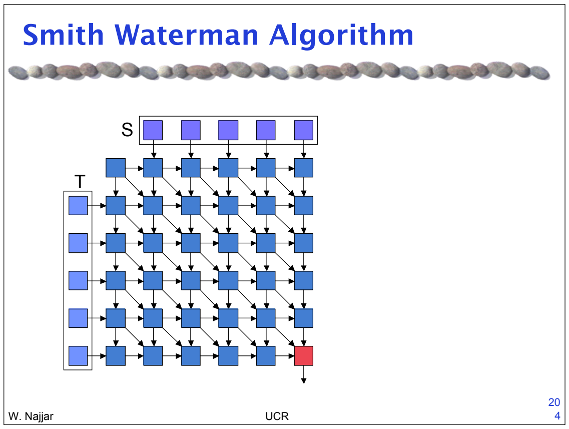
Smith Waterman Algorithm

W. Najjar UCR 202



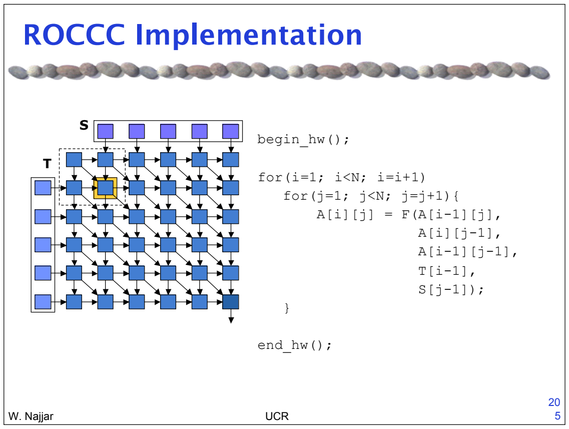
Smith Waterman Algorithm

W. Najjar UCR 203



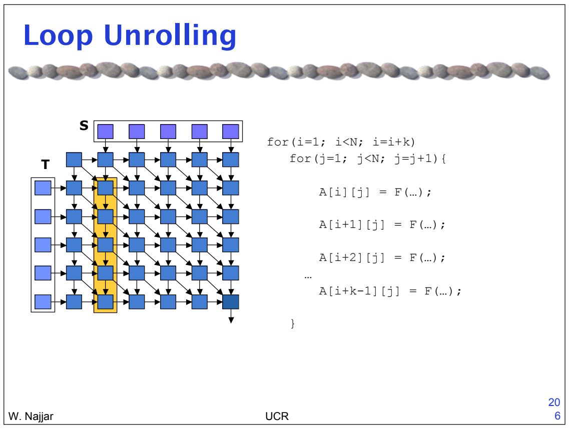
Smith Waterman Algorithm

W. Najjar UCR 204



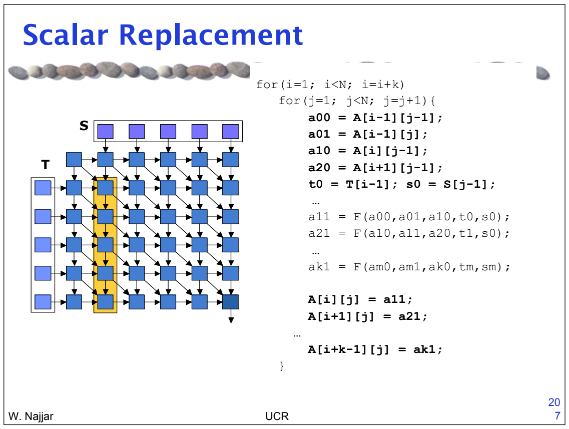
ROCCC Implementation

```
begin_hw();

for(i=1; i<N; i=i+1)
   for(j=1; j<N; j=j+1){
       A[i][j] = F(A[i-1][j],
                   A[i][j-1],
                   A[i-1][j-1],
                   T[i-1],
                   S[j-1]);
   }

end_hw();
```



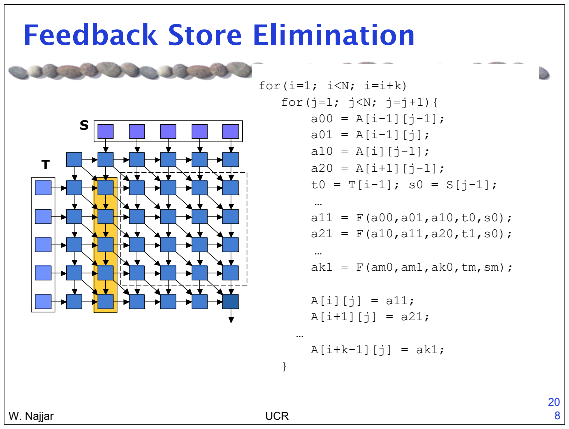
Loop Unrolling

```
for(i=1; i<N; i=i+k)
   for(j=1; j<N; j=j+1){

       A[i][j] = F(…);

       A[i+1][j] = F(…);

       A[i+2][j] = F(…);
     …
       A[i+k-1][j] = F(…);

   }
```



Scalar Replacement

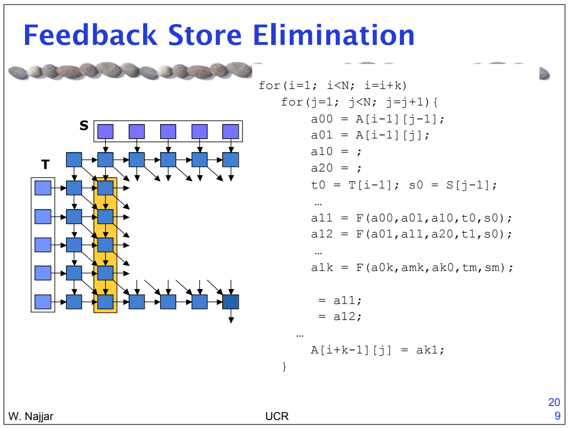
```
for(i=1; i<N; i=i+k)
   for(j=1; j<N; j=j+1){
       a00 = A[i-1][j-1];
       a01 = A[i-1][j];
       a10 = A[i][j-1];
       a20 = A[i+1][j-1];
       t0 = T[i-1]; s0 = S[j-1];
        …
       a11 = F(a00,a01,a10,t0,s0);
       a21 = F(a10,a11,a20,t1,s0);
        …
       ak1 = F(am0,am1,ak0,tm,sm);

       A[i][j] = a11;
       A[i+1][j] = a21;
     …
       A[i+k-1][j] = ak1;
   }
```



Feedback Store Elimination

```
for(i=1; i<N; i=i+k)
   for(j=1; j<N; j=j+1){
       a00 = A[i-1][j-1];
       a01 = A[i-1][j];
       a10 = A[i][j-1];
       a20 = A[i+1][j-1];
       t0 = T[i-1]; s0 = S[j-1];
        …
       a11 = F(a00,a01,a10,t0,s0);
       a21 = F(a10,a11,a20,t1,s0);
        …
       ak1 = F(am0,am1,ak0,tm,sm);

       A[i][j] = a11;
       A[i+1][j] = a21;
     …
       A[i+k-1][j] = ak1;
   }
```

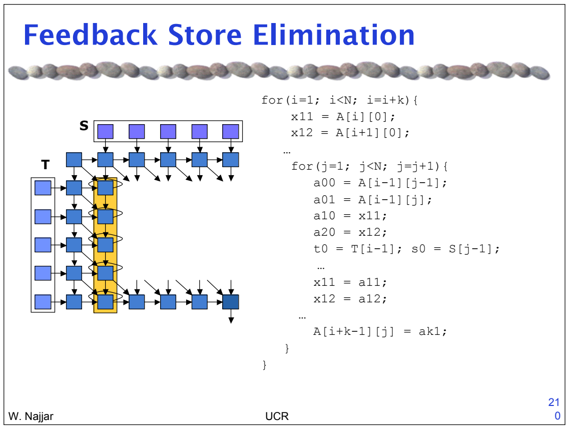
W. Najjar UCR 208



Feedback Store Elimination

```
for(i=1; i<N; i=i+k)
   for(j=1; j<N; j=j+1){
       a00 = A[i-1][j-1];
       a01 = A[i-1][j];
       a10 = ;
       a20 = ;
       t0 = T[i-1]; s0 = S[j-1];
        …
       a11 = F(a00,a01,a10,t0,s0);
       a12 = F(a01,a11,a20,t1,s0);
        …
       a1k = F(a0k,amk,ak0,tm,sm);

        = a11;
        = a12;
     …
       A[i+k-1][j] = ak1;
   }
```

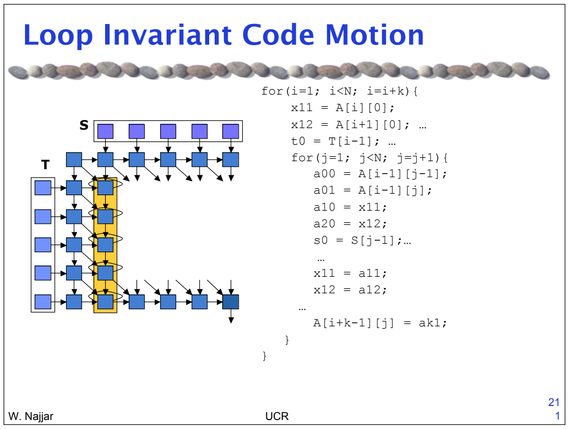
W. Najjar UCR 209



Feedback Store Elimination

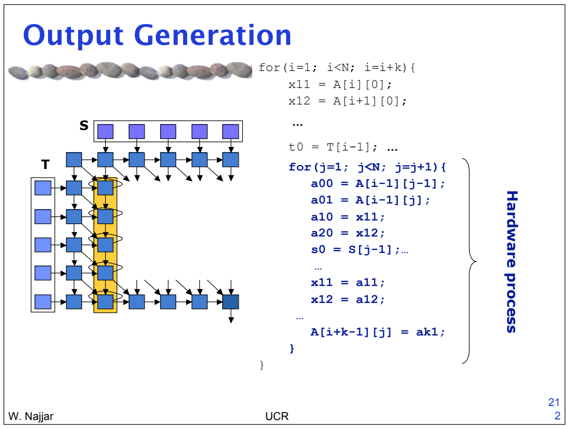
```
for(i=1; i<N; i=i+k){
    x11 = A[i][0];
    x12 = A[i+1][0];
   …
    for(j=1; j<N; j=j+1){
       a00 = A[i-1][j-1];
       a01 = A[i-1][j];
       a10 = x11;
       a20 = x12;
       t0 = T[i-1]; s0 = S[j-1];
        …
       x11 = a11;
       x12 = a12;
     …
       A[i+k-1][j] = ak1;
    }
}
```

W. Najjar UCR 210



Loop Invariant Code Motion

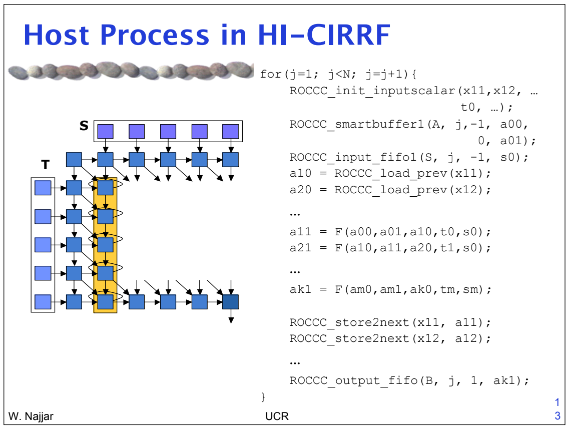
```
for(i=1; i<N; i=i+k){
    x11 = A[i][0];
    x12 = A[i+1][0]; …
    t0 = T[i-1]; …
    for(j=1; j<N; j=j+1){
       a00 = A[i-1][j-1];
       a01 = A[i-1][j];
       a10 = x11;
       a20 = x12;
       s0 = S[j-1];…
        …
       x11 = a11;
       x12 = a12;
     …
       A[i+k-1][j] = ak1;
   }
}
```



Output Generation

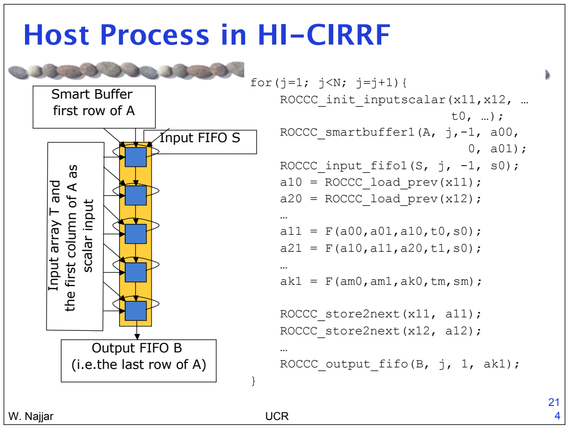
```
for(i=1; i<N; i=i+k){
    x11 = A[i][0];
    x12 = A[i+1][0];

     …

    t0 = T[i-1]; …
    for(j=1; j<N; j=j+1){
       a00 = A[i-1][j-1];
       a01 = A[i-1][j];
       a10 = x11;
       a20 = x12;
       s0 = S[j-1];…
        …
       x11 = a11;
       x12 = a12;
     …
       A[i+k-1][j] = ak1;
    }
}
```

Hardware process



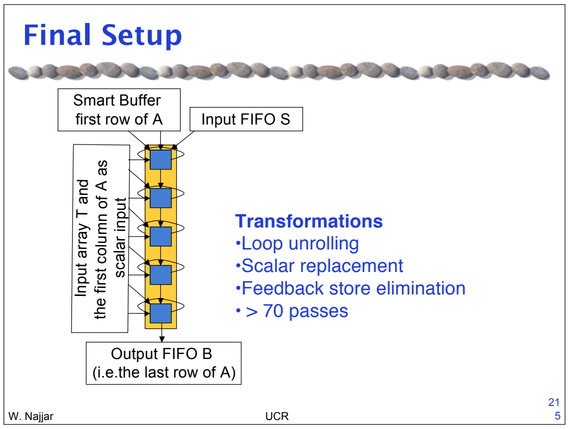
Host Process in HI-CIRRF

```
for(j=1; j<N; j=j+1){
    ROCCC_init_inputscalar(x11,x12, …
                           t0, …);
    ROCCC_smartbuffer1(A, j,-1, a00,
                              0, a01);
    ROCCC_input_fifo1(S, j, -1, s0);
    a10 = ROCCC_load_prev(x11);
    a20 = ROCCC_load_prev(x12);

    …

    a11 = F(a00,a01,a10,t0,s0);
    a21 = F(a10,a11,a20,t1,s0);

    …

    ak1 = F(am0,am1,ak0,tm,sm);

    ROCCC_store2next(x11, a11);
    ROCCC_store2next(x12, a12);

    …

    ROCCC_output_fifo(B, j, 1, ak1);
}
```



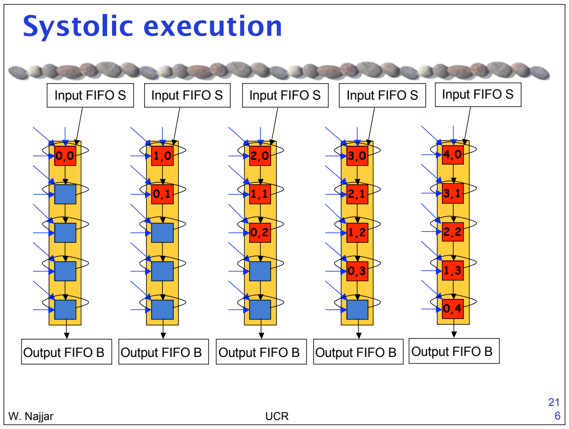
Host Process in HI-CIRRF

```
for(j=1; j<N; j=j+1){
    ROCCC_init_inputscalar(x11,x12, …
                           t0, …);
    ROCCC_smartbuffer1(A, j,-1, a00,
                              0, a01);
    ROCCC_input_fifo1(S, j, -1, s0);
    a10 = ROCCC_load_prev(x11);
    a20 = ROCCC_load_prev(x12);
    …
    a11 = F(a00,a01,a10,t0,s0);
    a21 = F(a10,a11,a20,t1,s0);
    …
    ak1 = F(am0,am1,ak0,tm,sm);

    ROCCC_store2next(x11, a11);
    ROCCC_store2next(x12, a12);
    …
    ROCCC_output_fifo(B, j, 1, ak1);
}
```



Final Setup

Transformations
- •Loop unrolling
- •Scalar replacement
- •Feedback store elimination
- • > 70 passes

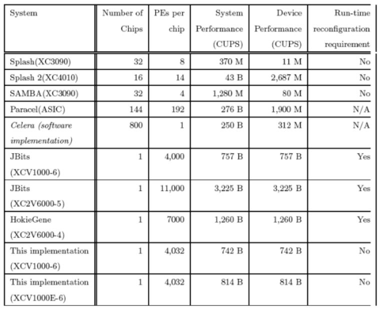


Systolic execution

SW Performance

Smith Waterman					
	Xeon	Itanium 2	V4LX200		
Cells			200	512	1024
Area (%)			2.8%	6.9%	17%
Clock	2.8 GHz	1.6 GHz	191 MHz	188 MHz	174 MHz
GCUPS	0.049	0.084	38.2	96.25	178.64
Speedup	1	1.7	780	1964	3645

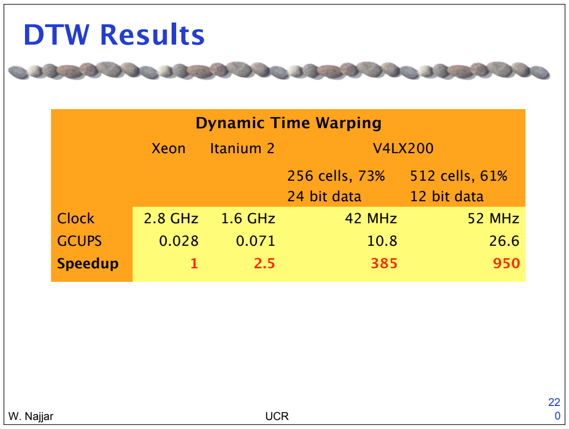
SW Results



System	Number of Chips	PEs per chip	System Performance (CUPS)	Device Performance (CUPS)	Run-time reconfiguration requirement
Splash(XC3090)	32	8	370 M	11 M	No
Splash 2(XC4010)	16	14	43 B	2,687 M	No
SAMBA(XC3090)	32	4	1,280 M	80 M	No
Paracel(ASIC)	144	192	276 B	1,900 M	N/A
Celera (software implementation)	800	1	250 B	312 M	N/A
JBits (XCV1000-6)	1	4,000	757 B	757 B	Yes
JBits (XC2V6000-5)	1	11,000	3,225 B	3,225 B	Yes
HokieGene (XC2V6000-4)	1	7000	1,260 B	1,260 B	Yes
This implementation (XCV1000-6)	1	4,032	742 B	742 B	No
This implementation (XCV1000E-6)	1	4,032	814 B	814 B	No
ROCCC	**1**	**512**	**70.5B**	**70.5B**	**No**

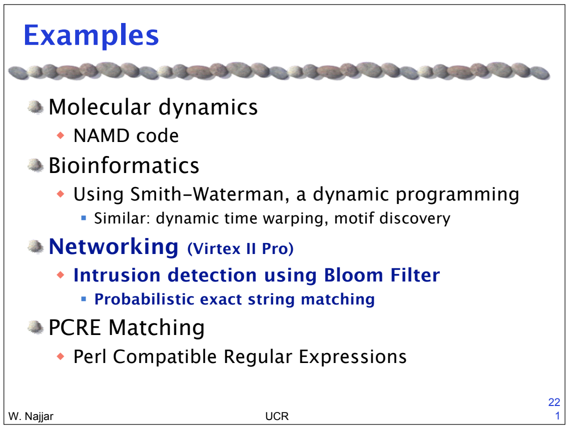
Dynamic Time Warping

- Data mining application
  - Motif discovery
  - Uses dynamic programming code
  - Multiplication of integer in each cell computation



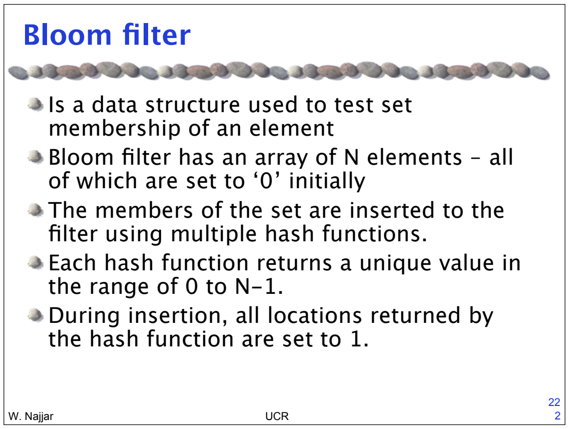
DTW Results

Dynamic Time Warping				
	Xeon	Itanium 2	V4LX200	
			256 cells, 73% 24 bit data	512 cells, 61% 12 bit data
Clock	2.8 GHz	1.6 GHz	42 MHz	52 MHz
GCUPS	0.028	0.071	10.8	26.6
Speedup	**1**	**2.5**	**385**	**950**



Examples

- Molecular dynamics
  - NAMD code
- Bioinformatics
  - Using Smith-Waterman, a dynamic programming
    - Similar: dynamic time warping, motif discovery
- **Networking (Virtex II Pro)**
  - **Intrusion detection using Bloom Filter**
    - **Probabilistic exact string matching**
- PCRE Matching
  - Perl Compatible Regular Expressions



Bloom filter

- Is a data structure used to test set membership of an element
- Bloom filter has an array of N elements – all of which are set to ‘0’ initially
- The members of the set are inserted to the filter using multiple hash functions.
- Each hash function returns a unique value in the range of 0 to N-1.
- During insertion, all locations returned by the hash function are set to 1.

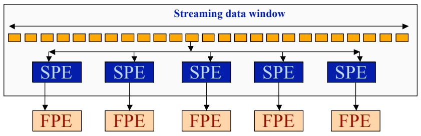
Search operation in a bloom filter

- During a search operation, multiple hash functions are applied to an incoming value.
- If all the locations returned by the hash function contain ‘1’, then the element belongs to the set with a probability P.
- For a Bloom filter with m elements, the probability of a false positive is given by : $\left(1 - e^{-kn/m}\right)^k$

Where,
K denotes the number of hash functions
m is the number of bits in the Bloom filter array
n is the number of elements inserted into the Bloom filter

Bloom filter for virus detection

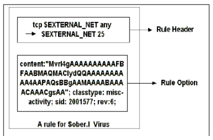
- Ours is the first bloom filter based virus detection code automatically generated from C.
- Each Signature Processing Engine (SPE) contains the generated bloom filter code and is used to detect signatures
- Bloom filter output contains false positives. Hence a RAM is used for absolute string comparison and to eliminate false positives.



Legend : SPE : Signature Processing Engine FPE: False Positive Eliminator

Virus signatures

- Signatures are unique bit patterns that correspond to a virus/malware
- We used the virus rules in the *bleeding snort* database.
- Each rule consists of a rule header and an option.
- Header contains information to be used in packet classification.
- Rule option contains the signatures to be used in intrusion detection.
- Most of the signatures in bleeding-snort database were under 32 bytes.

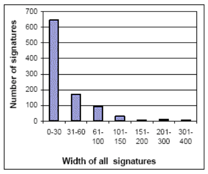


tcp $EXTERNAL_NET any → $EXTERNAL_NET 25

content:"Mvrl4gAAAAAAAAAAFBFAABMAQMAClydQQAAAAAAAAAA4AAPAQsBBgAAMAAAABAAAACAAACgsAA"; classtype: misc-activity; sid: 2001577; rev:6;

Rule Header
Rule Option

A rule for Sober.I Virus



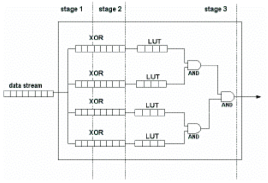
C code

- We implemented a functional protoype where we compared signatures of width 1 byte.
- Hash functions are implemented as XOR operations
- The innerloop processes the each byte of the incoming 8-byte value in parallel.

```
for(i=0;i<8;i++)
 {
   temp = value &0xff;
   result_location1 = temp ^
     hash_function1[i];;
   result_location2 = temp ^
     hash_function2[i];;
   result_location3 = temp ^
     hash_function3[i];;
   result_location4 = temp ^
     hash_function4[i];;

  found = bit_array[result_location1] &
             bit_array[result_location2] &
            bit_array[result_location3] &
            bit_array[result_location4] ;
  value = value >> 8;
 }
 return (found);
```

Datapath Analysis

- Compiler exploits ILP by grouping instructions into different execution levels.
- Each level corresponds to a loop iteration and the instructions are executed simultaneously.
- ROCC automatically places latches for pipelining



- Each latched level corresponds to one pipeline stage and has a delay of one cycle.
- In the 3-stage pipeline each box of XOR corresponds to one byte of input being XORed with a hashing function

Throughput evaluation

- The generated code does not have loop-carried dependency and the compiler pipelines the datapath fully.
- Clock frequency of the synthesized circuit was found to be 73MHz.
- The BRAM on our target FPGA can process 32 bytes per cycle.
- Throughput = bits per cycle * clock frequency

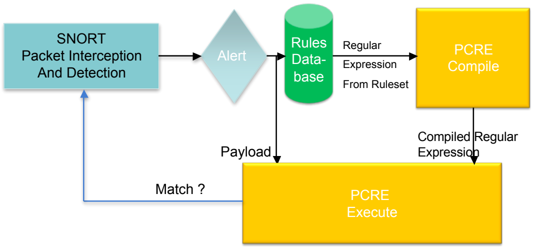
  32*8 * 73* 100,000 bits/sec

  ~ 18.6 Gbps

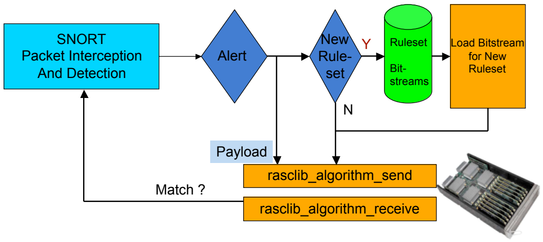
Examples

- Molecular dynamics
  - NAMD code
- Bioinformatics
  - Using Smith-Waterman, a dynamic programming
    - Similar: dynamic time warping, motif discovery
- Networking (Virtex II Pro)
  - Intrusion detection using Bloom Filter
    - Probabilistic exact string matching
- **PCRE Matching**
  - **Perl Compatible Regular Expressions**

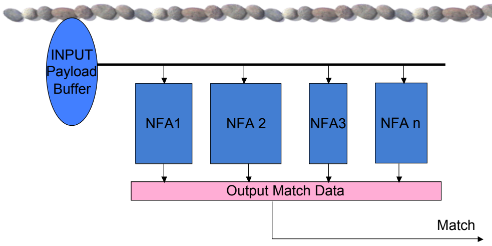
SNORT IDS with PCRE Regexp matching



SNORT IDS with RASC RC100 based Regexp Matching

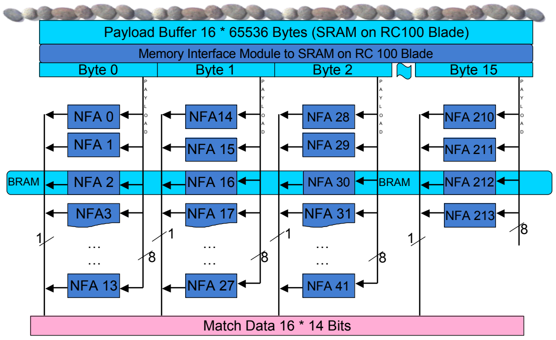


Chained NFA engines



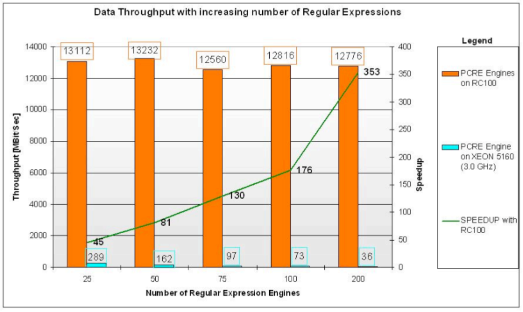
W. Najjar UCR 23 2

Architecture of NFA engines

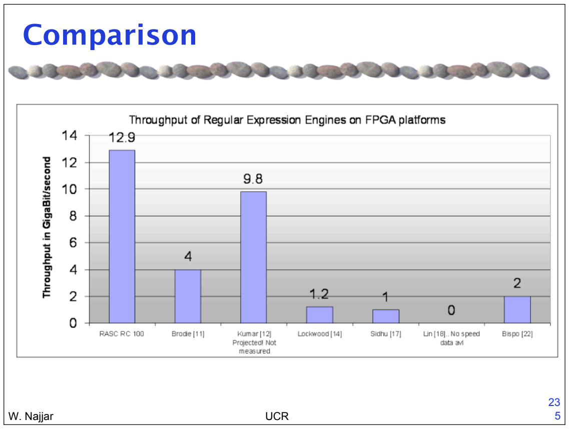


W. Najjar UCR 23 3

Throughput evaluation



W. Najjar UCR 23 4



Comparison

W. Najjar UCR 235