CS577: C-Based VLSI Design

Dr. Chandan Karfa

Department of Computer Science and Engineering

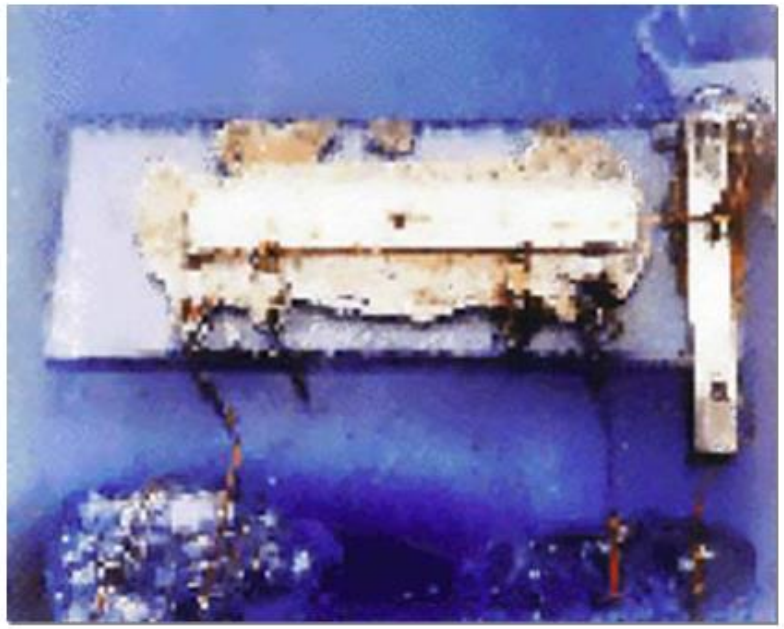
Indian Institute of Technology Guwahati

Lectures: 1-3

What is CS577: C-Based VLSI Design all about?

- What is Very Large Scale Integration (VLSI)?
  - **VLSI** is the process of creating an integrated circuit (IC) by combining millions of MOS transistors onto a single chip.
- How can we built an IC consists of millions of Transistors? Manually?
  - Need Automation
- Can we built a IC starting from C code in an automated way?
  - Yes

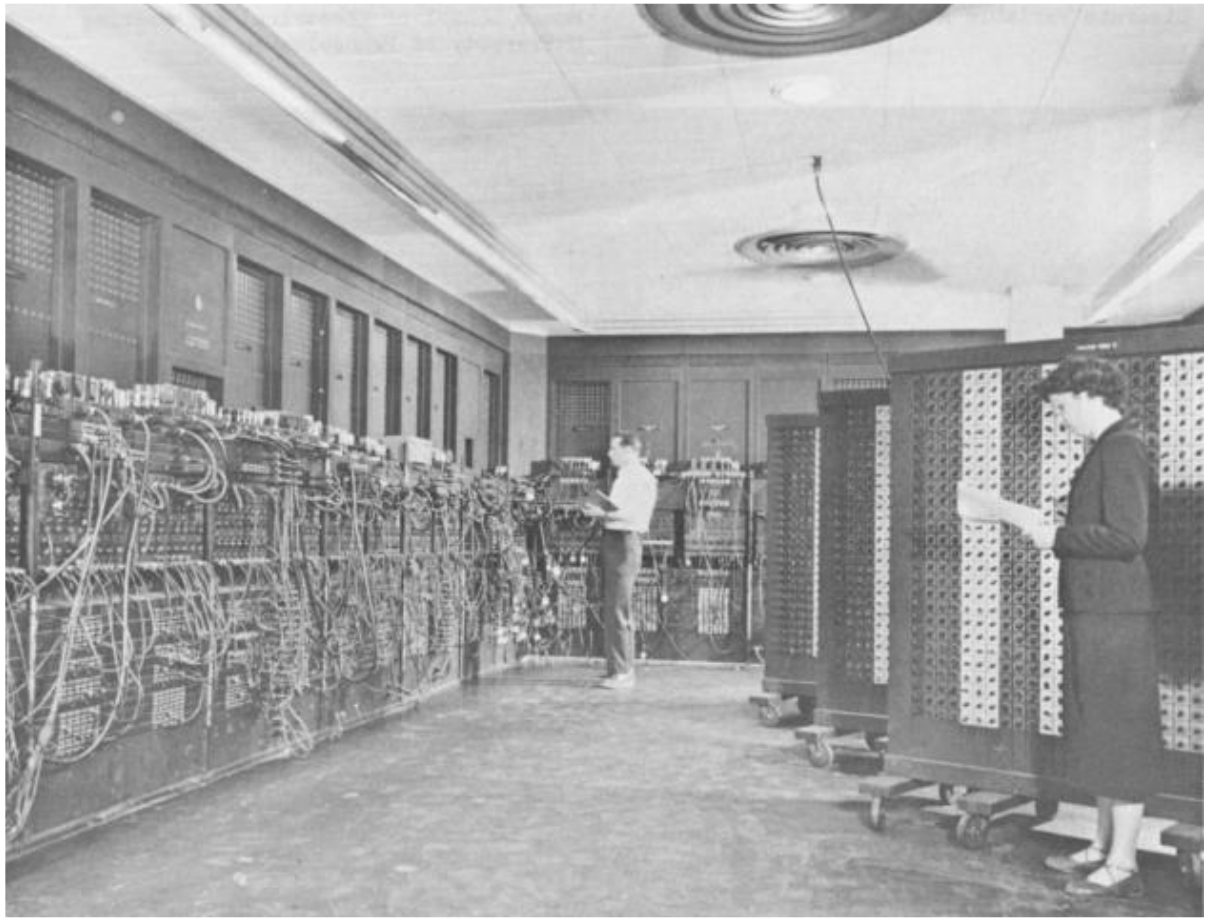
Integrated Circuit (IC) Revolution



1958: First integrated circuit (germanium)
Built by Jack Kilby at Texas Instruments
Contailed five components : transistors, resistors and capacitors

2000: Intel Pentium 4 Processor
Clock speed: 1.5 GHz
Transistors: 42 million
Technology: 0.18µm CMOS

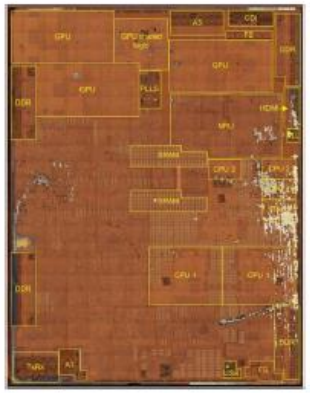
How Chips Have Shrunk?



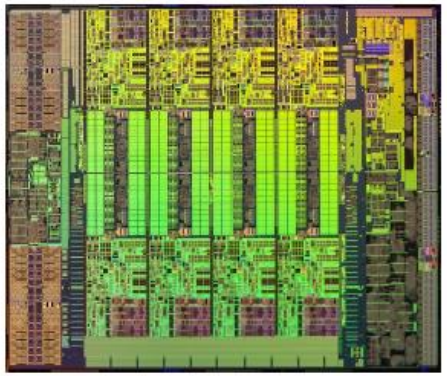
•1946 in UPenn
•Measured in cubic ft.

Intel/Altera Stratix 10
~30B transistors

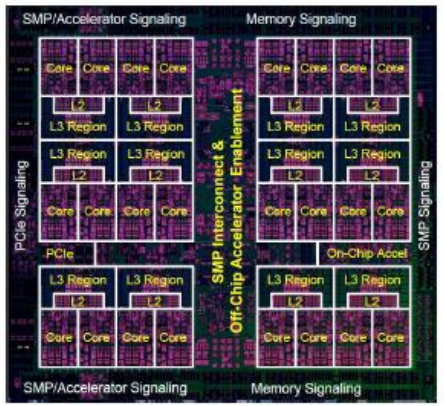
Era of Billion-Transistor Chips



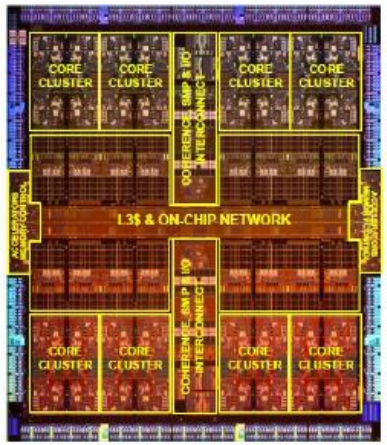
Apple A11
~4B transistors (?)



Intel Haswell-EP Xeon E5
~7B transistors



IBM Power9
~8B transistors

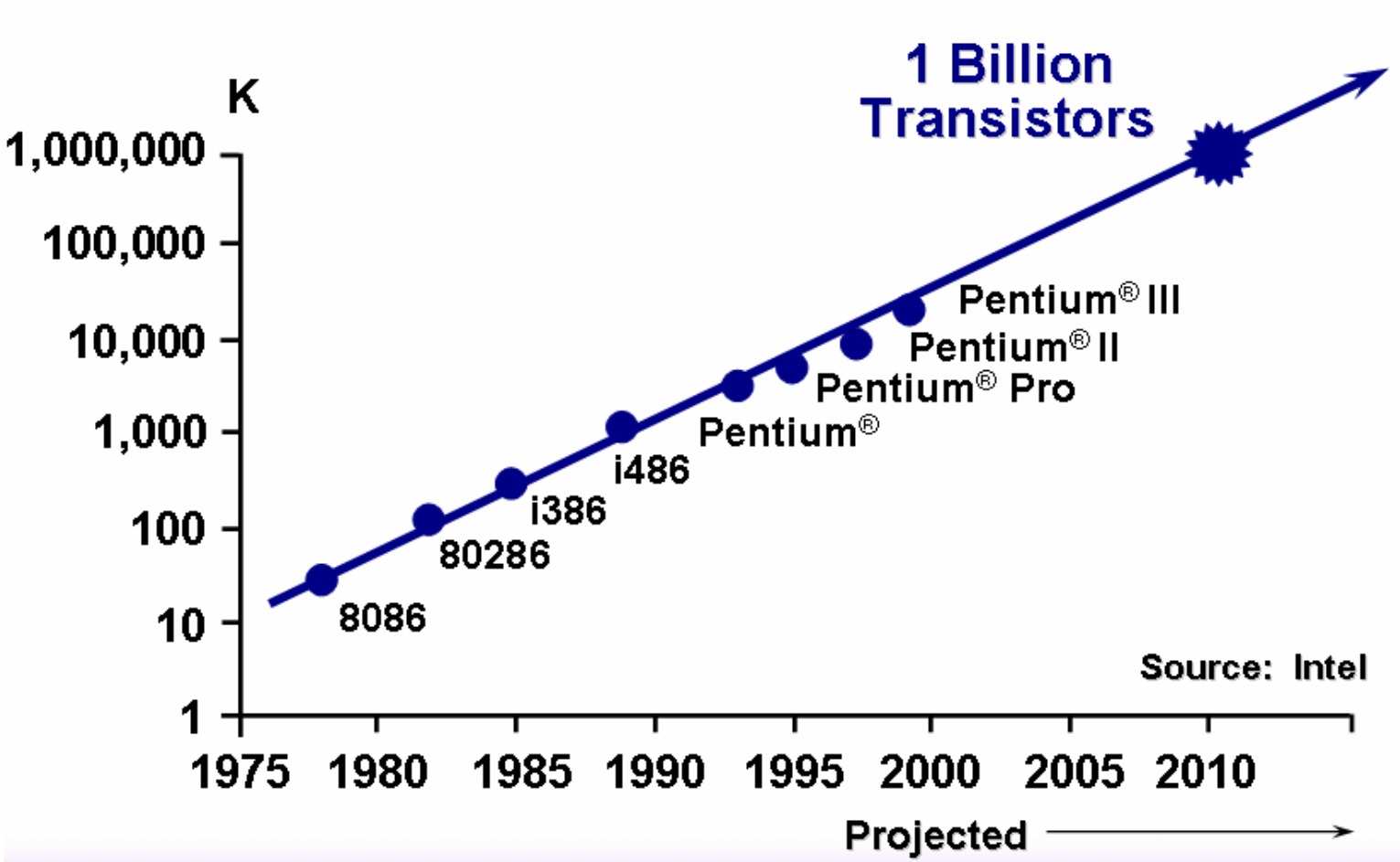


Oracle SPARC M7
~10B transistors

NVIDIA V100 Pascal
~21B transistors

Intel/Altera Stratix 10
~30B transistors

Evolution in IC Complexity



If Transistors are Counted as Seconds

4004	< 1 hr
8080	< 2 hrs
8086	8 hrs
80286	1.5 days
80386 DX	3 days
80486	13 days
Pentium	> 1 month
Pentium Pro	2 months
P II	3 months
P III	~1 year
P4	~ 1.5 years

Implication

- More functionality
- More complexity
- Cost ??

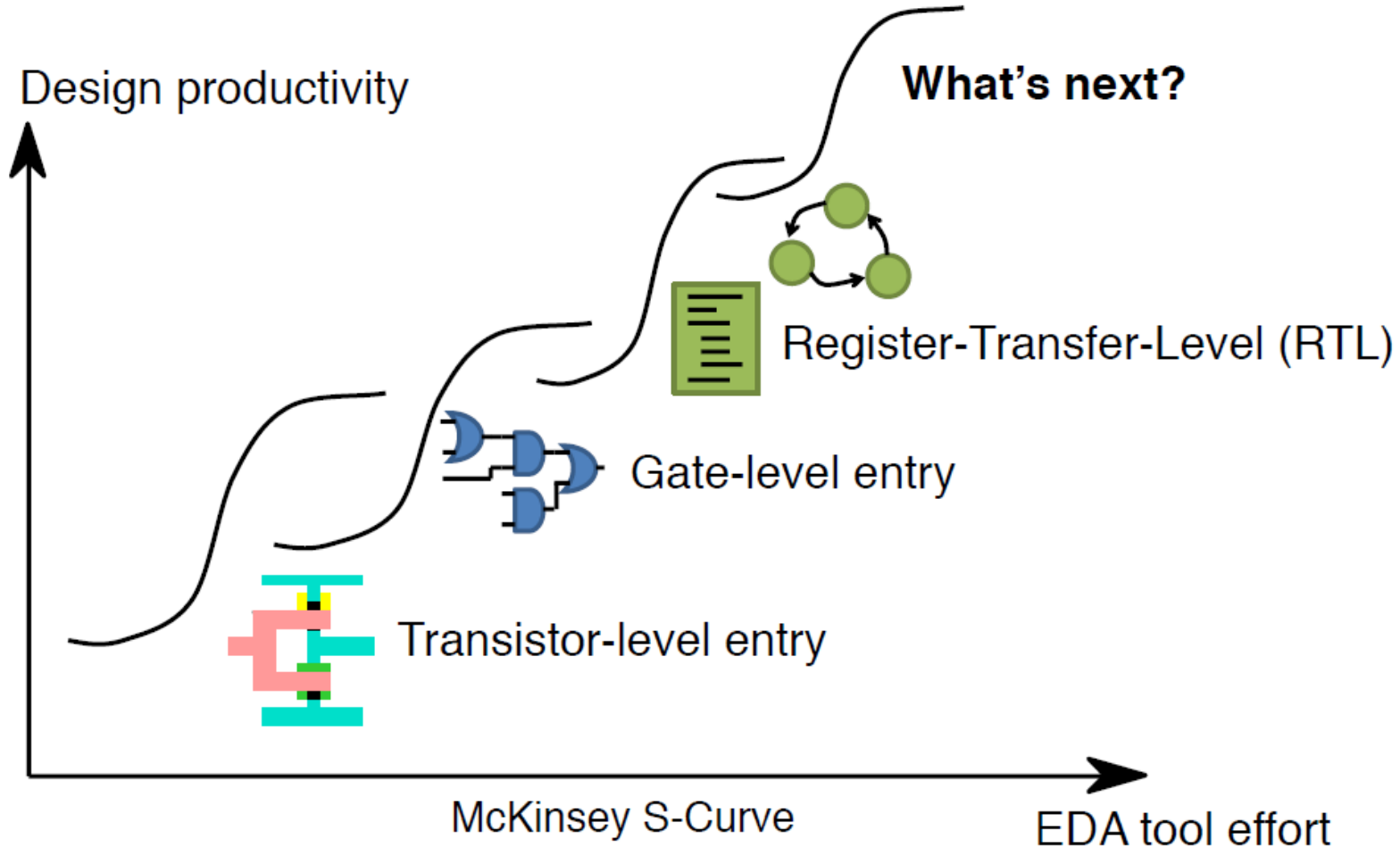
How to design a 10B transistors design?

- “This incredible growth rate could not be achieved by hiring an exponentially growing number of design engineers. It was fulfilled by adopting new design methodologies and by i**ntroducing innovative design automation software** at every processor generation”

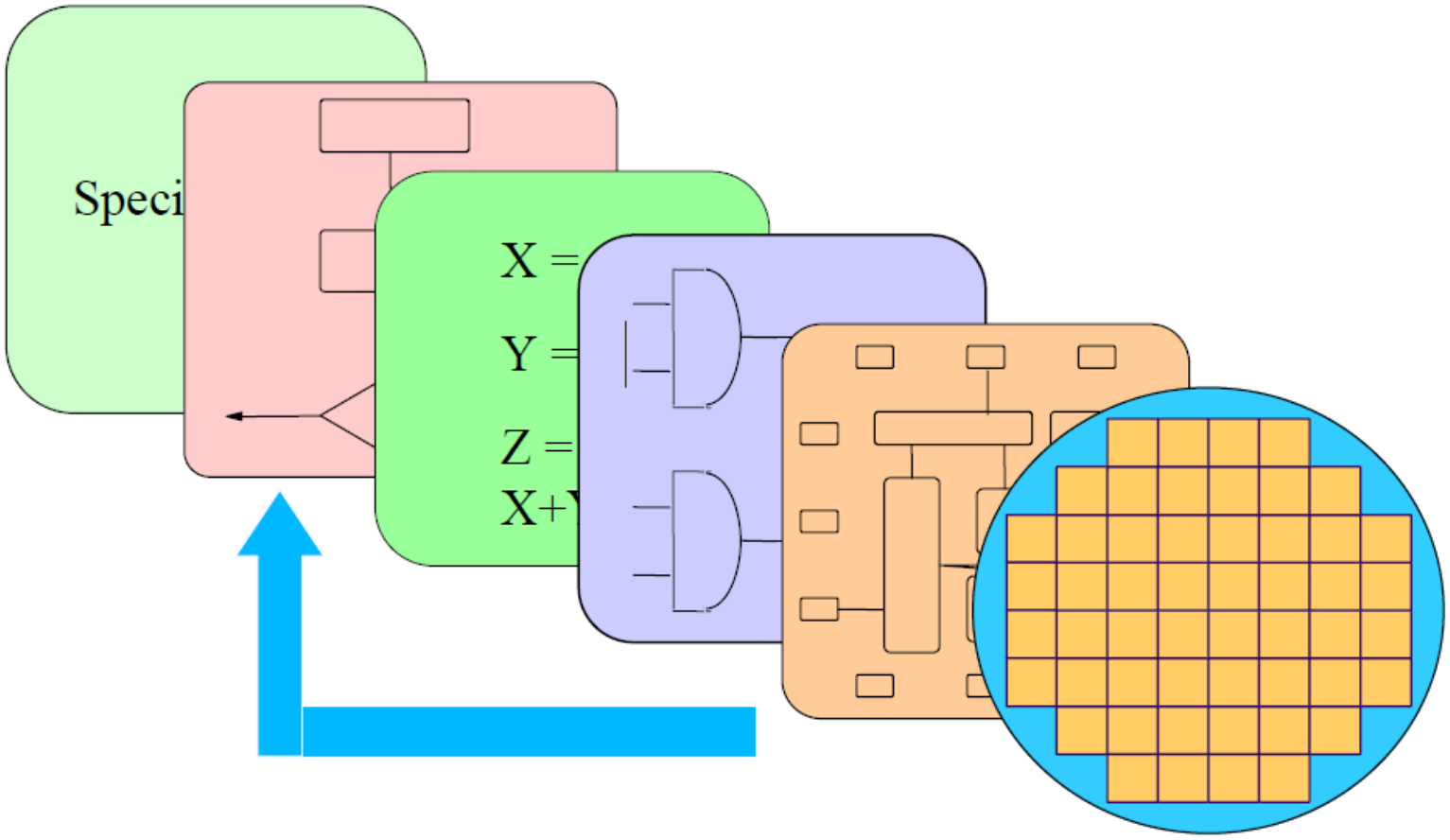
TABLE 1. INTEL PROCESSORS, 1971–1993.

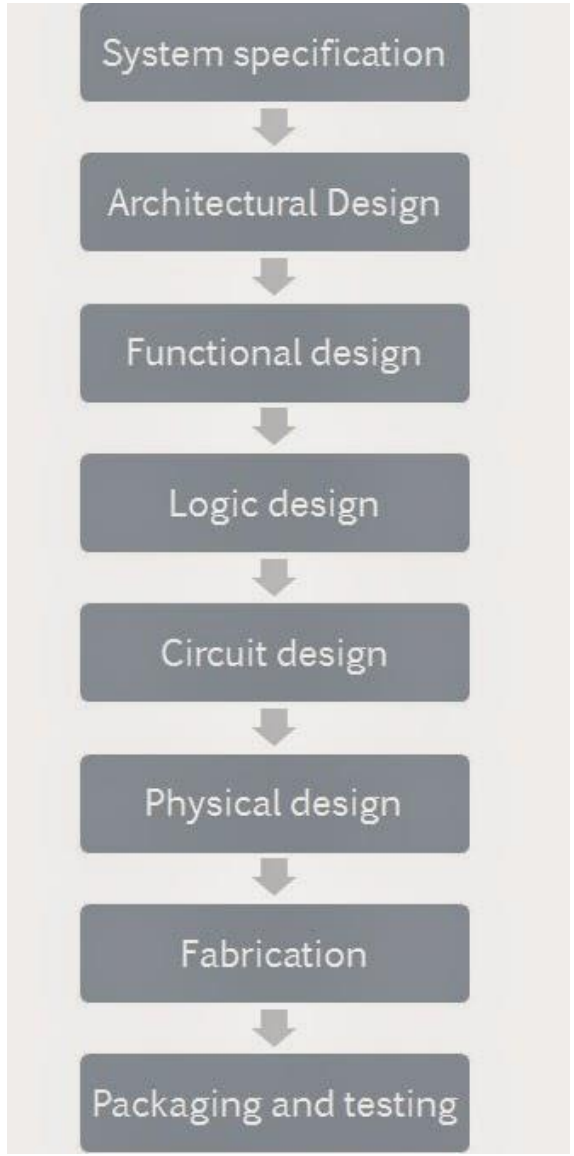
PROCESSOR	INTRO DATE	PROCESS	TRANSISTORS	FREQUENCY
4004	1971	10 μm	2,300	108 KHz
8080	1974	6 μm	6,000	2 MHz
8086	1978	3 μm	29,000	10 MHz
80286	1982	1.5 μm	134,000	12 MHz
80386	1985	1.5 μm	275,000	16 MHz
Intel 486 DX	1989	1 μm	1.2 M	33 MHz
Pentium	1993	0.8 μm	3.1 M	60 MHz

Evolution of EDA and Design Abstraction



VLSI Design Flow

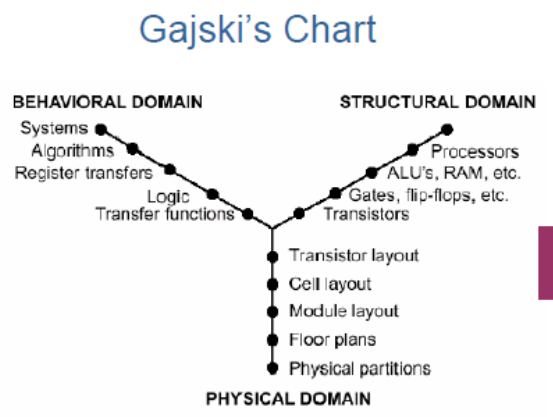


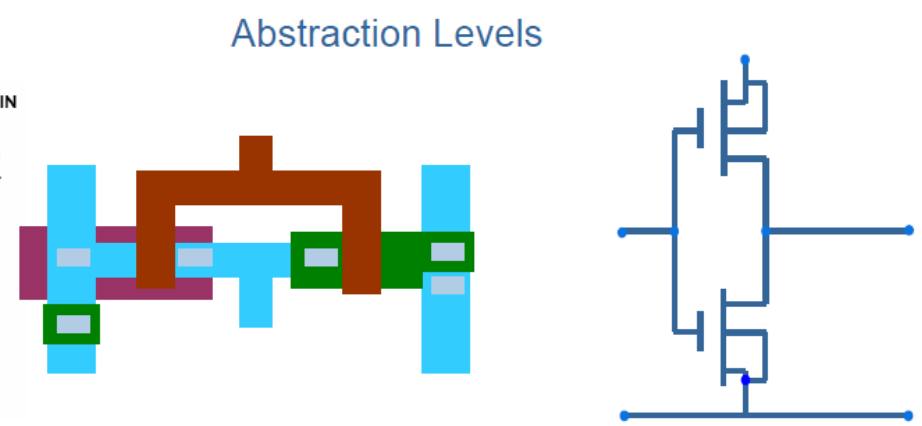


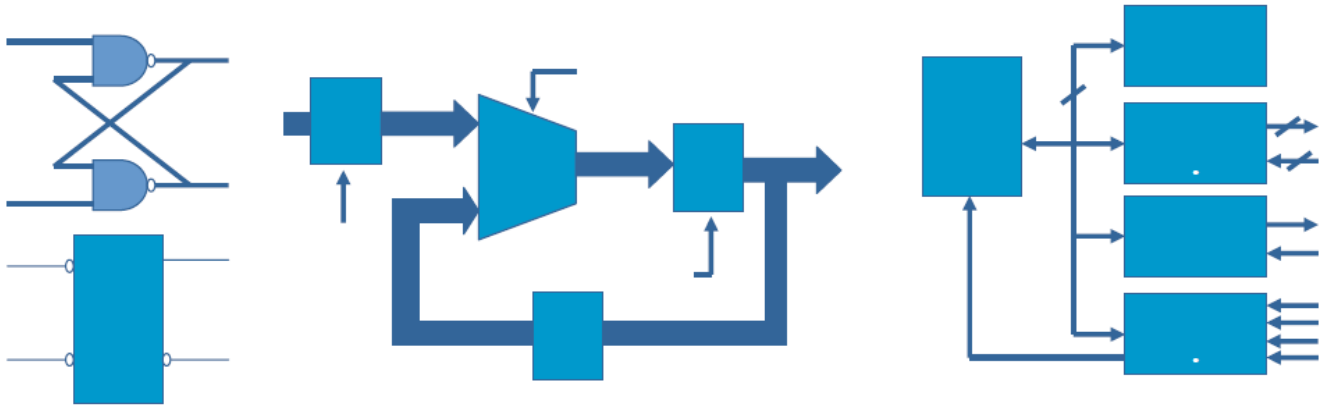
Importance of Design Automation

- Shorter design cycle
- Design space exploration
- Fewer errors in the design
- Specification driven optimization at the higher abstraction level

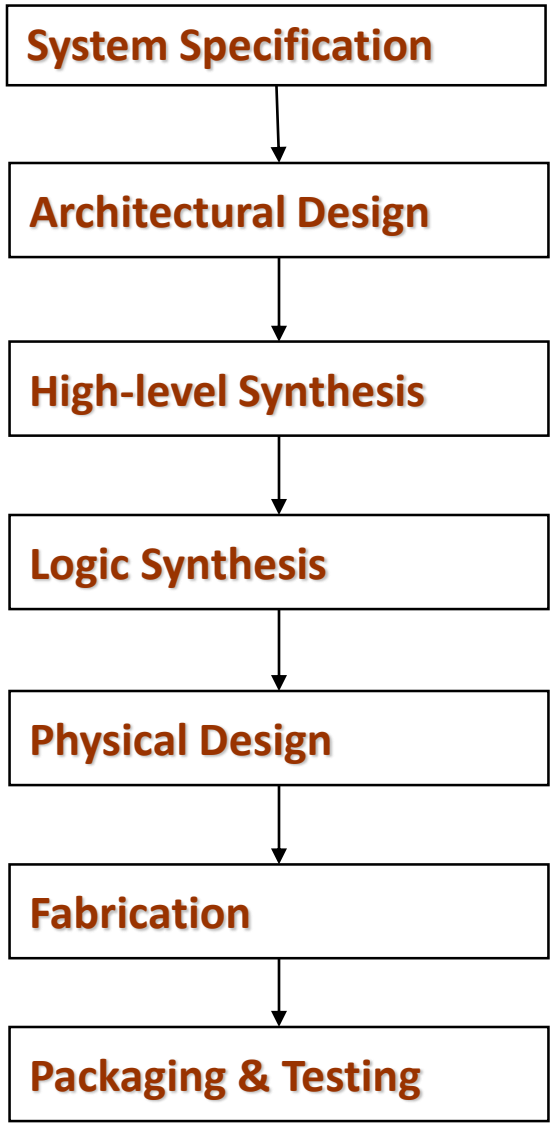
EDA Flow


Gajski's Chart
BEHAVIORAL DOMAIN
STRUCTURAL DOMAIN
Systems
Algorithms
Register transfers
Logic
Transfer functions
Processors
ALU's, RAM, etc.
Gates, flip-flops, etc.
Transistors
Transistor layout
Cell layout
Module layout
Floor plans
Physical partitions
PHYSICAL DOMAIN

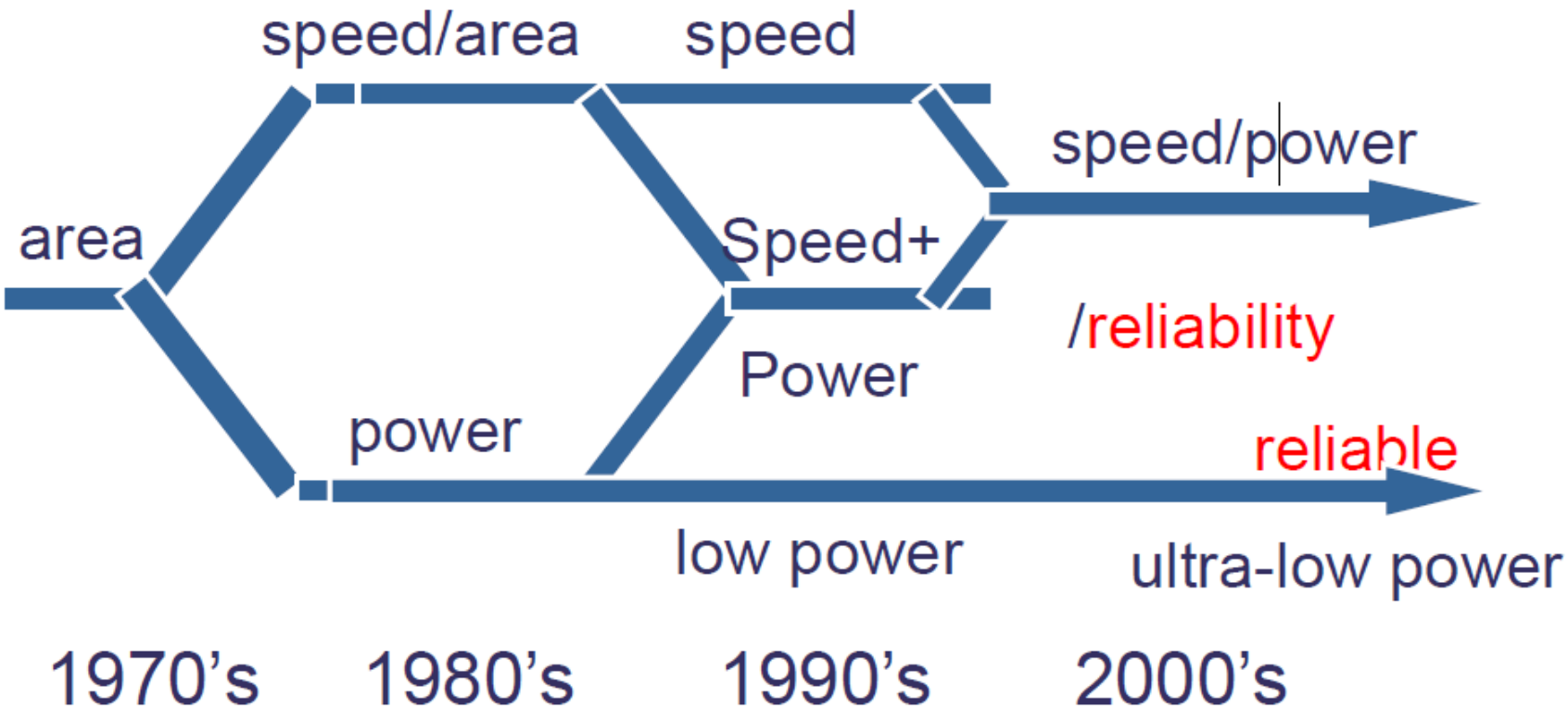

Abstraction Levels



VLSI Design Flow



Design Goals Over Time

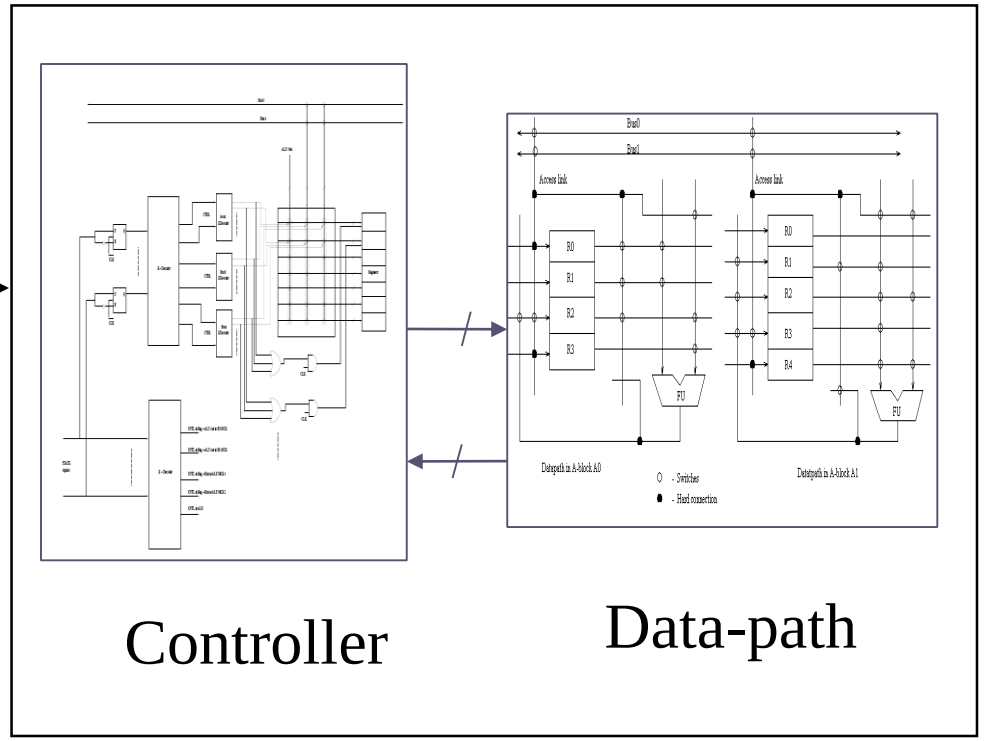


High-level Synthesis (C to Hardware/RTL)

```
while (x_var < a_var) loop
    t1 := u_var * dx_var;
    t2 := 3 * x_var;
    t3 := 3 * y_var;
    t4 := t1 * t2;
    t5 := dx_var * t3;
    t6 := u_var - t4;
    u_var := t6 - t5;
    y1 := u_var * dx_var;
    y_var := y_var + y1;
    x_var := x_var + dx_var;
 end loop;
X <= x_var;
Y <= y_var;
U <= u_var;
```

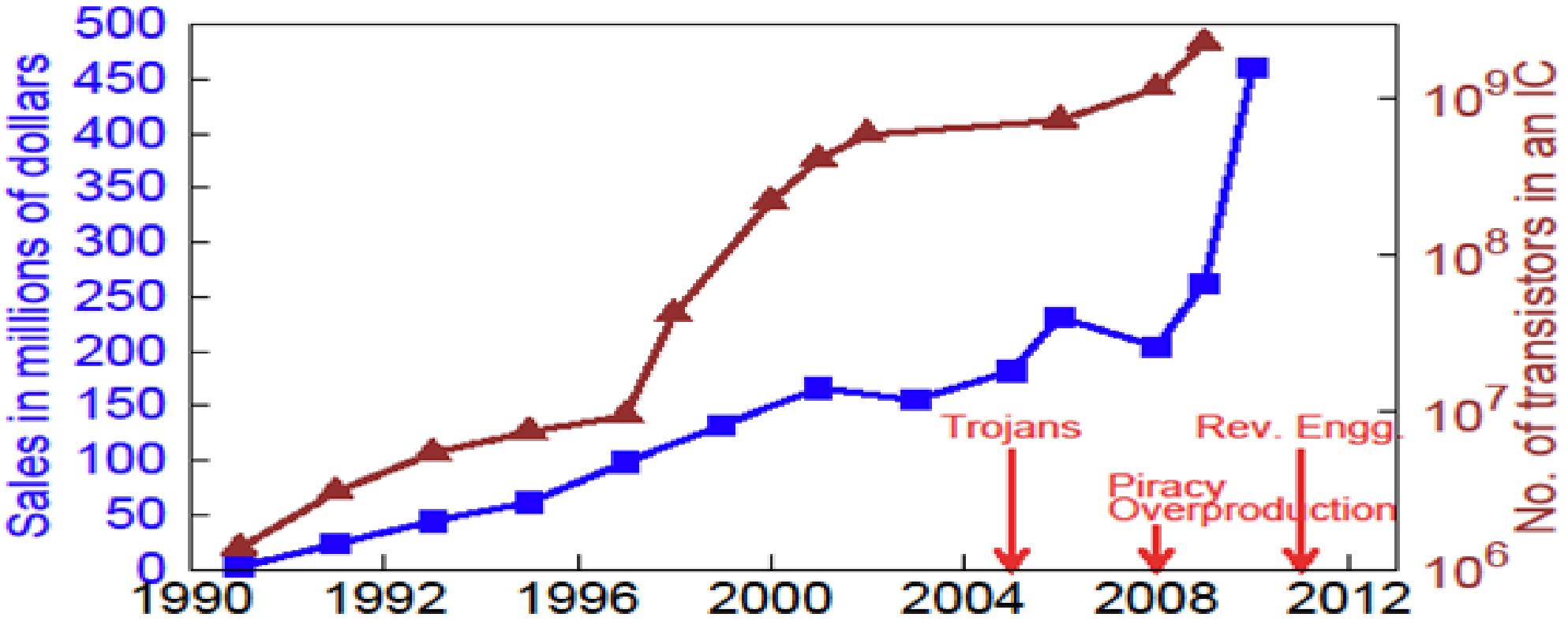
High-level Behaviour

HLS



Register Transfer Level Description

High-Level Synthesis (HLS)/C-Based VLSI Design



- C ➔ Gates
- Design time ↓
- Design complexity ↓ (10x)
- Verification effort ↓
- Hardware/software co-design ↑

Synfora ↔ SYNOPSYS®

XILINX®

intel FPGA

Oasys ↔ Mentor Graphics®

CALYPTO Design - Optimize - Verify ↔

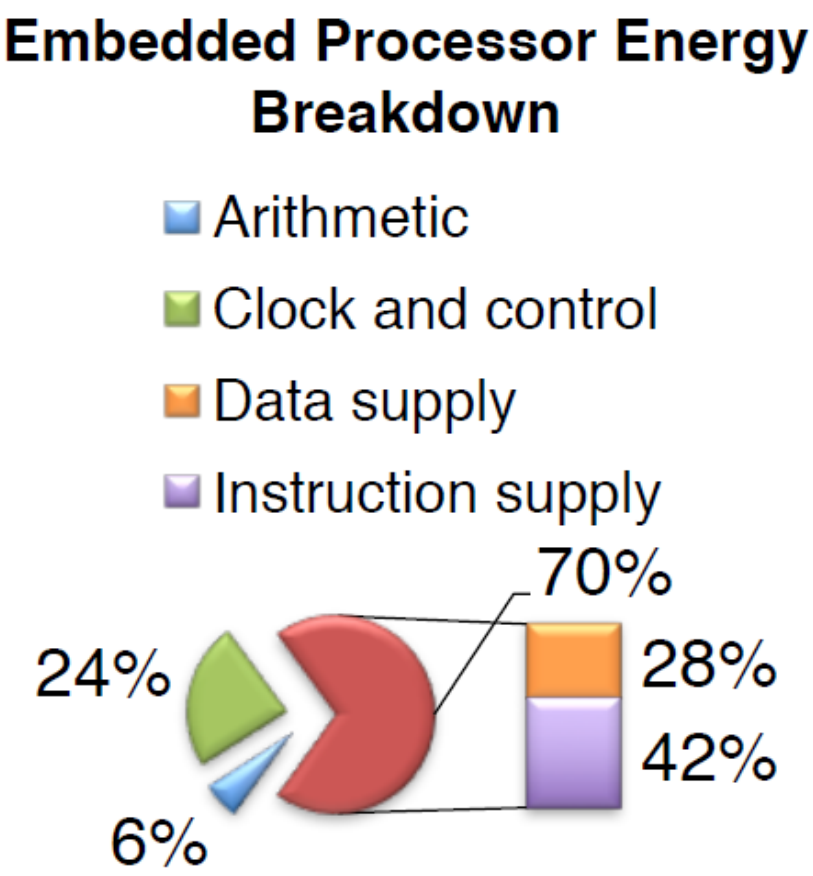
bluespec

FORTE DESIGN SYSTEMS ↔ cadence™

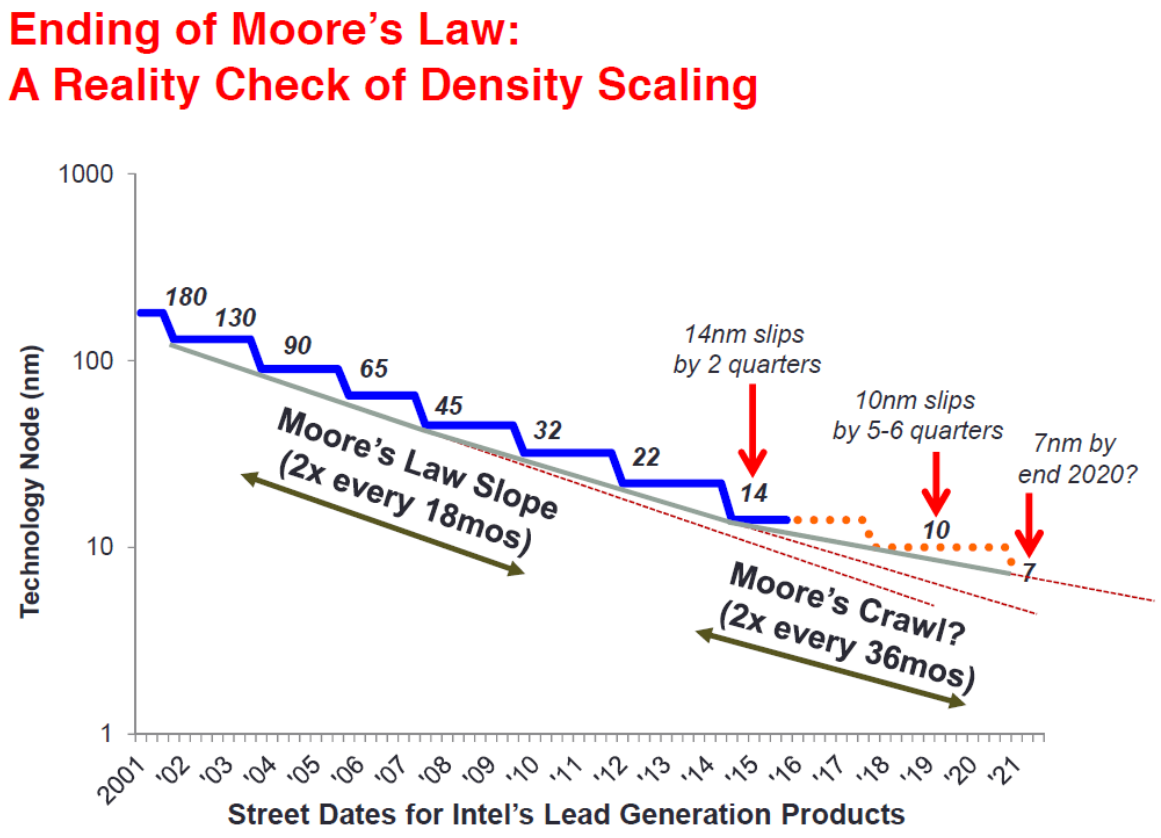
- Why do need Specialized hardware?
- Why the general purpose processors are not sufficient?

Inefficiency of General-Purpose Computing

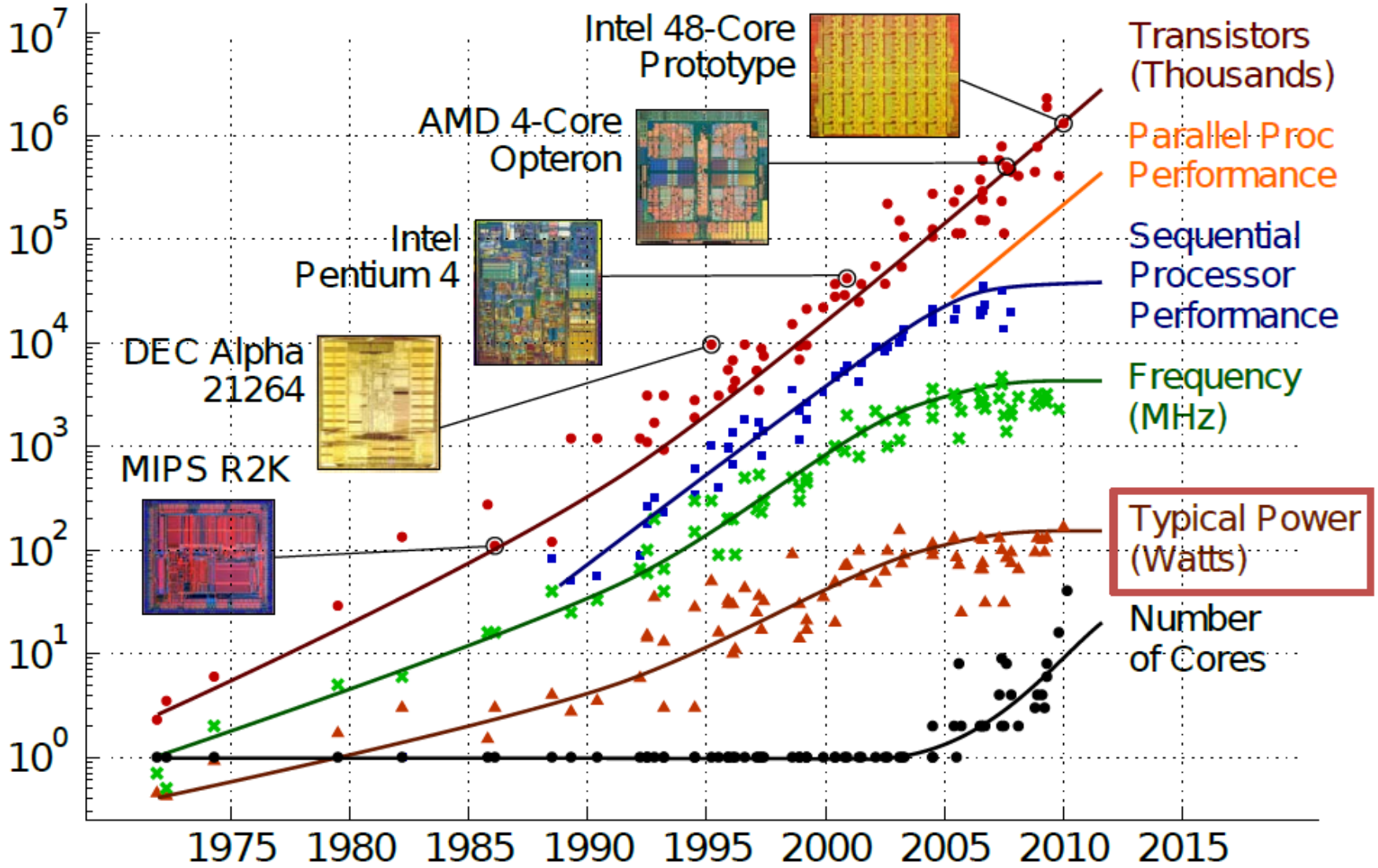
- Typical energy overhead for every 10pJ arithmetic operations
  - 70pJ on instruction supply
  - 47pJ on data supply
- Only 59% of the instructions are arithmetic



Dennard scaling: roughly, as transistors get smaller, their power density stays constant, so that the power use stays in proportion with area; both voltage and current scale (downward) with length



Power becomes the Key Constraint



Data partially collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond

Power-Constrained Modern Computers

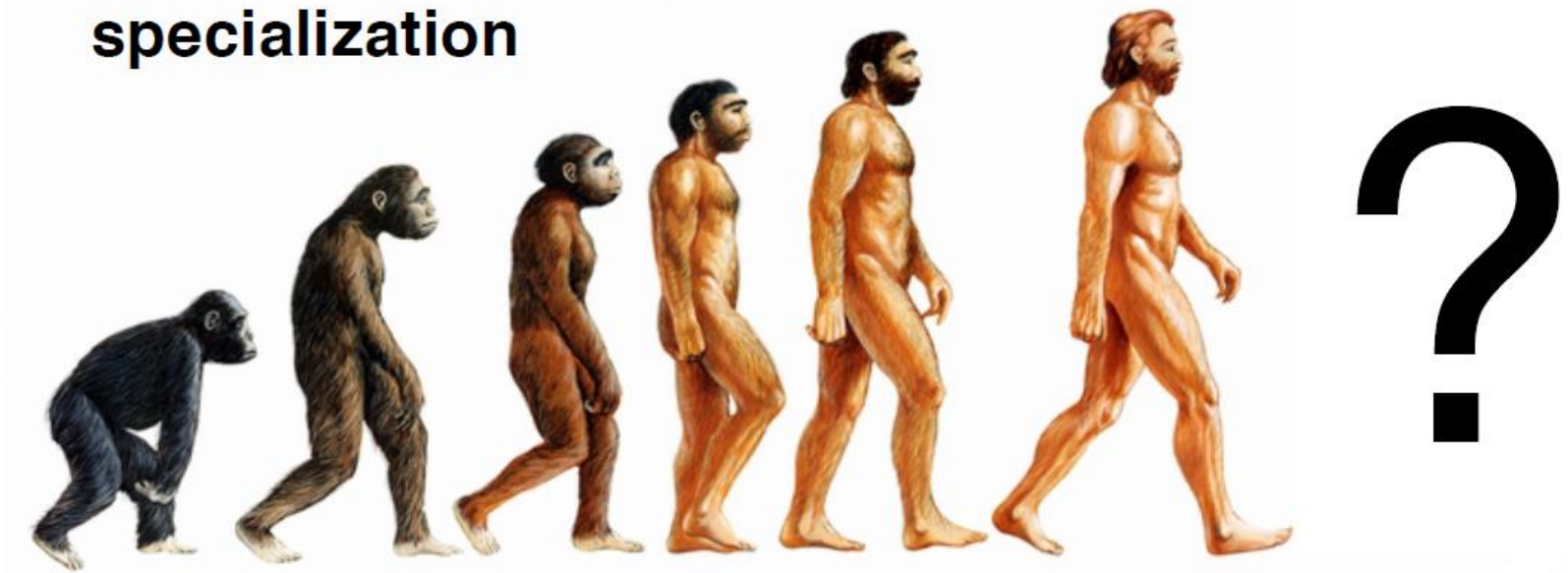
<<1W/chip ~1W/chip ~15W/chip ~50W/chip ~100W/chip >100W/chip

$$Power = \frac{Energy}{Second} = \frac{Energy}{Op} \times \frac{Ops}{Second}$$

- **Energy efficiency must improve!**
- **Limitations of general-purpose multicore scaling**
  - Amdahl's law
  - Dark silicon

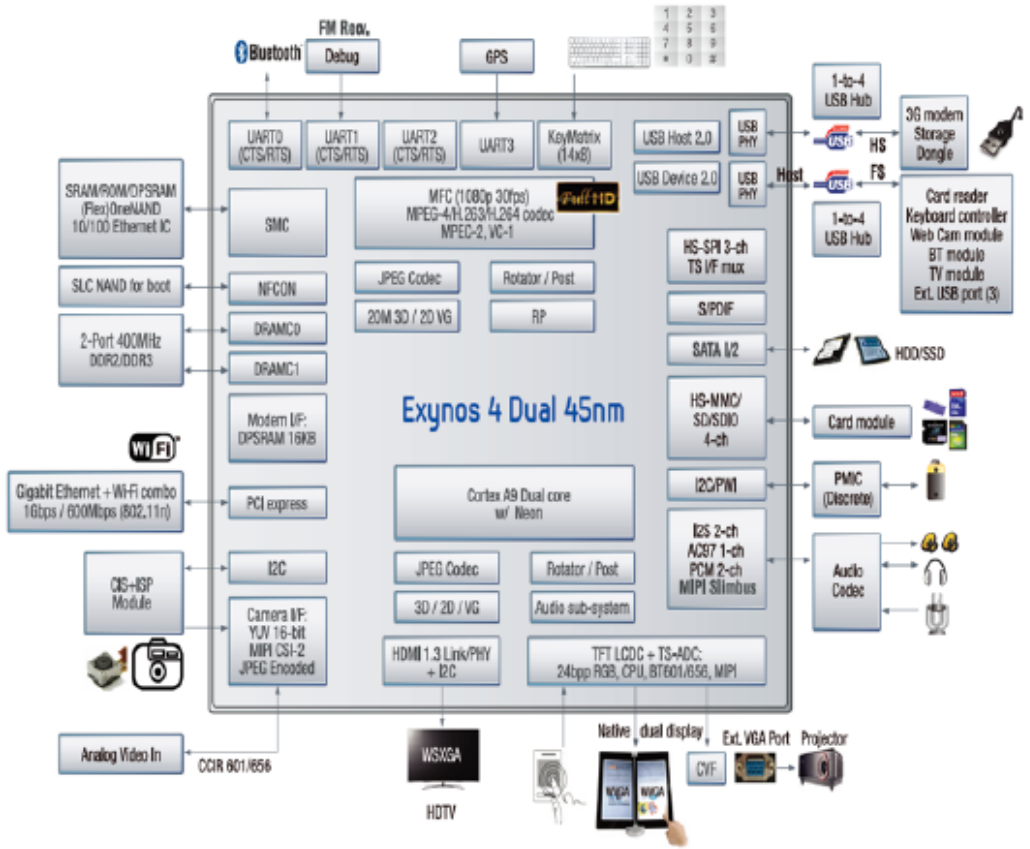
Advance of Civilization

- For humans, Moore's Law scaling of the brain has ended a long time ago
  - Number of neurons and their firing rate did not change significantly
- Remarkable advancement of civilization via **specialization**



Computers are Following the Same Path

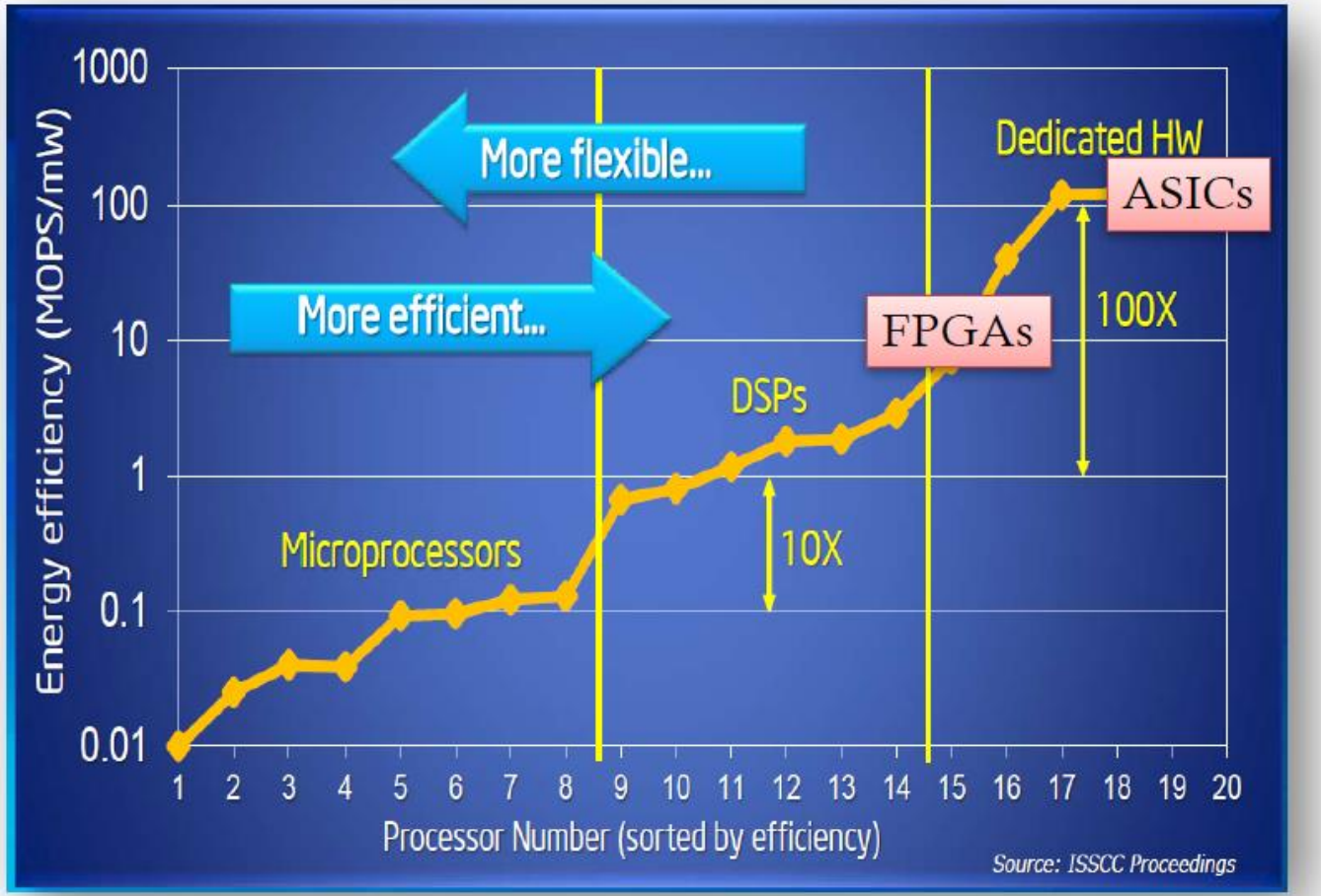
System on chip (SoC)



[source: Samsung]

- Modern SoCs integrate a rich set of **specialized accelerators**
  - Speed up critical tasks
  - Reduce power consumption and cost
  - Increase energy efficiency

Hardware Specialization for Higher Efficiency

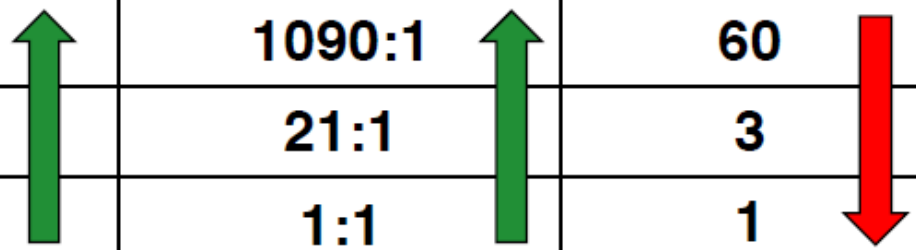


[source: Bob Broderson, Berkeley Wireless group]

Development Effort of Hardware Accelerators

- Comparative study on Monte Carlo option pricing:
  - Data for a single option pricing, using 524,288 simulation paths

Platform	Normalized Speed-Up	Normalized Performance/ Watt	Development Time in Days
FPGA	545:1	1090:1	60
GPU	50:1	21:1	3
GPP	1:1	1:1	1



Source: "Reconfigurable Computing in the Multi-Core Era," Khaled Benkrid, HEART'2010.

Conventional hardware design practice requires extensive hand-coding in RTL and manual tuning

RTL Verilog vs. Untimed C

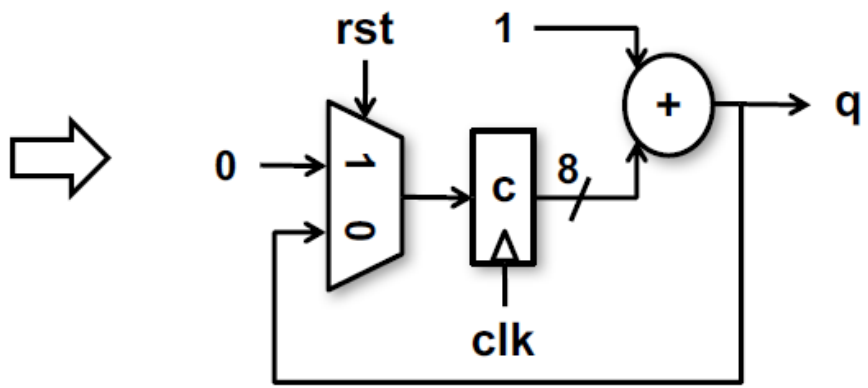
```
module dut(rst, clk, q);
  input rst;
  input clk;
  output q;
  reg [7:0] c;

  always @ (posedge clk)
  begin
    if (rst == 1b'1) begin
      c <= 8'b00000000;
  end
  else begin
    c <= c + 1;
  end

  assign q = c;
endmodule
```

RTL Verilog

vs.

```
uint8 dut() {
  static uint8 c;
  c+=1;
}
```

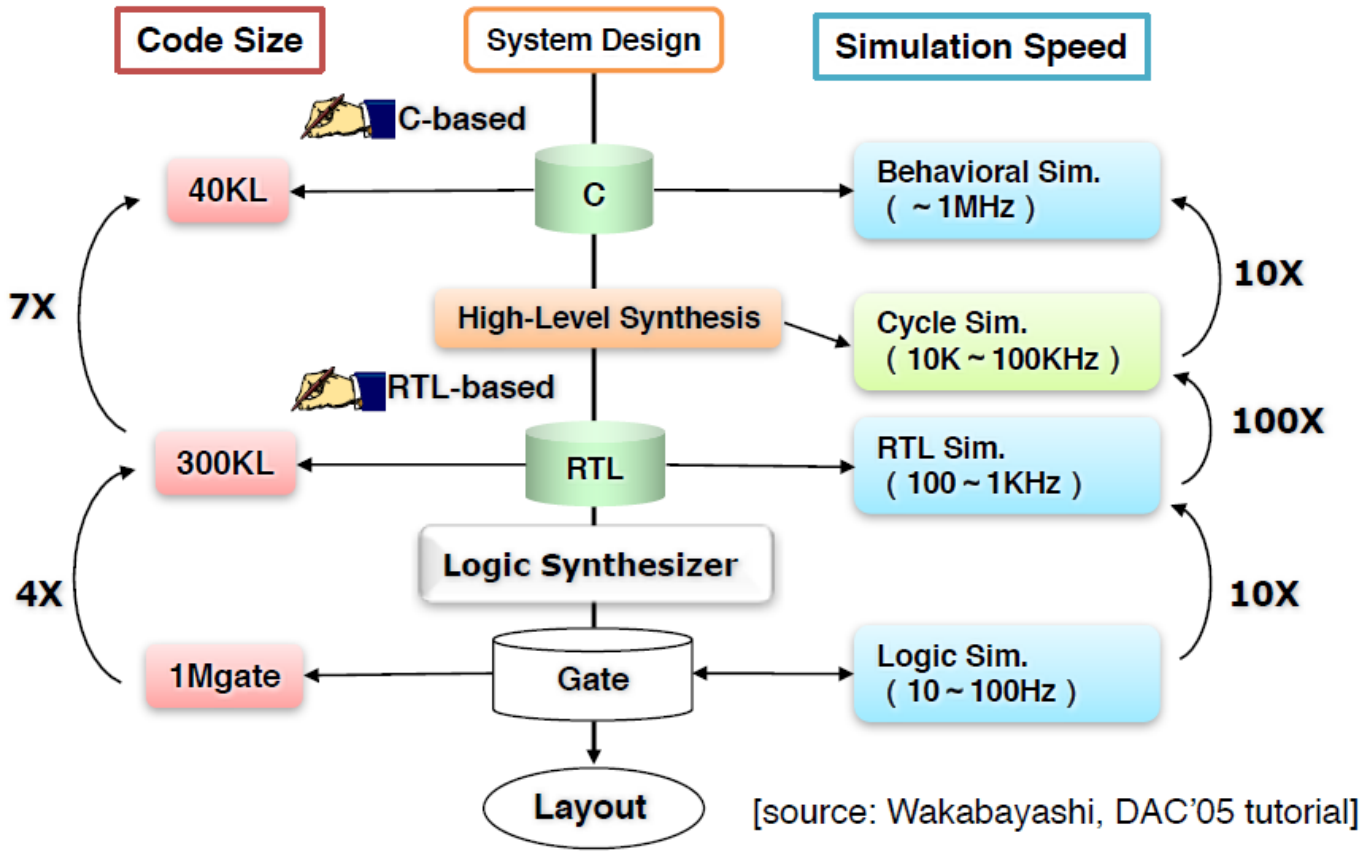
High-Level Synthesis



An 8-bit counter

High-Level Design Automation to Manage Design Complexity

- Significant code size reduction
- Shorter simulation/verification cycle

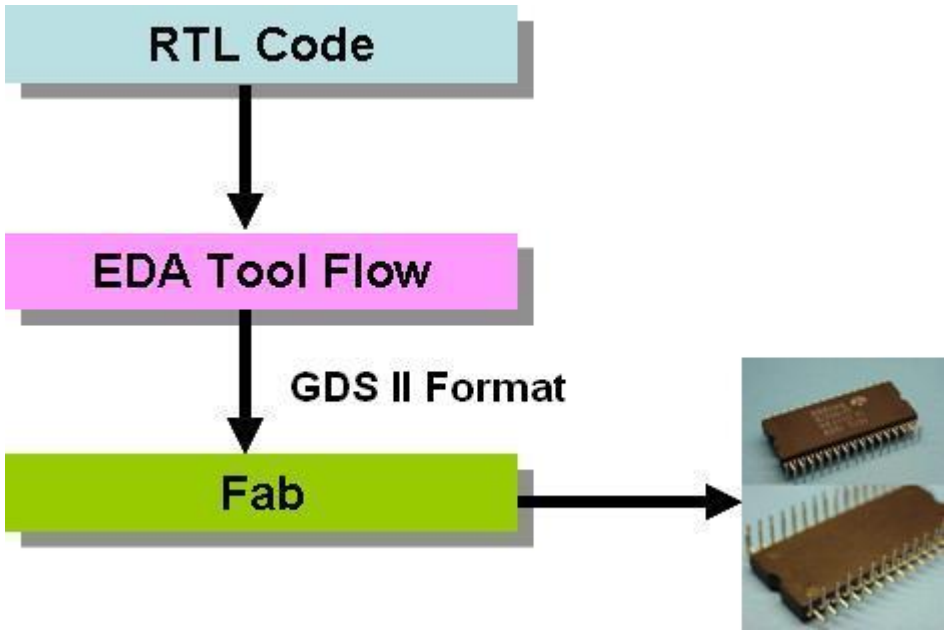


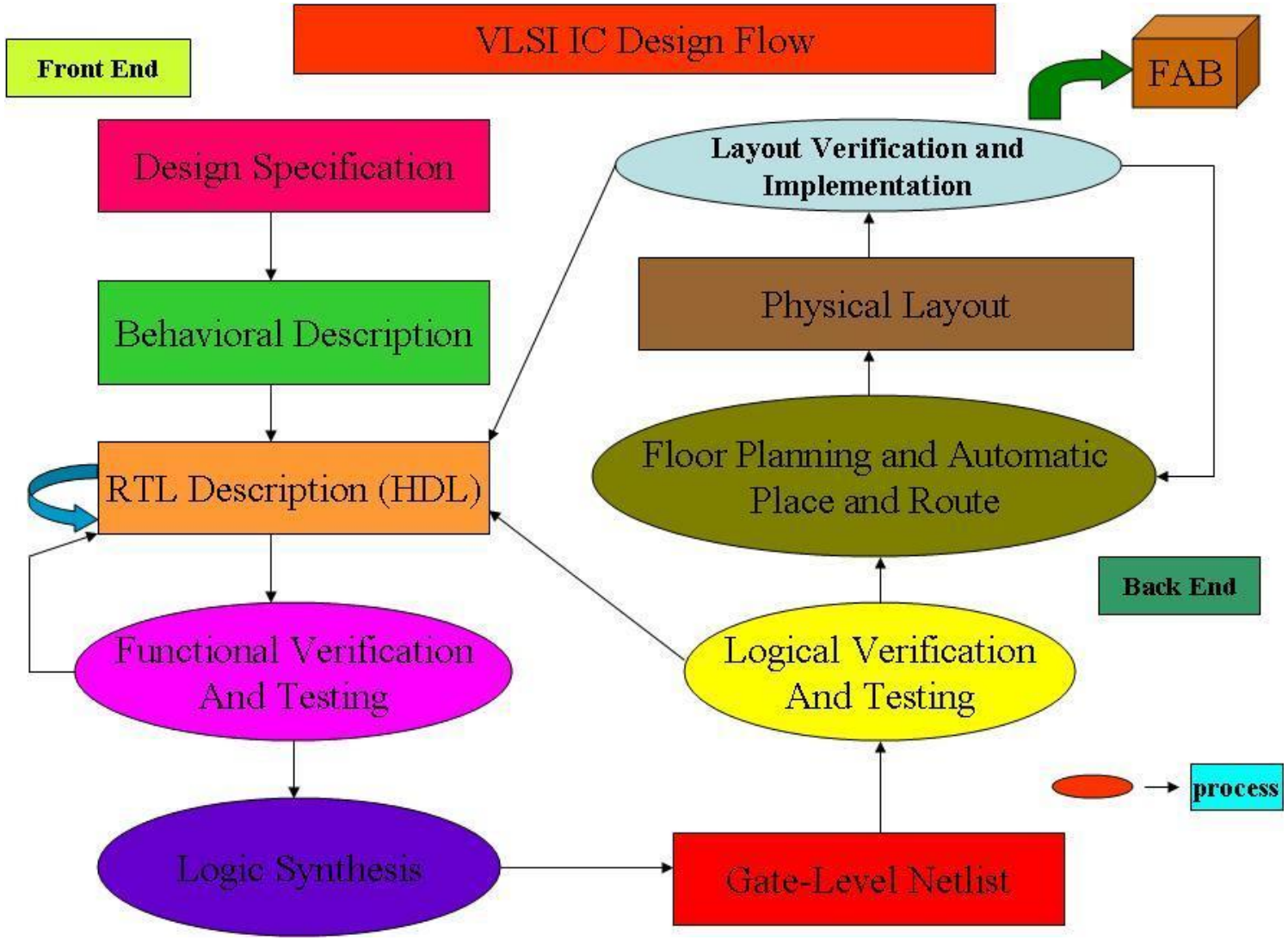
[source: Wakabayashi, DAC'05 tutorial]

ML and EDA

Electronic Design Automation Flow

- Synthesis
- Verification
- Test

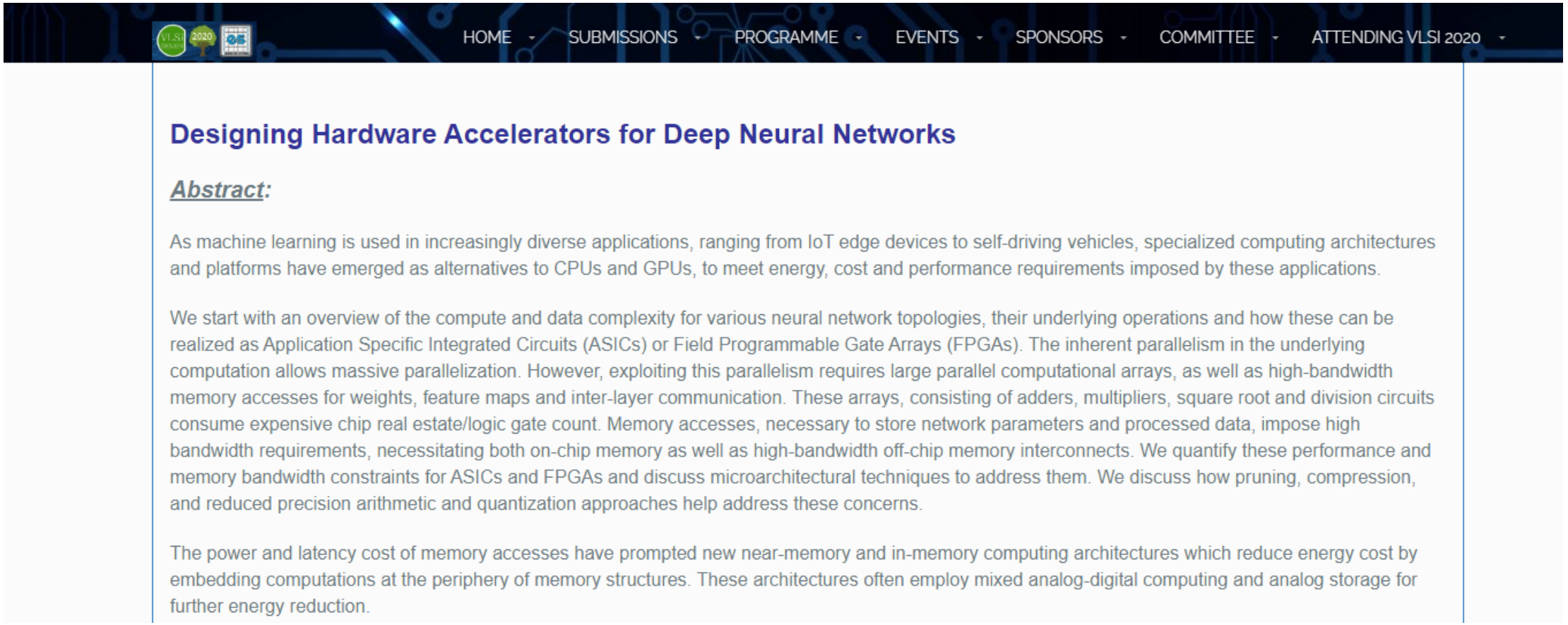




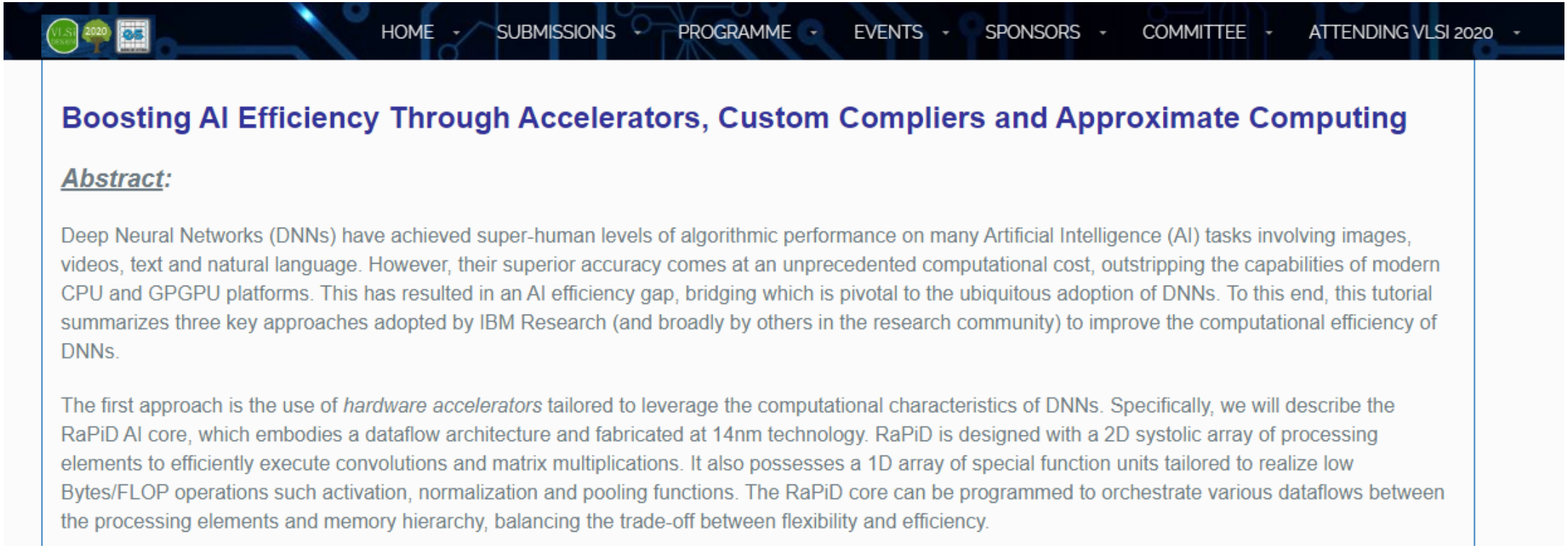
Two Directions of Works

- Use machine learning algorithms to improve the synthesis steps
  - Million of gates
  - Most of the synthesis problems are NP complete
  - Synthesis steps are heuristic based.
  - May not work well with new scenarios
  - I am Skeptical about the impact of ML in EDA.
- Use EDA to improve execution of execution of machine learning algorithms
  - Use EDA tools to develop AI/ML specific processor
  - Arm Machine Learning Processor
  - A **tensor processing unit** (**TPU**) is an AI accelerator application-specific integrated circuit (ASIC) developed by Google.
  - Automatic synthesis of ML algorithms to FPGA
  - FPGA implementations of ML applications can often run much faster that software implementations and can consume significantly less power.
  - Quite promising area of research.

VLSID 2020


HOME
SUBMISSIONS
PROGRAMME
EVENTS
SPONSORS
COMMITTEE
ATTENDING VLSI 2020
Designing Hardware Accelerators for Deep Neural Networks
Abstract:
As machine learning is used in increasingly diverse applications, ranging from IoT edge devices to self-driving vehicles, specialized computing architectures and platforms have emerged as alternatives to CPUs and GPUs, to meet energy, cost and performance requirements imposed by these applications.
We start with an overview of the compute and data complexity for various neural network topologies, their underlying operations and how these can be realized as Application Specific Integrated Circuits (ASICs) or Field Programmable Gate Arrays (FPGAs). The inherent parallelism in the underlying computation allows massive parallelization. However, exploiting this parallelism requires large parallel computational arrays, as well as high-bandwidth memory accesses for weights, feature maps and inter-layer communication. These arrays, consisting of adders, multipliers, square root and division circuits consume expensive chip real estate/logic gate count. Memory accesses, necessary to store network parameters and processed data, impose high bandwidth requirements, necessitating both on-chip memory as well as high-bandwidth off-chip memory interconnects. We quantify these performance and memory bandwidth constraints for ASICs and FPGAs and discuss microarchitectural techniques to address them. We discuss how pruning, compression, and reduced precision arithmetic and quantization approaches help address these concerns.
The power and latency cost of memory accesses have prompted new near-memory and in-memory computing architectures which reduce energy cost by embedding computations at the periphery of memory structures. These architectures often employ mixed analog-digital computing and analog storage for further energy reduction.

VLSID 2020


HOME
SUBMISSIONS
PROGRAMME
EVENTS
SPONSORS
COMMITTEE
ATTENDING VLSI 2020
Boosting AI Efficiency Through Accelerators, Custom Compliers and Approximate Computing
Abstract:
Deep Neural Networks (DNNs) have achieved super-human levels of algorithmic performance on many Artificial Intelligence (AI) tasks involving images, videos, text and natural language. However, their superior accuracy comes at an unprecedented computational cost, outstripping the capabilities of modern CPU and GPGPU platforms. This has resulted in an AI efficiency gap, bridging which is pivotal to the ubiquitous adoption of DNNs. To this end, this tutorial summarizes three key approaches adopted by IBM Research (and broadly by others in the research community) to improve the computational efficiency of DNNs.
The first approach is the use of hardware accelerators tailored to leverage the computational characteristics of DNNs. Specifically, we will describe the RaPiD AI core, which embodies a dataflow architecture and fabricated at 14nm technology. RaPiD is designed with a 2D systolic array of processing elements to efficiently execute convolutions and matrix multiplications. It also possesses a 1D array of special function units tailored to realize low Bytes/FLOP operations such activation, normalization and pooling functions. The RaPiD core can be programmed to orchestrate various dataflows between the processing elements and memory hierarchy, balancing the trade-off between flexibility and efficiency.

DATE 2020


DATE 20
Design, Automation and Test in Europe Conference
Call for Papers
Venue and Hotels
Registration
Exhibition
Home » M08 An industry approach to deploying deep learning network on FPGA
M08 An industry approach to deploying deep learning network on FPGA
Start: Monday, 9 March 2020 14:00
End: Monday, 9 March 2020 18:00

ASP-DAC 2019

Tutorial-5: Monday, January 21, 9:30 - 12:30 @Room Mercury

Machine Learning in Test

Organizer and Speaker:

Yu Huang (Mentor, A Siemens Business)

Abstract:

Machine learning has become a very hot topic in recent years due to its successful application in diverse areas such as computer vision, natural language processing, and intelligent gaming. Exciting new applications, such as autonomous driving, robotics, and AI assisted medical diagnosis, continue to emerge on a regular basis. However, machine learning has still not made much headway in the area of IC testing, since most researchers are unfamiliar with the underlying theory and algorithms, and are unsure of how to apply these techniques in the test domain. We believe machine learning is a powerful and innovative new technology, and can make a significant difference in the area of testing in the near future. This tutorial will review the basics of machine learning, its applications in testing, and forecast future applications of machine learning in testing. It will include the following two parts that are presented in an interleaving manner around each topic:
In the first part, we will provide the background necessary to understand the applications of machine learning in testing. We will start by covering the basics of

ASP-DAC 2019

Tutorial-9: Monday, January 21, 14:00 - 17:00 @Room Mars

Machine Learning for Reliability of ICs and Systems

Organizer:

Mehdi B. Tahoori (Karlsruhe Institute of Technology)

Speakers:

Mehdi B. Tahoori (Karlsruhe Institute of Technology)

Krishnendu Chakrabarty (Duke University)

Abstract:

With increasing the complexity of digital systems and the use of advanced nanoscale technology nodes, various process and runtime variabilities threaten the correct operation of these systems. The interdependence of these reliability detractors and their dependencies to circuit structure as well as running workloads makes it very hard to derive simple deterministic models to analyze and target them. As a result, machine learning techniques can be used to extract useful information which can be used to effectively monitor and improve the reliability of digital systems. These learning schemes are typically performed offline on large data sets in order to obtain various regression models which then are used during runtime operation to predict the health of the system and guide appropriate adaptation and countermeasure schemes. The purpose of this tutorial is to discuss and evaluate various learning schemes in order to analyze the reliability of the ICs and systems due to various runtime failure mechanisms which originate from process and runtime variabilities such as thermal and voltage fluctuations, device and interconnect aging mechanisms, as well as radiation-induced soft errors. The tutorial will also describe how time-series data analytics based on key performance indicators can be used to detect anomalies and predict failure in complex electronic systems. A

DAC 2019

TUTORIAL 4: MACHINE LEARNING IN DIGITAL IC DESIGN AND EDA: LATEST RESULTS AND OUTLOOK

Date: Monday, Jun 3, 2019
Time: 10:30am to 12:00pm
Location: N252
Event Type: Monday Tutorial

Registration Type: Conference, One Day Only Available
Topic Area(s): EDA, Machine Learning/AI
Keyword(s): Front End Design, Back End Design, Implementation

TUTORIAL 5: DESIGNING APPLICATION SPECIFIC AI PROCESSORS

Date: Monday, Jun 3, 2019
Time: 10:30am to 3:00pm
Location: N254
Event Type: Monday Tutorial

Registration Type: Conference, One Day Only Available
Topic Area(s): Machine Learning/AI, Design
Keyword(s): Emerging Technologies, Architecture & System Design, Any

MICROSOFT: ACCELERATING SILICON DESIGN WITH CLOUD AND AI/ML

Date: Monday, Jun 3, 2019
Time: 11:00am to 11:45am
Location: Design-On-Cloud Pavilion
Event Type: Design-on-Cloud Pavilion

Registration Type: Conference, Designer/IP Special, One Day Only Available, I Love DAC
Keyword(s): Any

Course Overview

This course will help the students to understand

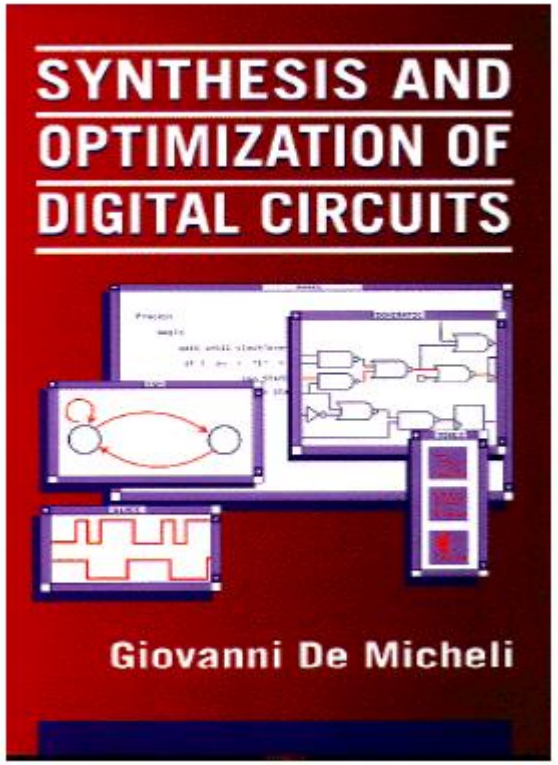
- the overall C to RTL synthesis flow,
- how a C-code will be converted to its equivalent hardware
- how to write C-code for efficient hardware generation,
- how the common software compiler optimizations can help to improve the circuit performance.
- Hardware Acceleration of Machine Learning Algorithm
- Secure Hardware generation using HLS
- Equivalence checking between C and RTL.
- The overall EDA tool flow.
- Design Optimizations

Pre-requisites:

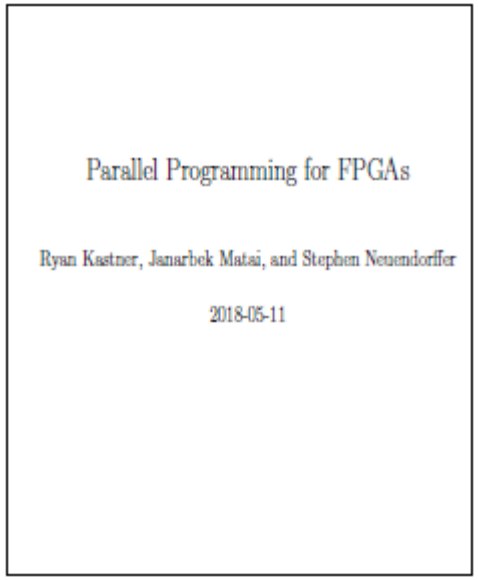
- Basic knowledge of digital design
  - Register, Clock, FSM, Gates, etc.
- Basic knowledge of Data structures and algorithms
  - Graph Algorithms
  - Search/sort
  - Shortest paths etc

Text Books:

- G. De Micheli. Synthesis and optimization of digital circuits, McGraw Hill, India Edition



- Ryan Kastner, Janarbek Matai, and Stephen Neuendorffer, Parallel Programming for FPGAs, 2018.n, 2003.
- Conference and Journal Papers



Reference Books

- Steve Kilts, Advanced FPGA Design, Wiley, 2007.
- K. Parhi: VLSI Digital Signal Processing Systems: Design and Implementation, Jan 1999, Wiley.
- D. D. Gajski, N. D. Dutt, A.C.-H. Wu and S.Y.-L. Lin, High-Level Synthesis: Introduction to Chip and System Design, Springer, 1st edition, 1992
- Mike Fingeroff, High-Level Synthesis Blue Book, Mentor Graphics Corporation, 2010.

Grading

- **Examination: 60%**
  - Four modules with an examination for each module. Each module has 15% weightage.
- **Project: 20%.**
  - A group of 4/5 students to be assigned to a project.
- **Class participation: 10%**
  - There will be pop quizzes during lectures session.
  - Your attendance in the class will also be monitored.
- **Student-led discussion: 10%**.
  - A group of 4/5 students to be assigned to a topic (The group will be same as project group). There will be a student-led presentation for each group on the assigned research topics.

Last two years' Grading summary

2021: Number of Students: 84

- ??

2020: Number of Students: 94

- **AS: 5, AA: 25, AB: 12, BB: 32, BC: 12, C: 1**

2019: Number of Students: 52

- **AS: 1, AA: 6, AB: 10, BB: 6, BC: 11, CC: 8, CD: 5, DD: 4**

Course feedback so far

		Average*	Average*
	About the Instructor :		
1.	Overall, the teaching by the instructor was excellent	4.45	4.28
2.	The concepts were explained with clarity	4.50	4.23
3.	Questions and discussions were encouraged.	4.52	4.32
4.	Allotted number of classes was held	4.65	4.32
5.	Evaluation was done regularly and feedback was given.	4.55	4.10
	About the Course:		
1	The course was highly enjoyable.	4.40	3.91
2	The content of the course was appropriate.	4.50	4.11
3	Text/Reference materials were appropriate for the course.	4.48	4.12

Teachers

Chandan Karfa

Debabrata Senapati

Priyanka Panigrahi

Modammed Abderahman

Arshdeep Kaur

Melbin John

Rupak Gupta

Jayprakash Patidar

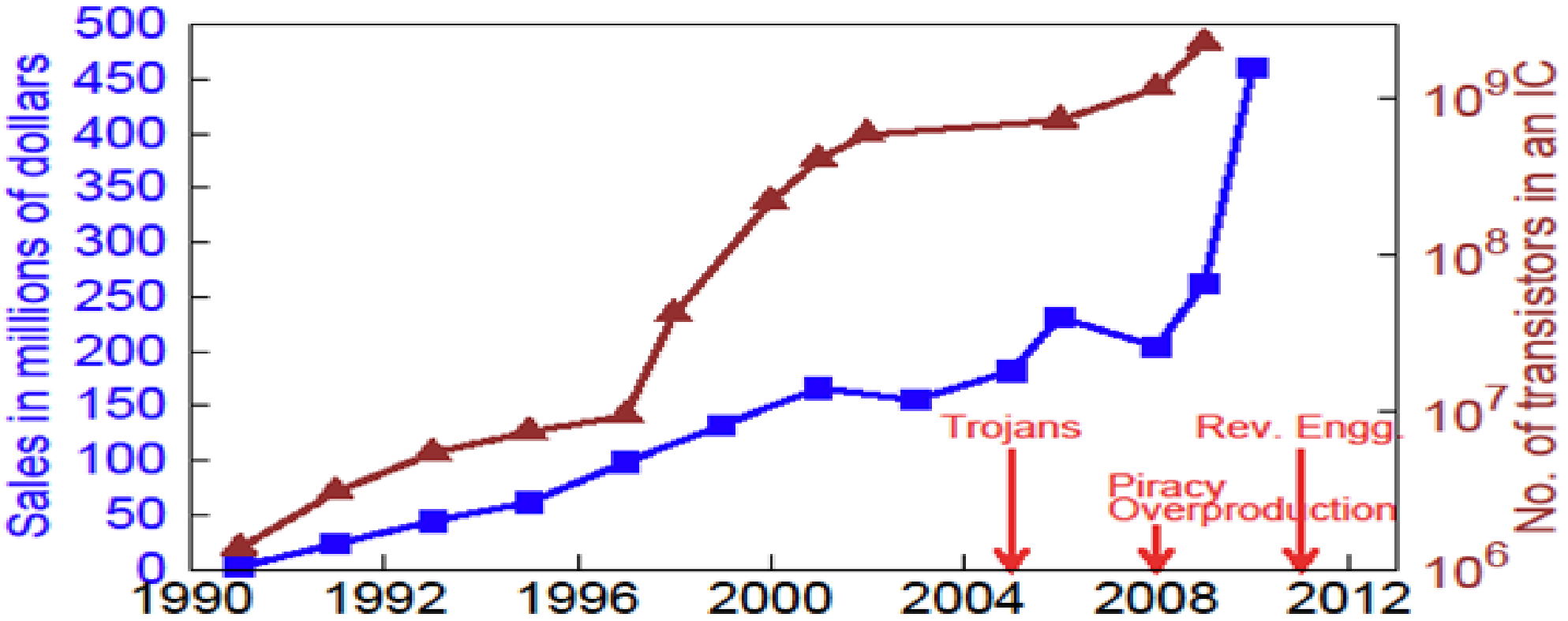
Acknowledgements

- These slides contain/adapt materials developed by
  - – Prof. Jason Cong (UCLA)
  - - Prof. Zhiru Zhang (Cornel)

Thank You

High-level Synthesis - Overview

High-Level Synthesis (HLS)/C-Based VLSI Design



- C ➔ Gates
- Design time ↓
- Design complexity ↓ (10x)
- Verification effort ↓
- Hardware/software co-design ↑

Synfora ↔ SYNOPSYS®

XILINX®

intel FPGA

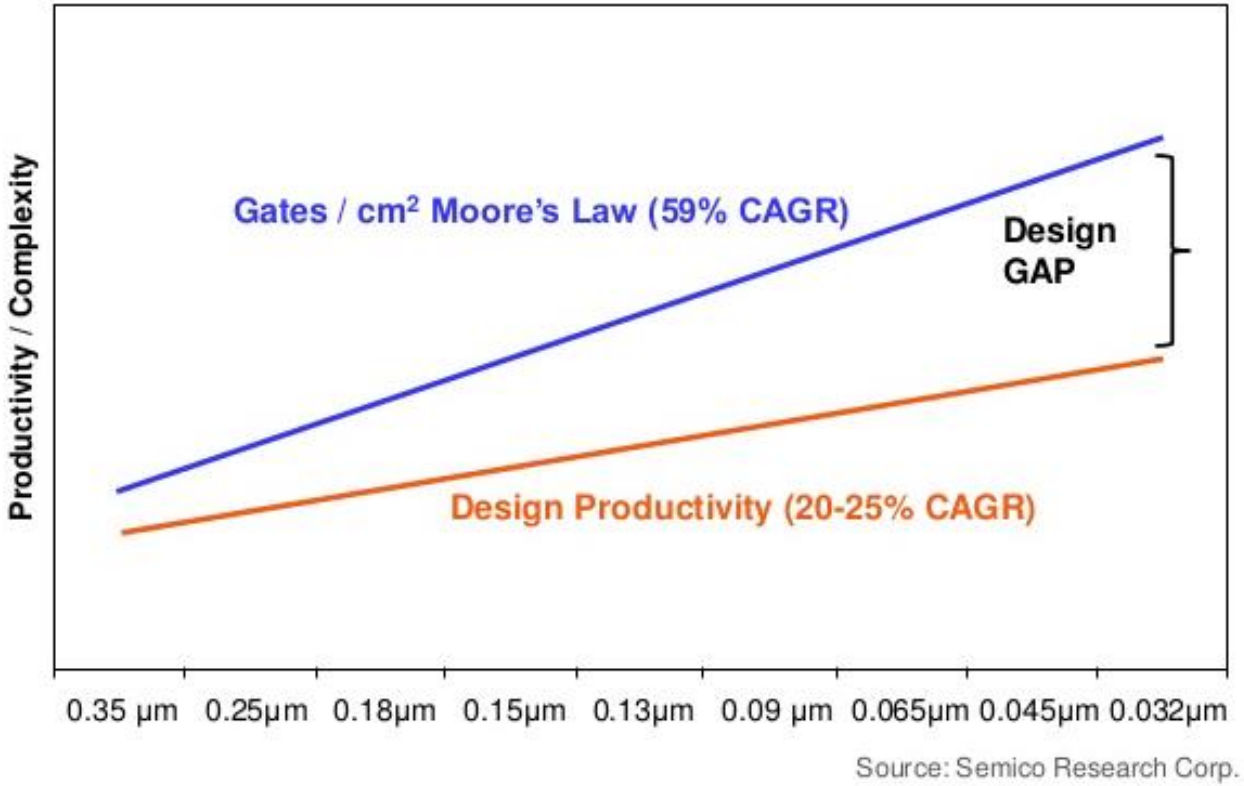
Oasys ↔ Mentor Graphics®

CALYPTO Design - Optimize - Verify ↔

bluespec

FORTE DESIGN SYSTEMS ↔ cadence™

HLS is a Productivity Tool



- Raises abstraction from RTL to C/C++; allows early design exploration
- Applications: image processing, video compression, vision, crypto, ML

HLS

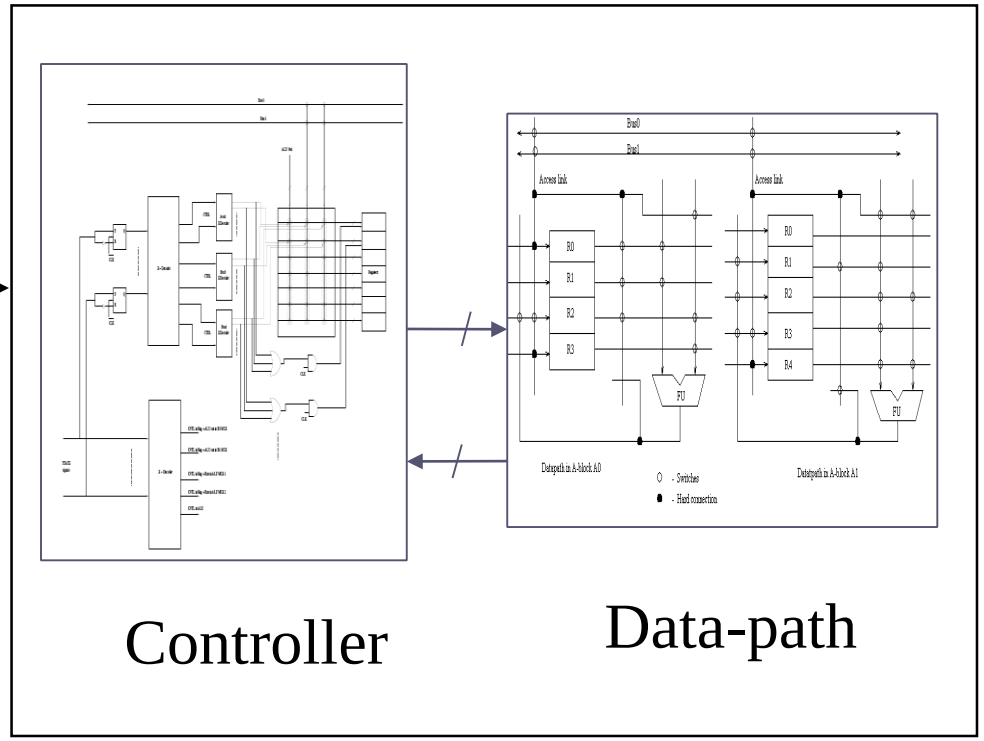
- Enables designs at higher abstraction level (e.g., C, C++, Java)
- 14 out of the top-20 semiconductor companies use HLS tools
- Communications, signal processing, computation, crypto, healthcare, etc.
- Tailor implementation to match characteristics of target technology (e.g., speed, resources, area budget)
  - Video components in Tegra X1 chip designed using Catapult HLS.
  - NVIDIA 4K processing was designed with C-based HLS.
  - Qualcomm designing parts of Snapdragon with Catapult HLS.
  - Vivado HLS is part of xilinx design flow
  - intel HLS is part of quartus design flow

High-level Synthesis

```
while (x_var < a_var) loop
    t1 := u_var * dx_var;
    t2 := 3 * x_var;
    t3 := 3 * y_var;
    t4 := t1 * t2;
    t5 := dx_var * t3;
    t6 := u_var - t4;
    u_var := t6 - t5;
    y1 := u_var * dx_var;
    y_var := y_var + y1;
    x_var := x_var + dx_var;
 end loop;
X <= x_var;
Y <= y_var;
U <= u_var;
```

High-level Behaviour

HLS



Register Transfer Level Description

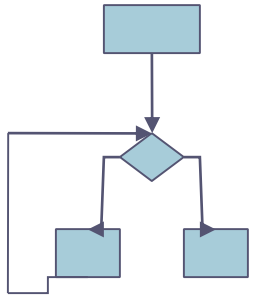
High-level Synthesis Steps

- Preprocessing: Intermediate representation (CDFG) construction, data-dependency, live variable analysis, compiler optimization.
- Scheduling: Assigns control step to the operations of the input behaviour.
- Allocation: Computes minimum number of functional units and registers.
- Binding: Variables are mapped to registers, operation to functional units, data transfers to the interconnection units.
- Data path & Controller design: controller is designed based on inter connections among the data path elements, data transfer required in different control steps.

High-level Synthesis Steps

```
while (x_var < a_var) loop
    t1 := u_var * dx_var;
    t2 := 3 * x_var;
    t3 := 3 * y_var;
    t4 := t1 * t2;
    t5 := dx_var * t3;
    t6 := u_var - t4;
    u_var := t6 - t5;
    y1 := u_var * dx_var;
    y_var := y_var + y1;
    x_var := x_var + dx_var;
 end loop;
X <= x_var;
Y <= y_var;
U <= u_var;
```

pre-processing



Input behaviour

scheduling

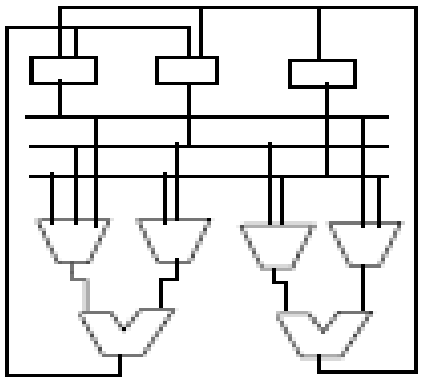
1.	< 1 *>		
2.		<2 + >	<0 * >
3.		<4 * >	| * |
4.		| * |	<3 *>
5.	<7 *>	<6 *>	| * |
6.	| * |	| * |	<5 - >
7.		<9 +>	<8 - >

Allocation & binding

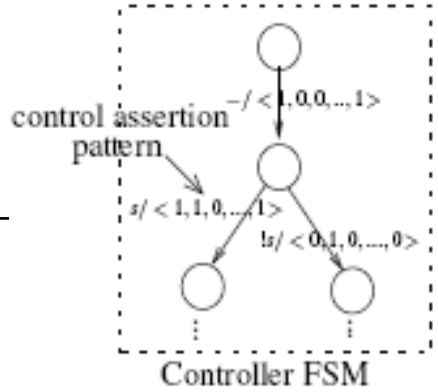
R1 : 3, v1
R2 : x u, v5
R3 : v0, v6
R4 : v3

FU1: op1, on3. ..
FU2: op2, op5, …
FU3: …

Data-path generation



Controller generation



Data-path

status signal

Control signal

Controller

RTL behaviour

High-level Synthesis -- working with an example

Example: 2nd order differential equation solver

```
Diffeq: (x, dx, u, a, clock, y)
input: x, dx, u, a, clock;
output: y
while(x < a)
        u = u-(3*x*u*dx)-(3*y*dx)
        y = y+(u*dx)
        x = x+dx
end
```

Preprocessing

Example: 2nd order differential equation solver

```
Diffeq: (x, dx, u, a, clock, y)
input: x, dx, u, a, y;
output: y
while(x < a)
        u = u-(3*x*u*dx)-(3*y*dx)
        y = y+(u*dx)
        x = x+dx
end
```

I

Read(p1, dx)

Read(p2, x)

Read(p3, a)

Read(p1,y)

Read(p2, u)

c = x < a

B2

Write(p1, y)

B_1

$V_1 : t_1 = u * dx$

$V_2 : t_2 = 3 * x$

$V_3 : t_3 = 3 * y$

$V_4 : t_4 = u * dx$

$V_5 : t_5 = t_1 * t_2$

$V_6 : t_6 = t_3 * dx$

$V_7 : t_7 = u - t_5$

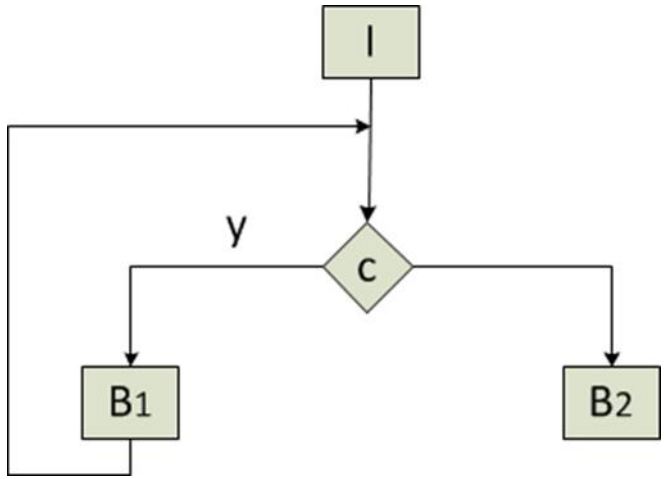
$V_8 : u = t_7 - t_6$

$V_9 : y = y + t_4$

$V_{10} : x = x + dx$

$V_{11} : c = x < a$

Basic Blocks with 3-address codes



Control and Dataflow graph (CDFG)

Preprocessing

B_1

$V_1 : t_1 = u * dx$

$V_2 : t_2 = 3 * x$

$V_3 : t_3 = 3 * y$

$V_4 : t_4 = u * dx$

$V_5 : t_5 = t_1 * t_2$

$V_6 : t_6 = t_3 * dx$

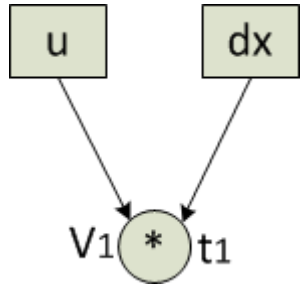
$V_7 : t_7 = u - t_5$

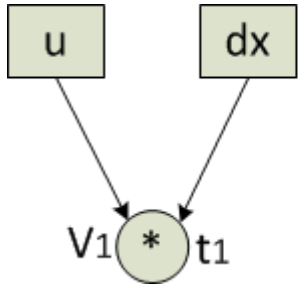
$V_8 : u = t_7 - t_6$

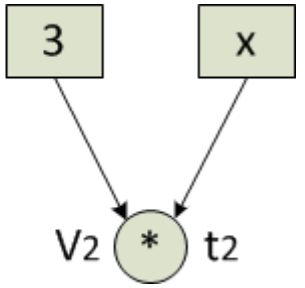
$V_9 : y = y + t_4$

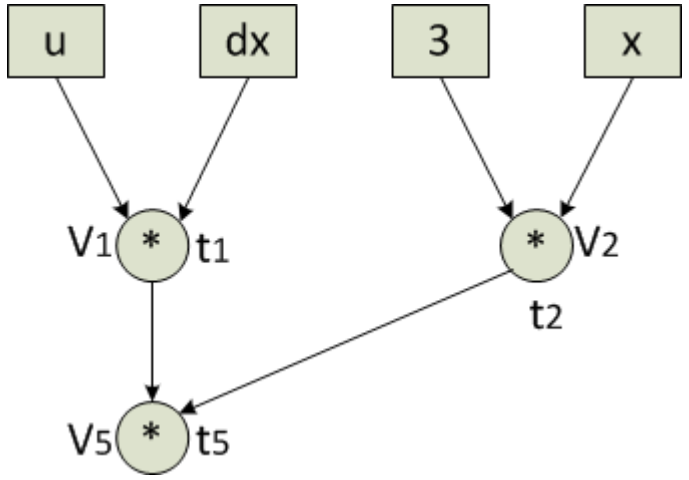
$V_{10} : x = x + dx$

$V_{11} : c = x < a$









Preprocessing

B_1

$V_1 : t_1 = u * dx$

$V_2 : t_2 = 3 * x$

$V_3 : t_3 = 3 * y$

$V_4 : t_4 = u * dx$

$V_5 : t_5 = t_1 * t_2$

$V_6 : t_6 = t_3 * dx$

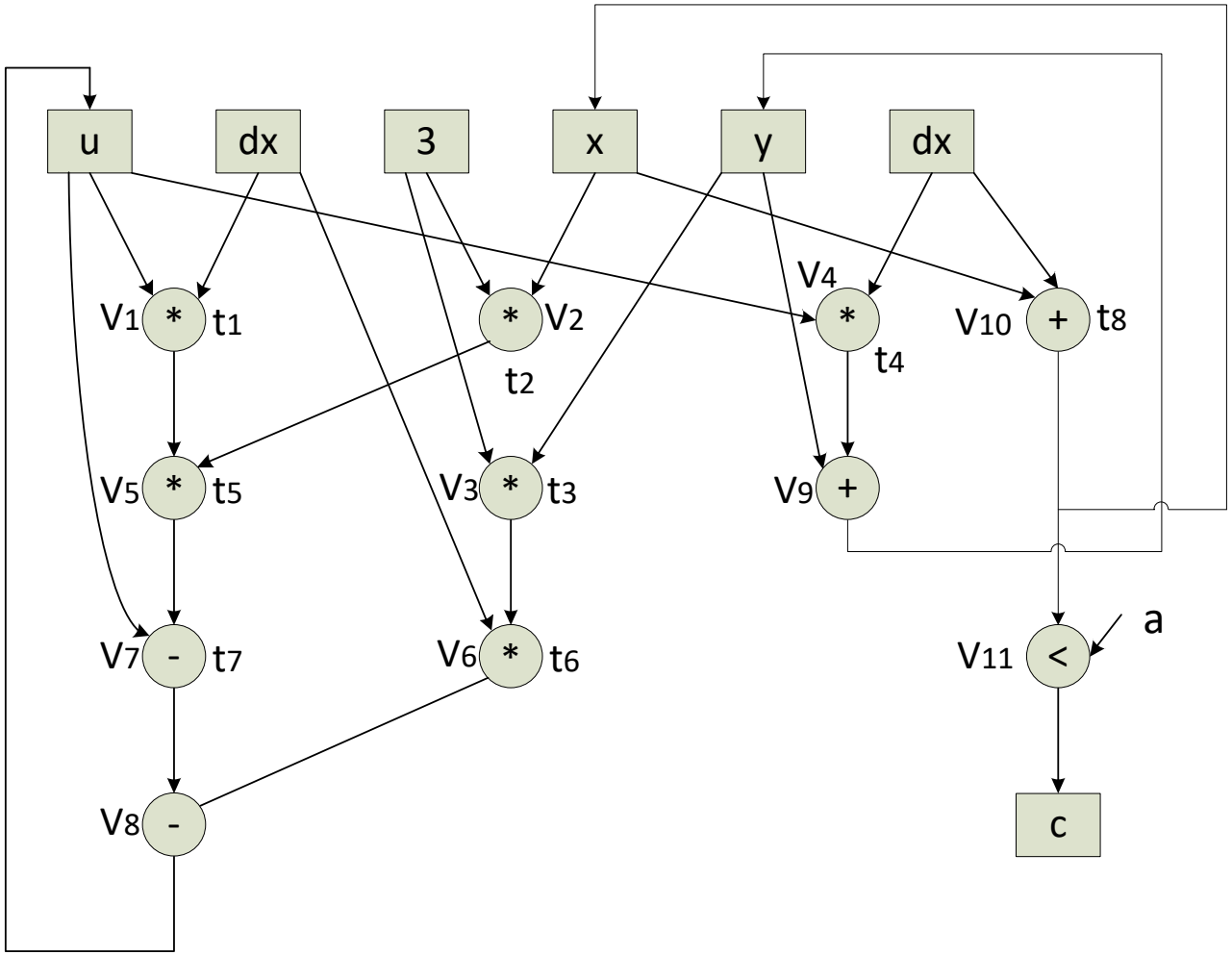
$V_7 : t_7 = u - t_5$

$V_8 : u = t_7 - t_6$

$V_9 : y = y + t_4$

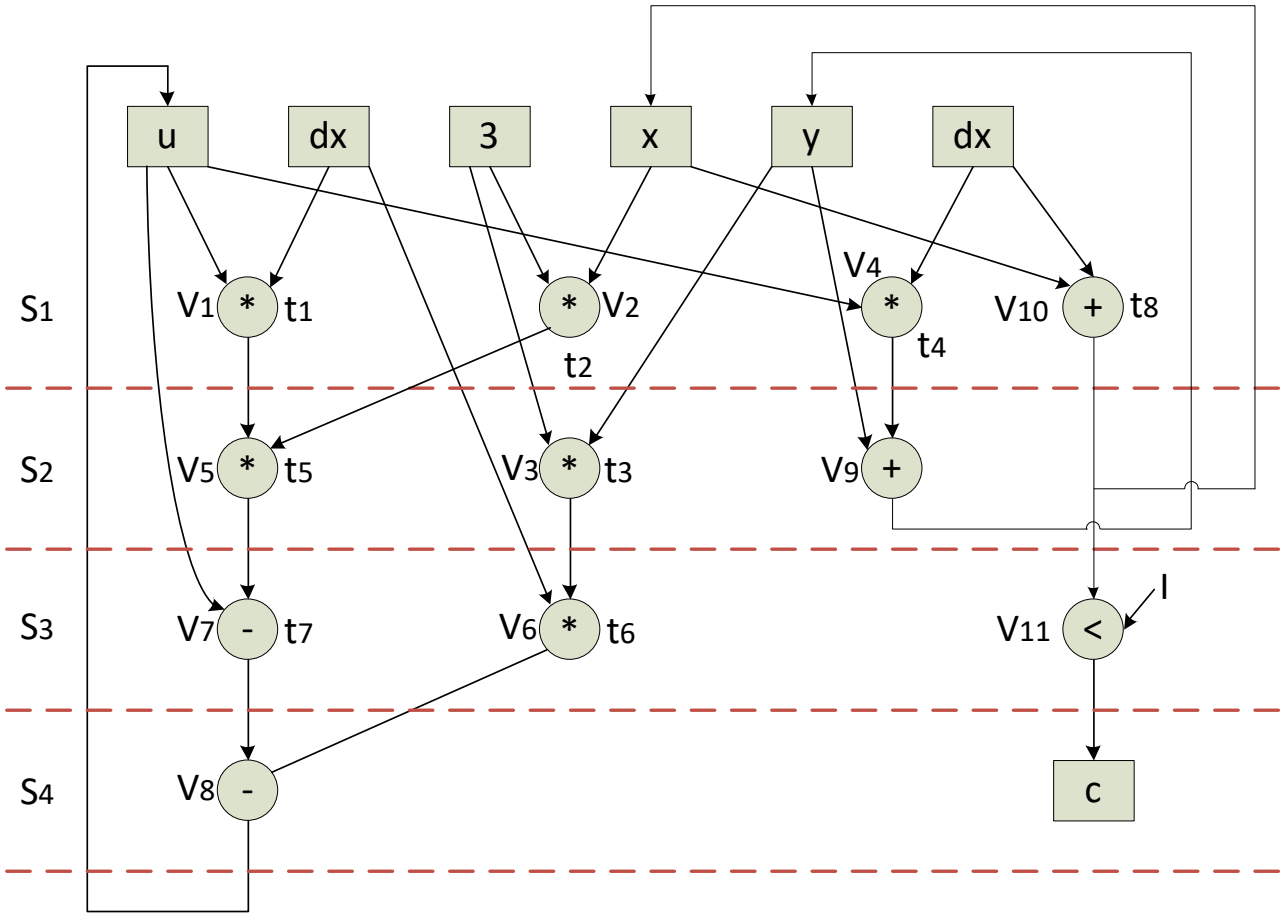
$V_{10} : x = x + dx$

$V_{11} : c = x < a$

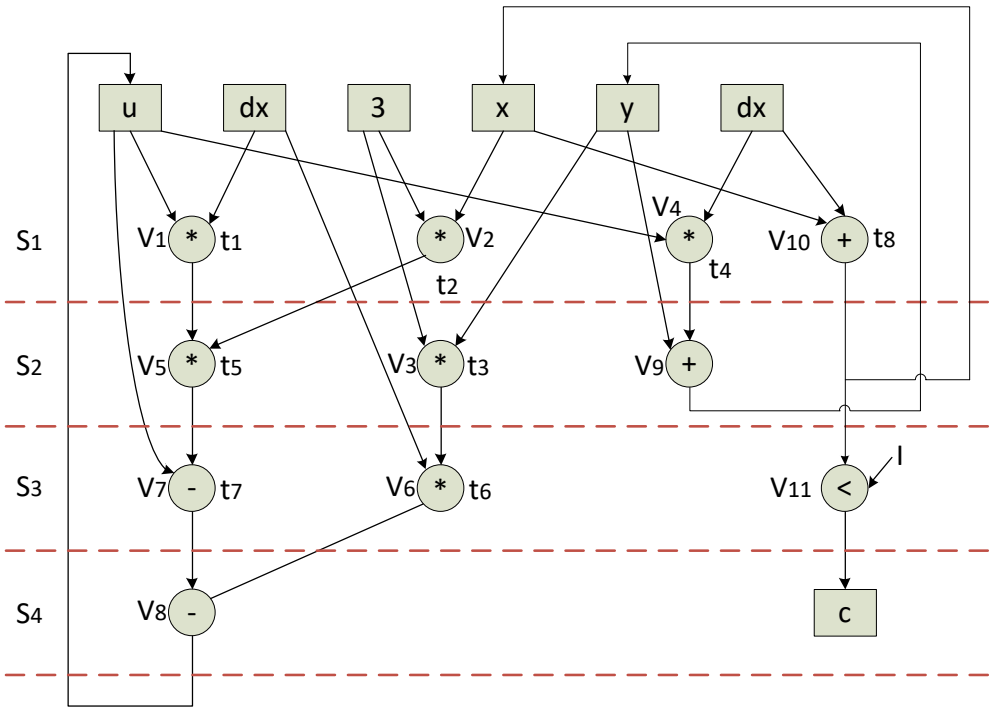


Date dependency graph

Scheduling


u
dx
3
x
y
dx
S1
S2
S3
S4
V1
*
t1
*
V2
t2
V4
*
t4
V10
+
t8
V5
*
t5
V3
*
t3
V9
+
V7
-
t7
V6
*
t6
V11
<
I
V8
-
c

Register Allocation and Binding

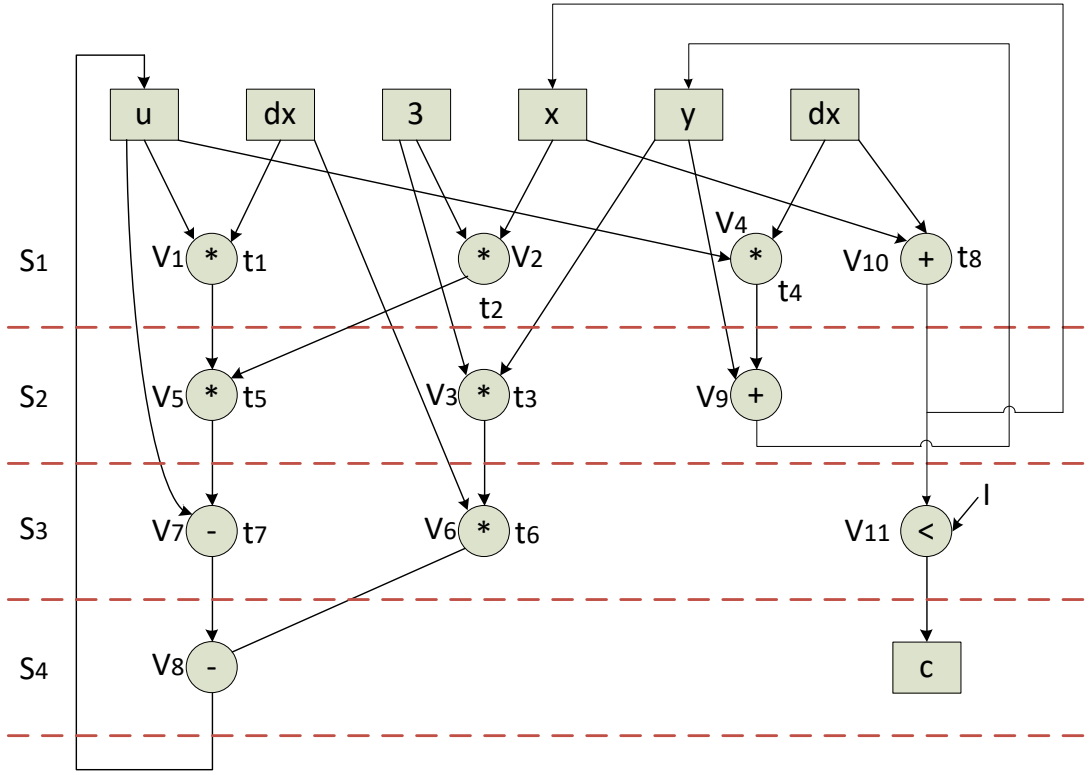


Var	S_1	S_2	S_3	S_4
t1		R1		
t2		R2		
t3			R1	
t4		R3		
t5			R2	
t6				R1
t7				R2
t8		R4		
u	R5			
x	R6			
dx	R7			
y	R8			
c	R9			
3	R10			
a	R11			

Interval graph

R_1: t_1, t_3, t_6
R_2: t_2, t_5, t_7
R_3: t_4
R_4: t_8
R_5: u
R_6: x
R_7: dx
R_8: y
R_9: c
R_{10}: 3
R_{11}: a

FU Allocation and Binding: Multiplier

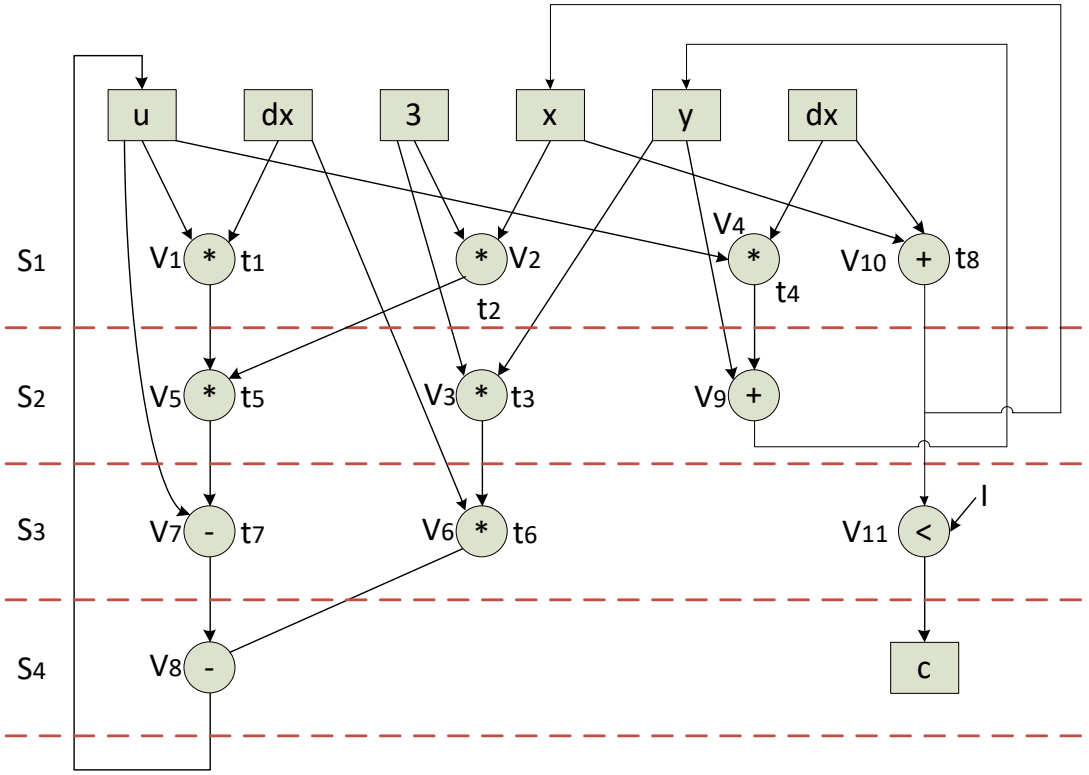


	S_1	S_2	S_3	S_4
V_1	M_1			
V_2	M_2			
V_3		M_2		
V_4	M_3			
V_5		M_1		
V_6			M_3	

Mult operations with non-overlapping schedule can be mapped to the same Multiplier FU

MULT: M_1: V_1, V_5
MULT: M_2: V_2, V_3
MULT: M_3: V_4, V_6

FU Allocation and Binding: Adder



Var	S_1	S_2	S_3	S_4
V_{10}	A_1			
V_9		A_1		
V_7			A_1	
V_8				A_1

Add/Sub operations with non-overlapping schedule can be mapped to the same adder FU

FU allocation and Binding
A1: v10, v9, v7, v6

Functional Unit Allocation and Binding

FU alloc and bind:

MULT: M_1: V_1, V_5

MULT: M_2: V_2, V_3

MULT: M_3: V_4, V_6

ADD: A_1: V_7, $V_{8,}$ V_9, V_{10}

COMP: C_1: V_{11}

Register Transfer Level (RTL) Behaviour

S_1:
$V_1 : t_1 = u * dx$
$V_2 : t_2 = 3 * x$
$V_4 : t_4 = u * dx$
$V_{10} : x = x + dx$

S_2:
$V_5 : t_5 = t_1 * t_2$
$V_3 : t_3 = 3 * y$
$V_9 : y = y + t_4$

Original behaviour

R_1:	t_1, t_3, t_6
R_2:	t_2, t_5, t_7
R_3:	t_4
R_4:	t_8
R_5:	u
R_6:	x
R_7:	dx
R_8:	y
R_9:	c
R_{10}:	3
R_{11}:	a

Register mapping

FU alloc and bind:

MULT: M_1: V_1, V_5

MULT: M_1: V_2, V_3

MULT: M_1: V_4, V_6

ADD: A_1: V_7, V_8, V_9, V_{10}

COMP: C_1: V_{11}

FU mapping

S_1:

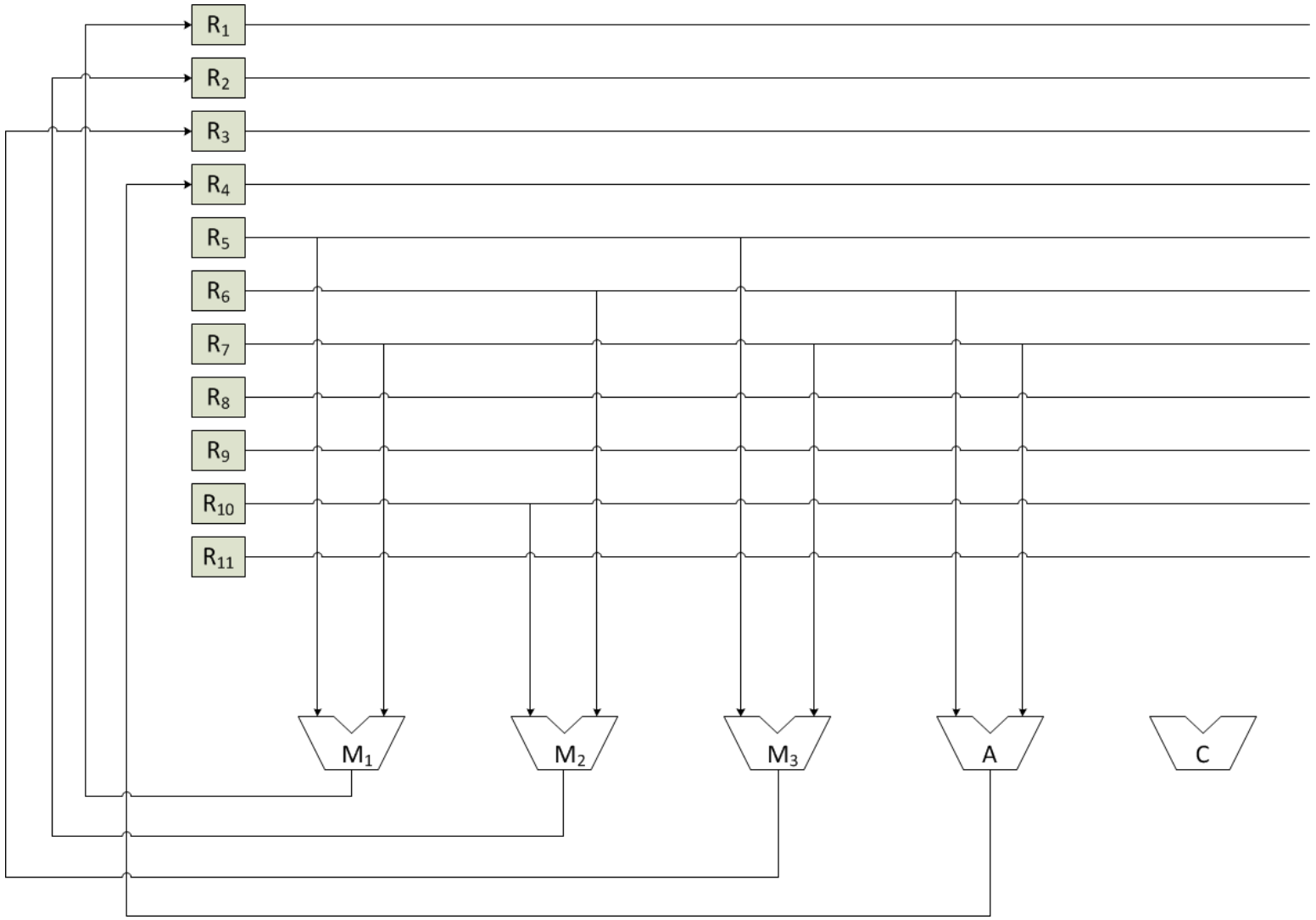
V_1	$R_1 = R_5 <M_1> R_7$
V_2	$R_2 = R_{10} <M_2> R_6$
V_4	$R_3 = R_5 <M_3> R_7$
V_{10}	$R_4 = R_6 <A> R_7$

S_2:

V_5	$R_2 = R_1 <M_1> R_2$
V_3	$R_1 = R_{10} <M_2> R_8$
V_9	$R_8 = R_8 <A> R_3$

RTL behaviour

Data Path Synthesis



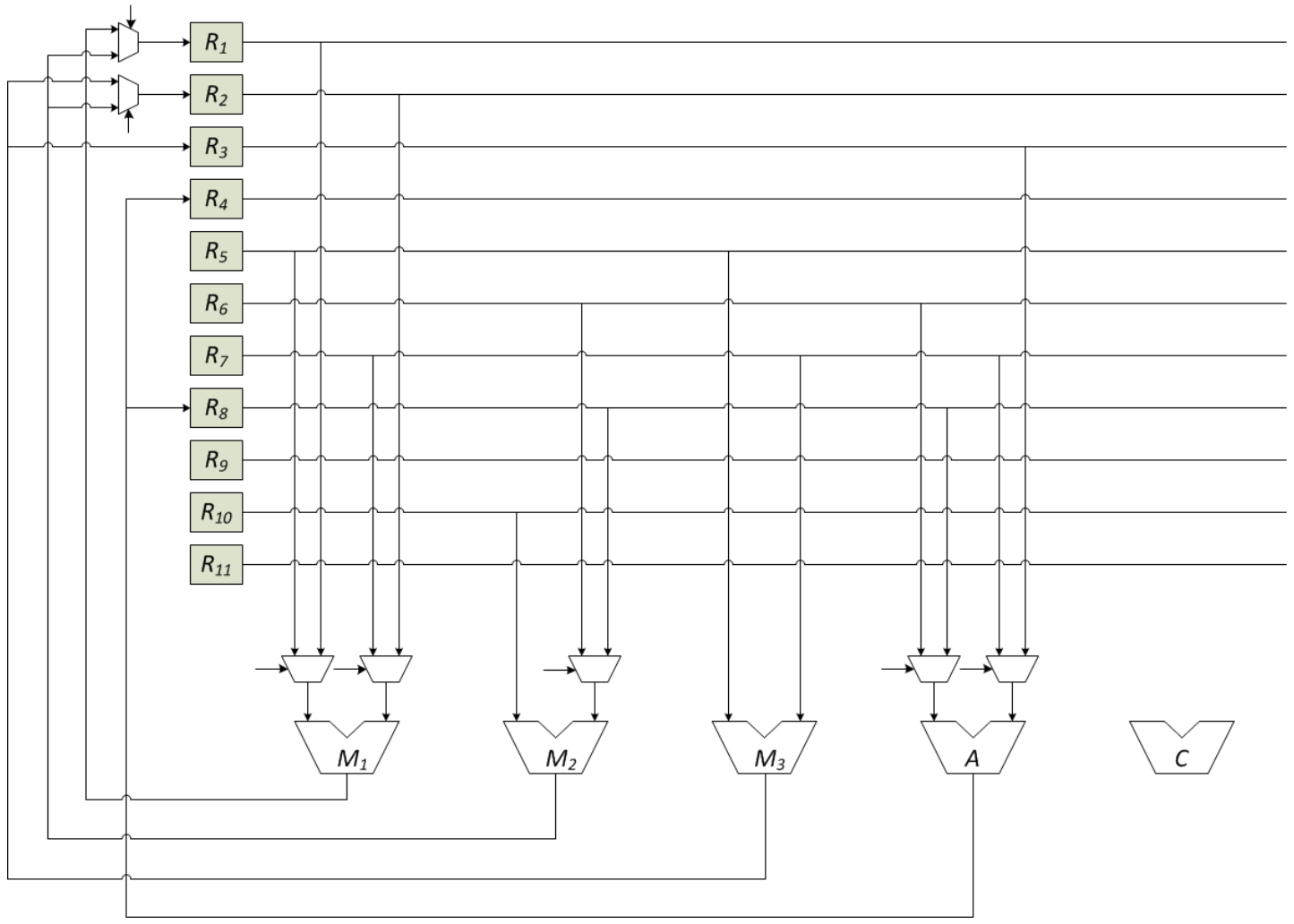
S_1:

V_1	$R_1 = R_5 <M_1> R_7$
V_2	$R_2 = R_{10} <M_2> R_6$
V_4	$R_3 = R_5 <M_3> R_7$
V_{10}	$R_4 = R_6 <A> R_7$

S_2:

V_5	$R_2 = R_1 <M_1> R_2$
V_3	$R_1 = R_{10} <M_2> R_8$
V_9	$R_8 = R_8 <A> R_3$

Data path Synthesis



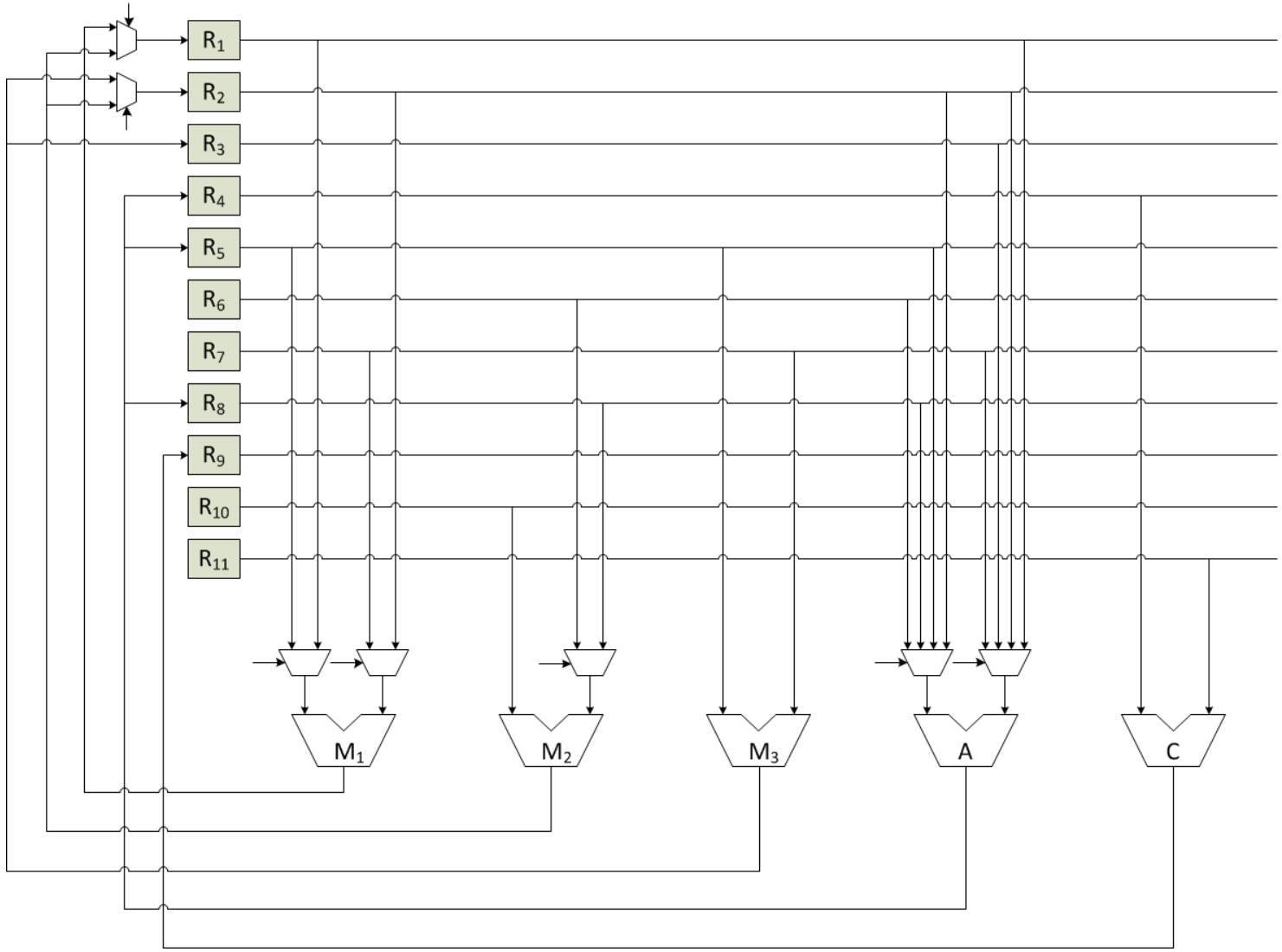
S_1:

V_1	$R_1 = R_5 <M_1> R_7$
V_2	$R_2 = R_{10} <M_2> R_6$
V_4	$R_3 = R_5 <M_3> R_7$
V_{10}	$R_4 = R_6 <A> R_7$

S_2:

V_5	$R_2 = R_1 <M_1> R_2$
V_3	$R_1 = R_{10} <M_2> R_8$
V_9	$R_8 = R_8 <A> R_3$

Data path Generation



Control Signals

Control Assertion Pattern: <FU, FU_MUX_in, Reg-en, Reg_Mux_in>

FU: 1 bit

FU_MUX_in: 7 bits

Reg_en: 11 bits

Reg_MUX_in: 2 bits

Total: 21 bits

S_1: <1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0 0, 0, 1>

S_2: <1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0 0, 1, 0>

S_3: <.....>

S_4: <.....>

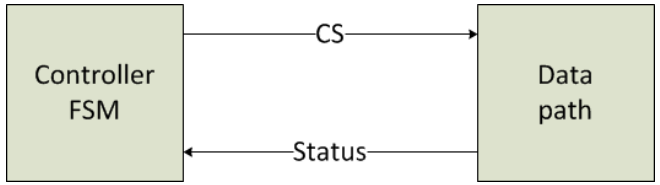
S_1:

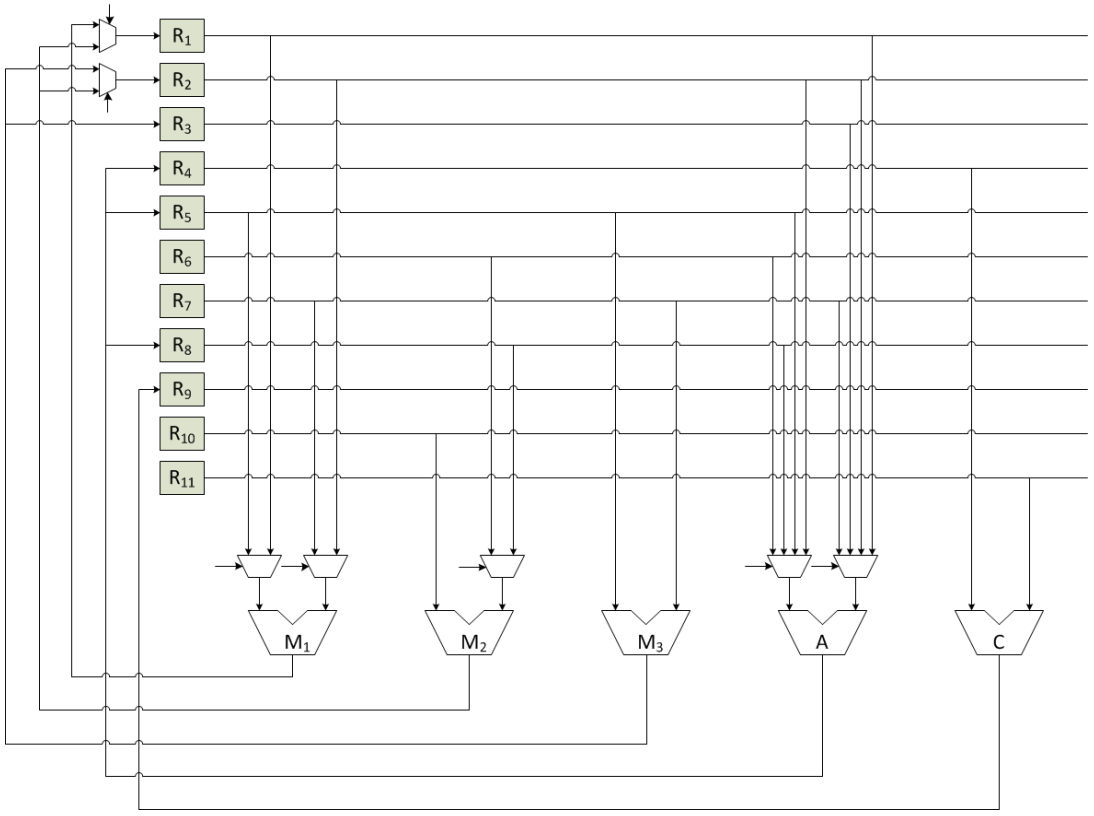
V_1	$R_1 = R_5 <M_1> R_7$
V_2	$R_2 = R_{10} <M_2> R_6$
V_4	$R_3 = R_5 <M_3> R_7$
V_{10}	$R_4 = R_6 <A> R_7$

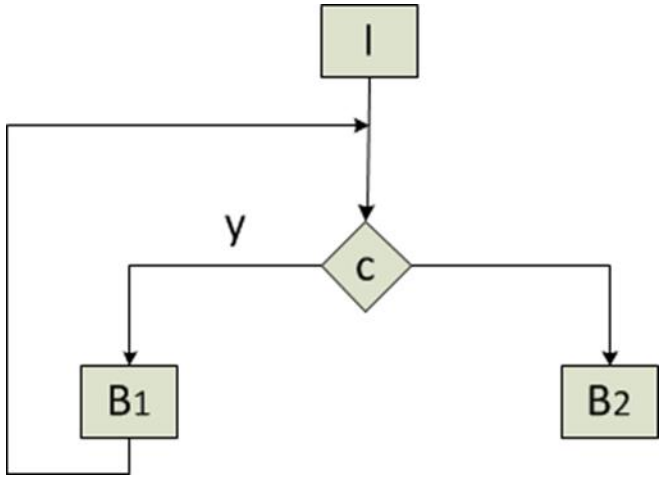
S_2:

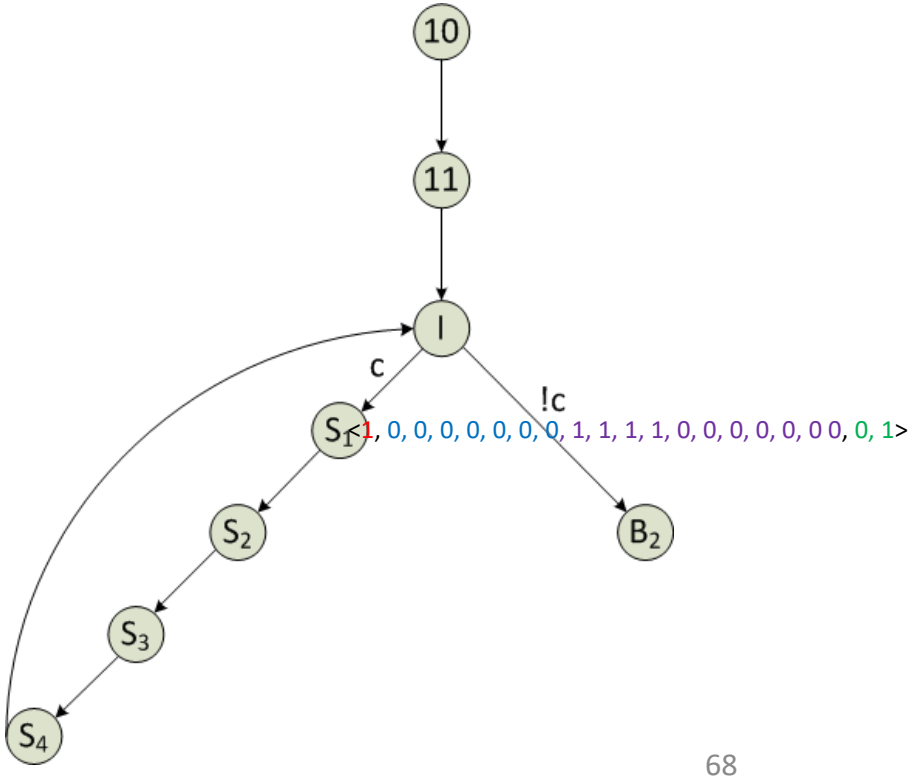
V_5	$R_2 = R_1 <M_1> R_2$
V_3	$R_1 = R_{10} <M_2> R_8$
V_9	$R_8 = R_8 <A> R_3$

Final RTL

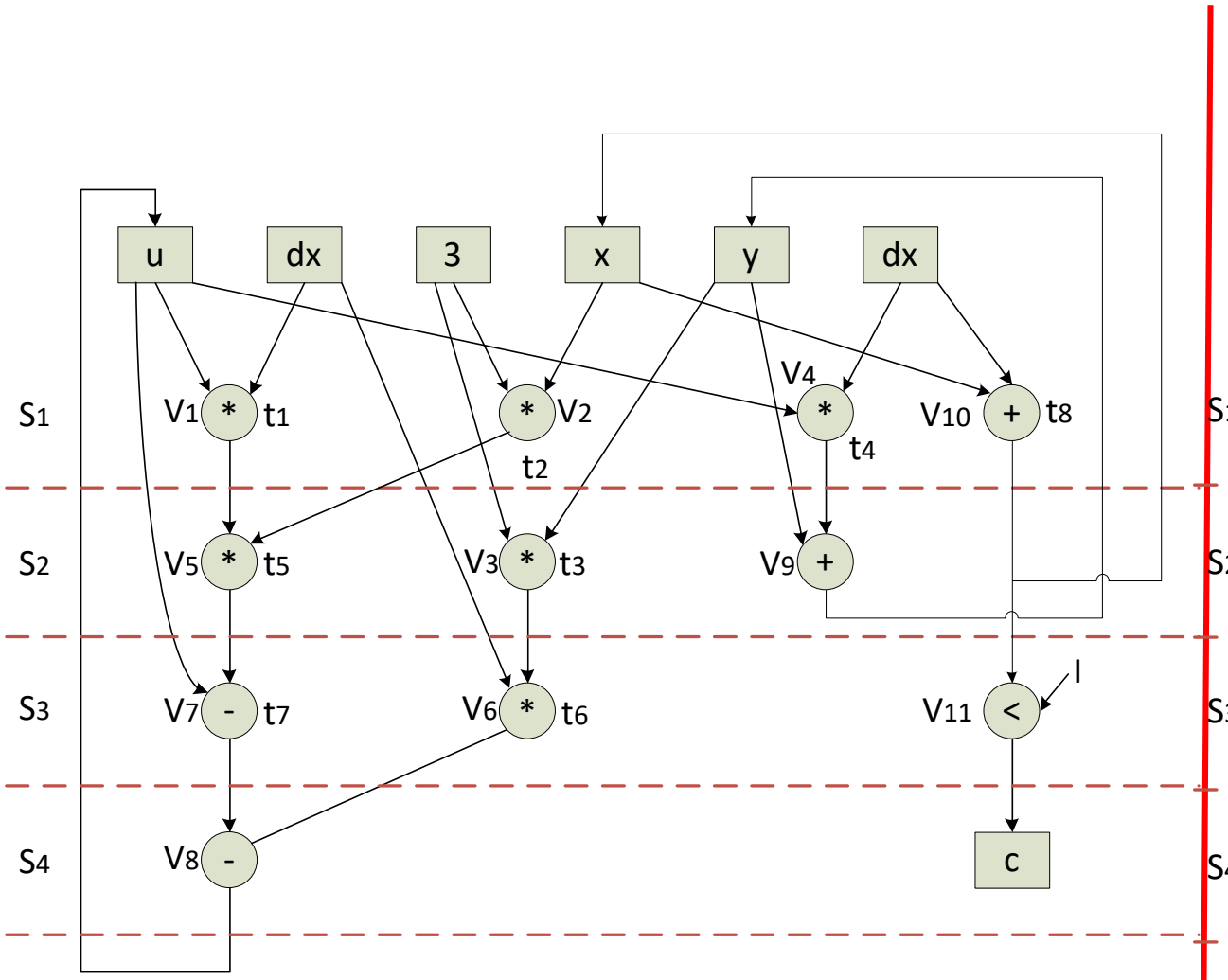

Controller FSM
CS
Status
Data path


R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
M1
M2
M3
A
C

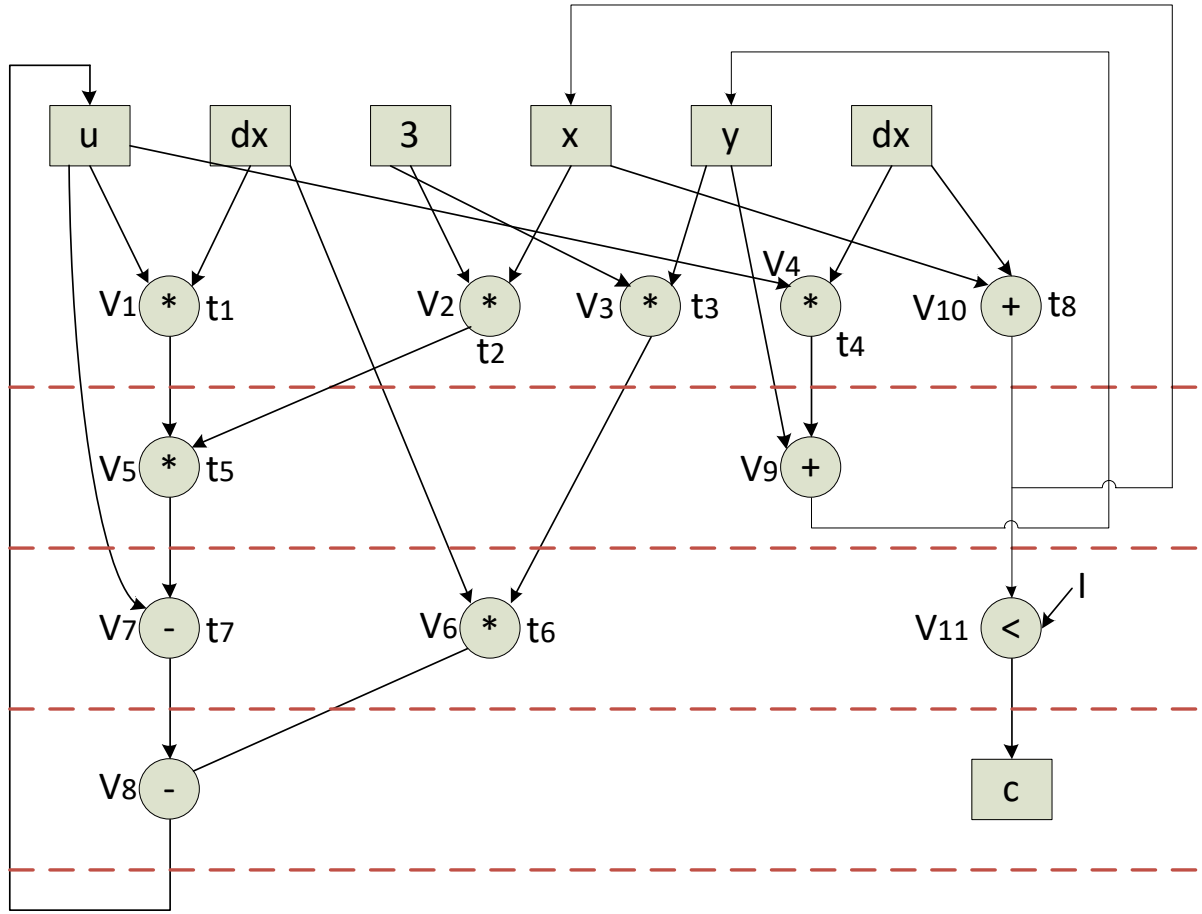

I
y
c
B1
B2


10
11
I
c
!c
S1
<1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0 0, 0, 1>
S2
B2
S3
S4

Scheduling Possibilities

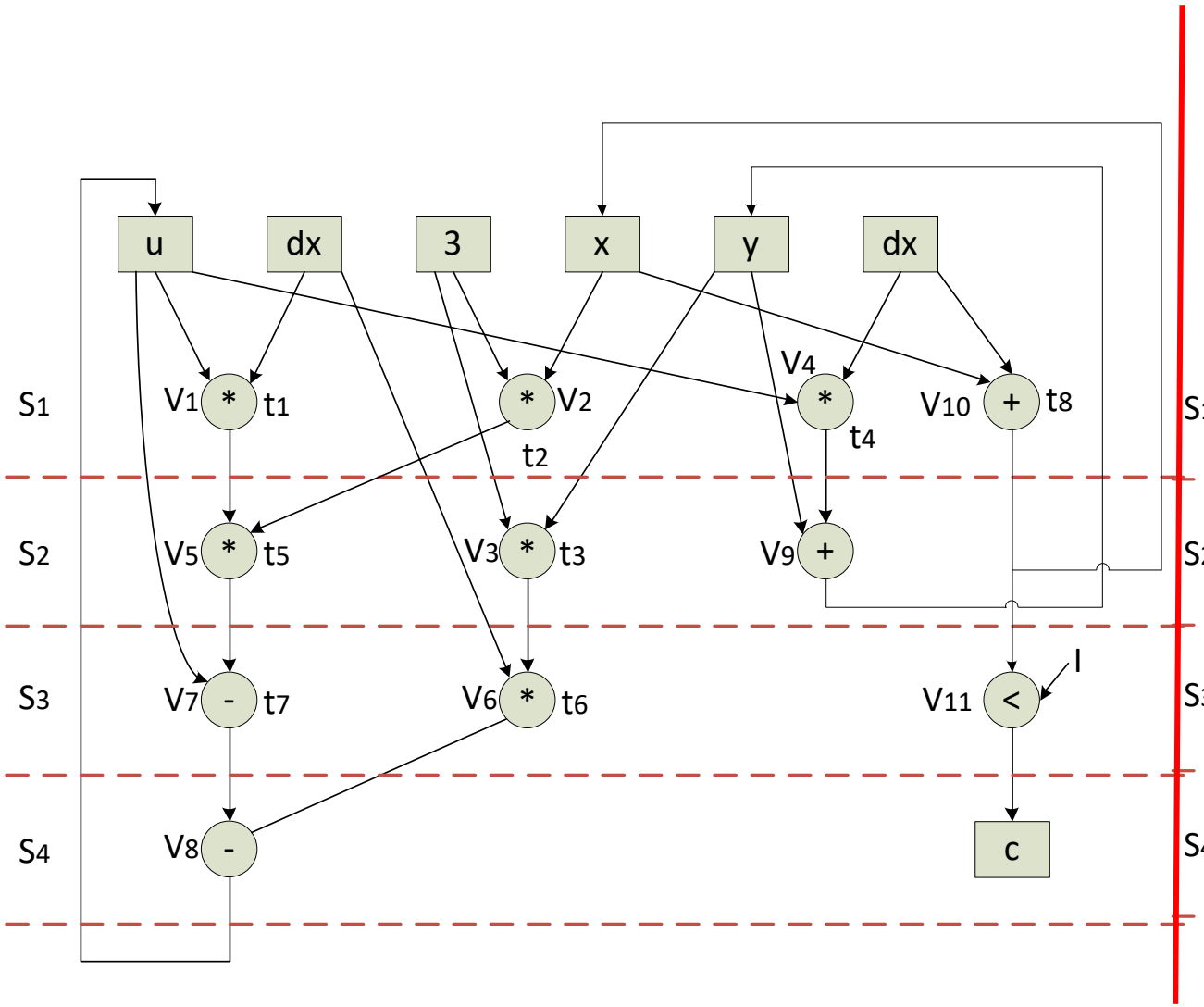


At least 3 Multipliers required

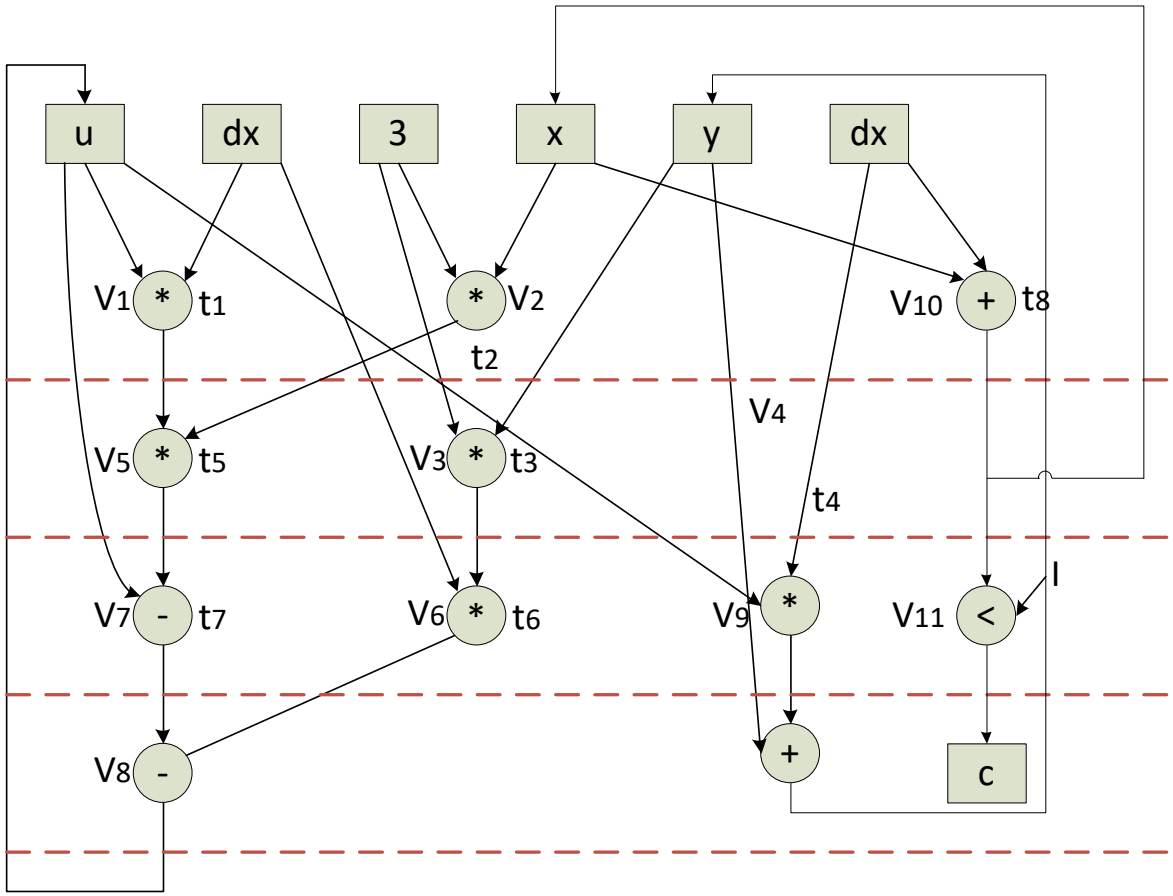


At least 4 Multipliers required

Scheduling Possibilities



At least 3 Multipliers required



At least 2 Multipliers required

Scheduling Techniques

- **Unconstraint Algorithms**
  1. As Soon As Possible (ASAP)
  2. As Late As Possible (ALAP)
- **Resource Constrained**
  1. List scheduling
- **Time Constraints**
  1. Force Directed Scheduling (FDS)
  2. Integer Linear Programming (ILP)
  3. Iterative Refinement

Automation of register and FU allocation and binding

- How to automate register allocation and binding?
- How to automate FU allocation and binding
- Map the problem to Graph colouring problem or clique partitioning problem and solve.

Data path and Controller Synthesis:

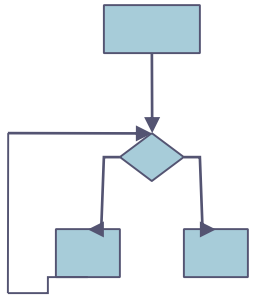
-Mux based and Bus based architecture

-Various way to optimize interconnections

High-level Synthesis Steps

```
while (x_var < a_var) loop
    t1 := u_var * dx_var;
    t2 := 3 * x_var;
    t3 := 3 * y_var;
    t4 := t1 * t2;
    t5 := dx_var * t3;
    t6 := u_var - t4;
    u_var := t6 - t5;
    y1 := u_var * dx_var;
    y_var := y_var + y1;
    x_var := x_var + dx_var;
 end loop;
X <= x_var;
Y <= y_var;
U <= u_var;
```

pre-processing



Input behaviour

scheduling

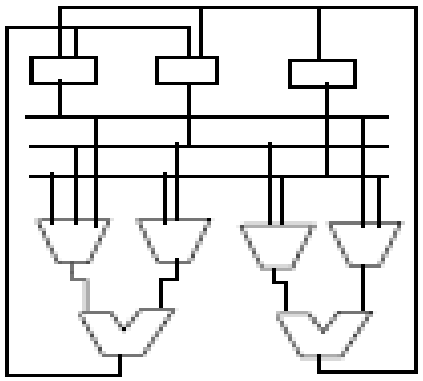
1.	< 1 *>		
2.		<2 + >	<0 * >
3.		<4 * >	| * |
4.		| * |	<3 *>
5.	<7 *>	<6 *>	| * |
6.	| * |	| * |	<5 - >
7.		<9 +>	<8 - >

Allocation & binding

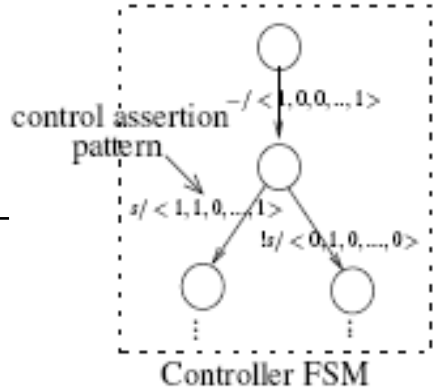
R1 : 3, v1
R2 : x u, v5
R3 : v0, v6
R4 : v3

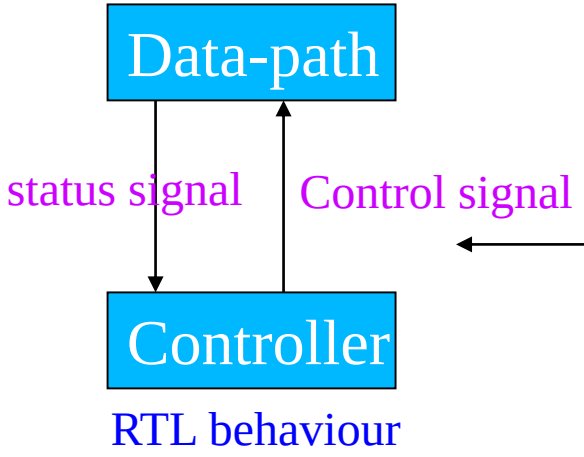
FU1: op1, on3. ..
FU2: op2, op5, …
FU3: …

Data-path generation



Controller generation





RTL behaviour

Thank You