ECE/CS 250
Computer Architecture

Course review

Tyler Bletsch
Duke University

Includes work by
Daniel J. Sorin (Duke), Amir Roth (Penn), and Alvin Lebeck (Duke)
INTRODUCTION
Course objective:
Evolve your understanding of computers
System Organization

I/O Bus

- CPU
- cache
- memory bus
- main memory

I/O Bridge
- disk controller
- graphics controller
- network interface

Disk
Disk
Graphics
Network
C PROGRAMMING
What is C?

- The language of UNIX
- Procedural language (no classes)
- Low-level access to memory
- Easy to map to machine language
- Not much run-time stuff needed
- Surprisingly cross-platform

**Why teach it now?**

To expand from basic programming to operating systems and embedded development.

Also, as a case study to understand computer architecture in general.
There is NO bounds checking in C

i.e., it’s legal (but not advisable) to refer to
`days_in_month[216]` or
`days_in_month[-35]`!

who knows what is stored there?
Structures

- Structures are sort of like Java objects
  - They have member variables
  - But they do NOT have methods!

- Structure definition with `struct` keyword
  ```c
  struct student_record {  
    int id;  
    float grade;  
  } rec1, rec2;
  ```

- Declare a variable of the structure type with `struct` keyword
  ```c
  struct student_record onerec;
  ```

- Access the structure member fields with dot (\`\.`), e.g. `structvar.member`
  ```c
  onerec.id = 12;  
onerec.grade = 79.3;
  ```
Let's look at memory addresses!

- You can find the address of ANY variable with:

\[ \& \]

The address-of operator

```c
int v = 5;
printf("%d\n", v);
printf("%p\n", &v);
```

```bash
$ gcc x4.c && ./a.out
5
0x7ffffffd232228c
```
What’s a pointer?

- It’s a **memory address** you treat as a **variable**
- You declare pointers with:

```c
int v = 5;
int* p = &v;
printf("%d\n", v);
printf("%p\n", p);
```

The *dereference* operator

```
$ gcc x4.c && ./a.out
5
0x7fffe0e60b7c
```
What’s a pointer?

• You can **look up** what’s stored *at* a pointer!
• You **dereference** pointers with:

\[
\text{int } v = 5; \\
\text{int* } p = \&v; \\
\text{printf(“%d\n”, v);} \\
\text{printf(“%p\n”, p);} \\
\text{printf(“%d\n”, *p);} \\
\]

```bash
$ gcc x4.c && ./a.out
5 0x7fffe0e60b7c 5
```
C Memory Allocation

- **void* malloc(nbytes)**
  - Obtain storage for your data (like `new` in Java)
  - Often use `sizeof(type)` built-in returns bytes needed for `type`
  - `int* my_ptr = malloc(64);` // 64 bytes = 16 ints
  - `int* my_ptr = malloc(64*sizeof(int));` // 64 ints

- **free(ptr)**
  - Return the storage when you are finished (no Java equivalent)
  - `ptr` must be a value previously returned from `malloc`
DATA REPRESENTATIONS AND MEMORY
Decimal to binary using remainders

<table>
<thead>
<tr>
<th>?</th>
<th>Quotient</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>457</td>
<td>228</td>
<td>1</td>
</tr>
<tr>
<td>228</td>
<td>114</td>
<td>0</td>
</tr>
<tr>
<td>114</td>
<td>57</td>
<td>0</td>
</tr>
<tr>
<td>57</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>28</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

111001001
### Decimal to binary using comparison

<table>
<thead>
<tr>
<th>Num</th>
<th>Compare $2^n$</th>
<th>$\geq$ ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>457</td>
<td>256</td>
<td>1</td>
</tr>
<tr>
<td>201</td>
<td>128</td>
<td>1</td>
</tr>
<tr>
<td>73</td>
<td>64</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

111001001
Binary to/from hexadecimal

• 0101101100100011₂ -->

• 0101 1011 0010 0011₂ -->

• 5  B  2  3₁₆

  1  F  4  B₁₆ -->

0001 1111 0100 1011₂ -->

0001111101001011₂
2’s Complement Integers

- Use large positives to represent negatives
  - \((-x) = 2^n - x\)
- This is 1’s complement + 1
  - \((-x) = 2^n - 1 - x + 1\)
- So, just invert bits and add 1

6-bit examples:

- \(010110_2 = 22_{10}; 101010_2 = -22_{10}\)
- \(1_{10} = 000001_2; -1_{10} = 111111_2\)
- \(0_{10} = 000000_2; -0_{10} = 000000_2 \rightarrow \text{good!}\)
- \(1_{10} = 000000_2; -1_{10} = 111111_2\)
Floating point

- **32-bit float format:**

  
  ![IEEE 754 Floating Point Standard](image)
  
  
  number = \((-1)^s \times (1.m) \times 2^{(e-127)}\)

- **64-bit double format:**
  
  (same thing, but with more bits)

  ![Double Precision](image)
<table>
<thead>
<tr>
<th>Dec</th>
<th>Hx</th>
<th>Oct</th>
<th>Html</th>
<th>Chr</th>
<th>Dec</th>
<th>Hx</th>
<th>Oct</th>
<th>Html</th>
<th>Chr</th>
<th>Dec</th>
<th>Hx</th>
<th>Oct</th>
<th>Html</th>
<th>Chr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>000</td>
<td>null</td>
<td>32</td>
<td>20</td>
<td>040</td>
<td> </td>
<td>Space</td>
<td>64</td>
<td>40</td>
<td>100</td>
<td> </td>
<td>0</td>
<td>96</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>SOH</td>
<td>(start of heading)</td>
<td>33</td>
<td>21</td>
<td>041</td>
<td> </td>
<td> </td>
<td>65</td>
<td>41</td>
<td>101</td>
<td> </td>
<td>0</td>
<td>97</td>
</tr>
<tr>
<td>2</td>
<td>002</td>
<td>STX</td>
<td>(start of text)</td>
<td>34</td>
<td>22</td>
<td>042</td>
<td> </td>
<td>!</td>
<td>66</td>
<td>42</td>
<td>102</td>
<td> </td>
<td>0</td>
<td>98</td>
</tr>
<tr>
<td>3</td>
<td>003</td>
<td>ETX</td>
<td>(end of text)</td>
<td>35</td>
<td>23</td>
<td>043</td>
<td> </td>
<td>&quot;</td>
<td>67</td>
<td>43</td>
<td>103</td>
<td> </td>
<td>0</td>
<td>99</td>
</tr>
<tr>
<td>4</td>
<td>004</td>
<td>EOT</td>
<td>(end of transmission)</td>
<td>36</td>
<td>24</td>
<td>044</td>
<td> </td>
<td>#</td>
<td>68</td>
<td>44</td>
<td>104</td>
<td> </td>
<td>100</td>
<td>64</td>
</tr>
<tr>
<td>5</td>
<td>005</td>
<td>ENQ</td>
<td>(enquiry)</td>
<td>37</td>
<td>25</td>
<td>045</td>
<td> </td>
<td>$</td>
<td>69</td>
<td>45</td>
<td>105</td>
<td> </td>
<td>101</td>
<td>65</td>
</tr>
<tr>
<td>6</td>
<td>006</td>
<td>ACK</td>
<td>(acknowledge)</td>
<td>38</td>
<td>26</td>
<td>046</td>
<td> </td>
<td> </td>
<td>70</td>
<td>46</td>
<td>106</td>
<td> </td>
<td>102</td>
<td>66</td>
</tr>
<tr>
<td>7</td>
<td>007</td>
<td>BEL</td>
<td>(bell)</td>
<td>39</td>
<td>27</td>
<td>047</td>
<td> </td>
<td> </td>
<td>71</td>
<td>47</td>
<td>107</td>
<td> </td>
<td>103</td>
<td>67</td>
</tr>
<tr>
<td>8</td>
<td>010</td>
<td>BS</td>
<td>(backspace)</td>
<td>40</td>
<td>28</td>
<td>050</td>
<td> </td>
<td>(</td>
<td>72</td>
<td>48</td>
<td>110</td>
<td> </td>
<td>104</td>
<td>68</td>
</tr>
<tr>
<td>9</td>
<td>011</td>
<td>TAB</td>
<td>(horizontal tab)</td>
<td>41</td>
<td>29</td>
<td>051</td>
<td> </td>
<td>)</td>
<td>73</td>
<td>49</td>
<td>111</td>
<td> </td>
<td>105</td>
<td>69</td>
</tr>
<tr>
<td>10</td>
<td>A</td>
<td>012</td>
<td>LF</td>
<td>(NL line feed, new line)</td>
<td>42</td>
<td>2A</td>
<td>052</td>
<td> </td>
<td>*</td>
<td>74</td>
<td>4A</td>
<td>112</td>
<td> </td>
<td>106</td>
</tr>
<tr>
<td>11</td>
<td>B</td>
<td>013</td>
<td>VT</td>
<td>(vertical tab)</td>
<td>43</td>
<td>2B</td>
<td>053</td>
<td> </td>
<td>+</td>
<td>75</td>
<td>4B</td>
<td>113</td>
<td> </td>
<td>107</td>
</tr>
<tr>
<td>12</td>
<td>C</td>
<td>014</td>
<td>FF</td>
<td>(NP form feed, new page)</td>
<td>44</td>
<td>2C</td>
<td>054</td>
<td> </td>
<td>,</td>
<td>76</td>
<td>4C</td>
<td>114</td>
<td> </td>
<td>108</td>
</tr>
<tr>
<td>13</td>
<td>D</td>
<td>015</td>
<td>CR</td>
<td>(carriage return)</td>
<td>45</td>
<td>2D</td>
<td>055</td>
<td> </td>
<td></td>
<td>77</td>
<td>4D</td>
<td>115</td>
<td> </td>
<td>109</td>
</tr>
<tr>
<td>14</td>
<td>E</td>
<td>016</td>
<td>SO</td>
<td>(shift out)</td>
<td>46</td>
<td>2E</td>
<td>056</td>
<td> </td>
<td>;</td>
<td>78</td>
<td>4E</td>
<td>116</td>
<td> </td>
<td>110</td>
</tr>
<tr>
<td>15</td>
<td>F</td>
<td>017</td>
<td>SI</td>
<td>(shift in)</td>
<td>47</td>
<td>2F</td>
<td>057</td>
<td> </td>
<td>/</td>
<td>79</td>
<td>4F</td>
<td>117</td>
<td> </td>
<td>111</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>020</td>
<td>DLE</td>
<td>(data link escape)</td>
<td>48</td>
<td>30</td>
<td>058</td>
<td> </td>
<td>80</td>
<td>50</td>
<td>120</td>
<td> </td>
<td>112</td>
<td>70</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>021</td>
<td>DC1</td>
<td>(device control 1)</td>
<td>49</td>
<td>31</td>
<td>059</td>
<td> </td>
<td>81</td>
<td>51</td>
<td>121</td>
<td> </td>
<td>113</td>
<td>71</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>022</td>
<td>DC2</td>
<td>(device control 2)</td>
<td>50</td>
<td>32</td>
<td>060</td>
<td> </td>
<td>82</td>
<td>52</td>
<td>122</td>
<td> </td>
<td>114</td>
<td>72</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>023</td>
<td>DC3</td>
<td>(device control 3)</td>
<td>51</td>
<td>33</td>
<td>061</td>
<td> </td>
<td>83</td>
<td>53</td>
<td>123</td>
<td> </td>
<td>115</td>
<td>73</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>024</td>
<td>DC4</td>
<td>(device control 4)</td>
<td>52</td>
<td>34</td>
<td>062</td>
<td> </td>
<td>84</td>
<td>54</td>
<td>124</td>
<td> </td>
<td>116</td>
<td>74</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>025</td>
<td>NAK</td>
<td>(negative acknowledge)</td>
<td>53</td>
<td>35</td>
<td>063</td>
<td> </td>
<td>85</td>
<td>55</td>
<td>125</td>
<td> </td>
<td>117</td>
<td>75</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>026</td>
<td>SYN</td>
<td>(synchronous idle)</td>
<td>54</td>
<td>36</td>
<td>064</td>
<td> </td>
<td>86</td>
<td>56</td>
<td>126</td>
<td> </td>
<td>118</td>
<td>76</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>027</td>
<td>ETB</td>
<td>(end of trans. block)</td>
<td>55</td>
<td>37</td>
<td>065</td>
<td> </td>
<td>87</td>
<td>57</td>
<td>127</td>
<td> </td>
<td>119</td>
<td>77</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>030</td>
<td>CAN</td>
<td>(cancel)</td>
<td>56</td>
<td>38</td>
<td>070</td>
<td> </td>
<td>88</td>
<td>58</td>
<td>130</td>
<td> </td>
<td>120</td>
<td>78</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>031</td>
<td>EM</td>
<td>(end of medium)</td>
<td>57</td>
<td>39</td>
<td>071</td>
<td> </td>
<td>89</td>
<td>59</td>
<td>131</td>
<td> </td>
<td>121</td>
<td>79</td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>032</td>
<td>SUB</td>
<td>(substitute)</td>
<td>58</td>
<td>3A</td>
<td>072</td>
<td> </td>
<td>90</td>
<td>5A</td>
<td>132</td>
<td> </td>
<td>122</td>
<td>7A</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>033</td>
<td>ESC</td>
<td>(escape)</td>
<td>59</td>
<td>3B</td>
<td>073</td>
<td> </td>
<td>91</td>
<td>5B</td>
<td>133</td>
<td> </td>
<td>123</td>
<td>7B</td>
</tr>
<tr>
<td>28</td>
<td>1C</td>
<td>034</td>
<td>FS</td>
<td>(file separator)</td>
<td>60</td>
<td>3C</td>
<td>074</td>
<td> </td>
<td>92</td>
<td>5C</td>
<td>134</td>
<td> </td>
<td>124</td>
<td>7C</td>
</tr>
<tr>
<td>29</td>
<td>1D</td>
<td>035</td>
<td>GS</td>
<td>(group separator)</td>
<td>61</td>
<td>3D</td>
<td>075</td>
<td> </td>
<td>93</td>
<td>5D</td>
<td>135</td>
<td> </td>
<td>125</td>
<td>7D</td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td>036</td>
<td>RS</td>
<td>(record separator)</td>
<td>62</td>
<td>3E</td>
<td>076</td>
<td> </td>
<td>94</td>
<td>5E</td>
<td>136</td>
<td> </td>
<td>126</td>
<td>7E</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>037</td>
<td>US</td>
<td>(unit separator)</td>
<td>63</td>
<td>3F</td>
<td>077</td>
<td> </td>
<td>95</td>
<td>5F</td>
<td>137</td>
<td> </td>
<td>127</td>
<td>7F</td>
</tr>
</tbody>
</table>
Memory Layout

- Memory is an array of bytes, but there are conventions as to what goes where in this array.
  - **Text**: instructions (the program to execute)
  - **Data**: global variables
  - **Stack**: local variables and other per-function state; starts at top & grows down
  - **Heap**: dynamically allocated variables; grows up
- What if stack and heap overlap?????
LEARNING ASSEMBLY LANGUAGE WITH MIPS
The MIPS architecture

- 32-bit word size
- 32 registers ($0$ is zero, $31$ is return address)
- Fixed size 32-bit aligned instructions

### Types of instructions:

- **Math and logic:**
  - or $\$1$, $\$2$, $\$3$ → $\$1 = \$2 \mid \$3$
  - add $\$1$, $\$2$, $\$3$ → $\$1 = \$2 + \$3$

- **Loading constants:**
  - li $\$1$, 50 → $\$1 = 50$

- **Memory:**
  - lw $\$1$, 4($\$2$) → $\$1 = *($\$2 + 4$)$
  - sw $\$1$, 4($\$2$) → *($\$2 + 4$) = $\$1$

- **Control flow:**
  - j label → PC = label
  - bne $\$1$, $\$2$, label → if ($\$1 \neq \$2$) PC=label
Control Idiom: If-Then-Else

- **Control idiom:** *if-then-else*
  
  ```
  if (A < B) A++;  // assume A in register $1
  else B++;       // assume B in $2
  ```

  ```
  slt $3,$1,$2     // if $1<$2, then $3=1
  beqz $3,else    // branch to else if !condition
  addi $1,$1,1
  j    join        // jump to join
  else: addi $2,$2,1
  join:           
  ```

*ICQ: assembler converts “else” operand of beqz into immediate → what is the immediate?*
## MIPS Register Usage/Naming Conventions

### Important:
The only general purpose registers are the $s and $t registers.

Everything else has a specific usage:
- $a = arguments
- $v = return values
- $ra = return address

### Also 32 floating-point registers: $f0 .. $f31

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>zero constant</td>
</tr>
<tr>
<td>1</td>
<td>reserved for assembler</td>
</tr>
<tr>
<td>2</td>
<td>v0 expression evaluation &amp;</td>
</tr>
<tr>
<td>3</td>
<td>v1 function results</td>
</tr>
<tr>
<td>4</td>
<td>a0 arguments</td>
</tr>
<tr>
<td>5</td>
<td>a1</td>
</tr>
<tr>
<td>6</td>
<td>a2</td>
</tr>
<tr>
<td>7</td>
<td>a3</td>
</tr>
<tr>
<td>8</td>
<td>t0 temporary: caller saves</td>
</tr>
<tr>
<td>15</td>
<td>t7</td>
</tr>
<tr>
<td>16</td>
<td>s0 callee saves</td>
</tr>
<tr>
<td>17</td>
<td>...</td>
</tr>
<tr>
<td>23</td>
<td>s7</td>
</tr>
<tr>
<td>24</td>
<td>t8 temporary (cont’d)</td>
</tr>
<tr>
<td>25</td>
<td>t9</td>
</tr>
<tr>
<td>26</td>
<td>k0 reserved for OS kernel</td>
</tr>
<tr>
<td>27</td>
<td>k1</td>
</tr>
<tr>
<td>28</td>
<td>gp pointer to global area</td>
</tr>
<tr>
<td>29</td>
<td>sp stack pointer</td>
</tr>
<tr>
<td>30</td>
<td>fp frame pointer</td>
</tr>
<tr>
<td>31</td>
<td>ra return address</td>
</tr>
</tbody>
</table>

### Registers Summary:
- **General Purpose Registers:** $s0, $s7, $t0, $t7
- **Floating-Point Registers:** $f0 .. $f31
- **Reserved Registers:** $a0 .. $a3, $t8 .. $t15, $k0, $k1, $gp, $sp, $fp, $ra

---

24
MIPS Instruction Formats

- 3 variations on theme from previous slide
  - All MIPS instructions are either R, I, or J type
  - Note: all instructions have opcode as first 6 bits

<table>
<thead>
<tr>
<th>R-type</th>
<th>Op(6)</th>
<th>Rs(5)</th>
<th>Rt(5)</th>
<th>Rd(5)</th>
<th>Sh(5)</th>
<th>Func(6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-type</td>
<td>Op(6)</td>
<td>Rs(5)</td>
<td>Rt(5)</td>
<td>Immed(16)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>J-type</td>
<td>Op(6)</td>
<td>Target(26)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Memory Addressing Issue: Endian-ness

Byte Order

- **Big Endian:** byte 0 is 8 most significant bits IBM 360/370, Motorola 68k, MIPS, SPARC, HP PA-RISC
- **Little Endian:** byte 0 is 8 least significant bits Intel 80x86, DEC Vax, DEC/Compaq Alpha
COMBINATIONAL LOGIC
Truth Tables

- Map any number of inputs to any number of outputs
- Example:
  \[(A \& B) \mid \neg C\]

Start with Empty TT
- Column per input
- Column per output

Fill in Inputs
- Counting in Binary

Compute Output

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Convert truth table to function

- Given a Truth Table, find the formula?

Write down every “true” case

Then OR together:

\( (!A \& !B \& !C) \mid !A \& !B \& C \mid !A \& B \& !C \mid A \& B \& !C \mid A \& B \& C) \)
# Summary of all Boolean axioms

<table>
<thead>
<tr>
<th>Name</th>
<th>AND form</th>
<th>OR form</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identity law</td>
<td>$1 \land A = A$</td>
<td>$0 \lor A = A$</td>
</tr>
<tr>
<td>Null law</td>
<td>$0 \land A = 0$</td>
<td>$1 \lor A = 1$</td>
</tr>
<tr>
<td>Idempotent law</td>
<td>$A \land A = A$</td>
<td>$A \lor A = A$</td>
</tr>
<tr>
<td>Inverse law</td>
<td>$A \land \neg A = 0$</td>
<td>$A \lor \neg A = 1$</td>
</tr>
<tr>
<td>Commutative law</td>
<td>$A \land B = B \land A$</td>
<td>$A \lor B = B \lor A$</td>
</tr>
<tr>
<td>Associative law</td>
<td>$(A \land B) \land C = A \land (B \land C)$</td>
<td>$(A \lor B) \lor C = A \lor (B \lor C)$</td>
</tr>
<tr>
<td>Distributive law</td>
<td>$A \lor (B \land C) = (A \lor B) \land (A \lor C)$</td>
<td>$A \land (B \lor C) = (A \land B) \lor (A \land C)$</td>
</tr>
<tr>
<td>Absorption law</td>
<td>$A \land (A \lor B) = A$</td>
<td>$A \lor (A \land B) = A$</td>
</tr>
<tr>
<td>De Morgan’s law</td>
<td>$\neg (A \land B) = \neg A \lor \neg B$</td>
<td>$\neg (A \lor B) = \neg A \land \neg B$</td>
</tr>
<tr>
<td>Double negation law</td>
<td>$\neg \neg A = A$</td>
<td></td>
</tr>
</tbody>
</table>

Adapted from [http://studytronics.weebly.com/boolean-algebra.html](http://studytronics.weebly.com/boolean-algebra.html)
Guide to Remembering your Gates

Straight like an A
AND \((a, b)\)

Curved, like an O
OR \((a, b)\)

XOR looks like OR (curved line), but has two lines (like an X does)
XOR \((a, b)\)

Circle means NOT
NAND \((a, b)\)
NOR \((a, b)\)
XNOR \((a, b)\)

(NXOR is 1-bit “equals” by the way)

NOT \((a)\)
Designing a 1-bit adder

- So we’ll need to add three bits (including carry-in)
- Two-bit output is the **carry-out** and the **sum**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>Cin</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11</td>
</tr>
</tbody>
</table>

Turn into expression, simplify, circuit-ify, yadda yadda yadda…
A 1-bit Full Adder

![Diagram of a 1-bit Full Adder](image)

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>$C_{in}$</th>
<th>Sum</th>
<th>$C_{out}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Logisim example: basic_logic.circ · full-adder
Example: Adder/Subtractor

Add/Sub

C_{out}

S3
Full Adder

S2
Full Adder

S1
Full Adder

S0
Full Adder

a3
b3

a2
b2

a1
b1

a0
b0

Logisim example
basic_logic.circ - 4bit-addsub
The ALU

[Diagram showing a circuit for an ALU with inputs a and b, carry-in $C_{in}$, 16-bit add/sub module, carry-out $C_{out}$, and output Q.]
SEQUENTIAL LOGIC
D flip flops

- Stores one bit
- Inputs:
  - The data D
  - The clock ‘>’
  - An “enable” signal E
- Outputs:
  - The stored bit output Q (and also its inverse !Q)
- “Commits” the input bit on clock rise, and only if E is high

![D flip flop diagram](image)
Register

- **Register**: N flip flops working in parallel, where N is the word size
Register file

- A set of registers with multiple ports so numbered registers can be read/written.

- How to write:
  - Use decoder to convert reg # to one hot
  - Send write data to all regs
  - Use one hot encoding of reg # to enable right reg

- How to read:
  - 32 input mux (the way we’ve made it) not realistic
  - To do this: expand our world from \{1,0\} to \{1, 0, Z\}
FINITE STATE MACHINES
How FSMs are represented

- **State 1**
  - What input we need to see to do this state transition: 3 / 0
  - What we change the circuit output to as a result of this state transition

- **State 2**
  - “Self-edges” are possible: 7 / 1

"Self-edges" are possible
Mealy vs Moore

Mealy machine: outputs on TRANSITIONS in red

Moore machine: outputs on STATES in red
State Transition Diagram → Truth Table

<table>
<thead>
<tr>
<th>Current State</th>
<th>Input</th>
<th>Next state</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>3</td>
<td>Saw 3</td>
<td>0 (closed)</td>
</tr>
<tr>
<td>Start</td>
<td>Not 3</td>
<td>Start</td>
<td>0</td>
</tr>
<tr>
<td>Saw 3</td>
<td>8</td>
<td>Saw 38</td>
<td>0</td>
</tr>
<tr>
<td>Saw 3</td>
<td>3</td>
<td>Saw 3</td>
<td>0</td>
</tr>
<tr>
<td>Saw 3</td>
<td>Not 8 or 3</td>
<td>Start</td>
<td>0</td>
</tr>
<tr>
<td>Saw 38</td>
<td>4</td>
<td>Saw 384</td>
<td>1 (open)</td>
</tr>
<tr>
<td>Saw 38</td>
<td>3</td>
<td>Saw 3</td>
<td>0</td>
</tr>
<tr>
<td>Saw 38</td>
<td>Not 4 or 3</td>
<td>Start</td>
<td>0</td>
</tr>
<tr>
<td>Saw 384</td>
<td>Any</td>
<td>Saw 384</td>
<td>1</td>
</tr>
</tbody>
</table>
### State Transition Diagram $\rightarrow$ Truth Table

<table>
<thead>
<tr>
<th>Current State</th>
<th>Input</th>
<th>Next state</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 (start)</td>
<td>3</td>
<td>01</td>
<td>0 (closed)</td>
</tr>
<tr>
<td>00</td>
<td>Not 3</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>8</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>3</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>Not 8 or 3</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>11</td>
<td>1 (open)</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>Not 4 or 3</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>Any</td>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

4 states $\rightarrow$ 2 flip-flops to hold the current state of the FSM
inputs to flip-flops are $D_1D_0$
outputs of flip-flops are $Q_1Q_0$
### State Transition Diagram → Truth Table

<table>
<thead>
<tr>
<th>Q1</th>
<th>Q0</th>
<th>Input</th>
<th>D1</th>
<th>D0</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0 (closed)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Not 3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Not 8 or 3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1 (open)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Not 4 or 3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Any</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Input can be 0-9 → requires 4 bits  
input bits are in3, in2, in1, in0
From here, it's just like combinational logic design! Write out product-of-sums equations, optimize, and build.
### State Transition Diagram $\rightarrow$ Truth Table

<table>
<thead>
<tr>
<th>Q1</th>
<th>Q0</th>
<th>In3</th>
<th>In2</th>
<th>In1</th>
<th>In0</th>
<th>D1</th>
<th>D0</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

Output = (Q1 & !Q0 & !In3 & In2 & !In1 & !In0) | (Q1 & Q0)

D1 = (!Q1 & Q0 & In3 & !In2 & !In1 & !In0) | (Q1 & !Q0 & !In3 & In2 & !In1 & !In0) | (Q1 & Q0)

D0 = do the same thing
### State Transition Diagram → Truth Table

<table>
<thead>
<tr>
<th>Q1</th>
<th>Q0</th>
<th>In3</th>
<th>In2</th>
<th>In1</th>
<th>In0</th>
<th>D1</th>
<th>D0</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Remember, these represent **DFF outputs** …and these are the **DFF inputs**

The DFFs are how we store the **state**.
Truth Table → Sequential Circuit

D1 = (!Q1 & Q0 & !in3 & !in2 & !in1 & !in0) | (Q1 & !Q0 & !in3 & in2 & !in1 & !in0) | (Q1 & Q0)

Follow a similar procedure for D0...
How to think about the FSM circuit

Steps:
1. Do truth table
2. Convert to logic circuit
3. Slap down DFFs
4. Hook up DFFs
5. Hook up inputs/outputs

<table>
<thead>
<tr>
<th>Current state</th>
<th>Input</th>
<th>Next state</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Q_1 )</td>
<td>( Q_0 )</td>
<td>( \text{In}_1 )</td>
<td>( \text{In}_0 )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Inputs: \( \text{In}_0 \), \( \text{In}_1 \)

Outputs: \( \text{Out} \)

Combo logic circuit

(Yields) DFF

(A bunch of logic gates)
CPU DATAPATH AND CONTROL
The overall datapath
Exceptions

- **Exceptions and interrupts**
  - Infrequent (exceptional!) events
    - I/O, divide-by-0, illegal instruction, page fault, protection fault, ctrl-C, ctrl-Z, timer

- Handling requires intervention from operating system
  - End program: divide-by-0, protection fault, illegal insn, ^C
  - Fix and restart program: I/O, page fault, ^Z, timer

- Handling should be transparent to application code
  - Don’t want to (can’t) constantly check for these using insns
  - Want “Fix and restart” equivalent to “never happened”
CACHING
Big Concept: Memory Hierarchy

- Use hierarchy of memory components
  - Upper components (closer to CPU)
    - Fast ↔ Small ↔ Expensive
  - Lower components (further from CPU)
    - Slow ↔ Big ↔ Cheap
  - Bottom component (for now!) = what we have been calling “memory” until now

- Make average access time close to L1’s
  - How?
    - Most frequently accessed data in L1
    - L1 + next most frequently accessed in L2, etc.
  - **Automatically** move data up&down hierarchy
Terminology

- **Hit**: Access a level of memory and find what we want
- **Miss**: Access a level of memory and DON’T find what we want
- **Block**: a group of spatially contiguous and aligned bytes
- **Temporal locality**: Recently accessed stuff likely to be accessed again soon
- **Spatial locality**: Stuff near recently accessed thing likely to be accessed soon
Memory Performance Equation

• For memory component L1
  • Access: read or write to L1
  • Hit: desired data found in L1
  • Miss: desired data not found in L1
    • Must get from another (slower) component
  • Fill: action of placing data in L1

• %miss (miss-rate): #misses / #accesses
• t_{hit}: time to read data from (write data to) L1
• t_{miss}: time to read data into M from lower level

• Performance metric
  • t_{avg}: average access time
  \[ t_{avg} = t_{hit} + (\%_{miss} \times t_{miss}) \]
Abstract Hierarchy Performance

How do we compute $t_{\text{avg}}$?

$$
= t_{\text{avg}} - L1 \\
= t_{\text{hit}} - L1 + (\%_{\text{miss}} - L1 \times t_{\text{miss}} - L1) \\
= t_{\text{hit}} - L1 + (\%_{\text{miss}} - L1 \times t_{\text{avg}} - L2) \\
= t_{\text{hit}} - L1 + (\%_{\text{miss}} - L1 \times (t_{\text{hit}} - L2 + (\%_{\text{miss}} - L2 \times t_{\text{miss}} - L2))) \\
= t_{\text{hit}} - L1 + (\%_{\text{miss}} - L1 \times (t_{\text{hit}} - L2 + (\%_{\text{miss}} - L2 \times t_{\text{avg}} - L3))) \\
= \ldots 
$$

Note: Miss at level $X$ = access at level $X+1$
Where to Put Blocks in Cache

• How to decide which frame holds which block?
  • And then how to find block we’re looking for?

• Some more cache structure:
  • Divide cache into sets
    • A block can only go in its set → there is a 1-to-1 mapping from block address to set
  • Each set holds some number of frames = set associativity
    • E.g., 4 frames per set = 4-way set-associative

• At extremes
  • Whole cache has just one set = fully associative
    • Most flexible (longest access latency)
  • Each set has 1 frame = 1-way set-associative = “direct mapped”
    • Least flexible (shortest access latency)
Cache structure math

- Given capacity, block_size, ways (associativity), and word_size.

- Cache parameters:
  - num_frames = capacity / block_size
  - sets = num_frames / ways = capacity / block_size / ways

- Address bit fields:
  - offset_bits = \(\log_2(\text{block\_size})\)
  - index_bits = \(\log_2(\text{sets})\)
  - tag_bits = word_size - index_bits - offset_bits

- Way to get offset/index/tag from address (bitwise & numeric):
  - block_offset = \(\text{addr} \& \text{ones}(\text{offset\_bits})\) = \(\text{addr} \mod \text{block\_size}\)
  - index = \((\text{addr} \gg offset\_bits) \& \text{ones}(\text{index\_bits})\)
    = \((\text{addr} / \text{block\_size}) \mod \text{sets}\)
  - tag = \(\text{addr} \gg (\text{offset\_bits} + \text{index\_bits})\) = \(\text{addr} / (\text{sets} \times \text{block\_size})\)

ones(n) = a string of \(n\) ones = \(((1\ll n)-1)\)
Cache Replacement Policies

- Set-associative caches present a new design choice
  - On cache miss, which block in set to replace (kick out)?
- Some options
  - Random
  - LRU (least recently used)
    - Fits with temporal locality, LRU = least likely to be used in future
  - NMRU (not most recently used)
    - An easier-to-implement approximation of LRU
      - NMRU=LRU for 2-way set-associative caches
  - FIFO (first-in first-out)
    - When is this a good idea?
ABCs of Cache Design

• Architects control three primary aspects of cache design
  • And can choose for each cache independently
• A = Associativity
• B = Block size
• C = Capacity of cache

• Secondary aspects of cache design
  • Replacement algorithm
  • Some other more subtle issues we’ll discuss later
Analyzing Cache Misses: 3C Model

- Divide cache misses into three categories
  - **Compulsory (cold)**: never seen this address before
    - Easy to identify
  - **Capacity**: miss caused because cache is too small – would’ve been miss even if cache had been fully associative
    - Consecutive accesses to block separated by accesses to at least N other distinct blocks where N is number of frames in cache
  - **Conflict**: miss caused because cache associativity is too low – would’ve been hit if cache had been fully associative
    - All other misses
Stores: Write-Through vs. Write-Back

• When to propagate new value to (lower level) memory?
  • **Write-through**: immediately (as soon as store writes to this level)
    + Conceptually simpler
    + Uniform latency on misses
    – Requires additional bandwidth to next level
  • **Write-back**: later, when block is replaced from this level
    • Requires additional “dirty” bit per block → why?
    + Minimal bandwidth to next level
      • Only write back dirty blocks
    – Non-uniform miss latency
      • Miss that evicts clean block: just a fill from lower level
      • Miss that evicts dirty block: writeback dirty block and then fill from lower level
Stores: Write-allocate vs. Write-non-allocate

• What to do on a write miss?
  • **Write-allocate**: read block from lower level, write value into it
    + Decreases read misses
    – Requires additional bandwidth
  • Use with write-back
  • **Write-non-allocate**: just write to next level
    – Potentially more read misses
    + Uses less bandwidth
  • Use with write-through
### Example cache trace

<table>
<thead>
<tr>
<th>Term</th>
<th>Value</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache size</td>
<td>4096</td>
<td>given</td>
</tr>
<tr>
<td>block size</td>
<td>32</td>
<td>given</td>
</tr>
<tr>
<td>ways</td>
<td>2</td>
<td>given</td>
</tr>
<tr>
<td>frames</td>
<td>cache size / block size</td>
<td></td>
</tr>
<tr>
<td>sets</td>
<td>frames / ways</td>
<td></td>
</tr>
<tr>
<td>bits:index</td>
<td>log(_2)(sets)</td>
<td></td>
</tr>
<tr>
<td>bits:offset</td>
<td>log(_2)(block size)</td>
<td></td>
</tr>
<tr>
<td>bits:tag</td>
<td>64 minus the above</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>addr-dec</th>
<th>addr-hex</th>
<th>tag</th>
<th>index</th>
<th>offset</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>38</td>
<td>0026</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>001E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>003E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0005</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2049</td>
<td>0801</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2085</td>
<td>0825</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>003C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4130</td>
<td>1022</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2085</td>
<td>0825</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Example cache trace

<table>
<thead>
<tr>
<th>Term</th>
<th>Value</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache size</td>
<td>4096</td>
<td>given</td>
</tr>
<tr>
<td>block size</td>
<td>32</td>
<td>given</td>
</tr>
<tr>
<td>ways</td>
<td>2</td>
<td>given</td>
</tr>
<tr>
<td>frames</td>
<td>128</td>
<td>cache size / block size</td>
</tr>
<tr>
<td>sets</td>
<td>64</td>
<td>frames / ways</td>
</tr>
<tr>
<td>bits:index</td>
<td>6</td>
<td>(\log_2(\text{sets}))</td>
</tr>
<tr>
<td>bits:offset</td>
<td>5</td>
<td>(\log_2(\text{block size}))</td>
</tr>
<tr>
<td>bits:tag</td>
<td>53</td>
<td>64 minus the above</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>addr-dec</th>
<th>addr-hex</th>
<th>tag</th>
<th>index</th>
<th>offset</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>38</td>
<td>0026</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>miss compulsory</td>
</tr>
<tr>
<td>30</td>
<td>001E</td>
<td>0</td>
<td>0</td>
<td>30</td>
<td>miss compulsory</td>
</tr>
<tr>
<td>62</td>
<td>003E</td>
<td>0</td>
<td>1</td>
<td>30</td>
<td>hit</td>
</tr>
<tr>
<td>5</td>
<td>0005</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>hit</td>
</tr>
<tr>
<td>2049</td>
<td>0801</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>miss compulsory</td>
</tr>
<tr>
<td>2085</td>
<td>0825</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>miss compulsory</td>
</tr>
<tr>
<td>60</td>
<td>003C</td>
<td>0</td>
<td>1</td>
<td>28</td>
<td>hit</td>
</tr>
<tr>
<td>4130</td>
<td>1022</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>miss compulsory</td>
</tr>
<tr>
<td>2085</td>
<td>0825</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>miss conflict</td>
</tr>
</tbody>
</table>
VIRTUAL MEMORY
Figure: caching vs. virtual memory

**Caching**
- Copy if **popular**
- Drop

**Virtual Memory**
- Load if **needed**
- Swap out (RW) or drop (RO)

**Cache**
- Faster
- More expensive
- Lower capacity

**RAM (or SSD)**
- Slower
- Cheaper
- Higher capacity

**Hard disk** (or SSD)
High level operation

Virtual memory

Physical memory

"Page table"

HDD/SSD storage
Demand Paging

Done in hardware

Memory reference → Is in physical memory?

Is page stored on disk?

Success → Load it, success

Invalid reference, abort!

Done by OS (software)

Adapted from Operating System Concepts by Silberschatz, Galvin, and Gagne
Address translation

Adapted from Operating System Concepts by Silberschatz, Galvin, and Gagne
Address translation

Virtual address: \(0000000000000000111000000000101\)

Virtual page number

Page offset

Page table:

<table>
<thead>
<tr>
<th>Index</th>
<th>Data</th>
<th>Valid?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>463</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>116</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>460</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>407</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>727</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>719</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>203</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>192</td>
<td>1</td>
</tr>
</tbody>
</table>

Physical address: \(0000000000000000110000000000101\)

Physical page number

Page offset
Steps in Handling a Page Fault

1. Trap
2. Page is on backing store
3. Page is on backing store
4. Bring in missing page
5. Reset page table
6. Restart instruction

Adapted from Operating System Concepts by Silberschatz, Galvin, and Gagne
• Functionality problem? Add indirection!
• Performance problem? Add cache!

• Address translation too slow?
  • Cache translations in **translation buffer (TB)**
    + Small cache: 16–64 entries, often fully assoc
    + Exploits temporal locality in PT accesses
    + OS handler only on TB miss
Compromise: virtual-physical caches

- Indexed by VAs
- Tagged by PAs
- Cache access and address translation in parallel
  + No context-switching/aliasing problems
  + Fast: no additional $t_{hit}$ cycles

- A TB that acts in parallel with a cache is a TLB
  - Translation Lookaside Buffer

- Common organization in processors today
What Happens if There is no Free Frame?

• **Page replacement** – find *some page* in memory, but not really in use, page it out
  • Algorithm?
  • Want an algorithm which will result in minimum number of page faults
  • *This decision is just like choosing the caching replacement algorithm!*

Adapted from Operating System Concepts by Silberschatz, Galvin, and Gagne
Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system

- Thrashing \(\equiv\) a process is busy swapping pages in and out

Adapted from Operating System Concepts by Silberschatz, Galvin, and Gagne
Working-set model

- \( \Delta \equiv \text{working-set window} \equiv \text{a fixed number of page references} \)
  Example: 10,000 instructions

- \( WSS_i \) (working set of Process \( P_i \)) = total number of pages referenced in the most recent \( \Delta \) (varies in time)
  - if \( \Delta \) too small will not encompass entire locality
  - if \( \Delta \) too large will encompass several localities
  - if \( \Delta = \infty \) \( \Rightarrow \) will encompass entire program

- \( D = \sum WSS_i \) = total demand frames
  - Approximation of locality

- if \( D > m \) \( \Rightarrow \) Thrashing

- Policy if \( D > m \), then suspend or swap out one of the processes

Adapted from Operating System Concepts by Silberschatz, Galvin, and Gagne
Virtual memory summary

- Address translation via **page table**
  - Page table turns VPN to PPN (noting the valid bit)
- Page is marked ‘i’? **Page fault.**
  - If OS has stored page on disk, load and resume
  - If not, this is invalid access, kill app (seg fault)
- Governing policies:
  - Keep a certain **number of frames loaded** per app
  - Kick out frames based on a **replacement algorithm** (like LRU, etc.)
- Looking up page table in memory too slow, so cache it:
  - The **Translation Buffer (TB)** is a hardware cache for the page table
  - When applied at the same time as caching (as is common), it’s called a **Translation Lookaside Buffer (TLB)**.
- **Working set size** tells you how many pages you need over a time window.
- **DRAM** is slower than SRAM, but denser. Needs constant refreshing of data.
I/O
Protection and access

- I/O should be protected, with device access limited to OS

- User processes request I/O through the OS (not directly)

- User processes do so by triggering an **interrupt**, this causes the OS to take over and service the request

- The interrupt/exception facility is implemented in hardware, but triggers OS software
**Connectivity**

- **Bus**: A communication linkage with two or more devices on it
- Various topologies are possible
Communication models

- **Polling**: Ask continuously
  - Often a waste of processor time

- **Interrupts**: Have disk alert the CPU when data is ready
  - But if data packets are small, this interrupt overhead can add up

- **Direct Memory Access (DMA)**: The device itself can put the requested data directly into RAM without the CPU being involved
  - The CPU is alerted via interrupt when the *whole* transaction is done
  - Complication!
    - Now memory can change without notice; interferes with cache
    - Solution: cache listens on bus for DMA traffic, drops changed data
PIPELINING
5 Stage Pipelined Datapath

- Temporary values (PC, IR, A, B, O, D) re-latched every stage
  - Why? 5 insns may be in pipeline at once, they share a single PC?
  - Notice, PC not re-latched after ALU stage (why not?)
**Pipeline Diagram**

- **Pipeline diagram**: shorthand for what we just saw
  - Across: cycles
  - Down: insns
  - Convention: $X$ means \texttt{lw \$4,0\($\$5\$\)} finishes execute stage and writes into X/M latch at end of cycle 4

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{add $3,$2,$1}</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\texttt{lw $4,0($$5$)}</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\texttt{sw $6,4($$7$)}</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pipeline Hazards

• **Hazard**: condition leads to incorrect execution if not fixed
  • “Fixing” typically increases CPI
  • Three kinds of hazards

• **Structural hazards**
  • Two insns trying to use same circuit at same time
  • Fix by proper ISA/pipeline design: Each insn uses every structure exactly once for at most one cycle, always at same stage relative to Fetch

• **Data hazards**
  • Result of dependencies: Need data before it’s ready
  • Solve by (a) **stalling** pipeline (inject NOPs) and (b) having **bypasses** provide data before it formally hits destination memory/register.

• **Control hazards**
  • Result of jump/branch not being resolved until late in pipeline
  • Solve by flushing instructions that shouldn’t have been happening after branch is resolved
  • This incurs overhead: wasted time! Reduce with:
    • **Fast branches**: Add hardware to resolve branch sooner
    • **Delayed branch**: Always execute instruction after a branch (complicates compiler)
    • **Branch prediction**: Add hardware to speculate on if/where the branch goes
Stalling and Bypassing together

Stall = \((D/X.IR.OP == \text{LOAD}) \&\& (F/D.IR.RS1 == D/X.IR.RD) \|| ((F/D.IR.RS2 == D/X.IR.RD) \&\& (F/D.IR.OP != \text{STORE}))\)
Pipeline Diagram: Data Hazard

- Even with bypasses, stalls are sometimes necessary
- Examples:
  - Memory load -> ALU operation
  - Memory load -> Address component of memory load/store

- Example pipeline diagram for a stall due to a data hazard:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $3,$2,$1</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $4,0($3)</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi $6,$4,1</td>
<td>F</td>
<td>d*</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pipeline Diagram: Control Hazard

- Control hazards indicated with $c^*$ (or not at all)
  - “Default” penalty for taken branch is 2 cycles:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addi $3,$0,1</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>bnez $3,targ</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>sw $6,4($7)</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Fast branches reduce the penalty to 1 cycle:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addi $3,$0,1</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>bnez $3,targ</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>sw $6,4($7)</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MULTICORE
Types of parallelism

- **Pipelining** tries to exploit *instruction-level parallelism (ILP)*
  - “How can we simultaneously do steps in this otherwise sequential process?”

- **Multicore** tries to exploit *thread-level parallelism*
  - “How can we simultaneously do multiple processes?”

- **Thread**: A program has one (or more) threads of control
  - A thread has its own PC
  - Threads in a program share resources, especially memory (e.g. sharing a page table)
Two cases of multiple threads

- **Multiprogramming**: run multiple programs at once

- **Multithreaded programming**: write software to explicitly take advantage of multiple threads (divide problem into parallel tasks)
Multiprocessors

- Multiprocessors: have more than one CPU core
  - Historically: multiple discrete physical chips
  - Now: a single chip with multiple cores

**Multiprocessor**
Two drive-throughs, each with its own kitchen
Challenges of multicore

- Two main challenges:
  - **Topologies** of connection (rings, cubes, meshes, buses, etc.)
  - **Cache coherence**: If each core has a cache, then each CPU can have a diverging view of memory!! (BAD)
  - Solution: Intelligent caches that use snooping on the memory bus to spot sharing and react accordingly
  - Different coherence algorithms (performance/complexity tradeoffs)
## Basic differences

<table>
<thead>
<tr>
<th></th>
<th><strong>MIPS</strong></th>
<th><strong>Intel x86</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Design</strong></td>
<td>RISC</td>
<td>CISC</td>
</tr>
<tr>
<td><strong>ALU ops</strong></td>
<td>Register = Register ⊗ Register (3 operand)</td>
<td>Register ⊗ = &lt;Reg</td>
</tr>
<tr>
<td>** Registers**</td>
<td>32</td>
<td>8 (32-bit) or 16 (64-bit)</td>
</tr>
<tr>
<td><strong>Instruction size</strong></td>
<td>32-bit fixed</td>
<td>Variable: up to 15 <em>bytes</em>!</td>
</tr>
<tr>
<td><strong>Branching</strong></td>
<td>Condition in register (e.g. “slt”)</td>
<td>Condition codes set implicitly</td>
</tr>
<tr>
<td><strong>Endian</strong></td>
<td>Either (typically big)</td>
<td>Little</td>
</tr>
<tr>
<td><strong>Variants and extensions</strong></td>
<td>Just 32- vs. 64-bit, plus some graphics extensions in the 90s</td>
<td>A bajillion (x87, IA-32, MMX, 3DNow!, SSE, SSE2, PAE, x86-64, SSE3, SSE4, SSE5, AVX, AES, FMA)</td>
</tr>
<tr>
<td><strong>Market share</strong></td>
<td>Small but persistent (embedded)</td>
<td>80% server, similar for consumer (defection to ARM for mobile is recent)</td>
</tr>
</tbody>
</table>

*Note: x86 refers to the instruction set architecture by Intel.*
64-bit x86 primer

- Registers:
  - General: `rax r bx r cx r dx r di r si r8 r9 .. r15`
  - Stack: `rsp rb p`
  - Instruction pointer: `rip`
- Complex instruction set
  - Instructions are variable-sized & unaligned
- Hardware-supported call stack
  - `call / ret`
  - Parameters in registers `{rdi, rsi, rdx, rcx, r8, r9}`, return value in `rax`
- Little-endian
- These slides use Intel-style assembly language (destination first)
  - GNU tools like `gcc` and `objdump` use AT&T syntax (destination last)
Binary modification
(applies to *all* ISAs)

- Can disassemble binaries (turn into human-readable assembly)
- Do a bunch of cross-referencing to understand functionality (that’s what IDA Pro does)
- Basic blocks of code ending in branches form a flow chart
- Identify behavior and make inferences on author intent

- Can modify by overwriting binary with new instructions (can also *insert* instructions, but this changes layout of binary program, so various pointers have to be updated)

- Cheap and easy technique on x86: overwrite stuff you don’t want with **NOP (0x90)**
THE END