Lec_0_Introduction 2
Lec_1_Number_Systems 22
Lec_2_Switching_Circuits 60
Lec_3_4_Logic_Minimization 89
Lec_4_5_Combinational_Circuits 116
Lec_6_7_Sequential_Circuits 153
1_intro-to-comp-arch 199
2_parts-of-computer 216
3_ic-technology 225
4_instruction-set-design 229
1_instruction-encoding 252
2_function-call-support 257
3_hll-code-to-process 272
1_arithmetic-in-mips 279
2_comp-perf-quant 286
3_hw-impl-prelims 299
4_single-cycle-impl 305
2_single-cycle-extn 321
3_single-cycle-analysis 328

Digital Logic Design + Computer Architecture

Sayandeep Saha

Assistant Professor
Department of Computer Science and Engineering
Indian Institute of Technology Bombay

Introduction

Who is talking??

- Sayandeep Saha (https://www.cse.iitb.ac.in/~sayandeepsaha/)
  - Research Interests: Security and Cryptographic Engineering
  - Office: CC 215
  - Office Hours: Before and after the labs (1:30 PM to 5:30 PM) on **Tuesday**
  - Mailto: sayandeepsaha@cse.iitb.ac.in (must mention [CS230+231] in the subject).

I am a Computer Scientist…

CS230 Deep Learning

Deep Learning is one of the most highly sought after skills in AI. In this course, you will learn the foundations of Deep Learning, understand how to build neural networks, and learn how to lead successful machine learning projects. You will learn about Convolutional networks, RNNs, LSTM, Adam, Dropout, BatchNorm, Xavier/He initialization, and more.

Syllabus | Ed (link via Canvas)

Lecture videos (Canvas) | Lecture videos (Fall 2018)

Instructors

Andrew Ng

Instructor

Time and Location

Wednesday 9:30AM-11:20AM

Zoom

Sorry Guys… But this is also about…

- Machine and Learning…Better to say “learning about machines…”

How is it Related to “Machine Learning”?

- **Ever though about where your tensorflow/pytorch runs?**
- Machine Learning
  - Concepts were there since 1980’s
  - Then why such buzz in 2024??
- Those days we didn’t have such powerful machines to train our models

Why should I bother about chips??

- **It is Electronics… :(**
- Indeed it is….But then…

Why should I bother about chips??

- **It is Electronics… :(**
- Indeed it is….But then everyone wants their own processor, why???

Google's Next Pixel Phone Will Be Powered by a Custom Chip

Following the industry trend of tech giants manufacturing their own processors, the company will start putting bespoke silicon in its mobile hardware.

News / Technology Science

Meta Launches New AI Chip For Facebook, Instagram And WhatsApp

The chip, internally known as "Artemis," signifies Meta's move towards reducing reliance on Nvidia's AI chips and cutting down energy costs.

Authored by: TN Tech Desk | Updated Apr 11, 2024, 09:50 IST

Why should I bother about chips??

- **It is Electronics… :(**
- Hardware is the new software

Google's Next Pixel Phone Will Be Powered by a Custom Chip

Following the industry trend of tech giants manufacturing their own processors, the company will start putting bespoke silicon in its mobile hardware.

News / Technology Science

Meta Launches New AI Chip For Facebook, Instagram And WhatsApp

The chip, internally known as "Artemis," signifies Meta's move towards reducing reliance on Nvidia's AI chips and cutting down energy costs.

Authored by: TN Tech Desk | Updated Apr 11, 2024, 09:50 IST

Abstraction…

- **We only use our computers…do not bother what is inside…**
  - Don’t bother about unnecessary complexity…just use things as black box
  - Can you develop a game while thinking on how your electrons behave??

Abstraction…

https://www.cse.iitb.ac.in/~biswa/courses/CS230/lectures/L1.pdf

Abstraction…

- **But then, it only works when you don't care much about performance…**
  - Recent trend in industry…
    - Design and optimize everything…
  - Breaking the abstraction barrier..
    - But that's ok, performance gives money

The Computer Aided Design Industry

- **The role of a computer scientist in all this….**
- Imagine wiring your latest intel processor by hand …
  - Note: 10 billion transistors…
- We do it with computes since 80's.

The Computer Aided Design Industry

And Finally, Security…

- **Hardware hacking is the new "cool" thing…**

Intel ups protection against physical chip attacks in Alder Lake

Repurposes logic originally used for spotting variations in voltage, timing in older circuits to help performance

Dan Robinson — Fri 12 Aug 2022 | 15:00 UTC

BLACK HAT Intel has disclosed how it may be able to protect systems against some physical threats by repurposing circuitry originally designed to counter variations in voltage and timing that may occur as silicon circuits age.

The research was presented by Intel at the Black Hat USA 2022 cybersecurity conference this week, and details logic inside the system chipset that is intended to complement existing software mitigations for fault injection attacks, the chipmaker said.

It makes use of a Tunable Replica Circuit (TRC), logic developed at Intel Labs to monitor variations such as voltage droop, temperature, and aging in circuits to improve

TRENDING: Maingear MG-1 AMD PC Build | GeForce RTX 4070 | Radeon RX 7900 XTX | Ryzen 7 7800X3D | Alienw

HOME > NEWS

AMD And Intel CPUs Rocked By New Speculative Execution Attack With A Huge Performance Hit

by Zak Killian — Wednesday, July 13, 2022, 05:08 PM EDT

0 Comments | BECOME A PATRON

Crypto Keys Could Be Compromised by Intel and AMD 'Hertzbleed' Chip Vulnerability

2 mins

By Martin Young, 15 June 2022, 07:40 GMT+0000 | Updated by Geraint Price, 15 June 2022, 07:40 GMT+0000

In Brief

- The attack observes power signature of any cryptographic workload.
- Modern Intel Core and AMD Ryzen chips are affected.

Technology

Using just a $25 device a researcher hacked into Elon Musk's Starlink system

What will the technology mogul have to say about this?

AI power analysis breaks post-quantum security algorithm

Technology News | February 19, 2023

By Nick Flaherty

AUTHENTICATION & ENCRYPTION

One of the key post quantum security algorithms agreed as the latest standard has been broken by Swedish researchers

And Finally, Security…

- **Hardware hacking is the new "cool" thing…**
- We can just hack a hardware by writing a piece of code

Intel ups protection against physical chip attacks in Alder Lake

Repurposes logic originally used for spotting variations in voltage, timing in older circuits to help performance

Dan Robinson — Fri 12 Aug 2022 | 15:00 UTC

BLACK HAT Intel has disclosed how it may be able to protect systems against some physical threats by repurposing circuitry originally designed to counter variations in voltage and timing that may occur as silicon circuits age.

The research was presented by Intel at the Black Hat USA 2022 cybersecurity conference this week, and details logic inside the system chipset that is intended to complement existing software mitigations for fault injection attacks, the chipmaker said.

It makes use of a Tunable Replica Circuit (TRC), logic developed at Intel Labs to monitor variations such as voltage droop, temperature, and aging in circuits to improve

TRENDING: Maingear MG-1 AMD PC Build | GeForce RTX 4070 | Radeon RX 7900 XTX | Ryzen 7 7800X3D | Alienw

HOME > NEWS

AMD And Intel CPUs Rocked By New Speculative Execution Attack With A Huge Performance Hit

by Zak Killian — Wednesday, July 13, 2022, 05:08 PM EDT

0 Comments | BECOME A PATRON

Crypto Keys Could Be Compromised by Intel and AMD 'Hertzbleed' Chip Vulnerability

2 mins

By Martin Young, 15 June 2022, 07:40 GMT+0000 | Updated by Geraint Price, 15 June 2022, 07:40 GMT+0000

In Brief

The attack observes power signature of any cryptographic workload.

Modern Intel Core and AMD Ryzen chips are affected.

We use cookies to improve your experience

Technology

Using just a $25 device a researcher hacked into Elon Musk's Starlink system

What will the technology mogul have to say about this?

AI power analysis breaks post-quantum security algorithm

Technology News | February 19, 2023

By Nick Flaherty

AUTHENTICATION & ENCRYPTION

One of the key post quantum security algorithms agreed as the latest standard has been broken by Swedish researchers

And Finally, Security…

- **Hardware hacking is the new "cool" thing…**
- We can just hack a hardware by writing a piece of code

Intel ups protection against physical chip attacks in Alder Lake

Repurposes logic originally used for spotting variations in voltage, timing in older circuits to help performance

Dan Robinson — Fri 12 Aug 2022 | 15:00 UTC

BLACK HAT Intel has disclosed how it may be able to protect systems against some physical threats by repurposing circuitry originally designed to counter variations in voltage and timing that may occur as silicon circuits age.

The research was presented by Intel at the Black Hat USA 2022 cybersecurity conference this week, and details logic inside the system chipset that is intended to complement existing software mitigations for fault injection attacks, the chipmaker said.

It makes use of a Tunable Replica Circuit (TRC), logic developed at Intel Labs to monitor variations such as voltage droop, temperature, and aging in circuits to improve

TRENDING: Maingear MG-1 AMD PC Build | GeForce RTX 4070 | Radeon RX 7900 XTX | Ryzen 7 7800X3D | Alienw

HOME > NEWS

AMD And Intel CPUs Rocked By New Speculative Execution Attack With A Huge Performance Hit

by Zak Killian — Wednesday, July 13, 2022, 05:08 PM EDT

0 Comments | BECOME A PATRON

Crypto Keys Could Be Compromised by Intel and AMD 'Hertzbleed' Chip Vulnerability

2 mins

By Martin Young, 15 June 2022, 07:40 GMT+0000 | Updated by Geraint Price, 15 June 2022, 07:40 GMT+0000

In Brief

- The attack observes power signature of any cryptographic workload.
- Modern Intel Core and AMD Ryzen chips are affected.

Technology

Using just a $25 device a researcher hacked into Elon Musk's Starlink system

What will the technology mogul have to say about this?

AI power analysis breaks post-quantum security algorithm

Technology News | February 19, 2023

By Nick Flaherty

AUTHENTICATION & ENCRYPTION

One of the key post quantum security algorithms agreed as the latest standard has been broken by Swedish researchers

Let’s Pause

- **This is a foundational course for entering…**
  - The world on modern computer architecture, AI accelerators…
    - **Byproduct**: It will make you a better coder for sure…
  - CAD tools and algorithms
  - Modern computer security
  - And many more…
- Let’s now talk a bit on the course logistics…

Digital Logic + Computer Architecture

- **<u>Logic Part: (First 3 weeks)</u>**
  - Instructor: Sayandeep
  - Traditional Model
    - Class lectures
    - SAFE quiz — every class
    - Traditional quiz — 1
    - Lab on Verilog — 3 labs
    - Resources:
      - **BodhiTree**
      - https://sites.google.com/view/sayandeepsaha/digital-logic-and-computer-architecture-theory-lab
- **<u>Architecture Part:</u>**
  - Instructor: Bhaskar
  - **Flipped Classroom Model**

Thank you

Digital Logic Design + Computer Architecture

Sayandeep Saha

Assistant Professor
Department of Computer Science and Engineering
Indian Institute of Technology Bombay

Number Systems and Codes

Baby Step

- **Let's go back to the days when we were 2 years old…**
  - Learning numbers again….but in a new way…

Numbers in Computing

- We normally use the decimal number system.
- But computers understand only bits…
- How to compute on bits??

Generalization of Number Representation

- Numbers are represented in a "base".
- Decimal (Base 10): $953.78_{10} = 9 \times 10^2 + 5 \times 10^1 + 3 \times 10^0 + 7 \times 10^{-1} + 8 \times 10^{-2}$
- Binary (Base 2):

$$1011.11_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2}$$
$$= 8 + 0 + 2 + 1 + \tfrac{1}{2} + \tfrac{1}{4} = 11\tfrac{3}{4} = 11.75_{10}$$

- The general case — base b number:
$(N)_b = a_{q-1}b^{(q-1)} + a_{q-2}b^{(q-2)} + \cdots + a_2b^2 + a_1b^1 + a_0b^0 + a_{-1}b^{-1} + \cdots a_{-p}b^{-p}, \quad 0 \le a_i < b, b > 1$
  - $a_{(q-1)}$ is called the Most Significant Digit (MSD)
  - $a_{(-p)}$ is called the Least Significant Digit (MSD)
  - $a_{(-1)}$ — $a_{(-p)}$ are digits in the fractional part.

Generalization of Number Representation

	Base			
2	4	8	10	12
0000	0	0	0	0
0001	1	1	1	1
0010	2	2	2	2
0011	3	3	3	3
0100	10	4	4	4
0101	11	5	5	5
0110	12	6	6	6
0111	13	7	7	7
1000	20	10	8	8
1001	21	11	9	9
1010	22	12	10	α
1011	23	13	11	β
1100	30	14	12	10
1101	31	15	13	11
1110	32	16	14	12
1111	33	17	15	13

Base Conversion

- Octal (b=8) to Decimal (b=10): $(432.2)_8 = 4 \times 8^2 + 3 \times 8^1 + 2 \times 8^0 + 2 \times 8^{-1} = (282.25)_{10}$
- Binary (base 2) to Decimal (b=10): $(1010.011)_2 = 2^3 + 2^1 + 2^{-2} + 2^{-3} = (10.375)_{10}$
- General Rule: for $b_1 > b_2$

$$(N)_{b_1} = a_{q-1}b_2^{q-1} + a_{q-2}b_2^{q-2} + \cdots + a_1 b_2^1 + a_0 b_2^0$$

$$\frac{(N)_{b_1}}{b_2} = \underbrace{a_{q-1}b_2^{q-2} + a_{q-2}b_2^{q-3} + \cdots + a_1}_{Q_0} + \frac{a_0}{b_2}$$

$$\left(\frac{Q_0}{b_2}\right)_{b_1} = \underbrace{a_{q-1}b_2^{q-3} + a_{q-2}b_2^{q-4} + \cdots}_{Q_1} + \frac{a_1}{b_2}$$

Base Conversion: Decimal to…

To Binary:

$$
\begin{array}{lll}
2\,\big/\,53 & & \\
2\,\big/\,26 & \text{rem.} = 1 = a_0 & \\
2\,\big/\,13 & \text{rem.} = 0 = a_1 & \\
2\,\big/\,6 & \text{rem.} = 1 = a_2 & 53_{10} = 110101_2 \\
2\,\big/\,3 & \text{rem.} = 0 = a_3 & \\
2\,\big/\,1 & \text{rem.} = 1 = a_4 & \\
0 & \text{rem.} = 1 = a_5 &
\end{array}
$$

Convenient way of writing

Integer	Remainder
41	
20	1
10	0
5	0
2	1
1	0
0	1

101001 = answer

To Octal:

$$
\begin{array}{r|l}
153 & \\
19 & 1 \\
2 & 3 \\
0 & 2
\end{array}
\quad = (231)_8
$$

Base Conversion: Fractions

To Binary:

	Integer		Fraction	Coefficient
$0.6875 \times 2 =$	1	+	0.3750	$a_{-1} = 1$
$0.3750 \times 2 =$	0	+	0.7500	$a_{-2} = 0$
$0.7500 \times 2 =$	1	+	0.5000	$a_{-3} = 1$
$0.5000 \times 2 =$	1	+	0.0000	$a_{-4} = 1$

Answer: $(0.6875)_{10} = (0.a_{-1}a_{-2}a_{-3}a_{-4})_2 = \underline{(0.1011)_2}$

- Important!!! If a number has both integer and fraction parts, then conversion of these two parts are done separately.
- Simplest way of converting from base b_1 to b_2, is to convert from b_1 to base 10 and then base 10 base b_2.

To Octal:

$0.513 \times 8 = 4.104$

$0.104 \times 8 = 0.832$

$0.832 \times 8 = 6.656$

$0.656 \times 8 = 5.248$

$0.248 \times 8 = 1.984$

$0.984 \times 8 = 7.872$

$(0.513)_{10} = (0.406517\ .\ .\ .\ )_8$

$$(N)_{b_1} = a_{-1}b_2^{-1} + a_{-2}b_2^{-2} + \cdots + a_{-p}b_2^{-p}$$

$$b_2 \cdot (N)_{b_1} = a_{-1} + a_{-2}b_2^{-1} + \cdots + a_{-p}b_2^{-p+1}$$

Quiz Time

- Convert $(231.3)_4$ to base 7

Quiz Time

- Convert $(231.4)_4$ to base 7
- <u>Ans:</u>

$$(231.3)_4 = 2 \times 4^2 + 3 \times 4^1 + 1 \times 4^0 + 3 \times 4^{-1} = (45.75)_{10}$$

$$\begin{array}{ll} 45 & \\ 6 & rem = 3 \\ 0 & rem = 6 \end{array} \longrightarrow (63)_7$$

$$\begin{array}{ll} .75 \times 7 = 5.25 & intpart = 5 \\ 0.25 \times 7 = 1.75 & intpart = 1 \\ 0.75 \times 7 = 5.25 & intpart = 5 \end{array} \longrightarrow (.5151\cdots)_7$$

$$\boxed{(63.5151\cdots)_7}$$

Quiz Time

- Convert $(0.625)_{10}$ to base 2

Quiz Time

- Convert $(0.625)_{10}$ to base 2

$$
\begin{aligned}
.625 \times 2 &= 1.25 \quad & intpart &= 1 \\
0.25 \times 2 &= 0.5 \quad & intpart &= 0 \\
0.5 \times 2 &= 1.00 \quad & intpart &= 1
\end{aligned}
\quad \longrightarrow \quad (.101)_2
$$

Octal and Hexadecimal Numbers

- Base 8 and base 16 are very useful in computing.
- Hex: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
- Binary to Hex/Octal and vice versa:
-

$$(673.124)_8 = (\underbrace{110}_{6}\ \underbrace{111}_{7}\ \underbrace{011}_{3}\ .\ \underbrace{001}_{1}\ \underbrace{010}_{2}\ \underbrace{100}_{4})_2$$

$$(306.D)_{16} = (\underbrace{0011}_{3}\ \underbrace{0000}_{0}\ \underbrace{0110}_{6}\ .\ \underbrace{1101}_{D})_2$$

Octal and Hexadecimal Numbers

- Base 8 and base 16 are very useful in computing.
- Hex: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
- Binary to Hex/Octal and vice versa:
-

$$(673.124)_8 = (\underbrace{110}_{6}\ \underbrace{111}_{7}\ \underbrace{011}_{3}\ .\ \underbrace{001}_{1}\ \underbrace{010}_{2}\ \underbrace{100}_{4})_2$$

$$(306.D)_{16} = (\underbrace{0011}_{3}\ \underbrace{0000}_{0}\ \underbrace{0110}_{6}\ .\ \underbrace{1101}_{D})_2$$

Binary Arithmetic

Bits		*Sum*	*Carry*	*Difference*	*Borrow*	*Product*
a	b	$a+b$		$a-b$		$a \cdot b$
0	0	0	0	0	0	0
0	1	1	0	1	1	0
1	0	1	0	1	0	0
1	1	0	1	0	0	1

Binary Arithmetic

Binary addition:

$$
\begin{array}{rl}
1111 & = \text{carries of } 1 \\
1111.01 & = (15.25)_{10} \\
0111.10 & = (\ \ 7.50)_{10} \\
\hline
10110.11 & = (22.75)_{10}
\end{array}
$$

Binary subtraction:

$$
\begin{array}{rl}
1 & = \text{borrows of } 1 \\
10010.11 & = (18.75)_{10} \\
01100.10 & = (12.50)_{10} \\
\hline
00110.01 & = (\ \ 6.25)_{10}
\end{array}
$$

Binary Arithmetic

Binary multiplication:

```
      11001.1 = (25.5)10
         110.1 = (  6.5)10
      ---------------------
      110011
     000000
    110011
   110011
   ----------------------
  10100101.11 = (165.75)10
```

Binary division:

```
                 10110 = quotient
11001 | 1000100110
        11001
        ----------
        00100101
           11001
        ----------
          0011001
             11001
        ----------
             00000   = remainder
```

Complements

- Simplest way to represent negative numbers.
- The most intuitive way to represent a negative number is to use an extra bit for sign — sign-magnitude representation
- But the computation with this representation is little complex, we'll see this later.
- Complements are more handy
- (b-1)'s complement of $(N)_b$: $(b^n - 1) - N$, where n is the number of digits.
- b's complement of $(N)_b$: $b^n - N$, where n is the number of digits

Complements

- Let's say we want to compute $(1234)_{10} - (110)_{10}$
- First we take the 10'complement of 110: $10^4 - 110 = 9890$ (Important!!! Number of digits has to be the same)
- Now add: $1234 + 9890 = 11124$
- 11124 > 10000: so there will be an end carry in the addition — just discard the carry
- 11124 —> 1124, which is the answer.
- Now, what is the purpose????

Complements — in the binary world

- 2's complement of N: $2^n - N$
- 1's complement of N: $(2^n - 1) - N$
- We have a super easy way to compute these: can you guess why?

Complements in the binary world

- 2's complement of N: $2^n - N$
- 1's complement of N: $(2^n - 1) - N$
- We have a super easy way to compute these: can you guess why?
  - Observe that for any n
    - $(2^n - 1)$ is basically 1111… n times.
    - Now if you subtract N, all the bits of N are flipped.
    - Example: 1111 - 1011 = 0100
  - So, we do not need to do any subtraction really, —just flip the bits.
  - 2's complement = 1's complement + 1

Complements in the binary world

- Compute $M - N$
- Step 1: compute 2's complement of N: $2^n - N = comp(N) + 1$
- Step 2: $M + (2^n - N) = 2^n + (M - N)$
- Now if $M \geq N$: the sum will be $> 2^n$ — so there will be a carry. Just discard it and output the result.
- If $M < N$ the result is $2^n + (N - M) < 2^n$. This is basically the 2's complement of $(N - M)$. No carry will be produced. The output will be 2's complement of the result, with a negative sign in the front.

Complements in the binary world

- Let X = 1010100, Y = 1000011; compute X-Y and Y - X.

Complements in the binary world

- Let X = 1010100, Y = 1000011; compute X-Y and Y - X.

$$
\begin{array}{rr}
X = & 1010100 \\
\text{2's complement of } Y = & +\ \underline{0111101} \\
\text{Sum} = & 10010001 \\
\text{Discard end carry } 2^7 = & -\underline{10000000} \\
\text{Answer: } X - Y = & 0010001
\end{array}
$$

$$
\begin{array}{rr}
Y = & 1000011 \\
\text{2's complement of } X = & +\ \underline{0101100} \\
\text{Sum} = & 1101111
\end{array}
$$

There is no end carry.

Answer: $Y - X = -(\text{2's complement of } 1101111) = -0010001$

- Important!!! We are doing unsigned subtraction!!
- Food of thought: we can do the same with 1's complement too!!, then why 2's complement???

Complements in the binary world

- Let X = 1010100, Y = 1000011; compute X-Y and Y - X.

$X - Y = 1010100 - 1000011$

$$
\begin{array}{rr}
X = & 1010100 \\
\text{1's complement of } Y = & +\ \underline{0111100} \\
\text{Sum} = & 10010000 \\
\text{End-around carry} & \longrightarrow\ \underline{+\ 1} \\
\text{Answer: } X - Y = & 0010001
\end{array}
$$

$Y - X = 1000011 - 1010100$

$$
\begin{array}{rr}
Y = & 1000011 \\
\text{1's complement of } X = & +\ \underline{0101011} \\
\text{Sum} = & 1101110
\end{array}
$$

There is no end carry.

Answer: $Y - X = -(\text{1's complement of } 1101110) = -0010001$

Complements in the binary world

- We can do the same with 1's complement too!!, then why 2's complement???
-

$+N$	Positive Integers (all systems)	$-N$	Negative Integers: Sign and Magnitude	Negative Integers: 2's Complement N^*	Negative Integers: 1's Complement $\bar{N}$
+0	0000	−0	1000	——	1111
+1	0001	−1	1001	1111	1110
+2	0010	−2	1010	1110	1101
+3	0011	−3	1011	1101	1100
+4	0100	−4	1100	1100	1011
+5	0101	−5	1101	1011	1010
+6	0110	−6	1110	1010	1001
+7	0111	−7	1111	1001	1000
		−8	——	1000	——

- For signed magnitude, we use 4 bits to represent [-7,7] with the 4th bit being the sign bit

Complements in the binary world

- We can do the same with 1's complement too!!, then why 2's complement???
-

$+N$	Positive Integers (all systems)	$-N$	Negative Integers: Sign and Magnitude	Negative Integers: 2's Complement N^*	Negative Integers: 1's Complement $\bar{N}$
+0	0000	−0	1000	——	1111
+1	0001	−1	1001	1111	1110
+2	0010	−2	1010	1110	1101
+3	0011	−3	1011	1101	1100
+4	0100	−4	1100	1100	1011
+5	0101	−5	1101	1011	1010
+6	0110	−6	1110	1010	1001
+7	0111	−7	1111	1001	1000
		−8	——	1000	——

- For 2's complement, we still use 4 bits to represent [-7,7]; but here the encoding is different.
- 1's complement has a negative 0

Signed Binary Numbers

- Leftmost bit is the sign bit — 0 implies positive number and 1 implies negative number
- **Signed magnitude** representation of -9: 10001001
- **Signed 2's complement representation**: 11110111 — take 2's complement of the positive number including the signed bit.
- Signed 2's complement is generally used for computer arithmetic

Signed Binary Numbers

Decimal	Signed-2's Complement	Signed-1's Complement	Signed Magnitude
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000
−0	—	1111	1000
−1	1111	1110	1001
−2	1110	1101	1010
−3	1101	1100	1011
−4	1100	1011	1100
−5	1011	1010	1101
−6	1010	1001	1110
−7	1001	1000	1111
−8	1000	—	—

Signed Binary Numbers

- The addition of two signed binary numbers with negative numbers represented in signed- 2's-complement form is obtained from the addition of the two numbers, including their sign bits. **A carry out of the sign-bit position is discarded**

- **Subtraction:** Take the 2's complement of the subtrahend (including the sign bit) and add it to the minuend (including the sign bit). A carry out of the sign-bit position is discarded.

- Therefore, computers need only one common hardware circuit to handle both signed and unsigned arithmetic.

+ 6	00000110	− 6	11111010
+13	00001101	+13	00001101
+19	00010011	+ 7	00000111
+ 6	00000110	− 6	11111010
−13	11110011	−13	11110011
− 7	11111001	−19	11101101

Binary Codes

- Solves our problems of interpretation.
- **How to represent a decimal digit with bits?**
- Several encoding techniques can be used
- Note that we have 10 digits — what is the minimum number of bits we need?

Binary Codes

- Solves our problems of interpretation.
- **How to represent a decimal digit with bits?**
- Several encoding techniques can be used
- Note that we have 10 digits — what is the minimum number of bits we need? — 4
- But we can represent total 16 digits with 4 bits, so many different encodings are possible.
- **Weighted code**: If x_1, x_2, x_3, x_4 are the binary digits, with weights w_1, w_2, w_3, w_4, then the decimal digit is:

  $$N=w_4x_4+w_3x_3+w_2x_2+w_1x_1$$

  We say, the sequence (x_1, x_2, x_3, x_4) denotes the code word for N.

Binary Codes

Decimal digit	$w_4w_3w_2w_1$											
	8	4	2	1	2	4	2	1	6	4	2	−3
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1	0	1	0	1
2	0	0	1	0	0	0	1	0	0	0	1	0
3	0	0	1	1	0	0	1	1	1	0	0	1
4	0	1	0	0	0	1	0	0	0	1	0	0
5	0	1	0	1	1	0	1	1	1	0	1	1
6	0	1	1	0	1	1	0	0	0	1	1	0
7	0	1	1	1	1	1	0	1	1	1	0	1
8	1	0	0	0	1	1	1	0	1	0	1	0
9	1	0	0	1	1	1	1	1	1	1	1	1

BCD

Self-complementing Codes

Self-complementing code: Code word of 9's complement of N obtained by interchanging 1's and 0's in the code word of N

Binary Codes

Decimal digit	*Excess-3*				*Cyclic*			
0	0	0	1	1	0	0	0	0
1	0	1	0	0	0	0	0	1
2	0	1	0	1	0	0	1	1
3	0	1	1	0	0	0	1	0
4	0	1	1	1	0	1	1	0
5	1	0	0	0	1	1	1	0
6	1	0	0	1	1	0	1	0
7	1	0	1	0	1	0	0	0
8	1	0	1	1	1	1	0	0
9	1	1	0	0	0	1	0	0

Add 3 to BCD

Successive code words differ in only one digit

Binary Codes

Decimal digit	*Excess-3*				*Cyclic*			
0	0	0	1	1	0	0	0	0
1	0	1	0	0	0	0	0	1
2	0	1	0	1	0	0	1	1
3	0	1	1	0	0	0	1	0
4	0	1	1	1	0	1	1	0
5	1	0	0	0	1	1	1	0
6	1	0	0	1	1	0	1	0
7	1	0	1	0	1	0	0	0
8	1	0	1	1	1	1	0	0
9	1	1	0	0	0	1	0	0

Add 3 to BCD

Successive code words differ in only one digit

Binary Codes

Decimal number	*Gray* g_3	g_2	g_1	g_0	*Binary* b_3	b_2	b_1	b_0
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	1	0	0	1	0
3	0	0	1	0	0	0	1	1
4	0	1	1	0	0	1	0	0
5	0	1	1	1	0	1	0	1
6	0	1	0	1	0	1	1	0
7	0	1	0	0	0	1	1	1
8	1	1	0	0	1	0	0	0
9	1	1	0	1	1	0	0	1
10	1	1	1	1	1	0	1	0
11	1	1	1	0	1	0	1	1
12	1	0	1	0	1	1	0	0
13	1	0	1	1	1	1	0	1
14	1	0	0	1	1	1	1	0
15	1	0	0	0	1	1	1	1

Example:

Binary:

Gray:

Gray-to-binary:

- $b_i = g_i$ if no. of 1's preceding g_i is even
- $b_i = g_i'$ if no. of 1's preceding g_i is odd

Thank you

Digital Logic Design + Computer Architecture

Sayandeep Saha

Assistant Professor
Department of Computer Science and Engineering
Indian Institute of Technology Bombay

Switching Algebra

Switches

Algebra of Switches

- **Warning!!! Too informal**: An <u>algebraic system</u> consists of a set and some well-defined operators on the elements of the set and possibly some relations defined over the elements.

- We will be studying one of the simplest system defined over the set {0, 1} — but ironically this set is going to complicate the rest of your life. :P (just joking)

- **Set**: {0,1}

- **Operators**:
  - AND, OR (Binary)
  - NOT (Unary)

$$0+0=0, \quad 0+1=1, \quad 1+0=1, \quad 1+1=1.$$

$$0\cdot 0=0, \quad 0\cdot 1=0, \quad 1\cdot 0=0, \quad 1\cdot 1=1.$$

$$0'=1, \quad 1'=0.$$

Algebra of Switches: Basic Properties

- **Idempotency**: $x + x = x$
  $x \,.\, x = x$

How do we prove this property?
Perfect induction*: proving a theorem by verifying every combination of values that the variables may assume*

Proof of $x + x = x$: $1 + 1 = 1$ and $0 + 0 = 0$

- If x is a switching variable, then: $x + 1 = 1$
  $x \,.\, 0 = 0$
  $x + 0 = x$
  $x \,.\, 1 = x$

- **Commutativity**: $x + y = y + x$
  $x \,.\, y = y \,.\, x$

Algebra of Switches: Basic Properties

- **Associativity**: $(x + y) + z = x + (y + z)$
  $(x \,.\, y) \,.\, z = x \,.\, (y \,.\, z)$

- **Complementation**: $x + x' = 1$
  $x \,.\, x' = 0$

- **Distributivity**: $x \,.\, (y + z) = x \,.\, y + x \,.\, z$
  $x + y \,.\, z = (x + y) \,.\, (x + z)$

Proofs are easy, you can write down the entire truth table

x	y	z	xy	xz	$y+z$	$x(y+z)$	$xy+xz$
0	0	0	0	0	0	0	0
0	0	1	0	0	1	0	0
0	1	0	0	0	1	0	0
0	1	1	0	0	1	0	0
1	0	0	0	0	0	0	0
1	0	1	0	1	1	1	1
1	1	0	1	0	1	1	1
1	1	1	1	1	1	1	1

Algebra of Switches: Principle of Duality

- **Observe**: Preceding properties grouped in pairs
- <u>One statement can be obtained from the other by interchanging operations OR and AND and constants 0 and 1</u>
- The two statements are said to be <u>dual</u> of each other
- **Implication**: necessary to prove only one of each pair of statements

Algebra of Switches: Some Basic Laws

- **Switching expression**: combination of finite number of switching variables and constants via switching operations (AND, OR, NOT)
  - Any constant or switching variable is a switching expression
  - If T_1 and T_2 are switching expressions, so are T_1', T_2', T_1+T_2 and T_1T_2
  - No other combination of constants and variables is a switching expression

- **Absorption law**: $x + xy = x$

  $x(x + y) = x$

Proof: $x + xy = x1 + xy$ [basic property]

$= x(1 + y)$ [distributivity]

$= x1$ [commutativity and basic property]

(Note we write in the law y + 1 = y, so we apply commutativity)

$= x$ [basic property]

Algebra of Switches: Some Basic Laws

- **Switching expression**: combination of finite number of switching variables and constants via switching operations (AND, OR, NOT)
  - Any constant or switching variable is a switching expression
  - If T_1 and T_2 are switching expressions, so are T_1', T_2', T_1+T_2 and T_1T_2
  - No other combination of constants and variables is a switching expression

- **Absorption law**: $x + xy = x$

  $x(x + y) = x$

Proof: $x + xy = x1 + xy$ [basic property]

$= x(1 + y)$ [distributivity]

$= x1$ [commutativity and basic property]

(Note we write in the law y + 1 = y, so we apply commutativity)

$= x$ [basic property]

Simplification of Switching Expressions

- **Law 2:** $x + x'y = x + y$, $x(x' + y) = xy$
- **Law 3**: $xy + x'z + yz = xy + x'z$, $(x + y)(x' + z)(y + z) = (x + y)(x' + z)$. **(Consensus Theorem)**
- **Law 4**: $(x')' = x$. **(Involution)**
- **Law 5**: $(x+y)' = x'y'$, $(xy)' = x' + y'$ **(De-Morgan's Theorem)**

Food of thought: Can we extend it for n variables???

x	y	x'	y'	$x+y$	$(x+y)'$	$x'y'$
0	0	1	1	0	1	1
0	1	1	0	1	0	0
1	0	0	1	1	0	0
1	1	0	0	1	0	0

Try the proofs!!!

Simplification of Switching Expressions

Example: Simplify $T(x,y,z) = (x + y)[x'(y' + z')]' + x'y' + x'z'$

Simplification of Switching Expressions

Example: Simplify $T(x,y,z) = (x + y)[x'(y' + z')]' + x'y' + x'z'$

$$
\begin{aligned}
(x + y)[x'(y' + z')]' + x'y' + x'z' &= (x + y)(x + yz) + x'y' + x'z' \\
&= (x + xyz + yx + yz) + x'y' + x'z' \\
&= x + yz + x'y' + x'z' \\
&= x + yz + y' + z' \\
&= x + z + y' + z' \\
&= x + y' + 1 \\
&= 1
\end{aligned}
$$

Thus, $T(x,y,z) = 1$, independently of the values of the variables

Simplification of Switching Expressions

Example: Prove $xy + x'y' + yz = xy + x'y' + x'z$

Simplification of Switching Expressions

Example: Prove $xy + x'y' + yz = xy + x'y' + x'z$

- From consensus theorem, $x'z$ can be added to LHS
- Consensus theorem can be applied again to first, third and fourth terms in $xy + x'y' + yz + x'z$ to eliminate yz and reduce it to RHS

Simplification of Switching Expressions

Example: Prove $xy + x'y' + yz = xy + x'y' + x'z$

- From consensus theorem, $x'z$ can be added to LHS
- Consensus theorem can be applied again to first, third and fourth terms in $xy + x'y' + yz + x'z$ to eliminate yz and reduce it to RHS

- LHS : $xy + x'y' + yz = xy + (x'y' + yz) = xy + (y'x' + (y')'z + x'z) = xy + y'x' + yz + x'z = (xy + x'z + yz) + x'y' = xy + x'z + x'y' = RHS$

Switching Functions

- **Switching function $f(x_1, x_2, \ldots, x_n)$**: values assumed by an expression for all combinations of variables $x_1, x_2, \ldots, x_n$
- **Complement function**: $f'(x_1, x_2, \ldots, x_n)$ assumes value 0 (1) whenever $f(x_1, x_2, \ldots, x_n)$ assumes value 1 (0)
- **Logical sum of two functions**: $f(x_1, x_2, \ldots, x_n) + g(x_1, x_2, \ldots, x_n) = 1$ for every combination in which either f or g or both equal 1
- **Logical product of two functions**: $f(x_1, x_2, \ldots, x_n) \,.\, g(x_1, x_2, \ldots, x_n) = 1$ for every combination for which both f and g equal 1

x	y	z	f	g	f'	$f+g$	fg
0	0	0	1	0	0	1	0
0	0	1	0	1	1	1	0
0	1	0	1	0	0	1	0
0	1	1	1	1	0	1	1
1	0	0	0	1	1	1	0
1	0	1	0	0	1	0	0
1	1	0	1	1	0	1	1
1	1	1	1	0	0	1	0

Canonical Forms

- Truth tables are one of the most standard way of representing switching functions.
- But a functions with n variables require 2^n rows each of size $(n+1)$ bits. — quite some overhead.
- **Let's consider the problem of deriving a compact expression of a function from a given truth table**

Decimal code	x	y	z	f
0	0	0	0	1
1	0	0	1	0
2	0	1	0	1
3	0	1	1	1
4	1	0	0	0
5	1	0	1	0
6	1	1	0	1
7	1	1	1	1

$$f = x'y'z' + x'yz' + x'yz + xyz' + xyz$$

- Find the sum (OR) of all terms for which the function evaluates to 1.
  - In this case, for (0, 2, 3, 6, 7).
- Each term is a product of the variables on which the function depends
- Variable x_i appears in uncomplemented (complemented) form in the product if has value 1 (0) in the combination

Canonical Forms: Sum of Products

- **Minterm**: a product term that contains each of the *n* variables as factors in either complemented or uncomplemented form
    - It assumes value 1 for exactly one combination of variables
    - In the following table 000, 010, 011, 110, 111 are minterms, which are usually represented with their decimal equivalent (0,2,3,6,7). In terms of variables *x'y'z', x'yz', x'yz, xyz', xyz*

- **Canonical sum-of-products**: sum of all minterms derived from combinations for which function is 1
    - Also called **disjunctive normal expression**

- **Compact representation**: $\sum(0,2,3,6,7)$

Decimal code	x	y	z	f
0	0	0	0	1
1	0	0	1	0
2	0	1	0	1
3	0	1	1	1
4	1	0	0	0
5	1	0	1	0
6	1	1	0	1
7	1	1	1	1

Canonical Forms: Product of Sums

- **Maxterm**: a sum term that contains each of the n variables in either complemented or uncomplemented form
  - It assumes value 0 for exactly one combination of variables
  - Variable x_i appears in uncomplemented (complemented) form in the sum if it has value 0 (1) in the combination
- **Canonical product-of-sums**: product of all maxterms derived from combinations for which function is 0
  - Also called **conjunctive normal expression**

- **Compact representation**: $\prod(1,4,5)$

$$f = (x + y + z')(x' + y + z)(x' + y + z')$$

Decimal code	x	y	z	f
0	0	0	0	1
1	0	0	1	0
2	0	1	0	1
3	0	1	1	1
4	1	0	0	0
5	1	0	1	0
6	1	1	0	1
7	1	1	1	1

Derivation of SOP Forms

1. Examine each term: if it is a minterm, retain it; continue to next term

2. In each product which is not a minterm: check the variables that do not occur; for each x_i that does not occur, multiply the product by $(x_i + x_i')$

3. Multiply out all products and eliminate redundant terms

Example: $T(x,y,z) = x'y + z' + xyz$

$$= x'y(z + z') + (x + x')(y + y')z' + xyz$$
$$= x'yz + x'yz' + xyz' + xy'z' + x'yz' + x'y'z' + xyz$$
$$= x'yz + x'yz' + xyz' + xy'z' + x'y'z' + xyz$$

Canonical product-of-sums obtained in a dual manner

Example:

$T = x'(y' + z)$

$$= (x' + yy' + zz')(y' + z + xx')$$
$$= (x' + y + z)(x' + y + z')(x' + y' + z)(x' + y' + z')(x + y' + z)(x' + y' + z)$$
$$= (x' + y + z)(x' + y + z')(x' + y' + z)(x' + y' + z')(x + y' + z)$$

Transforming Canonical Forms

- **Example**: Find the canonical product-of-sums for

$$T(x,y,z) = x'y'z' + x'y'z + x'yz + xyz + xy'z + xy'z'$$

$$T = (T')' = [(x'y'z' + x'y'z + x'yz + xyz + xy'z + xy'z')']'$$

Complement T' consists of minterms not contained in T. Thus,

$$T = [x'yz' + xyz']' = (x + y' + z)(x' + y' + z)$$

- Canonical forms are unique

- **Two switching functions are equivalent if and only if their corresponding canonical forms are identical**

Introducing Exclusive-OR (XOR)

Exclusive-OR: modulo-2 addition, i.e., $A \oplus B = 1$ if either A or B is 1, but not both.

Commutativity: $A \oplus B = B \oplus A$

Associativity: $(A \oplus B) \oplus C = A \oplus (B \oplus C) = A \oplus B \oplus C$

Distributivity: $(AB) \oplus (AC) = A(B \oplus C)$

If $A \oplus B = C$, then

$$A \oplus C = B$$
$$B \oplus C = A$$
$$A \oplus B \oplus C = 0$$

Exclusive-OR of an even (odd) number of elements, whose value is 1, is 0 (1)

Functional Completeness

- A set of operations is **functionally complete** (or **universal**) if and only if every switching function can be expressed by operations from this set

- Every switching function can be expressed in canonical form consisting of a finite number of switching variables, constants and operations OR, AND, NOT

- **Example**: Set {+, ., ’}
- **Example**: Set {+, ’} since using De Morgan’s theorem, $x \,.\, y = (x' + y')'$. Thus, + and ’ can replace the . in any switching function
- **Example**: Set {., ’} for similar reasons
- **Example**: NAND since NAND(x,x) = x' and NAND[NAND(x,y),NAND(x,y)] = xy
- **Example**: NOR since NOR(x,x) = x' and NOR[NOR(x,y),NOR(x,y)] = $x + y$

What Can be done with Switching Algebra?

Isomorphism: Two algebraic systems are isomorphic if

- For every operation in one system, there exists a corresponding operation in the second system
- To each element x_i in one system, there corresponds a unique element y_i in the other system, and vice versa
- If each operation and element in every postulate of one system is replaced by the corresponding operation and element in the other system, then the resulting postulate is valid in the second system

Thus, two algebraic systems are isomorphic if and only if they are identical except the labels and symbols used to represent the operations and elements.

- Switching algebra is isomorphic to proposition calculus — something which encodes declarative statements with values "True" or "False", but never both.
  - "If it rains, then we play"

- **Most important**!!!Switching algebra is isomorphic to the **network of logic gates**, something which we build with transistors.

At The Gates

NOT

BUF

OR

AND

XOR

NAND

$Y = \overline{AB}$

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

NOR

$Y = \overline{A + B}$

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

One Level Below

CMOS Level View

A Bit of GK

Thank you

Digital Logic Design + Computer Architecture

Sayandeep Saha

Assistant Professor
Department of Computer Science and Engineering
Indian Institute of Technology Bombay

Logic Minimization

Life of an Engineer

Logic Minimization: Why?

- Consider a switching expression. **How many gates do you need to implement this?** Consider each gate is 2-input, 1 output except the NOTs — 5 OR, 12 ANDs, 3 NOTs

$$f(x, y, z) = x'yz' + x'y'z' + xy'z' + x'yz + xyz + xy'z$$

- Now consider the following expression:' $x'z' + y'z' + yz + xz$
- Observe that both implements the same logic function!!! Now you need 4 ORs, 4 ANDs, and 3 NOTs.
- Can you do better?? — Yes $f(x, y, z) = x'z' + xy' + yz$
- Turns out that there can be more such expressions.
- Lower gate count => Lower transistor count => Lower area (and perhaps less power, and time)…
- So, now we have an engineering problem in hand — **how to minimize the switching expressions???**

Bigger Picture

- Modern circuits contains billions of gates — e.g. AMD Instinct is a GPU processor containing 146,000,000,000 transistors; so a few billions of gates (if not trillions)…
- How do people minimized their gate network…Fortunately we have tools for that.
- Today we will be studying some of the fundamental techniques behind these tools.
  - Of course, a very very rudimentary intro

Bigger Picture

- Modern circuits contains billions of gates — e.g. AMD Instinct is a GPU processor containing 146,000,000,000 transistors; so a few billions of gates (if not trillions)…
- How do people minimized their gate network…Fortunately we have tools for that.
- Today we will be studying some of the fundamental techniques behind these tools.

Bigger Picture

- Modern circuits contains billions of gates — e.g. AMD Instinct is a GPU processor containing 146,000,000,000 transistors; so a few billions of gates (if not trillions)…
- How do people minimized their gate network…Fortunately we have tools for that.
- Today we will be studying some of the fundamental techniques behind these tools.

The Map Method

- **Karnaugh map**: modified form of truth table
- Combine terms using the **Aa + Aa' = A (combining theorem)**

z \ xy	00	01	11	10
0	0	2	6	4
1	1	3	7	5

(*a*) Location of minterms in a three-variable map.

z \ xy	00	01	11	10
0		1	1	
1			1	

(*b*) Map for function $f(x,y,z) = \sum(2,6,7) = yz' + xy$.

yz \ wx	00	01	11	10
00	0	4	12	8
01	1	5	13	9
11	3	7	15	11
10	2	6	14	10

(*c*) Location of minterms in a four-variable map.

yz \ wx	00	01	11	10
00		1	1	1
01		1	1	
11			1	
10			1	

(*d*) Map for function $f(w,x,y,z) = \sum(4,5,8,12,13,14,15) = wx + xy' + wy'z'$.

The Map Method

- **Karnaugh map**: modified form of truth table
- Combine terms using the **Aa + Aa' = A (combining theorem)**
- **<u>Cube</u>**:
  - Collection of 2^m cells, each adjacent to ***m*** cells of the collection
  - The cube is said to **cover** the cells it is involved with
  - Expressed by a product of **n-m literals** for a function containing **n variables**
  - **m literals** not in the product said to be eliminated

z \ xy	00	01	11	10
0	0	2	6	4
1	1	3	7	5

(*a*) Location of minterms in a three-variable map.

z \ xy	00	01	11	10
0		1	1	
1			1	

(*b*) Map for function $f(x,y,z) = \sum(2,6,7) = yz' + xy$.

yz \ wx	00	01	11	10
00	0	4	12	8
01	1	5	13	9
11	3	7	15	11
10	2	6	14	10

(*c*) Location of minterms in a four-variable map.

yz \ wx	00	01	11	10
00		1	1	1
01		1	1	
11			1	
10			1	

(*d*) Map for function $f(w,x,y,z) = \sum(4,5,8,12,13,14,15) = wx + xy' + wy'z'$.

The Map Method

- **Karnaugh map**: modified form of truth table
- Combine terms using the **Aa + Aa' = A (combining theorem)**
- **<u>Cube</u>**:
  - Collection of 2^m cells, each adjacent to ***m*** cells of the collection
  - The cube is said to **cover** the cells it is involved with
  - Expressed by a product of **n-m literals** for a function containing **n variables**
  - **m literals** not in the product said to be eliminated
  - **More Clarification**:
    - Consider the squares 2 and 6 in Fig (a)
    - The minterms are z'x'y and z'xy
    - Now apply the **combining theorem**.
    - Literal x and x' are eliminated.
    - The result is a 2-cube.

z \ xy	00	01	11	10
0	0	2	6	4
1	1	3	7	5

(*a*) Location of minterms in a three-variable map.

z \ xy	00	01	11	10
0		1	1	
1			1	

(*b*) Map for function $f(x,y,z) = \sum(2,6,7) = yz' + xy$.

yz \ wx	00	01	11	10
00	0	4	12	8
01	1	5	13	9
11	3	7	15	11
10	2	6	14	10

(*c*) Location of minterms in a four-variable map.

yz \ wx	00	01	11	10
00		1	1	1
01		1	1	
11			1	
10			1	

(*d*) Map for function $f(w,x,y,z) = \sum(4,5,8,12,13,14,15) = wx + xy' + wy'z'$.

The Map Method

z \ xy	00	01	11	10
0	0	2	6	4
1	1	3	7	5

(*a*) Location of minterms in a three-variable map.

z \ xy	00	01	11	10
0		1	1	
1			1	

(*b*) Map for function $f(x,y,z) = \sum(2,6,7) = yz' + xy$.

- **Example**: $f = yz' + xy$
  - **Use of cell 6 in forming both cubes justified by idempotent law**
  - Corresponding algebraic manipulations:

$$
\begin{aligned}
f &= x'yz' + xyz' + xyz \\
&= x'yz' + \underline{xyz' + xyz'} + xyz \quad \textit{(idempotent law)} \\
&= yz'(x' + x) + xy(z' + z) \\
&= yz' + xy
\end{aligned}
$$

The Map Method

- **Example**: w'xy'z' + w'xy'z + wxy'z' + wxy'z = xy'(w'z' + w'z + wz' + wz) = xy'
- **Trick**:
  - In a cube, just keep the variables not changing their value.

yz \ wx	00	01	11	10
00	0	4	12	8
01	1	5	13	9
11	3	7	15	11
10	2	6	14	10

(*c*) Location of minterms in a four-variable map.

yz \ wx	00	01	11	10
00		1	1	1
01		1	1	
11			1	
10			1	

(*d*) Map for function $f(w,x,y,z) = \sum(4,5,8,12,13,14,15) = wx + xy' + wy'z'$.

The Map Method

Rules for minimization:

- **Step 1**: cover those 1 cells by cubes that cannot be combined with other 1 cells; continue to 1 cells that have a single adjacent 1 cell (thus can form cubes of only two cells)
- **Step 2—:** Combine 1 cells that yield cubes of four cells, but are not part of any cube of eight cells, and so on..
  - A cube contained in a larger cube must never be selected
  - A cube contained in any combination of other cubes already selected in the cover is redundant (**consensus theorem)**
  - If there are more than one way of covering the map with cubes, select the cover with larger cubes
  - **Minimal expression**: collection of cubes that are as large and as few in number as possible, such that each 1 cell is covered by at least one cube
  - **Irredundant expressions**:
    - An SOP from where no term or literal can be deleted.
    - Not necessarily minimal
  - **Minimal and irredundant expressions may not be unique**
  - **But a minimal expression is always irredundant.**

(*d*) Map for function $f(w,x,y,z)$
$=\sum(4,5,8,12,13,14,15) = wx + xy' + wy'z'$.

Let's try this..

The Map Method

(*a*) $f = x'y'z' + w'xy' + wy'z + xz$ is an irredundant expression.

(*b*) $f = w'y'z' + wx'y' + xz$ is the unique minimal expression.

Example: Two irredundant expressions for $f(w,x,y,z) = \sum(0,4,5,7,8,9,13,15)$

The Map Method

Example: $f(w,x,y,z) = \sum(1,5,6,7,11,12,13,15)$

- Only one irredundant form: f = wxy’ + wyz + w’xy + w’y’z

The Map Method: Earth is not Flat

A \ BC	00	01	11	10
0	000	001	011	010
1	100	101	111	110

A \ BC	00	01	11	10
0	1			1
1				

Minimal Product-of-Sums

- **Dual procedure**: product of a minimum number of sum factors, provided there is no other such product with the same number of factors and fewer literals
  - **<u>Variable corresponding to a 1 (0) is complemented (uncomplemented)</u>**
  - Cubes are formed of 0 cells
- **Example**: either one of minimal sum-of-products or minimal product-of-sums can be better than the other in literal count

yz \ wx	00	01	11	10
00				
01		1		1
11				
10		1		1

(*a*) Map of $f(x,y,z) = \sum(5,6,9,10)$
$= w'xy'z + wx'y'z + w'xyz' + wx'yz'$.

yz \ wx	00	01	11	10
00	0	0	0	0
01	0	1	0	1
11	0	0	0	0
10	0	1	0	1

(*b*) Map of $f(x,y,z)$
$=\prod(0,1,2,3,4,7,8,11,12,13,14,15)$
$= (y + z)(y' + z')(w + x)(w' + x')$.

Let's Try it..

- Implement **complement of** $f(A, B, C, D) = \prod(7,9,13)$.

Let's Try it..

- Implement $f(A, B, C, D) = \sum(0,2,8,12,13)$ with minimum number of gates.

Don’t-care Combinations

- **Don’t-care combination** : combination for which the value of the function is not specified.
  - Either input combinations may be invalid
  - Or precise output value is of no importance

- Since each don’t-care can be specified as either 0 or 1
  - a function with k don’t-cares corresponds to a class of 2^k distinct functions.
  - Our aim is to choose the function with the minimal representation
- Assign 1 to some don’t-cares and 0 to others in order to increase the size of the selected cubes whenever possible
- No cube containing only don’t-care cells may be formed

Code Converter

Example: code converter from BCD to excess-3 code
Combinations 10 through 15 are don't-cares

Truth table

Decimal number	*BCD inputs* w	x	y	z	*Excess-3 outputs* f_4	f_3	f_2	f_1
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0

$$f_1 = \sum(0,2,4,6,8) + \sum_{\phi}(10,11,12,13,14,15)$$
$$f_2 = \sum(0,3,4,7,8) + \sum_{\phi}(10,11,12,13,14,15)$$
$$f_3 = \sum(1,2,3,4,9) + \sum_{\phi}(10,11,12,13,14,15)$$
$$f_4 = \sum(5,6,7,8,9) + \sum_{\phi}(10,11,12,13,14,15)$$

Code Converter

yz \ wx	00	01	11	10
00	1	1	ϕ	1
01			ϕ	
11			ϕ	ϕ
10	1	1	ϕ	ϕ

f_1 map

yz \ wx	00	01	11	10
00	1	1	ϕ	1
01			ϕ	
11	1	1	ϕ	ϕ
10			ϕ	ϕ

f_2 map

yz \ wx	00	01	11	10
00		1	ϕ	
01	1		ϕ	1
11	1		ϕ	ϕ
10	1		ϕ	ϕ

f_3 map

yz \ wx	00	01	11	10
00			ϕ	1
01		1	ϕ	1
11		1	ϕ	ϕ
10		1	ϕ	ϕ

f_4 map

Increase the size of the cubes without making it necessary to increase the number of cubes, than would be required with fewer don't cares assigned one.

Lets do it!!!

Code Converter

yz \ wx	00	01	11	10
00	1	1	ϕ	1
01			ϕ	
11			ϕ	ϕ
10	1	1	ϕ	ϕ

f_1 map

yz \ wx	00	01	11	10
00	1	1	ϕ	1
01			ϕ	
11	1	1	ϕ	ϕ
10			ϕ	ϕ

f_2 map

Increase the size of the cubes without making it necessary to increase the number of cubes, than would be required with fewer don't cares assigned one.

yz \ wx	00	01	11	10
00		1	ϕ	
01	1		ϕ	1
11	1		ϕ	ϕ
10	1		ϕ	ϕ

f_3 map

yz \ wx	00	01	11	10
00			ϕ	1
01		1	ϕ	1
11		1	ϕ	ϕ
10		1	ϕ	ϕ

f_4 map

$$f_1 = z'.$$
$$f_2 = y'z' + yz$$
$$f_3 = x'y + x'z + xy'z'$$
$$f_4 = w + xy + xz$$

Logic Network for Code Converter

Two-level AND-OR realization:

Five-variable Map

General five-variable map

yz \ vwx	000	001	011	010	110	111	101	100
00	0	4	12	8	24	28	20	16
01	1	5	13	9	25	29	21	17
11	3	7	15	11	27	31	23	19
10	2	6	14	10	26	30	22	18

Example: Minimize $f(v,w,x,y,z) = \sum(1,2,6,7,9,13,14,15,17,22,23,25,29,30,31)$

	000	001	011	010	110	111	101	100
	1		1	1	1	1		1
		1	1			1	1	
	1	1	1			1	1	

$f(v,w,x,y,z) = x'y'z + wxz + xy + v'w'yz'$

Limitation of Simple Maps

- Maps are useful up to 5-6 variables, after that the calculation becomes formidable..
- How many cells are there in a 6 variable map??
- We also need something which is more amenable to a computer program.
  - QM Method
    - Finds out the useful logic cubes called ‘prime implicants’
    - Systematic procedure — amenable to programming
    - Will see it if time permits..

Let's Try it..

- Implement $f(A, B, C, D) = \sum(0,2,8,12,13)$ with minimum number of gates.

Digital Logic Design + Computer Architecture

Sayandeep Saha

Assistant Professor
Department of Computer Science and Engineering
Indian Institute of Technology Bombay

Combinational Circuits

Do You Want to Design Some Day?

Design with Gates

- **Logic gates**: perform logical operations on input signals
- **Positive (negative) logic polarity**: constant 1 (0) denotes a high voltage and constant 0 a low (high) voltage
- **Combinational circuits:** No memorization
- **Synchronous sequential circuits**: have memory; driven by a clock that produces a train of equally spaced pulses
- **Propagation delay**: time to propagate a signal through a gate
- **Asynchronous circuits**: are almost free-running and do not depend on a clock; controlled by initiation and completion signals

Combinational Circuits

Circuit analysis: determine the Boolean function that describes the circuit

- Done by tracing the output of each gate, starting from circuit inputs and continuing towards each circuit output

Example: a multi-level realization of a full binary adder

$$C_0 = AB + (A + B)C$$
$$= AB + AC + BC$$

$$S = (A + B + C)[AB + (A + B)C]' + ABC$$
$$= (A + B + C)(A' + B')(A' + C')(B' + C') + ABC$$
$$= AB'C' + A'BC' + A'B'C + ABC$$
$$= A \oplus B \oplus C$$

Combinational Circuits: Parity-bit Generator

Parity-bit generator: produces output value 1 if and only if an odd number of its inputs have value 1

z \ xy	00	01	11	10
0	0	1	0	1
1	1	0	1	0

(*a*) Map.

(*b*) Implementation.

$$P = x'y'z + x'yz' + xy'z' + xyz = x \oplus y \oplus z$$

Combinational Circuits: Serial to Parallel

Serial-to-parallel converter: distributes a sequence of binary digits on a serial input to a set of different outputs, as specified by external control signals

Control		*Output lines*				*Logic equations*
C_1	C_2	L_1	L_2	L_3	L_4	
0	0	x	0	0	0	$L_1 = xC_1'C_2'$
0	1	0	x	0	0	$L_2 = xC_1'C_2$
1	0	0	0	x	0	$L_3 = xC_1C_2'$
1	1	0	0	0	x	$L_4 = xC_1C_2$

Combinational Circuits: Comparators

***n*-bit comparator**: compares the magnitude of two numbers X and Y, and has three outputs f_1, f_2, and f_3

- $f_1 = 1$ iff $X > Y$
- $f_2 = 1$ iff $X = Y$
- $f_3 = 1$ iff $X < Y$

(*a*) Block diagram.

y_1y_2 \ x_1x_2	00	01	11	10
00	2	1	1	1
01	3	2	1	1
11	3	3	2	3
10	3	3	1	2

(*b*) Map for f_1, f_2, and f_3.

$f_1 = ?$

$f_2 = ?$

$f_3 = ?$

Combinational Circuits: Comparators

***n*-bit comparator**: compares the magnitude of two numbers X and Y, and has three outputs f_1, f_2, and f_3

- $f_1 = 1$ iff $X > Y$
- $f_2 = 1$ iff $X = Y$
- $f_3 = 1$ iff $X < Y$

(*a*) Block diagram.

y_1y_2 \ x_1x_2	00	01	11	10
00	2	1	1	1
01	3	2	1	1
11	3	3	2	3
10	3	3	1	2

(*b*) Map for f_1, f_2, and f_3.

$$f_1 = x_1x_2y_2' + x_2y_1'y_2' + x_1y_1'$$
$$= (x_1 + y_1')x_2y_2' + x_1y_1'$$

$$f_2 = x_1'x_2'y_1'y_2' + x_1'x_2y_1'y_2 + x_1x_2'y_1y_2' + x_1x_2y_1y_2$$
$$= x_1'y_1'(x_2'y_2' + x_2y_2) + x_1y_1(x_2'y_2' + x_2y_2)$$
$$= (x_1'y_1' + x_1y_1)(x_2'y_2' + x_2y_2)$$

$$f_3 = x_2'y_1y_2 + x_1'x_2'y_2 + x_1'y_1$$
$$= x_2'y_2(y_1 + x_1') + x_1'y_1$$

(*c*) Circuit for f_1.

Combinational Circuits: Comparators

Four-bit comparator: 8 inputs (four for A, four for B, and three outputs $A>B$, $A<B$ and $A=B$

$$x_i = A_i B_i + A_i' B_i' \qquad i = 0, 1, 2, 3$$

$$(A = B) = x_3 x_2 x_1 x_0$$

$$(A > B) = A_3 B_3' + x_3 A_2 B_2' + x_3 x_2 A_1 B_1' + x_3 x_2 x_1 A_0 B_0'$$

$$(A < B) = A_3' B_3 + x_3 A_2' B_2 + x_3 x_2 A_1' B_1 + x_3 x_2 x_1 A_0' B_0$$

Combinational Circuits: Multiplexers

Multiplexer: electronic switch that connects one of n inputs to the output

Data selector: application of multiplexer

- n data input lines, $D_0, D_1, \ldots, D_{n-1}$
- m select digit inputs $s_0, s_1, \ldots, s_{m-1}$
- 1 output
- **Can you design a simple data selectors with 2 input data lines?**

(*a*) Block diagram.

(*b*) Logic diagram.

Combinational Circuits: Multiplexers

Data selectors: can implement arbitrary switching functions

Example: implementing two-variable functions

$z = sD_1 + s'D_0$

If $s = A$, $B = D_0$, and $B' = D_1$, then $z = A \oplus B$.

If $s = A$, $D_0 = 1$, and $D_1 = B'$, then $z = A' + B'$.

Implementing Switching Function with Mux

To implement an *n*-variable function: a data selector with n-1 select inputs and 2^{n-1} data inputs

Implementing three-variable functions:

$$z = s_2's_1'D_0 + s_2's_1D_1 + s_2s_1'D_2 + s_2s_1D_3$$

Example: $s_1 = A$, $s_2 = B$, $D_0 = C$, $D_1 = 1$, $D_2 = 0$, $D_3 = C'$

$$z = A'B'C + AB' + ABC'$$
$$= AC' + B'C$$

General case: Assign n-1 variables to the select inputs and last variable and constants 0 and 1 to the data inputs such that desired function results

Implementing Switching Function with Mux

- Y = AC’ + B’C
- Make A, B as select lines.
  - A,B=0,0 => Y=C
  - A,B=0,1 => Y=0
  - A,B=1,0 => Y=C’+C=1
  - A,B=1,1 => Y=C’

Implementing Switching Function with Mux

$$F(A,B,C) = \sum(1,3,5,6)$$

Minterm	A	B	C	F
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

	I_0	I_1	I_2	I_3
A'	0	1	2	3
A	4	5	6	7
	0	1	A	A'

Adders: Half Adder

Add two variables and generate the sum and carry

x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$S = x \oplus y$$

$$C = xy$$

Adders: Full Adder

Add two variables and an input carry..

x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Try it yourself…!!!

Adders: Full Adder

Add two variables and an input carry..

x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$S = x \oplus y \oplus z$$

$$C = xy + yz + zx$$

Adders: Full Adder with Half Adders

Use two half adder and something else to generate a full adder

Try it yourself…!!!

Priority Encoders

Priority encoder: n input lines and $\log_2 n$ output lines

- Input lines represent units that may request service
- When inputs p_i and p_j, such that $i > j$, request service simultaneously, line p_i has priority over line p_j
- Encoder produces a binary output code indicating which of the input lines requesting service has the highest priority

Example: Eight-input, three-output priority encoder

(*a*) Block diagram.

Input lines								Outputs		
p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	z_4	z_2	z_1
1	0	0	0	0	0	0	0	0	0	0
ϕ	1	0	0	0	0	0	0	0	0	1
ϕ	ϕ	1	0	0	0	0	0	0	1	0
ϕ	ϕ	ϕ	1	0	0	0	0	0	1	1
ϕ	ϕ	ϕ	ϕ	1	0	0	0	1	0	0
ϕ	ϕ	ϕ	ϕ	ϕ	1	0	0	1	0	1
ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	1	0	1	1	0
ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	1	1	1	1

(*b*) Truth table.

$$z_4 = p_4p_5'p_6'p_7' + p_5p_6'p_7' + p_6p_7' + p_7 = p_4 + p_5 + p_6 + p_7$$
$$z_2 = p_2p_3'p_4'p_5'p_6'p_7' + p_3p_4'p_5'p_6'p_7' + p_6p_7' + p_7 = p_2p_4'p_5' + p_3p_4'p_5' + p_6 + p_7$$
$$z_1 = p_1p_2'p_3'p_4'p_5'p_6'p_7' + p_3p_4'p_5'p_6'p_7' + p_5p_6'p_7' + p_7 = p_1p_2'p_4'p_6' + p_3p_4'p_6' + p_5p_6' + p_7$$

Priority Encoders

(*c*) Logic diagram.

Decoders

Decoders with *n* inputs and 2^n outputs: for any input combination, only one output is 1

Useful for:

- Routing input data to a specified output line, e.g., in addressing memory
- Basic building blocks for implementing arbitrary switching functions
- Code conversion
- Data distribution

Example: 2-to-4- decoder

Decoders with Enable

4-16 decoder using 3-8 decoder

Usefulness of Enable to expand circuits.

Realizing Arbitrary Functions

Idea: Realize a distinct minterm at each output

Demultiplexer

Demultiplexers: decoder with 1 data input and *n* address inputs

- Directs input to any one of the 2^n outputs

Example: A 4-output demultiplexer

Adders Again

Full adder: performs binary addition of three binary digits

- Inputs: arguments A and B and carry-in C
- Outputs: sum S and carry-out C_0

A	B	C	S	C_0
0	0	0	0	0
0	0	1	1	0
0	1	1	0	1
0	1	0	1	0
1	1	0	0	1
1	1	1	1	1
1	0	1	0	1
1	0	0	1	0

(*a*) Truth table for S and C_0.

(*b*) Block diagram.

$$S = A \oplus B \oplus C$$

$$C_0 = AB + BC + CA$$

Ripple Carry Adder

Ripple-carry adder: Stages of full adders

- C_f: forced carry
- $C_{0(n-1)}$: overflow carry

$$S_i = A_i \oplus B_i \oplus C_i$$

$$C_{0i} = A_i B_i + B_i C_{0i} + C_{0i} A_i$$

Time required:

- **Carry propagation takes longest time — in the worst case, the carry propagates through all the stages**
- Time per full adder: 2 units (assuming each gate takes one unit of time)
  - Time for carry generation
  - Assumption: two level circuit realisation with 2 input gates
- Time for ripple-carry adder: **2*n* units**

Ripple Carry Adder

https://ee.usc.edu/~redekopp/ee457/slides/EE457Unit2b_FastAdders.pdf

Carry Lookahead Adder

Carry-lookahead adder: several stages simultaneously examined and their carries generated in parallel

- Generate signal $D_i = A_i B_i$
- Propagate signal $T_i = A_i \oplus B_i$
- Thus, $C_{0i} = D_i + T_i C_i$

To generate carries in parallel: convert recursive form to nonrecursive

$$
\begin{aligned}
C_{0i} &= D_i + T_i C_i \\
C_i &= C_{0(i-1)} \\
C_{0i} &= D_i + T_i(D_{i-1} + T_{i-1}C_{i-1}) \\
&= D_i + T_i D_{i-1} + T_i T_{i-1}(D_{i-2} + T_{i-2}C_{i-2}) \\
&= D_i + T_i D_{i-1} + T_i T_{i-1} D_{i-2} + T_i T_{i-1} T_{i-2} C_{i-2} \\
\ldots \quad & \ldots\ldots \\
C_{0i} &= D_i + T_i D_{i-1} + T_i T_{i-1} D_{i-2} + \ldots + T_i T_{i-1} T_{i-2} \ldots T_0 C_f
\end{aligned}
$$

Thus, $C_{0i} = 1$ if it has been generated in the ith stage or originated in a preceding stage and propagated to all subsequent stages

Carry Lookahead Adder

Implementation of lookahead for the complete adder impractical:

- Divide the n stages into groups
- Full carry lookahead within group
- Ripple carry between groups

Example: Three-digit adder group with full carry lookahead

(*a*) Block diagram of initial three-stage group

Carry Lookahead Adder

Implementation of lookahead for the complete adder impractical:

- Divide the n stages into groups of 3-bit adders
- Full carry lookahead within group
- Ripple carry between groups

Example: Three-digit adder group with full carry lookahead

(*a*) Block diagram of initial three-stage group

(*b*) The carry networks

Time taken:

- 4 time units for C_{g1} why 4?
- Only 2 time units for C_{g2} and other group carries why?

Carry Lookahead Adder

Example: divide n stages into groups of **three bits** for a **30 bit adder**

- **Very very important!!!**
  - First stage requires more time (2 units extra to generate T_i).
  - Last stage requires more time (2 extra units for the final sum, for the other units it's not coming in the critical path)
- **Time taken**: $4 + 2n/3$ time units
- 50% additional hardware for a threefold speedup

Adder Subtractor

- Implements two's complement subtraction
- The idea is to have both an adder and subtractor in the same circuit
- **Overflow detection**

Adder Subtractor: The Overflow

- **Overflow**: When two numbers with n digits each are added and the sum is a number occupying n + 1 digits, we say that an overflow occurred.
- This is true for binary or decimal numbers, signed or unsigned
- <u>An overflow condition can be detected by observing the carry into the sign bit position and the carry out of the sign bit position. If these two carries are not equal, an overflow has occurred.</u>
- If the numbers are unsigned
  - C bit detects the overflow
- If the numbers are signed
  - V bit detections the overflow

carries:	0	1	carries:	1 0
+70		0 1000110	−70	1 0111010
+80		0 1010000	−80	1 0110000
+150		1 0010110	−150	0 1101010

BCD Adder

- **Adds two BCD Numbers**
- First add in binary and then convert the sum to BCD
- Values beyond 9 requires "correction"
  - For values beyond 9, 0110 needs to added to generate the BCD encoded sum
- We use a bit C to detect when correction is needed
- C = 1, when
  - Carry K or the binary sum is 1
  - Z8 = 1 and Z4 = 1
  - Z8 = 1 and Z2 = 1
- **C = K + Z8Z4 + Z8Z2**

Binary Sum					BCD Sum					Decimal
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

BCD Adder

Addend
Augend
Carry out
K
4-bit binary adder
Carry in
Z_8 Z_4 Z_2 Z_1
Output carry
0
4-bit binary adder
S_8 S_4 S_2 S_1

Digital Logic Design + Computer Architecture

Sayandeep Saha

Assistant Professor
Department of Computer Science and Engineering
Indian Institute of Technology Bombay

Sequential Circuits

A Circuit that Remembers

- **How do you remember things?**
  - Memory
- **Can we design a circuit which remembers?**
  - A formal way to model this capability is called a **state**
  - **So we will be modelling circuits to create a state.**

A Circuit that Remembers

- **Every digital logic you see in real life is sequential**
  - Your processors — that you going to see in the rest of the course
  - Your washing machine — it remembers your setting and washes accordingly
  - Your elevator — it remembers which floors to stop
  - Your ATM machine — it remembers your choice and updates your account after despatching money

Sequential Circuits and Finite State Machines

Sequential circuit: its outputs a function of external inputs as well as stored information (aka. State)

Finite-state machine (FSM): abstract model to describe the synchronous sequential machines. It has finite memory.

Serial binary adder example: block diagram, addition process, <u>state table</u> and <u>state diagram</u>
Let A denote the state of the adder at t_i if the carry 0 is generated at t_{i-1}
Let B denote the state of the adder at t_i if the carry 1 is generated at t_{i-1}

$$
\begin{array}{rcl}
t_5\ t_4\ t_3\ t_2\ t_1 & & \\
0\ 1\ 1\ 0\ 0 & = & X_1 \\
+\ 0\ 1\ 1\ 1\ 0 & = & X_2 \\
\hline
1\ 1\ 0\ 1\ 0 & = & Z
\end{array}
$$

PS	NS, z			
	$x_1x_2 = 00$	01	11	10
A	$A, 0$	$A, 1$	$B, 0$	$A, 1$
B	$A, 1$	$B, 0$	$B, 1$	$B, 0$

Sequential Circuits and Finite State Machines

Two states capable of storing information regarding the **presence or absence of carry**:

delay element with input Y and output y

- Two states: $y = 0$ and $y = 1$
- The capability of the device to store information is the result of the fact that it takes some time for the input signal Y to pass to the output y
  - Compare with a combinational gate where the output changes almost immediately. In this device there is some well-defined time required before the input Y passes to the output y. **During that time-window, the old value stays!**
- Since the **present input value *Y*** of the delay element is equal to its **next output value**: the input value is referred to as the next state of the delay
  - $y(t+1)=Y(t)$

Example: assign state $y = 0$ to state A of the adder and $y = 1$ to B

- The value of y at t_i corresponds to the value of the carry generated at t_{i-1}
- Process of assigning the states of a physical device to the states of the serial adder: called **state assignment**
- Output value y: referred to as the **state variable**
- **Transition/output table** for the serial adder:

y	*Next state Y* x_1x_2 00	01	11	10	*Output z* x_1x_2 00	01	11	10
0	0	0	1	0	0	1	0	1
1	0	1	1	1	1	0	1	0

$$Y = x_1x_2 + x_1y + x_2y$$

$$z = x_1 \oplus x_2 \oplus y$$

Sequential Circuits and Finite State Machines

Memory Element: Latches

Latch: remains in one state indefinitely until an input signals directs it to do otherwise

Set-reset of *SR* latch:

(*a*) Block diagram.

(*b*) NOR latch.

(*c*) NAND latch.

Memory Element: Latches

Characteristic table and excitation requirements:

$y(t)$	$S(t)$	$R(t)$	$y(t+1)$
0	0	0	0
0	0	1	0
0	1	1	?
0	1	0	1
1	1	0	1
1	1	1	?
1	0	1	0
1	0	0	1

Circuit change		*Required value*	
From:	*To:*	S	R
0	0	0	–
0	1	1	0
1	0	0	1
1	1	–	0

$$RS = 0$$
$$y(t+1) = R'y(t) + S$$

Clocked *SR* latch: all state changes synchronized to clock pulses

- Restrictions placed on the length and frequency of clock pulses: so that the circuit changes state no more than once for each clock pulse

(*a*) Block diagram.

(*b*) Logic diagram.

Memory Element: Latches

Why is the (1,1) input forbidden?

$y(t)$	$S(t)$	$R(t)$	$y(t+1)$
0	0	0	0
0	0	1	0
0	1	1	?
0	1	0	1
1	1	0	1
1	1	1	?
1	0	1	0
1	0	0	1

$$RS = 0$$

$$y(t+1) = R'y(t) + S$$

Memory Element: Latches

- A **clock** is a periodic signal that is used to keep time in sequential circuits.
- **Duty Cycle** is the ration of t_{on}/T_{period}
- We want to keep t_{on} small so that in the same clock pulse only a single computation is performed.
- We want to keep T_{period} sufficient so that there is enough time for the next input to be computed.

Memory Element: T Latch

Value 1 applied to its input triggers the latch to change state

(*a*) Block diagram.

(*b*) Deriving the T latch from the clocked SR latch.

Excitations requirements:

Circuit change		*Required*
From:	*To:*	*value T*
0	0	0
0	1	1
1	0	1
1	1	0

"Q" is basically "y"

Characteristic Table

T Flip-Flop

T	$Q(t+1)$	
0	$Q(t)$	No change
1	$Q'(t)$	Complement

$$y(t+1) = Ty'(t) + T'y(t)$$
$$= T \oplus y(t)$$

Memory Element: JK Latch

Unlike the *SR* latch, *J* = *K* = 1 is permitted: when it occurs, the latch acts like a trigger and switches to the complement state

(*a*) Block diagram.

(*b*) Constructing the JK latch from the clocked SR latch.

Excitation requirements:

Circuit change		*Required value*	
From:	*To:*	*J*	*K*
0	0	0	–
0	1	1	–
1	0	–	1
1	1	–	0

"Q" is basically "y"

Characteristic Table

***JK* Flip-Flop**			
J	***K***	***Q*(*t* + 1)**	
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	$Q'(t)$	Complement

Can you write the characteristic equation?

Memory Element: JK Latch

Unlike the *SR* latch, *J* = *K* = 1 is permitted: when it occurs, the latch acts like a trigger and switches to the complement state

(*a*) Block diagram.

(*b*) Constructing the JK latch from the clocked SR latch.

Excitation requirements:

Circuit change		*Required value*	
From:	*To:*	*J*	*K*
0	0	0	–
0	1	1	–
1	0	–	1
1	1	–	0

"Q" is basically "y"

Characteristic Table

***JK* Flip-Flop**			
J	***K***	***Q*(*t* + 1)**	
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	$Q'(t)$	Complement

Can you write the characteristic equation?

$$y(t+1) = Jy(t)' + K'y(t)$$

D Latch — The Latch of Your Life

The next state of the D latch is equal to its present excitation:

$$y(t+1) = D(t)$$

D Flip-Flop

D	Q(t + 1)	
0	0	Reset
1	1	Set

(*a*) Block diagram.

(*b*) Transforming the JK latch to the D latch.

Excitation Table

Q(t)	Q(t+1)	D
0	0	0
0	1	1
1	0	0
1	1	1

How is Your Clock?

Clocked latch: changes state only in synchronization with the clock pulse and no more than once during each occurrence of the clock pulse

Duration of clock pulse: determined by circuit delays and signal propagation time through the latches

- Must be long enough to allow latch to change state, and
- Short enough so that the latch will not change state twice due to the same excitation

Excitation of a *JK* latch within a sequential circuit:

- Length of the clock pulse must allow the latch to generate the *y*'s
- But should not be present when the values of the *y*'s have propagated through the combinational circuit

Master Slave Flip-Flop

Master-slave flip-flop: a type of synchronous memory element that eliminates the timing problems by isolating its inputs from its outputs

Master-slave *SR* flip-flop:

Master-slave *JK* flip-flop: since master-slave *SR* flip-flop suffers from the problem that both its inputs cannot be 1, it can be converted to a *JK* flip-flip

Edge Triggered Flip-Flop

Positive (negative) edge-triggered *D* flip-flip: stores the value at the *D* input when the clock makes a 0 -> 1 (1 -> 0) transition

- Any change at the *D* input after the clock has made a transition does not have any effect on the value stored in the flip-flop

A negative edge-triggered *D* flip-flop:

- When the clock is high, the output of the bottommost (topmost) NOR gate is at *D'* (*D*), whereas the *S*-*R* inputs of the output latch are at 0, causing it to hold previous value
- When the clock goes low, the value from the bottommost (topmost) NOR gate gets transferred as *D* (*D'*) to the *S* (*R*) input of the output latch
  - Thus, output latch stores the value of *D*
- If there is a change in the value of the *D* input after the clock has made its transition, the bottommost NOR gate attains value 0
  - However, this cannot change the *SR* inputs of the output latch

Edge Triggered Flip-Flop

Changes to the input D does not affect the output y and y' when the clock is high.

Edge Triggered Flip-Flop

Output RS-Latch stores a 0 or 1, depending on the value of D.

What happens when D changes and clock is low?

Synthesis of Synchronous Sequential Circuits

Main steps:

1. From a word description of the problem, form a **state diagram** or **table**
2. Check the table to determine if it contains any **redundant states**
   - If so, remove them (We will see this briefly)
3. Select a **state assignment** and determine the type of memory elements
4. Derive **transition** and **output** tables
5. Derive an **excitation table** and obtain **excitation** and **output functions** from their respective tables
6. Draw a **circuit diagram**

Synthesis of Synchronous Sequential Circuits

One-input/one-output sequence detector: produces output value 1 every time sequence 0101 is detected, else 0

- Example: 010101 -> 000101

State diagram and state table:

PS	NS, z $x=0$	$x=1$
A	$B,0$	$A,0$
B	$B,0$	$C,0$
C	$D,0$	$A,0$
D	$B,0$	$C,1$

Transition and output tables:

y_1y_2	Y_1Y_2 $x=0$	$x=1$	z $x=0$	$x=1$
$A \rightarrow 00$	01	00	0	0
$B \rightarrow 01$	01	11	0	0
$C \rightarrow 11$	10	00	0	0
$D \rightarrow 10$	01	11	0	1

D Latch — The Latch of Your Life

The next state of the D latch is equal to its present excitation:

$$y(t+1) = D(t)$$

D Flip-Flop

D	Q(t + 1)	
0	0	Reset
1	1	Set

(*a*) Block diagram.

(*b*) Transforming the JK latch to the D latch.

Excitation Table

Q(t)	Q(t+1)	D
0	0	0
0	1	1
1	0	0
1	1	1

Synthesis of Synchronous Sequential Circuits

Excitation and output maps:

y_1y_2	Y_1Y_2 $x=0$	Y_1Y_2 $x=1$	z $x=0$	z $x=1$
$A \rightarrow 00$	01	00	0	0
$B \rightarrow 01$	01	11	0	0
$C \rightarrow 11$	10	00	0	0
$D \rightarrow 10$	01	11	0	1

Q(t)	Q(t+1)	D
0	0	0
0	1	1
1	0	0
1	1	1

y_1y_2 \ x	0	1
00		
01		
11		
10		1

(*a*) *z* map.

y_1y_2 \ x	0	1
00		
01		1
11	1	
10		1

(*b*) Y_1 map.

y_1y_2 \ x	0	1
00	1	
01	1	1
11		
10	1	1

(*c*) Y_2 map.

$$z = xy_1y_2'$$
$$y_1 = x'y_1y_2 + xy_1'y_2 + xy_1y_2'$$
$$y_2 = y_1y_2' + x'y_1' + y_1'y_2$$

Logic diagram:

Synthesis of Synchronous Sequential Circuits

Another state assignment:

y_1y_2	Y_1Y_2 $x=0$	Y_1Y_2 $x=1$	z $x=0$	z $x=1$
$A \rightarrow 00$	01	00	0	0
$B \rightarrow 01$	01	10	0	0
$C \rightarrow 10$	11	00	0	0
$D \rightarrow 11$	01	10	0	1

$z = xy_1y_2$

$Y_1 = x'y_1y_2' + xy_2$

$Y_2 = x'$

Binary Counter

One-input/one-output modulo-8 binary counter: produces output value 1 for every eighth input 1 value

State diagram and state table:

	NS		$Output$	
PS	$x=0$	$x=1$	$x=0$	$x=1$
S_0	S_0	S_1	0	0
S_1	S_1	S_2	0	0
S_2	S_2	S_3	0	0
S_3	S_3	S_4	0	0
S_4	S_4	S_5	0	0
S_5	S_5	S_6	0	0
S_6	S_6	S_7	0	0
S_7	S_7	S_0	0	1

Binary Counter

Transition and output tables:

PS	NS		z	
$y_3y_2y_1$	$x=0$	$x=1$	$x=0$	$x=1$
000	000	001	0	0
001	001	010	0	0
010	010	011	0	0
011	011	100	0	0
100	100	101	0	0
101	101	110	0	0
110	110	111	0	0
111	111	000	0	1

	$T_3T_2T_1$	
$y_3y_2y_1$	$x=0$	$x=1$
000	000	001
001	000	011
010	000	001
011	000	111
100	000	001
101	000	011
110	000	001
111	000	111

Excitation table for *T*

Circuit change		*Required*
From:	*To:*	*value T*
0	0	0
0	1	1
1	0	1
1	1	0

$$T_1 = x$$
$$T_2 = xy_1$$
$$T_3 = xy_1y_2$$
$$z = xy_1y_2y_3$$

Binary Counter with SR Flip Flops

Transition and output tables:

PS	NS		z	
$y_3y_2y_1$	$x=0$	$x=1$	$x=0$	$x=1$
000	000	001	0	0
001	001	010	0	0
010	010	011	0	0
011	011	100	0	0
100	100	101	0	0
101	101	110	0	0
110	110	111	0	0
111	111	000	0	1

	$x=0$			$x=1$		
$y_3y_2y_1$	S_3R_3	S_2R_2	S_1R_1	S_3R_3	S_2R_2	S_1R_1
000	0–	0–	0–	0–	0–	10
001	0–	0–	–0	0–	10	01
010	0–	–0	0–	0–	–0	10
011	0–	–0	–0	10	01	01
100	–0	0–	0–	–0	0–	10
101	–0	0–	–0	–0	10	01
110	–0	–0	0–	–0	–0	10
111	–0	–0	–0	01	01	01

Excitation table for *SR* flip-flops and logic diagram:

- **Trivially extensible to modulo-16 counter**

Circuit change		*Required value*	
From:	*To:*	*S*	*R*
0	0	0	–
0	1	1	0
1	0	0	1
1	1	–	0

$$S_1 = xy_1'$$
$$R_1 = xy_1$$
$$S_2 = xy_1y_2'$$
$$R_2 = xy_1y_2$$
$$S_3 = xy_1y_2y_3'$$
$$R_3 = z = xy_1y_2y_3$$

Sequential Circuit as a Controller

Control element: streamlines computation by providing appropriate control signals

Example: digital system that computes the value of $(4a + b)$ modulo 16

- a, b: four-bit binary number
- X: register containing four flip-flops
- x: number stored in X
- Register can be loaded with: either b or $a + x$
- Addition performed by: a four-bit parallel adder
- K: modulo-4 binary counter, whose output L equals 1 whenever the count is 3 modulo 4

Sequential Circuit as a Controller

Sequential circuit *M*:

- Input u: initiates computation
- Input L: gives the count of K
- Outputs: α, β, γ, z
- When $\alpha = 1$: contents of b transferred to X
- When $\beta = 1$: values of x and a added and transferred back to X
- When $\gamma = 1$: count of K increased by 1
- $z = 1$: whenever final result available in X

Sequential Circuit as a Controller

Sequential circuit *M*:

- *K*, *u*, *z*: initially at 0
- When *u* = 1: computation starts by setting $\alpha = 1$
  - Causes *b* to be loaded into *X*
- To add *a* to *x*: set $\beta = 1$ and $\gamma = 1$ to keep track of the number of times *a* has been added to *x*
- After four such additions: *z* = 1 and the computation is complete
- At this point: *K* = 0 to be ready for the next computation

State diagram:

Sequential Circuit as a Controller

State assignment, transition table, maps and logic diagram:

PS y_1y_2	NS Y_1Y_2
00	0u
01	11
11	1L'
10	00

(*a*) Transition table.

(*b*) Maps for Y_1 and Y_2.

$$\alpha = y_1'y_2$$
$$\beta = \gamma = y_1y_2$$
$$z = y_1y_2'$$
$$Y_1 = y_2$$
$$Y_2 = y_1'y_2 + uy_1' + L'y_2$$

(*c*) Logic diagram.

Registers: Your Main Sequential Element

- Used to store data
- Basically an **array of D-flip-flops**
- You can **load data**, **reset it to zero**, and **shift it to left and right**

Registers: Your Main Sequential Element

- 4-bit register
- Asynchronous Reset
- On a clock tick, the data in I0, I1, I2, I3 gets available in A0, A1, A2, A3.

Registers: Your Main Sequential Element

- 4-bit register with parallel load
- Asynchronous Reset
- The main difference from the previous design is that in the former case the stored data deliberately changes at every clock tick. But here we have a control through the *Load* line.
- This is your "**the building block**"
- Don't worry you do not need to code it down. Verilog does this internally for you.
  - But maybe you should give it a try
- Your processor in the rest of the course contains such registers!!!

Shift Registers

- Shift the bits left and right
- Again, very easy to write in Verilog

Shift Registers

- Universal Shift Register

(a)

Mode Control		
s_1	s_0	Register Operation
0	0	No change
0	1	Shift right
1	0	Shift left
1	1	Parallel load

Mealy and Moore Machines

Example: 01/10 Detector

Moore

reset	input	current state	next state	current output
1	–	–	A	0
0	0	A	B	0
0	1	A	C	0
0	0	B	B	0
0	1	B	D	0
0	0	C	E	0
0	1	C	C	0
0	0	D	E	1
0	1	D	C	1
0	0	E	B	1
0	1	E	D	1

Mealy

reset	input	current state	next state	current output
1	–	–	A	0
0	0	A	B	0
0	1	A	C	0
0	0	B	B	0
0	1	B	C	1
0	0	C	B	1
0	1	C	C	0

Example: 01/10 Detector

Moore

reset	input	current state	next state	current output
1	–	–	A	0
0	0	A	B	0
0	1	A	C	0
0	0	B	B	0
0	1	B	D	0
0	0	C	E	0
0	1	C	C	0
0	0	D	E	1
0	1	D	C	1
0	0	E	B	1
0	1	E	D	1

Mealy

reset	input	current state	next state	current output
1	–	–	A	0
0	0	A	B	0
0	1	A	C	0
0	0	B	B	0
0	1	B	C	1
0	0	C	B	1
0	1	C	C	0

State Machine Minimization

More states $\longrightarrow$ more flip-flops and logic:

- Cost saving is always important
- Remember circuit minimization
- We shall now see how to minimize the number of states in a state machine
- Again some complex stuff: Sorry :P

State Machine Minimization

n-state machine ⟶ ceil[$\log_2$(n)] state variables

- Sometimes we have redundant states
- Redundant states are also called *equivalent states*

- ***k*-distinguishable states**: Two states S_i and S_j of a machine M are distinguishable iff there exists at least one finite input sequence that, when applied to M, causes different output sequence depending on whether S_i or S_j is the initial state. M is called *k-distinguishable* if the length of the distinguishing sequence is **k.**

	NS, z	
PS	$x = 0$	$x = 1$
A	$E, 0$	$D, 1$
B	$F, 0$	$D, 0$
C	$E, 0$	$B, 1$
D	$F, 0$	$B, 0$
E	$C, 0$	$F, 1$
F	$B, 0$	$C, 0$

- The pair (AB) is 1-distinguishable
- (AE) is 3-distinguishable for the input X = 111 — check it!!

State Machine Minimization: State Equivalence

- **Equivalent states**: Two states S_i and S_j of a machine M are equivalent iff for every possible input sequence the same output sequence is produced regardless of whether S_i or S_j is the initial state.
- **Theorem**: If two state S_i and S_j in M are distinguishable, then they are distinguishable by a sequence of length n-1, where n is the number of states in M.
  - In other words, if two states are k-equivalent for all $k \leq n - 1$, then they are equivalent
- If $S_i = S_j$ and $S_j = S_k$, then $S_i = S_k$. Also, $S_i = S_j \implies S_j = S_i$, and $S_i = S_i$, so this is an *equivalence relation.*
- **Therefore, the set of states can be partitioned into disjoint equivalence classes**
  - This is the key idea used in machine minimization
- If the machine is **completely specified**, (that is, its state transitions and outputs are defined for all inputs) then this equivalence partition is **unique**, —> there is a unique minimal machine.
- But if some of the states and outputs are **not specified**, it is **not unique**
  - Why? Simple — because you have to fill in the unspecified states and outputs (just like don't cares) and you can do it in many ways.

A Simple Special Case

Input Sequence	Present State	Next State $X = 0$	Next State $X = 1$	Present Output $X = 0$	Present Output $X = 1$
reset	*A*	*B*	*C*	0	0
0	*B*	*D*	*E*	0	0
1	*C*	*F*	*G*	0	0
00	*D*	*H*	*I*	0	0
01	*E*	*J*	*K*	0	0
10	*F*	*L*	*M*	0	0
11	*G*	*N*	*P*	0	0
000	*H*	*A*	*A*	0	0
001	*I*	*A*	*A*	0	0
010	*J*	*A*	*A*	0	1
011	*K*	*A*	*A*	0	0
100	*L*	*A*	*A*	0	1
101	*M*	*A*	*A*	0	0
110	*N*	*A*	*A*	0	0
111	*P*	*A*	*A*	0	0

- **Observation**: H, I, K, M, N, P all goes back to A and have the same outputs
- **Observation**: J, L goes back to A and have the same outputs
- So, these states (H, I, K, M, N, P) are clearly equivalent. So is the set (J, L)

A Simple Special Case

Input Sequence	Present State	Next State $X = 0$	Next State $X = 1$	Present Output $X = 0$	Present Output $X = 1$
reset	*A*	*B*	*C*	0	0
0	*B*	*D*	*E*	0	0
1	*C*	*F*	*G*	0	0
00	*D*	*H*	*I*	0	0
01	*E*	*J*	*K*	0	0
10	*F*	*L*	*M*	0	0
11	*G*	*N*	*P*	0	0
000	*H*	*A*	*A*	0	0
001	*I*	*A*	*A*	0	0
010	*J*	*A*	*A*	0	1
011	*K*	*A*	*A*	0	0
100	*L*	*A*	*A*	0	1
101	*M*	*A*	*A*	0	0
110	*N*	*A*	*A*	0	0
111	*P*	*A*	*A*	0	0

Present State	Next State $X = 0$	Next State $X = 1$	Present Output $X = 0$	Present Output $X = 1$
A	*B*	*C*	0	0
B	*D*	*E*	0	0
C	~~*F*~~ *E*	~~*G*~~ *D*	0	0
D	*H*	~~*I*~~ *H*	0	0
E	*J*	~~*K*~~ *H*	0	0
~~*F*~~	~~*L*~~ *J*	~~*M*~~ *H*	~~0~~	~~0~~
~~*G*~~	~~*N*~~ *H*	~~*P*~~ *H*	~~0~~	~~0~~
H	*A*	*A*	0	0
~~*I*~~	~~*A*~~	~~*A*~~	~~0~~	~~0~~
J	*A*	*A*	0	1
~~*K*~~	~~*A*~~	~~*A*~~	~~0~~	~~0~~
~~*L*~~	~~*A*~~	~~*A*~~	~~0~~	~~1~~
~~*M*~~	~~*A*~~	~~*A*~~	~~0~~	~~0~~
~~*N*~~	~~*A*~~	~~*A*~~	~~0~~	~~0~~
~~*P*~~	~~*A*~~	~~*A*~~	~~0~~	~~0~~

A Simple Special Case

Present State	Next State $X = 0$	Next State $X = 1$	Output $X = 0$	Output $X = 1$
A	B	C	0	0
B	D	E	0	0
C	E	D	0	0
D	H	H	0	0
E	J	H	0	0
H	A	A	0	0
J	A	A	0	1

- **Remember**: This simple technique only works when the circuit resets to its initial state after receiving a fixed number of inputs. So this is a special case only

CS305
Computer Architecture

What is Computer Architecture?

Why Study Computer Architecture?

Bhaskaran Raman

Room 406, KR Building

Department of CSE, IIT Bombay

http://www.cse.iitb.ac.in/~br

Computer Architecture

- “Architecture”
  - The art and science of designing and constructing buildings
  - A style and method of design and construction
  - Design, the way components fit together
- Computer Architecture
  - The overall design or structure of a computer system, including the hardware and the software required to run it, especially the internal structure of the microprocessor

Pre-Requisites

- Data Structures and Algorithms (CS213)
  - Arrays, pointers, stack, queue
- Logic Design (CS210)
  - Switching theory
  - Number systems, computer arithmetic
  - Logic circuits, combinatorial logic, K-maps
  - Finite state machines in hardware
  - Arithmetic unit, control unit design
  - CAD, FPGA, VHDL

Course Contents

- Computer organization, von Neumann arch.
- Instruction set design
- Measuring performance, Amdahl's law, CPI
- Datapath and control path
- Pipelining, hazards

Course Contents (continued)

- Memory hierarchy, cache design, cache performance
- Disk storage
- RAID
- Error correction codes, Hamming codes
- I/O Buses

Relation to Other Topics/Courses

Text Book References

4th edn: ARM

MIPS

- **"Computer Organization and Design: The Hardware/Software Interface"**, 3rd edition, David A. Patterson and John L. Hennessy, Elsevier (Restricted South Asia Edition).
  - 5th edition available, ok to follow, I'll follow 3rd edn. closely
- **"Computer Architecture and Organization"**, John P. Hayes, 3rd edition, McGraw Hill.
- Low-price editions, e-books available on amazon/flipkart, buy them, no piracy please!
- Notes from other computer architecture courses

Why Study Computer Architecture?

Q: Why do you think Computer Architecture is important (or unimportant)?

Identify Computer Architecture around you

Example-1: This Video

Watching this video on a computer: several computing devices involved in-between

Example-2: Cell-Phones to PCs

A variety of personal devices: the continuum between cell-phones and PCs

Example-3: Servers, Data Centers, Cloud Computing

Data storage and computing in the cloud: backbone of major Internet services

Example-4: Supercomputers

Specialized but important applications, high-end research

Example-5: Embedded Computers

Small but large in number, very critical roles
Home appliances, vehicles, industry automation

Personal Computing Devices in Numbers

Growth in Processing Power

Moore's Law

Summary: Why Study Computer Architecture?

- Computing central to *information age*
- Computer systems range from very small to very large, low-end to super-computers
- New computing devices, end-user devices
  - How are they designed?
  - What affects their performance?
  - What are the performance optimization metrics?
  - How to optimize these metrics?

CS305
Computer Architecture

Parts of a Computer

Bhaskaran Raman

Room 406, KR Building

Department of CSE, IIT Bombay

http://www.cse.iitb.ac.in/~br

A Hierarchy of Languages

Natural language (almost), with some symbols

Concise; structured, yet human readable

Human readable representation of machine language; not concise

Binary numbers (0's and 1's)

Electrical signals

An Example

Algorithm:
Compute the sum of…

C code:

```
a = b + c;
d = e + f;
```

Assembly code:

```
lw      $s1, 4($s0)
lw      $s2, 8($s0)
add     $s3, $s1, $s2
sw      ($s0), $s3
lw      $s3, 16($s0)
lw      $s4, 20($s0)
add     $s5, $s3, $s4
sw      12($s0), $s5
```

Machine code:

```
...0...1...
...0...1...
...0...1...
...0...1...
...0...1...
...0...1...
...0...1...
...0...1...
```

Digital logic:

Von Neumann Architecture

- The stored program concept: program (instructions) as well as data are stored in memory
- Processor *fetches* instructions from memory, and *executes* them on *data* (also fetched from/to memory)
  - Example from previous slide: LW and SW instructions

The Five Components

- All computers have these five components: input, output, memory, [data path + control path = processor]
- Underlined aspects: topics in this course

- Input: keyboard, mouse; also disk, network
- Output: monitor; also disk, network
- Memory: different kinds of memory
- Data path + control path = processor

Inside a Computer...

Inside a Computer (continued...)

External modem card

PCI slots for expansion (add-on cards)

main memory (RAM) chips

empty slots for RAM expansion in future

Magnetic Tape

Inside a Computer: Summary

- Integrated circuits, or chips:
  - Flat and black
  - Processor (CPU), main memory, cache memory, etc.
- Motherboard:
  - Houses the various chips
  - Also has many I/O interfaces (PCI, USB, Serial, etc.)
- Secondary memory: non-volatile
  - Magnetic disks, optical (CD/DVD), tape, flash-based (e.g. USB pen-drives, CF cards), floppies (obsolete)

CS305
Computer Architecture

SSI

Integrated Circuit (IC) Technology: An Overview

MSI

LSI - 10^4

VLSI

ULSI

Bhaskaran Raman

Room 406, KR Building

Department of CSE, IIT Bombay

http://www.cse.iitb.ac.in/~br

The IC Manufacturing Process

conductors

insulators

transistors

Silicon

chip

faulty

working

yield

IC Cost

$$\text{Cost per die} = \frac{\text{Cost per wafer}}{\text{Dies per wafer} \times \text{yield}}$$

Straightforward algebra

$$\text{Dies per wafer} \approx \frac{\text{Area of wafer}}{\text{Area of die}}$$

Approximation

$$\text{Yield} = \frac{1}{[1 + (\text{Defects per area} \times \text{Area of die}/2)]^2}$$

From experience

- Unit cost of chip decreases with volume of production
  - Fixed costs amortised: design, masks in chip manufacture
  - Tuning to improve yield

Limits to IC Density

- Fundamental physical dimension limits
- Power consumption
  - ~ 2.6 cm^2, 4 GHz Intel Core i7, 88W power
- Fan needed to sink the heat
- Frequency scaling employed

CS305
Computer Architecture

Instruction Set Design

Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

http://www.cse.iitb.ac.in/~br

Instruction Set: What and Why

HLL code examples:

C/C++

```
f()->next = (g() > h()) ? k() : NULL;
```

Perl

```
$line =~ s/abc/xyz/g;
```

- Simple for programmers to write
- But, too complex to be implemented *directly* in hardware
- Solution: express complex statements as a sequence of simple statements
- Instruction set: the set of (relatively) simple instructions using which higher level language statements can be expressed

A Simple Example

C code:

```
a = b + c;
d = e + f;
```

Compiler

Assembly code:

```
lw      $s1, 4($s0)
lw      $s2, 8($s0)
add     $s3, $s1, $s2
sw      ($s0), $s3
lw      $s3, 16($s0)
lw      $s4, 20($s0)
add     $s5, $s3, $s4
sw      12($s0), $s5
```

Assembler: instruction encoding (straight-forward)

Machine code:

```
...0...1...
...0...1...
...0...1...
...0...1...
...0...1...
...0...1...
...0...1...
...0...1...
```

Instruction Set

Instruction set is the interface between hardware and software

Interface: instruction set

Interface design:

- Central part of any system design
- Interface before implementation!
- Allows abstraction, independence
- Challenge: should be easy to use by the layer above

Instruction Set Defines a Machine

HLL code examples:

C/C++

```
f()->next = (g() > h()) ? k() : NULL;
```

Perl

```
$line =~ s/abc/xyz/g;
```

MIPS's instruction set

x86's instruction set

ARM's instruction set

Assembly code (machine code) for MIPS

Assembly code (machine code) for x86

Assembly code (machine code) for ARM

Instruction Set and the Stored Program Concept

- At the processor, two steps in a loop:
  - Fetch instruction from memory
  - Execute instruction
    - May involve data transfer from/to memory

The Two Aspects of an Instruction

- Instruction: operation, operand
  - Example: a := b + c
    - Operation is addition
    - Operands are b, c, a
    - For our discussion: “result” is also considered an operand
- What should be the instruction set? ==
  - What set of operations to support?
  - What set of operands to support?
- We’ll learn these in context of: the <u>MIPS instruction set</u>

Registers: Very Fast Operands

- <u>Registers:</u> very small memory, inside the processor
  - Small ==> <u>fast</u> to read/write
  - Small also ==> easy to encode instructions (as we'll see)
  - Integral part of the instruction set architecture (i.e. the hardware-software interface) [NOT a cache]
- MIPS has 32 registers, each of 32-bits

Some Terminology

- 32-bits = 4-bytes = 1-word
- 16-bits = 2-bytes = half-word
- 1-word is the (common-case) unit of data in MIPS
  - 32-bit architecture, also called MIPS32
  - 64-bit MIPS architecture also exists: MIPS64
  - 32-bit & 64-bit architectures are common
  - Low end embedded platforms: 8-bit or 16-bit architectures

Your First MIPS Instruction

```
add     <res>, <op1>, <op2>
Example:
add     $s3, $s1, $s2
```

- The add instruction has <u>exactly 3 operands</u>
  - Why not support more operands? Variable number?
  - Regularity ==> <u>simplicity</u> in hardware implementation
  - Simplicity ==> <u>fast</u> implementation
- All 3 operands are registers

  $0 - $31
  - In MIPS: 32 registers numbered 0-31
  - $s0-$s7 are assembly language names for register numbers 16-23 respectively (why? answered later)

Constant or Immediate Operands

```
addi    <res>, <op1>, <const>
Example:
addi    $s3, $s1, 123
```

- HLL constructs use immediate operands frequently
- Design principle: *make the common case fast*
  - Most instructions have a version with immediate operands
- Question: common case use of constant addition in C?

Memory Operations: Load and Store

```
lw      <dst_reg>, <offset>(<base_reg>)
sw      <offset>(<base_reg>), <src_reg>
Example:
lw      $s1, 4($s0)
sw      12($s0), $s5
```

- Load and store in units of 1-word: terminology w.r.t. CPU
- Also called data transfer instns: memory ↔ registers
- Address: 32-bit value, specified as base register + offset
- Question: why is this useful?
- Alignment restriction: address has to be a unit of 4 (why? answered later)

Test Your Understanding...

- Is a subi instruction needed? Why or why not?
- Is sub instruction needed? Why or why not?

Test Your Understanding (continued)...

- Translate the following C-code into assembly lang.:
  - Ex1: a[300]=x+a[200]; // all 32-bit int
  - What more information do you need?
  - Ex2: a[300]=x+a[i+j]; // all 32-bit int
  - Can you do it using instructions known to you so far?

```
# a in s0, x in s1
lw      $t0, 800($s0)
add     $t1, $t0, $s1
sw      1200($s0), $t1
```

Registers $t0-$t9 usually used for temporary values

```
# a in s0, x in s1
# i in s2, j in s3
add     $t2, $s2, $s3
muli    $t2, $t2, 4
add     $t3, $t2, $s0
lw      $t0, 0($t3)
add     $t1, $t0, $s1
sw      1200($s0), $t1
```

Notion of Register Assignment

- Registers are *statically assigned* by the compiler to registers
- Register management during code generation: one of the important jobs of the compiler
- Example from previous exercise...

Instructions for Bit-Wise Logical Operations

Logical Operators	C/C++/Java Operators	MIPS Instructions
Shift Left	<<	sll
Shift Right	>>	srl
Bit-by-bit AND	&	and, andi
Bit-by-bit OR	|	or, ori
Bit-by-bit NOT	~	nor

The Notion of the Program Counter

- In MIPS: (only) special instructions for PC manipulation
- PC not part of the register file
- In some other architectures: arithmetic or data transfer instructions can also be used to manipulate the PC

- The program is fetched and executed instruction-by-instruction
- Program Counter (PC)
- A special 32-bit register
- Points to the current instruction
- For sequential execution: PC += 4 for each instruction
- Non-sequential execution implemented through manipulation of the PC

Branching Instructions

- Stored program concept: usually *sequential* execution
- Many cases of non-sequential execution:
  - If-then-else, with nesting
  - Loops
  - Procedure/function calls
  - Goto (bad programming normally)
  - Switch: special-case of nested if-then-else
- Instruction set support for these is required...

Conditional and Unconditional Branches

- Two conditional branch instructions:
  - beq <reg1>, <reg2>, <branch_target>
  - bne <reg1>, <reg2>, <branch_target>
- An unconditional branch, or jump instruction:
  - j <jump_target>
- Branch (or jump) target specification:
  - In assembly language: it is a label
  - In machine language, it is a PC-relative offset
    - Assembler computes this offset from the program

Using Branches for If-Then-Else

```
if(x == 0) { y=x+y; } else { y=x-y; }
```

Convention in my slides:
s0, s1... assigned to variables# in order of appearance

```
# s0 is x, s1 is y
bne     $s0, $zero, ELSE
add     $s1, $s0, $s1
j       EXIT
ELSE:
sub     $s1, $s0, $s1
EXIT:
# Further instructions below
```

$0

Note: use of $zero register (make the common case fast)

Using Branches for Loops

```
while(a[i] != 0) i++;
```

```
# s0 is a, s1 is i
BEGIN:
sll     $t0, $s1, 2
add     $t0, $t0, $s0
lw      $t1, 0($t0)
beq     $t1, $zero, EXIT
addi    $s1, $s1, 1
j       BEGIN
EXIT:
# Further instructions below
```

Q: is $t1 really needed above?

Testing Other Branch Conditions

- slt <dst>, <reg1>, <reg2>
  - slt == set less than
  - <dst> is set to 1 if <reg1> is less than <reg2>, 0 otherwise
- slti <dst>, <reg1>, <immediate>
- Can be followed by a bne or beq instruction
- How about <= or > or >= comparisons?
- Note: register 0 in the register file is always ZERO
  - Denoted $zero in the assembly language
  - Programs use 0 in comparison operations very frequently
- Why not single blt or blti instruction?

For Loop: An Example

```
for(i = 0; i < 10; i++) { a[i] = 0; }
```

```
# s0 is i, s1 is a
addi    $s0, $zero, 0
BEGIN:
slti    $t0, $s0, 10
beq     $t0, $zero, EXIT
sll     $t1, $s0, 2
add     $t2, $t1, $s1
sw      0($t2), $zero
addi    $s0, $s0, 1
j       BEGIN
EXIT:
# Further instructions below
```

Q: min # temporary registers needed for above code secn?

CS305
Computer Architecture

Instruction Encoding

Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

http://www.cse.iitb.ac.in/~br

A Simple Example

C code:

```
a = b + c;
d = e + f;
```

Compiler

Assembly code:

```
lw      $s1, 4($s0)
lw      $s2, 8($s0)
add     $s3, $s1, $s2
sw      ($s0), $s3
lw      $s3, 16($s0)
lw      $s4, 20($s0)
add     $s5, $s3, $s4
sw      12($s0), $s5
```

Assembler: instruction encoding (straight-forward)

Machine code:

```
...0...1...
...0...1...
...0...1...
...0...1...
...0...1...
...0...1...
...0...1...
...0...1...
```

Instruction Encoding

- Encoding: representing instructions as numbers/bits
  - Recall: instructions are also stored in memory!
  - Encoding == (assembly language → machine language)
- MIPS: all instructions are encoded as 32 bits (why?)
- Also, all instructions have *similar* format (why?)

Regularity ⇒ simplicity ⇒ efficient implementation

MIPS Instruction Format

add
sub

opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)

R-type instruction: register-register operations

addi

opcode (6)	rs (5)	rt (5)	immediate/constant or offset (16)

PC+4

I-type instruction: loads, stores, all immediates, *conditional branch, jump register, jump and link register*

PC+4

opcode (6)	offset relative to PC+4 word (26)

J-type instruction: *jump, jump and link, trap and return*

Pseudo-direct addressing

Test Your Understanding…

- What format is used by the slt instruction?
- What instruction format is used by beq ?

CS305
Computer Architecture

Function Call Support in the Instruction Set

Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

http://www.cse.iitb.ac.in/~br

Recall: MIPS Instructions So Far

Arithmetic: add, addi, sub

Logical: and, or, nor

sll, srl

memory: lw, sw

branch: beq, bne, j

$0 - $31

$s0 - $s7

$t0 - $t9

$zero

Basic Function Call Support: What is Needed?

many calls to the same function

function call, return

j is sufficient

remember the return address

jal F_LBL

(1) $ra = PC+4

part of MIPS set of regs

(2) transfer ctrl to F_LBL

PC = F_LBL

jr $ra

jump register

can be any of the 32 regs

Arguments and Return Values

among the 32 MIPS regs

$a0, $a1, $a2, $a3 – arguments

$v0, $v1 – return values

convention – not enforced by MIPS h/w

What About Nested Function Calls?

main() → f() → g()

$ra gets overwritten

Arbitrary nesting of function calls

- where to store $ra?

Need for a Stack Data Structure

- before making a function call, store own return addr onto stack
- restore $ra from stack after function call

e.g. main → f() → g1() → g2()

```
f():
    :
    save $ra onto stack
    jal g1
    restore $ra from stack
    :
    save $ra onto stack
    jal g2
    restore $ra from stack
    :
```

```
f():
    save $ra onto stack
    :
    jal g1
    :
    jal g2
    :
    restore $ra from stack
```

more efficient

Stack Implementation

why does stack grow down?

Q: write code to push $ra onto stack

```
addi $sp, $sp, -4
sw   0($sp), $ra
```

Q: write code to pop $ra from stack

```
lw   $ra, 0($sp)
addi $sp, $sp, 4
```

Arguments and Return Values in Nested Calls

main() → f(...) → g(...)

$a0 – $a3, $v0, $v1 – caller has to save and restore if caller cares

What if arg/retval cannot be accommodated within given regs?

– use stack for these

Caller vs Callee Saving Conventions

main() → f() → g()

caller callee caller callee

caller saved (if it cares)

$a0 – $a3, $v0, $v1

unpreserved

$t0 – $t9

vs

callee saved (if it uses)

$ra

preserved

$s0 – $s7

$sp

f():

use $a2

save $a2:

jal g

restore $a2

use $a2

Stack Frame, or Call Activation Record

More arguments than 4 x 32 bit? Use stack
More return value than 2 x 32 bit? Use stack
Saving registers? Use stack
Local variables? Use stack

The Frame Pointer

$fp – makes it easier for compiler

```
f() {
    int x;
    :
L1 → x used here
    :
    for(int i=0; ...  ) {
L2 →    x used here
    }
    :
} // End f()
```

Ref to x at L2 and L1
will be <u>same</u> w.r.t $fp
will be <u>different</u> w.r.t $sp

$fp – preserved or
callee saved

Memory Organization: Stack, Heap, Text

Recursion Example

```
int factorial(int n) {
  if(n == 0)
    return 1;
  else
    return n*factorial(n-1);
}
```

```
FACT:
addi   $sp, $sp, -8
sw     4($sp), $ra # $ra in stack
sw     0($sp), $a0 # $a0 in stack
bne    $a0, $zero, ELSE
addi   $v0, $zero, 1 # base case
j      EXIT
ELSE:
addi   $a0, $a0, -1
jal    FACT  # recursive call
lw     $a0, 0($sp) # restore $a0
mul    $v0, $v0, $a0
EXIT:
lw     $ra, 4($sp) # restore $ra
addi   $sp, $sp, 8 # restore $sp
jr     $ra   # return to caller
```

A Few More Details: Pseudo-Instructions, Assembler Temporary, Dealing with Bytes/Half-Words

li — addi
ori

beq $s0, 10, EXIT

↓

assembler temporary

addi $at, $zero, 10
beq $s0, $at, EXIT

j FAR-AWAY

lui $at, something
ori $at, $at, something
jr $at

beq
bne
$zero

beq
bnz

lb, lh
sb, sh

lbu, lhu
sbu? shu?

Summary So Far

- MIPS instruction set design
- Instruction encoding
- Instructions for function call support

CS305
Computer Architecture

From HLL Code to Process: Object Files, Linking, Loading

Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

http://www.cse.iitb.ac.in/~br

Steps from HLL Code to Running Process

main.c → main.s → main.o

aux.c → aux.s → aux.o

Object files → main.exe

Linking

Exe file

Loading

library
- reusable code
.so
.a

libmath.a

Contents of an Object File

1. Header
2. Text segment
3. Static data
4. Relocation table: list of unresolved instructions
5. Symbol table: table of unresolved symbols
6. Debugging information

Object Files: An Example

main.s

```
.data Y
main:
…
jal  F          # I1
…
lw   $s0, X     # I2
…
G:
…
lw   $s1, Y     # I3
```

aux.s

```
.data X
F:
…
jal  G          # I4
…
lw   $s2, X     # I5
…
…
…
sw   Y, $s3     # I6
```

```
Relocation table: I1,I2,I3
Symbol table: main,F,G,X,Y
```

```
Relocation table: I4,I5,I6
Symbol table: F,G,X,Y
```

Functionalities of Linker, Loader

- Linker: - generate exe file
  - Organize text and data as it would appear in memory
  - Resolve symbols
  - Rewrite instructions referred to in relocation table
- Loader:
  - Place text and data in memory
  - Init stack, other regs, including args
  - Call main

Boot loader
↓
OS
↓
programs

Dynamic Linking

- Link files on demand
- Why?
  - Smaller exe files, better use of memory
  - Newer libraries can be used seamlessly
- How?
  - Rewrite indirect jump address
  - Use jump table of function pointers
  - **Note:** `jalr` instruction needed to support function pointers

Summary

- HLL code → Compiler → Assembler → Linker → Loader → Executing Process
- Object file contents
- Dynamic linking

CS305
Computer Architecture

Arithmetic in MIPS

Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

http://www.cse.iitb.ac.in/~br

Recall Integer Representation

- Possibilities
  - Sign-magnitude
  - 1's complement
  - 2's complement
- Which representation does MIPS use? Why?

Implications of Number Representation for ISA

- Unsigned versions of **`lb, lh: lbu, lhu,`** (**`lwu`** ?)
- Many instructions have unsigned versions
  - **`add`** → **`addu, addi`** → **`addiu, sub`** → **`subu`**
  - **`slt`** → **`sltu, slti`** → **`sltiu`**
- Questions:
  - Is result if **`add`** and **`addu`** always the same?
  - So why separate **`addu`** instruction ?
  - Using single **`sltu`** or **`sltiu`** to check array bounds

Integer Multiplication and Division in MIPS

- Use of special registers **`hi, lo`**
  - **`mult, multu`** store 64-bit result in **`hi, lo`**
  - **`div, divu`** store quotient in lo, remainder in hi
- How to get result from **`hi, lo`** ?
  - Special instructions: **`mfhi <reg>, mflo <reg>`**
  - Why not instructions for moving values to **`hi, lo`** ?

Recall Floating Point Representation

- Normalized scientific notation
  - Sign, significand, exponent (biased 2's complement)
  - Single precision: 32 = 1 + 8 + 23
  - Double precision: 64 = 1 + 11 + 52
  - Subnormal numbers, NaN
  - IEEE 754 standard
    - Supported in MIPS, same as in C float/double

MIPS Floating Point Instructions

- Arithmetic: **+, -, x, /** add.s, add.d sub.s/d mul.s/d div.s/d
- Comparisons: **eq, ne, lt, le, gt, ge**
  - Set a special bit c.eq.s c.le.d
  - To be used in next instruction: **bc1t, bc1f** co-processor
- Memory operations: **lwc1, ldc1, swc1, sdc1**
- MIPS has separate 32 x 32-bit FP registers
  - Why don't we need additional bit in instruction encoding?
  - Can be considered as 16 x 64-bit FP registers for double

Common Programming Bugs

- Signed versus unsigned
- Not handling overflow
- Assuming FP associativity
- Assuming FP precision

$a \times (b \times c) \neq (a \times b) \times c$

$x == y$

CS305
Computer Architecture

Computer Performance Quantification

Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

http://www.cse.iitb.ac.in/~br

Aspects of Computer Performance

Who cares?

Customer
Vendor
Computer designer (us)

Metrics

time to deliver
reliability — % past failed attempts
— % past delays
cost, per unit weight
max weight
delivery confirmation
online tracking
#destinations on offer
pickup service

Computer Performance Equation

customer = one executing program

metric - execution time ✓
response time

Linux: time

Sysad: throughput
multi-user environment

Prog. execn. time	= #cycles x clock-cycle-time
	= #cycles / clock-frequency
Prog. execn. time	= #instrns x [#cycles/instrn] x clock-cycle-time

executed

average across executed instructions

Mnemonic: $\frac{\text{Time}}{\text{Prog}} = \frac{\text{Instructions}}{\text{Prog}} \times \frac{\#\text{cycles}}{\text{instrn}} \times \frac{\text{Time}}{\text{cycle}}$

Use of the Computer Performance Equation: Example-1

Should MIPS support **blt** instruction?

Option-A: yes

Implications: #instructions = 5 million, 20% higher cycle time

Option-B: no

Implications: 10% instructions are blt, need to be replaced with 2 instructions

Execn time in A: 5 million x CPI x (1.2 t) slower

Execn time in B: 5.5 million x CPI x t faster

Use of the Computer Performance Equation: Example-2

Two implementations of the same instruction set:
Implementation-1: 2GHz, CPI=1.5
Implementation-2: 2.4GHz, CPI=2
Which is faster and by how much?

Execn. time
Option-1: $I \times 1.5 \times 1/(2\times10^9)$

Option-2: $I \times 2 \times \frac{1}{(2.4\times10^9)} \rightarrow \frac{20}{18}$

Use of the Computer Performance Equation: Example-3

Intel instruction set supports memory as operand in add:

Option-1: implement add as 3 cycles

Implication: #cycles increases from 2 million to 2.4 million

Option-2: additional hardware support

Implication: cycle length increases by 10%

Which option is better?

ET1: $(2.4 \times 10^6) \times t$

ET2: $(2 \times 10^6) \times 1.1t$ faster

Measuring the Factors in the Performance Equation

instrns × CPI × cycle time

executed

profiling

Simulators

h/w counters

instruction mix

$$CPI_{ave} = \sum CPI_i \times f_i$$

Types of instrns:

arithmetic

memory

branch

0.25×2

$+ 0.75 \times 1$

$= 1.25$

Factors Affecting Performance

Factor	Aspects Affected
Algorithm	#instructions, sometimes CPI
Programming Language	#instructions, CPI
Compiler	#instructions, CPI
ISA	#instructions, CPI, cycle time
Hardware implementation	CPI, cycle time

Algorithm HLL Compiler ISA H/w impl.

Compiler Design Decision Example

CPI for branch instructions: 2
CPI for lw/sw: 3
CPI for reg-reg: 1
Code-sequence-1: 8 branches, 8 loads, 2 stores, 8 reg-reg
Code-sequence-2: 2 branches, 14 loads, 2 stores, 8 reg-reg
Which is better? By what factor?

ET1: $(8\times2 + 8\times3 + 2\times3 + 8\times1)\times t$

ET2: $(2\times2 + 14\times3 + 2\times3 + 8\times1)\times t$

faster: $\frac{60}{54}$

Workload, Benchmark

- Which program to use for performance analysis?
- Benchmark: special or real ?
- SPEC: System Performance Evaluation Corporation
  - Since 1989
  - SPEC-2000 in textbook (includes gzip, gcc)
- How to summarize performance?
  - Think in terms of reproducibility of results
  - Give all possible details, e.g. input to program

Amdahl's Law

Performance improvement is limited by the fraction of program you are improving

$$\frac{1}{1-F+F/n} < \frac{1}{1-F}$$

Amdahl's Law: An Example

Intel wants to improve its CPU chip
Option-1: memory speedup 10x
Option-2: ALU speedup 2x
$F_{alu} = 0.5$, $F_{mem} = 0.2$, $F_{other} = 0.3$

Speedup 1 : $\dfrac{1}{1-0.2+\frac{0.2}{10}} \simeq 1.22$

Speedup 2 : $\dfrac{1}{1-0.5+\frac{0.5}{2}} \simeq 1.33$ ✓

Summary

- Computer performance quantification
  - Execution time is the primary metric
  - Helps answer various design questions quantitatively
- Role of benchmarks
- Amdahl's law

CS305
Computer Architecture

Hardware Implementation of MIPS: Preliminaries

Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

http://www.cse.iitb.ac.in/~br

Recall: Combinatorial vs Sequential Circuits

Combinatorial circuit

Combinational circuit

Sequential circuit

State elements

asynchronous

Synchronous
clock

level-triggered

edge-triggered

Sequential + Combinational Circuit

setup time

hold time

$c > d + s + h$

Control Path, Data Path Analogy

Datapath

- platforms
- rail lines

Control path / signals

- trains should not collide
- no deadlock

Control Path, Data Path in MIPS Implementation

Data path

- state elts
- combinational elts
- interconnections

Control path

- when to write
- what to write

Summary

- Synchronous (clocked) sequential circuit, edge-triggered
- Increasing complexity:
  - Single-cycle implementation
  - Multi-cycle implementation
  - Pipelined implementation
- **DRAW** circuit diagrams to **LEARN** effectively
  - Not enough to look at drawings

CS305
Computer Architecture

Single Cycle Implementation of MIPS ISA (Subset)

Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

http://www.cse.iitb.ac.in/~br

MIPS ISA Subset

- Reg-reg: **`add, sub, and, or, slt`**
- Memory: **`lw, sw`**
- Branch: **`beq`** (and **`j`** later)
- Subset ➔ keep it simple, understand techniques

Before Proceeding, Test Your Understanding

- What kind of arguments does **`add`** take ?
- How many registers need to be specified in **`lw`** ?
- How many registers need to be specified in **`sw`** ?
- What is the least integer value of offset in **`lw`** ?
- What is the largest integer value of offset in **`sw`** ?
- What instruction format is used by **`beq`** ?
- What is the role of the immediate operand in **`beq`** ?

Recall: MIPS Instruction Format

opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)

R-type instruction: register-register operations

opcode (6)	rs (5)	rt (5)	immediate/constant or offset (16)

I-type instruction: loads, stores, all immediates, *conditional branch, jump register, jump and link register*

opcode (6)	offset relative to PC (26)

J-type instruction: *jump, jump and link, trap and return*

Steps in Program Execution

(1) Fetch

Execution

add, sub, and, or, slt

lw, sw

beq

(2a) read register file

(2b) Specific to instruction

(2c) PC = PC+4

Hardware components

- put them together

Elements for Instruction Fetch

Element for Register Read/Write: The Register File

ALU: Arithmetic Logic Unit

Data Memory

Element for PC+4 Computation

Additional Elements to Implement beq

Putting it All Together

Summary of Control Lines

- RegDst (1): to decide Rd vs Rt
- RegWr (1): should register file be written?
- ALUSrc (1): to decide Rt vs SignExtImm
- MemRead (1): should data memory be read?
- MemWrite (1): should data memory be written?
- Mem2Reg (1): to decide ALUOut vs MemOut
- Branch (1): is this a **`beq`** instruction?
- AluOp (4): which ALU operation to perform

Main Control Unit, ALU Control Unit

Truth Table for Main Control Unit

	RegDst	RegWr	ALUSrc	MemRd	MemWr	Mem2Reg	Branch	ALUOp2
Reg-Reg	Rd (0)	1	Reg2Val (0)	0	0	ALUOut (0)	0	10
lw	Rt (1)	1	SgnExt-Imm (1)	1	0	MemOut (1)	0	00
sw	x	0	SgnExt-Imm (1)	0	1	x	0	00
beq	x	0	Reg2Val (0)	0	0	x	1	01

- Can produce optimized combinational circuit to implement truth table
- Similar truth table for ALU control unit as well
- Q: why is MemRd always explicitly enabled or disabled?

Summary

- Single cycle implementation of MIPS ISA subset
  - Sequential, combinational components for different instructions
  - Put together in a datapath
  - Control lines define the control path
  - Control lines generated from opcode + funccode
- Next: extending the implementation to support other instructions

CS305
Computer Architecture

Extending the Single Cycle MIPS Implementation

reg-reg
lw, sw
beq

Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

http://www.cse.iitb.ac.in/~br

Data Path, Control Path Extensions to Support `j`

Main Control Unit Truth Table Enhancement

	RegDst	RegWr	ALUSrc	Mem-Rd	Mem-Wr	Mem2-Reg	Bran-ch	ALU-Op2	Jump
Reg-Reg	Rd (0)	1	Reg2Val (0)	0	0	ALUOut (0)	0	10	0
lw	Rt (1)	1	SgnExt-Imm (1)	1	0	MemOut (1)	0	00	0
sw	x	0	SgnExt-Imm (1)	0	1	x	0	00	0
beq	x	0	Reg2Val (0)	0	0	x	1	01	0
j	x	0	x	0	0	x	x	x	1

Further Data Path, Control Path Extensions to Support `jr`

Main Control Unit Truth Table Enhancement

	Reg-Dst	Reg-Wr	ALUSrc	Mem-Rd	Mem-Wr	Mem2-Reg	Bran-ch	ALU-Op2	Jump	Jr
Reg-Reg	Rd (0)	1	Reg2Val (0)	0	0	ALUOut (0)	0	10	0	0
lw	Rt (1)	1	SgnExt-Imm (1)	1	0	MemOut (1)	0	00	0	0
sw	x	0	SgnExt-Imm (1)	0	1	x	0	00	0	0
beq	x	0	Reg2Val (0)	0	0	x	1	01	0	0
j	x	0	x	0	0	x	x	x	1	0
jr	x	0	x	0	0	x	x	x	x	1

Alternate Data Path, Control Path Modification to Support `j, jr`

Will be a function of opcode and zero?

Summary

- The data-path and control-path can be extended to support further instructions
  - In some cases, original data-path, control-path must be modified
- Understand general principle before proceeding
  - Identify hardware component(s) for instruction
  - String them together, enhance data-path, control-path
  - Enhance/modify main control unit (truth table)

CS305
Computer Architecture

Analysis of the Single Cycle MIPS Implementation

Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

http://www.cse.iitb.ac.in/~br

Single Cycle Implementation: When to Activate Each Element?

- Edge triggered versus level triggered?
  - Different (sequential) elements must trigger at different delays with respect to the start of an instruction fetch+execution
  - This can happen only in a level triggered implementation
- Uncontrolled state change with level triggering?

  - Make PC-write level triggered with latter half of cycle
  - Also make RegWr level triggered with latter half of cycle
  - Also make MemWr level triggered with latter half of cycle

Single Cycle Implementation is Inefficient

- Additional delay in clock cycle
  - Each instruction must be as long as instruction with longest delay!
  - E.g. **`j`** versus **`lw`**
- Cost: several additional pieces of hardware
  - I vs D memory separation
  - Adder for PC+4 is separate
  - Adder for BrTgt is separate

Illustrating the Inefficiency

- Main components:
  - Instruction memory
  - Register file
  - ALU
  - Data memory
- Say, each component takes 100 pico-sec
- What is the max clock frequency?

$0.35 \times 400 + 0.25 \times 400 + 0.2 \times 500 + 0.2 \times 300$

Components involved in:

- Register-register instructions
- `sw`
- `lw`
- `beq`
- `j`

`lw` is longest: 5 x 100 ps = 0.5 ns

Max clock frequency = 2 GHz

Suppose we (magically) have variable clock cycles?

Instruction mix: R 35%, sw 25%, lw 20%, beq 20%

Avg. time per instrn. = 400 ps

Summary

- Single cycle implementation
  - Higher cost than what looks optimal
  - Slower than what looks optimal
- Multi-cycle implementation addresses both
  - Reuse of hardware components
  - Each instruction takes only as long as needed