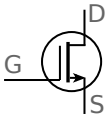
Lecture 1, Sep 8, 2022

- General purpose processors can be too slow for some applications because of overhead
  - Specialized hardware is used for e.g. WiFi
- Hardware is faster & more efficient
- Writing software involves many layers of abstraction
  - Code -> hardware adder -> logic gates -> transistor -> silicon
  - Makes it easier since it hides things we don't need to worry about
- Abstraction vs Complexity: need to manage complexity, find the right level of abstraction to succeed in labs



- Transistors: acting like a switch
  - When voltage on the gate is high, the drain and source are connected
  - Made of silicon chips, L is the length of the gate (current state of the art $L = 14\text{nm}$)
  - Transistors in chips are getting smaller and more powerful following Moore's Law

Why Build Hardware?

- Hardware is faster, but harder to produce and apply, and more expensive
- Why is hardware faster than software? Things get in the way with software:
  - Retrieval of instructions, operands, etc from memory
  - Write results back into memory
  - Keeps asking for the next computation
- Hardware is tailored to a specific purpose, so it doesn't have to ask what to do
- If not fast enough (throughput: things that can be done per unit time), just build more hardware!
- Hardware speed is bottlenecked by speed of electrical signal, wire resistance, capacitance, etc

When to Build Hardware?

- Software is easier to build, test, and manufacture
- Build hardware when software is simply too slow

Assembly Language

- High level languages are machine agnostic (doesn't care about the specific processor)
  - The compiler compiles this down to assembly
- Assembly language is a low level, machine specific language that is still human readable
- An assembler converts this to a native binary executable, which only runs on the specific architecture it was compiled/assembled for
- Assembly is typically only used in special circumstances:
  - Where high speed behaviour down to the instruction is needed
  - Where low level access to hardware is needed, e.g. device drivers
- ASM is a lot closer to hardware and is a stepping stone to learn computer architecture
- RISC-V: Reduced Instruction Set Computer, RISC-V is an open source ISA (instruction set architecture)

Lecture 2, Sep 12, 2022

Number Systems

- Different bases exist:
  - Decimal (base 10)

  - Binary (base 2)
    * Commonly prefixed with $0b$
  - Hexadecimal (base 16)
    * Commonly prefixed with $0x$ or $0h$
    * Each hex digit corresponds to 4 binary digits, allowing a more compact way to write it down

Lecture 3, Sep 13, 2022

Logic Circuits

- Transistors can be used as switches
  - Controlled by input x, either connects or disconnects A and B
  - $L(x) = x$
- Two transistors in series forms an AND gate: $L(x_1, x_2) = x_1 \cdot x_2$, or $L(x_1, x_2) = x_1 x_2$
- Two transistors in parallel forms an OR gate: $L(x_1, x_2) = x_1 + x_2$
- A transistor shorting to ground forms a NOT gate: $L(x) = \bar{x}$, or $L(x) = x'$
  - Also referred to the complement of x

Logic Gates

- Using transistors is tedious, so we can represent each of these with gates:
  - The NOT gate:

  - The AND gate:

  - The OR gate:

- Sometimes NOT gates are simplified to just a bubble before the input to a gate
- Example: $S = a\bar{b} + \bar{a}b$

Truth Tables

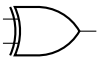
x_1	x_2	AND
0	0	0
0	1	0
1	0	0
1	1	1

x_1	x_2	OR
0	0	0
0	1	1
1	0	1
1	1	1

- Note AND and OR gates can be extended to an arbitrary number of inputs

Other Gates

- The XOR gate, output is 1 if two inputs are different:



  - $L = \bar{x}y + x\bar{y}$
    - When extended to an arbitrary number of inputs, its output is 1 if there are an odd number of 1 inputs
- The NAND gate, output is 0 if both inputs are 1 (i.e. AND + NOT):

  - $L = \overline{(xy)}$
    - An AND gate takes 6 transistors, but a NAND gate takes 4 transistors, so this is cheaper to build
    - NAND gates are functionally complete, i.e. you can build any gate with them
- The NOR gate, output is zero if at least one input is 1:

  - $L = \overline{(x + y)}$
    - NOR gates are also functionally complete

Lecture 4, Sep 15, 2022

Logic Expressions: Sum of Products and Products of Sums

- Terminology:
  - Literal: any variable or its complement, e.g. $x, y, \bar{x}$
  - Product term: an AND operation (since AND is denoted with $\cdot$)
  - Sum term: an OR operation (since OR is denoted with $+$)
- SOP and POS are a way to convert any arbitrary truth table to a logic expression

Sum of Products

Definition

Sum of products: An expression written as an OR operation of AND operations, e.g. $xy + \bar{x}\bar{y}$

- Minterm: A product term that evaluates to 1 for exactly one row of a truth table
  - Given a truth table, the min term is formed by including x_i if $x_i = 1$, or $\bar{x}_i$ if $x_i = 0$
- SOP specifies the truth table based on its ones
- Canonical SOP (Sum-of-Products): SOP expression for a function that comprises its minterms
  - Canonical SOPs are not simplified
- Example:

x	y	z	minterm
0	0	0	$\bar{x}\bar{y}\bar{z}$
0	0	1	$\bar{x}\bar{y}z$
0	1	0	$\bar{x}y\bar{z}$
0	1	1	$\bar{x}yz$
1	0	0	$x\bar{y}\bar{z}$
1	0	1	$x\bar{y}z$
1	1	0	$xy\bar{z}$
1	1	1	xyz

- Example: function comprised of minterms $f(x,y,z) = \sum m(0,1,2,3,6,7)$
  - Canonical SOP: $f(x,y,z) = \bar{x}\bar{y}\bar{z} + \bar{x}\bar{y}z + \bar{x}y\bar{z} + \bar{x}yz + xy\bar{z} + xyz$

Product of Sums

Definition

Product of sums: An expression written as an AND operation of OR operations

- Maxterm: A sum term that evaluates to 0 for exactly one row of a truth table
  - Given a truth table, include x_i if $x_i = 0$ in that row, else include $\bar{x}_i$
- POS specifies the truth table based on its zeroes
- Canonical POS: POS expression for a function that comprises its maxterms
- Example:

x	y	z	maxterm
0	0	0	$x+y+z$
0	0	1	$x+y+\bar{z}$
0	1	0	$x+\bar{y}+z$
0	1	1	$x+\bar{y}+\bar{z}$
1	0	0	$\bar{x}+y+z$
1	0	1	$\bar{x}+y+\bar{z}$
1	1	0	$\bar{x}+\bar{y}+z$
1	1	1	$\bar{x}+\bar{y}+\bar{z}$

- Example: $f(x,y,z) = \sum m(0,1,6,7) = \prod M(2,3,4,5)$
  - Canonical POS: $f(x,y,z) = (x+\bar{y}+z)(x+\bar{y}+\bar{z})(\bar{x}+y+z)(\bar{x}+y+\bar{z})$
- For any truth table, we can use its 1s to derive the SOP, or the 0s to derive the POS
- Example:

x	y	f
0	0	0
0	1	1
1	0	1
1	1	1

- Equivalent representations:
  - POS: $f = (x+y)$
  - SOP: $f = \bar{x}y + x\bar{y} + xy$
- Generally if you have fewer 0s, use POS, if you have fewer 1s, use SOP

Lecture 5, Sep 19, 2022

Boolean Algebra Basics

- Canonical SOP and POS representations can be large and inefficient; boolean algebra lets us simplify and optimize them
- Axioms of Boolean Algebra:
  1. $0 \cdot 0 = 0$
  2. $1 \cdot 1 = 1$
  3. $0 \cdot 1 = 1 \cdot 0 = 0$

4. $x = 0 \implies \bar{x} = 1$

- Duality: given a logic expression, swapping all 0 with 1 and $\cdot$ with $+$ leaves the expression still valid; this gives every axiom a *dual form*:
  1. $1 + 1 = 1$
  2. $0 + 0 = 0$
  3. $1 + 0 = 0 + 1 = 1$
  4. $x = 1 \implies \bar{x} = 0$
- Derived rules: ($\forall x$:)
  5. $x \cdot 0 = 0$, dual: $x + 1 = 1$
  6. $x \cdot 1 = x$, dual: $x + 0 = x$
  7. $x \cdot x = x$, dual: $x + x = x$
  8. $x \cdot \bar{x} = 0$, dual: $x + \bar{x} = 1$
  9. $\bar{\bar{x}} = x$
- Derived identities: ($\forall x, y, z$:)
  10. Commutativity: $x \cdot y = y \cdot x$, dual: $x + y = y + x$
  11. Associativity: $x(yz) = (xy)z$, dual: $x + (y + z) = (x + y) + z$
  12. Distributivity: $x(y + z) = xy + xz$, dual: $x + (yz) = (x + y)(x + z)$
  13. Absorption: $x + xy = x$, dual: $x(x + y) = x$
  14. Combination: $xy + x\bar{y} = x$, dual: $(x + y)(x + \bar{y}) = x$
  15. DeMorgan's Theorem: $\overline{xy} = \bar{x} + \bar{y}$, dual: $\overline{(x + y)} = \bar{x}\bar{y}$
      - Proof:
        $$\begin{aligned} \overline{xy} &= \bar{x}\bar{y} + \bar{x}y + x\bar{y} && \text{Canonical SOP} \\ &= \bar{x}\bar{y} + \bar{x}y + \bar{x}\bar{y} + x\bar{y} && \text{Rule 7} \\ &= \bar{x} + \bar{y} && \text{Combination} \end{aligned}$$
  16. $x + \bar{x}y = x + y$, dual: $x(\bar{x} + y) = xy$

Proof by Perfect Induction

- Proving a statement by enumerating all the possible cases
- Example: proving $x + (yz) = (x + y)(x + z)$

x	y	z	yz	$x + (yz)$	$x + y$	$x + z$	$(x + y)(x + z)$
0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0
0	1	0	0	0	1	0	0
0	1	1	1	1	1	1	1
1	0	0	0	1	1	1	1
1	0	1	0	1	1	1	1
1	1	0	0	1	1	1	1
1	1	1	1	1	1	1	1

Lecture 6, Sep 20, 2022

Functional Completeness of NAND and NOR

- DeMorgan's theorem allows us to implement any SOP circuit can be implemented using only NAND gates:
  - Example:
    $$\begin{aligned} f &= x_1x_2 + x_2x_3 \\ &= \overline{\overline{(x_1x_2)} + \overline{(x_2x_3)}} \\ &= \overline{\overline{x_1x_2} \cdot \overline{x_2x_3}} \end{aligned}$$
  - For a POS circuit, convert to SOP first
- Any POS circuit can be implemented using NOR gates:

- Example: $f = (x_1 + x_2)(x_2 + x_3)$
  $= \overline{\overline{x_1 + x_2}} \cdot \overline{\overline{x_2 + x_3}}$
  $= \overline{\overline{x_1 + x_2} + \overline{x_2 + x_3}}$
- Likely, for a SOP circuit, convert to POS first

Example

- Gumball factory
- s_2 normally 0, but 1 if gumball is too large
- s_1 normally 0, but 1 if too small
- s_0 normally 0, but 1 if too light
- Desired behaviour: $f = 1$ when gumball is either (too large) or (too small and too light)
  - By inspection, $f = s_2 + s_1 s_0$
- Truth table:

$s_2 s_1 s_0$	f
000	0
001	0
010	0
011	1
100	1
101	1
110	1
111	1

- Minterms are the last 5 rows:
  - $f = \bar{s}_1 s_1 s_0 + s_2 \bar{s}_1 \bar{s}_0 + s_2 \bar{s}_1 s_0 + s_2 s_1 \bar{s}_0 + s_2 s_1 s_0$
- Simplify: $f = \bar{s}_1 s_1 s_0 + s_2 \bar{s}_1 \bar{s}_0 + s_2 \bar{s}_1 s_0 + s_2 s_1 \bar{s}_0 + s_2 s_1 s_0$

$= s_1 s_0 (\bar{s}_2 + s_2) + s_2 \bar{s}_1 (s_0 + \bar{s}_0) + s_2 \bar{s}_0 (\bar{s}_1 + s_0)$	Rule 7 + Distributivity
$= s_1 s_0 + s_2 \bar{s}_1 + s_2 \bar{s}_0$	Combination
$= s_1 s_0 + s_2 (\bar{s}_1 + \bar{s}_0)$	Distributivity
$= s_1 s_0 + s_2 \overline{s_1 s_0}$	DeMorgan's Theorem
$= s_1 s_0 + s_2$	Rule 16

Example 2

- Derive a minimal POS expression for $f(x_1, x_2, x_3) = \prod M(0, 2, 4)$

$x_1 x_2 x_3$	f	$\bar{f}$
000	0	1
001	1	0
010	0	1
011	1	0
100	0	1
101	1	0
110	1	0
111	1	0

- $f = (x_1 + x_2 + x_3)(x_1 \bar{x}_2 + x_3)(\bar{x}_1 + x_2 + x_3)$
  $= (x_1 + x_3)(x_2 + x_3)$ Combination (dual)

- Using the min terms of $\bar{f}$:
  - $\bar{f} = \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1x_2\bar{x}_3 + x_1\bar{x}_2\bar{x}_3$
    $= \bar{x}_1\bar{x}_3 + \bar{x}_2\bar{x}_3$
  - Using DeMorgan's rule: $f = \bar{\bar{f}} = \overline{\bar{x}_1\bar{x}_3 + \bar{x}_2\bar{x}_3} = \overline{\bar{x}_1\bar{x}_2} \cdot \overline{\bar{x}_2\bar{x}_3} = (x_1 + x_2)(x_2 + x_3)$

Lecture 7, Sep 22, 2022

SystemVerilog HDL

- SystemVerilog is a hardware description language that allows the specification of logic functions using high level abstractions
- Modules are blocks of hardware with inputs and outputs
- Example:

```
module basic_logic(input logic a, b, // logic: type - indicates boolean variables
                   output logic w, x, y, z);
    // All these things happen at the same time -- not sequentially
    // assign: describes combinational logic
    assign w = a & b; // bitwise AND
    assign x = a | b; // bitwise OR
    assign y = ~a; // bitwise NOT
    assign z = a ^ b; // bitwise OR
endmodule
```

- Modules are not functions – they cannot call themselves

Multiplexers (Mux)

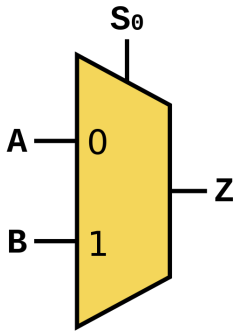


Figure 1: 2 to 1 multiplexer symbol

- A circuit that has an output f, controlled by either of two inputs x, y, based on an input s
  - If s is 0, then f is controlled by x, else y
- Truth table:

sxy	f
000	0
001	0
010	1
011	1
100	0
101	1

sxy	f
110	0
111	1

- Verilog:

```
module mux2to1(input logic x, y, s,
               output logic f);
    assign f = (~s & x) | (s & y);
endmodule
```

- Muxes can be extended to multiple inputs, and also multi-bit signals (buses)
  - A slash with a number is drawn on a wire to indicate that it is a bus
- Example: 2-bit mux
  - x and y are now 2-bit buses
  - Implemented with 2 muxes

```
module mux2to1_2bit(input logic [1:0] x, y, // The [1:0] indicates a 2-bit bus
                    input logic s,
                    output logic [1:0] f);
    // Note we cannot use a single assign statement
    // since s is a scalar, x is a vector, so s & x would be a mismatch
    assign f[1] = (~s & x[1]) | (s & y[1]);
    assign f[0] = (~s & x[0]) | (s & y[0]);
endmodule
```

Adders

- Half adder: adding two one-bit numbers
  - Max result can be 2, so output from the half adder is 2 bits s_1, s_0
- $s_1 = ab, s_2 = a \oplus b$ where $\oplus$ is the XOR operator

```
module ha(input logic a, b,
          output logic [1:0] s);
    assign s[1] = a & b;
    assign s[0] = a ^ b;
endmodule
```

- Full adder: includes a carry input
  - Adding multiple bits involves inputs a_i, b_i and also a carry c_i
  - Each column (except for the rightmost bit) adds 3 bits (2 inputs plus a carry)
  - Leftmost column produces a c_{out}

c_i a_i b_i	c_{i+1}	s_i
000	0	0
001	0	1
010	0	1
011	1	0
100	0	1
101	1	0
110	1	0
111	1	1

- $c_{i+1} = c_i a_i + c_i b_i + a_i b_i$ (carry is 1 if at least 2 inputs are 1, aka a *majority function*), $s_i = a_i \oplus b_i \oplus c_i$

  - s_i is a three-input XOR, equivalent to $(a_i \oplus b_i) \oplus c_i$, which produces 1 if an odd number of inputs is 1 (aka an *odd function*)
- The carry-out of each full adder is connected to the carry-in of the next bit (known as a ripple carry adder)
- Verilog:

```
// Single bit full adder module
module fa(input logic a, b, cin,
          output logic s, cout);
    assign s = a ^ b ^ cin;
    assign cout = (cin & a) | (cin & b) | (a & b);
endmodule
```

Lecture 8, Sep 26, 2022

Hierarchical Verilog

- Top-level module that is composed of simpler modules
- Makes code easier to read and reproduce
- Example: using full adder module to build adder for 2 3-bit inputs

```
module adder3(input logic [2:0] A, B,
              input logic cin,
              output logic [2:0] S,
              output logic cout);
    logic c1, c2;
    // Add first bits with carry in
    fa u0(A[0], B[0], cin, S[0], c1);
    // Add second bit and third bit
    fa u1(A[1], B[1], c1, S[1], c2);
    fa u2(A[2], B[2], c2, S[2], cout);
endmodule
```

- 3 instances of the full adder are needed, each needs a unique name

Example

- Example 2: a circuit that displays a sum R on a 7-segment display where R is either $a + b$ or $c + d$
  - Truth table for a 7-segment decoder (note a segment lights up when it is 0):

x_1 x_0	h_0 h_1 h_2 h_3 h_4 h_5 h_6 h_7
00	0000001
01	1001111
10	0010010
11	0000110

- Logic expressions for each column:
  - $h_0 = \bar{x}_1 x_0$
  - $h_1 = 0$
  - $h_2 = x_1 \bar{x}_0$
  - $h_3 = \bar{x}_1 x_0$
  - $h_4 = x_0$
  - $h_5 = x_1 + x_0$
  - $h_6 = \bar{x}_1$
- Verilog:

```
module seg7(input logic [1:0] x,
            output logic [6:0] h);
    assign h[0] = ~x[1] & x[0];
    // 1 represents number of bits, b indicates binary
    // d for decimal, h for hex
    assign h[1] = 1'b0;
    assign h[2] = x[1] & ~x[0];
    assign h[3] = ~x[1] & x[0];
    assign h[4] = x[0];
    assign h[5] = x[1] | x[0];
    assign h[6] = ~x[1];
endmodule
```

- Need a 2-bit 2-to-1 mux switching between a, b or c, d, with the output fed to a half adder, and then to a 7-segment decoder

```
module hier(input logic [4:0] SW,
            output logic [6:0] HEX0);
    logic [1:0] F, R;
    mux2to1_2bit(SW[1:0], SW[3:2], SW[4], F);
    ha u2(F[1], F[0], R);
    seg7 u3(R, HEX0);
endmodule
```

Lecture 9, Sep 27, 2022

Additional Verilog Statements

- `always` block
  - Statements in an always block execute sequentially
  - In an always block, we don't use `assign`
  - `=` (as opposed to `assign`) is a blocking assignment; must be used inside `always` blocks and enforces sequential execution order
- Conditionals such as if/else/else if must exist in an `always` block
  - *if* without *else* generates a latch

```
module mux(input logic x1, x2, s,
           output logic f);
    always_comb // comb for combinational logic
    begin // Enclose multiple statements, akin to {}
        if (s == 0)
            f = x1; // assign is not used
        else
            f = x2;
    end
endmodule
```

- `case` statements
  - Can be used to do pattern matching
  - Instead of deriving a logic expression, we can let Verilog do it for us
  - Also needs to be inside an `always` block
  - `default` catches unspecified cases; without this the compiler will generate latches (more on this later)

```
module seg7(input logic [3:0] sw,
            output logic [6:0] h);
```

```
    always_comb
    begin
        case (sw)
            0: HEX0 = 7'b1000000;
            1: HEX0 = 7'b1111001;
            // ...
            9: HEX0 = 7'b0000100;
            // Catch cases that have not been specified, since sw can go up to 15
            default: HEX0 = 7'b1111111;
        endcase
    end
endmodule
```

Karnaugh Maps (K-Maps)

- A method of optimizing logic expressions
- The point of logic simplification is to reduce the cost (area) of a circuit; for our purposes, our metric for cost is the number of gates and inputs
  - cost = # of gates + # of inputs
- Optimization using boolean algebra is awkward and error prone
  - When optimizing using boolean algebra, we need to combine terms, but seeing that those combinations are possible is challenging
- Karnaugh Maps are a type of truth table in which minterms that cam be combined are adjacent
- Example: 2-variable K-Map

x_2 \ x_1	0	1
0	m_0	m_2
1	m_1	m_3

- Looking at the first column, $f = m_0 + m_1 = \bar{x}_1\bar{x}_2 + \bar{x}_1x_2 = \bar{x}_1$
  - The second row: $f_2 = m_1 + m_2 = \bar{x}_1x_2 + x_1x_2 = x_2$
- Example: $f(x_1, x_2) = \sum m(0, 1, 3)$
  - $f = \bar{x}_1\bar{x}_2 + \bar{x}_1x_2 + x_1\bar{x}_2$
    - As it is the circuit has a cost of 17 (3 AND, 1 3-input OR, 2 NOT + 2 inputs per AND, 3 inputs per OR, 1 input per NOT)
  - K-Map:

x_2 \ x_1	0	1
0	1	0
1	1	1

- This lets us simplify our circuit to $f = \bar{x}_1 + x_2$ which only has a cost of 5
- To simplify using a K-Map, we group adjacent minterms in the map
  - The second row shows that regardless of x_1, as long as x_2 is 1, the expression is 1, so that row simplifies to x_2
  - The first column shows that regardless of x_2, as long as x_1 is 0, the expression is 1, so the row simplifies to $\bar{x}_1$
- We can only group terms in group sizes of powers of 2 (for a 2×2 K-Map, we can group 2 terms or 4 terms)

Lecture 10, Sep 29, 2022

3-Variable K-Maps

x_3 \ x_1x_2	00	01	11	10
0	m_0	m_2	m_6	m_4
1	m_1	m_3	m_7	m_5

- Group sizes are 1, 2, 4, 8
- Note the inputs are not enumerated in ascending order
  - This is a *grey code*, a sequence of bits where when transitioning between consecutive terms, only 1 bit changes
  - This ensures that adjacent entries in the table only differ by 1 input bit
- The way the table is arranged shouldn't matter for the final result, as long as minterms are mapped properly
- Example: $f = \sum m(3,7) = \bar{x}_1x_2x_3 + x_1x_2x_3 = x_2x_3(\bar{x}_1 + x_1) = x_2x_3$
  - In the K-Map these terms are adjacent, in the two where x_3 and x_2 are 1, and the value of x_1 does not matter, so this gives us x_2x_3 both non-inverted and no x_1
- Example: minterms m_2, m_6, m_3, m_7 simplifies to x_2
  - Notice that a product term that covers more adjacent cells is cheaper!
- The K-Map also wraps around, e.g. we can combine m_0, m_4 to $\bar{x}_2\bar{x}_3$

Terminology

- Implicant: for a function f, an *implicant* is any product term covered/included by f
  - Can be a simplified or unsimplified term
- Prime Implicant: an implicant for which it is impossible to remove any literal and still have a valid implicant
- Cover: any set of implicants that includes all minterms of a function (every 1 in a K-Map needs to be covered)
- Example: $f(x_1, x_2, x_3) = \sum m(1,4,5,6)$
  - Prime implicants are $x_1\bar{x}_3, x_1\bar{x}_2, \bar{x}_2x_3$
  - Minimal cost cover is $x_1\bar{x}_3 + \bar{x}_2x_3$ (notice $x_1\bar{x}_2$ is not included)

x_3 \ x_1x_2	00	01	11	10
0	0	0	1	1
1	1	0	0	1

- An essential prime implicant is a prime implicant that covers at least one minterm that is not covered by any other prime implicant
  - In the last example $x_1\bar{x}_2$ is not an essential prime implicant
- A minimal cost cover includes all essential prime implicants

4-Variable K-Maps

x_3x_4 \ x_1x_2	00	01	11	10
00	m_0	m_4	m_{12}	m_8

x_3x_4 \ x_1x_2	00	01	11	10
01	m_1	m_5	m_{13}	m_9
11	m_3	m_7	m_{15}	m_{11}
10	m_2	m_6	m_{14}	m_{10}

- Group sizes are 1, 2, 4, 8, 16
- Example: $f(x_1, x_2, x_3, x_4) = \sum m(2, 4, 5, 8, 10, 11, 12, 13, 15)$

x_3x_4 \ x_1x_2	00	01	11	10
00	0	1	1	1
01	0	1	1	0
11	0	0	1	1
10	1	0	0	1

- Prime implicants: $x_2\bar{x}_3, x_1\bar{x}_3\bar{x}_4, x_1x_2x_4, x_1x_3x_4, x_1\bar{x}_2x_3, \bar{x}_2x_3\bar{x}_4, x_1\bar{x}_2\bar{x}_4$
  - Note minterms 12 and 13 don't form a PI since it's completely inside the PI for minterms 4, 5, 12, 13
- Essential PIs: $x_2\bar{x}_3, \bar{x}_2x_3\bar{x}_4$
- Minimal cost cover: $f = x_2\bar{x}_3 + \bar{x}_2x_3\bar{x}_4 + x_2x_3x_4 + \begin{cases} x_1\bar{x}_3\bar{x}_4 \\ x_2\bar{x}_3\bar{x}_4 \end{cases}$
  - This shows that there can be multiple minimal cost covers

Lecture 11, Oct 3, 2022

Procedure for a Minimum Cost Cover

1. Find prime implicants
2. Identify prime implicants and include in the cover
3. Choose other PIs as needed until we cover all the 1s:
   - Do this using the largest power of two size grouping of only 1s
   - Use fewest circles to cover all the 1s
   - 1s can be circled multiple times if this allows fewer/larger circles to be used
   - Remember circles can wrap around the edges!
   - Note instead of circling 1s (minterms) for a SOP expression, we can also circle 0s (maxterms) for a POS expression

Don't Cares

- Sometimes we know specific inputs won't occur, or we don't care about what happens on an input combination
  - e.g. for a 7-segment display decoder if we only want to go from 0 to 9, we don't care when input is 1010 or higher
- Each don't care (d) term can independently be 0 or 1
- In a K-Map we put a d; we can either include it or exclude it, depending on which lets us use fewer/larger circles

Sequential Circuits

- As opposed to combinational circuits (outputs only determined by present inputs), sequential circuits' outputs depend on previous inputs/states as well

- The simplest way to do storage is with 2 inverters in sequence, with the final output feeding back into the input
  - Once the input is high, the 2 inverters store that high input and feed it back through, so that the output stays high
  - However this is missing a way to reset the storage state

RS Latch

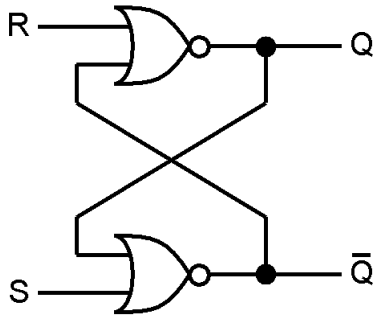


Figure 2: NOR gate RS latch

- Another way is to use 2 NOR gates
  - (Assume $S = 0$ at the start)
  - Reset Q to 0 by setting $R = 1$
    - When $R = 1$ it doesn't matter what the lower input to the NOR gate is, the output will always be a 0
    - Setting $R = 0$ again changes nothing, since the bottom NOR gate's output 1 drives the top NOR gate to output 0
  - Set S to 1, then the bottom NOR gate will always output 0, which makes Q a 1 (assuming $R = 0$)
  - Set S to 0 doesn't change Q, since the previous Q of 1 is still driving the output of the bottom NOR gate to 0
- R stands for reset, S stands for set
- This is built with what's known as *cross-coupled NOR gates*

Lecture 12, Oct 4, 2022

More on RS Latches

- Flaw with this design: it cannot support both set and reset going to high/low at the same time (this can cause the circuit to oscillate)

Characteristic Tables

- For sequential circuits, instead of truth tables we use characteristic tables

S	R	Q	$\bar{Q}$	Comment
0	0	0/1	1/0	(stored value)
0	1	0	1	
1	0	1	0	
1	1	0	0	(not used in practice)

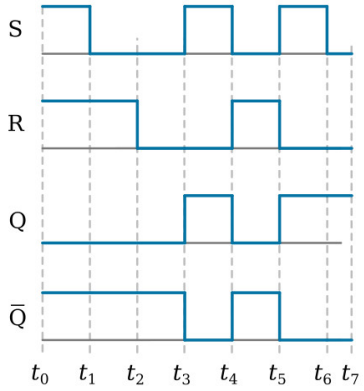
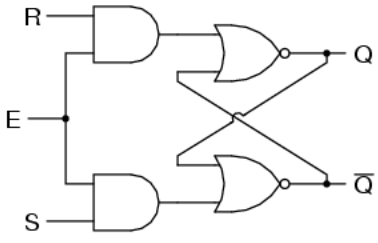


Figure 3: RS latch timing diagram



E	S	R	Q	$\overline{Q}$
0	**0**	**0**	latch	latch
0	**0**	1	latch	latch
0	1	**0**	latch	latch
0	1	1	latch	latch
1	**0**	**0**	latch	latch
1	**0**	1	**0**	1
1	1	**0**	1	**0**
1	1	1	**0**	**0**

Figure 4: NOR Gated RS latch schematic

Gated RS Latch

E	S	R	$Q(t+1)$
0	x	x	$Q(t)$
1	0	0	$Q(t)$
1	0	1	0
1	1	0	1
1	1	1	Not Useful

- When the clock is 0, $S' = R' = 0$ (none of the inputs pass through) so the stored value can't be changed
  - In a digital circuit the clock is typically a square wave generated by a crystal oscillator
- (Gated) RS latches can also be built out of NAND gates
  - We can obtain this by using DeMorgan's rule to convert an OR gate with both inputs inverted to a NAND gate

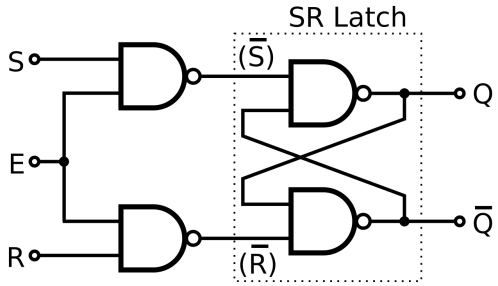
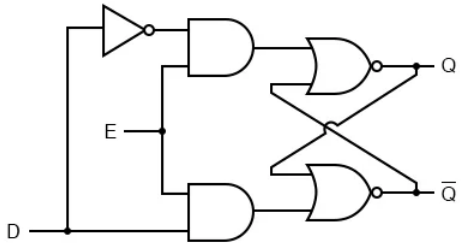


Figure 5: NAND Gated RS latch

Gated D Latch



E	D	Q	$\overline{Q}$
0	0	latch	latch
0	1	latch	latch
1	0	0	1
1	1	1	0

Figure 6: Gated D latch

- However S and R can still be 1; we don't want this, because this is not useful, and when they both drop to 0 from 1 the circuit oscillates
- We can let $S = D, R = \bar{D}$, so that we can't have S and R both be 1 at the same time
- This is represented with the symbol 
- D is the "data" input – when clock is 1, $Q = D$; when clock is 0, Q stores the last value of D
- This is known as a *level-sensitive latch*: the output Q is sensitive to the level of the clock (as opposed to edge sensitive latches that operate at the transition)
- This can also be represented using a mux

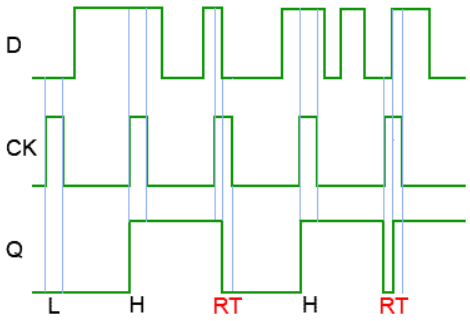


Figure 7: Gated D latch timing diagram

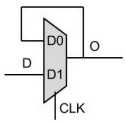


Figure 8: Gated D latch using a mux

A	B	$f(t+1)$
0	x	$f(t)$
1	0	0
1	1	1

Flip-Flops

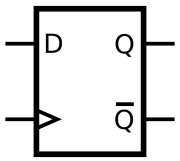


Figure 9: D Flip-Flop

- Another type of storage element
- Consider 2 gated D-latches in series, with Q leading to D of the next latch; the clock to the first latch is inverted and tied together with the second latch's clock
- This is known as a *D Flip-Flop*
- The triangle means that the circuit responds to the *edge* rather than the level of the clock
- Operation: Let D be the input signal to the first latch, Q_m be the output of the first latch (which is fed to the second latch), and Q be the final output of the second latch
  - Case 1: Clock = 0
    - First gated D-latch has clock 1, so middle signal Q_m tracks D, the input signal
    - Final output Q could be either 0 or 1 based on the previous stored value (since its clock is 0)
  - Case 2: Clock goes from 0 to 1 (rising/positive edge)
    - At this moment $Q = Q_m = D$, so the value of D is stored
    - Afterwards the clock changes to 1, so the first latch no longer tracks D, so the value of Q is stored
- The D flip-flop only changes its value on a clock edge

Lecture 13, Oct 6, 2022

More on Flip-Flops

- D-flip flops can be positive edge or negative edge triggered
  - A negative edge triggered D-flip flop is basically a positive one but with clock inverted; it stores a value on a clock negative edge (1 to 0)

```
            Q = D;
endmodule
```

- The `always_latch` makes it a latch
- If a `case` or `if` does not cover all cases, the compiler will infer a latch, i.e. it stores the previous value
  - In this case that's exactly what we want – we have no `else`, so the compiler creates a latch to store the previous value of `D` and assign it to `Q`
- For a flip-flop:

```
module D_FF(input logic D, clk,
            output logic Q);
    always_ff @(posedge clk)
        Q <= D;
endmodule
```

- `posedge` is a keyword that means the flip flop should trigger on the positive edge
  - `negedge` is a negative edge trigger
- `always_ff` crates a flip flop
- When code describes flip flops, the assignments should use `<=`, not `=`
  - `<=` is a nonblocking assignment used for sequential logic
  - `=` is a blocking assignment used for combinational logic
- The stuff inside `always_ff` only executes on a positive edge, so we don't need an `if` (whereas the latch needs to check)
- For an 8-bit register (in the schematic it's denoted by a D-flip flop with an 8-bit bus for D and Q):

```
module reg8(input logic [7:0] D,
            input logic clk,
            output logic [7:0] Q);
    always_ff @(posedge clk)
        Q <= D;
endmodule
```

- Note the only difference is `D` and `Q` are now buses, but inside the module the code is identical since we assign all 8 bits at once

Resets

- How do we reset a flip-flop to a known value?
  - Synchronously (on a clock edge) or asynchronously (independent of clock edge)
- In the synchronous case we can AND together D and a reset signal and then feed into the flip flop
  - This is known as an *active-low* reset because when the reset signal is 0, the value is reset

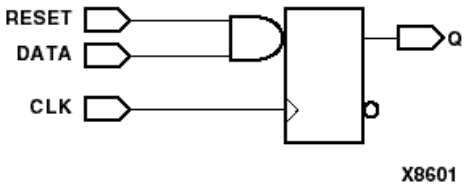


Figure 10: Synchronous reset D-flip flop

```
module D_FF(input logic D, clk, resetn,
            output logic Q);
    always_ff @(posedge clk)
        if (resetn == 0)
            Q <= 1'b0;
```

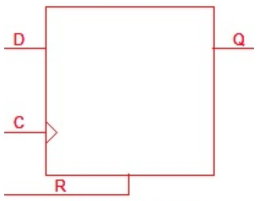


Figure 11: Synchronous reset D-flip flop symbol

```
        else
            Q <= D;
end
```

- In the asynchronous case, the flip-flop can be reset regardless of the clock

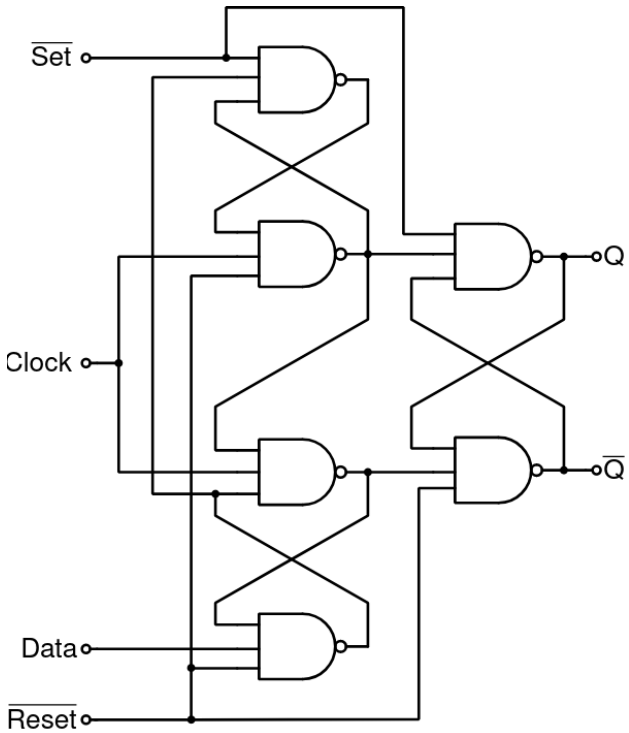


Figure 12: Asynchronous reset D-flip flop

- Regardless of the state of the clock, as long as reset is 0, Q will be reset
- Asynchronous reset is marked by an AR instead of just R

Lecture 14, Oct 11, 2022

D-Flip Flop With Reset in Verilog

```
module D_FF(input logic D, clock, resetn,
            output logic Q);
    always_ff @(posedge clock, negedge resetn)
        if (resetn == 0)
            Q <= 1'b0;
```

```
        else
            Q <= D;
endmodule
```

- `negedge resetn` is added to the sensitivity list because it directly affects `Q`
  - Whenever there's a negative edge on `resetn`, `Q` is reset regardless of clock state

Multi-Bit Register With Reset

```
module reg8(input logic [7:0] D,
            input logic resetn, clock,
            output logic [7:0] Q);
    // Note this is a synchronous reset
    always_ff @(posedge clock)
        if (!resetn)
            Q <= 8'b0;
        else
            Q <= D;
endmodule
```

- We can also add an enable to determine whether data is loaded on a clock edge
  - This can be done via a mux that passes a new D when enable is true and otherwise cycles Q back into D
  - We don't just add an AND gate onto the clock because generally in a modern high speed circuit we don't want to add additional gates onto the clock, since that introduces delays that cause timing issues

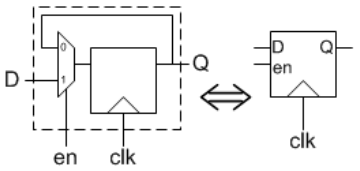


Figure 13: D-flip flop with enable

Counters

Cycle	$Q_2Q_1Q_0$
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111
8	000

- Circuits that count up or down every clock cycle
- Can we find a pattern in the bits?
  - Q_0, the LSB, toggles every clock cycle, so $Q_0(t+1) = Q_0 \oplus 1$
  - Q_1 toggles only in the cycle after Q_0 is 1, so $Q_1(t+1) = Q_1 \oplus Q_0$
  - Q_2 toggles only when both Q_0 and Q_1 are 1, so $Q_2(t+1) = Q_2 \oplus (Q_0Q_1)$

- A T-flip flop toggles its value on each clock cycle if T is 1; it's made by cycling the output of a D-flip flop back into the input with an XOR with T

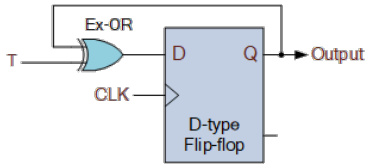


Figure 14: T-flip flop implemented with a D-flip flop

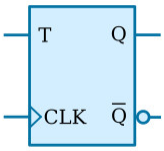


Figure 15: T-flip flop symbol

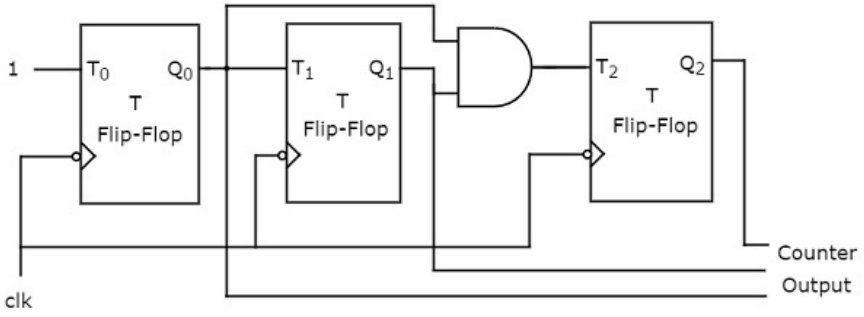


Figure 16: 3-bit synchronous up-counter implemented with T-flip flops

- Note we should add a reset for all the flip-flops because when the circuit powers on, there is no guarantee that they will initialize to 0
- This called a synchronous counter because all the clocks are the synchronized together
- Instead of a fixed 1 being fed into the first flip flop we can also use an enable input, so when enable is 0 the counter won't count up
- Instead of resetting all the flip flops we might also want to load a preset value into all of them; this is called a parallel load

```
module upcount(input logic [3:0] R,
               input logic resetn, clock, E, L,
               output logic [3:0] Q);
    always_ff @(posedge clock, negedge resetn)
    begin
        if (!resetn)
            Q <= 4'b0;
        else if (L)
            Q <= R;
        else if (E)
            Q <= Q + 1;
```

```
    end
endmodule
```

Lecture 15, Oct 13, 2022

Clock Dividers (Prescalers)

- How do we generate a different clock frequency from what's given?

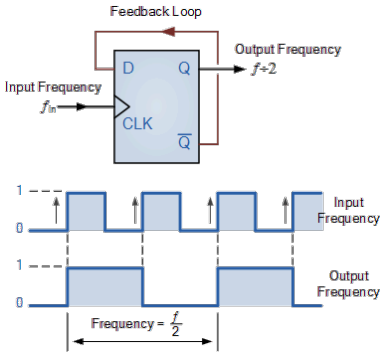


Figure 17: Clock division by 2

- Example: Given a 100MHz clock, derive a circuit using D-flip flops to generate 50MHz and 25MHz clocks
  - The 50MHz clock can be made by tying the input clock into the clock of a T-flip flop
  - The 25MHz clock can be made by putting the 50MHz clock into the clock of a T-flip flop
  - This method is restricted to clock divisions by powers of two
- More generally we can use a counter in order to divide by any factor
  - A NOR gate over all the counter bits can be used as a comparator to check for 0
  - The comparator output can be ANDed together with some signal so the signal only passes through when the counter reaches the desired value

Finite State Machines (FSM)

- Any sequential circuit can be modelled by some set of inputs w_i →combinational circuit A →set of flip flops →combinational circuit B →output
- A finite state machine is so named because a circuit with k registers (flip flops) can only be in one of a finite number of states (2^k states)

Example FSM

- Motor outputs its status on w
- If status is ok, the FSM should maintain $z = 0$
- If motor outputs a sequence of $1, 0, 1$ then an error is occurred, so the FSM should output $z = 1$
- z is determined by the history of w
- First we need a state diagram:

- From this state diagram we can make a state table:

State	Next $w = 0$	Next $w = 1$	z
A	A	B	0
B	C	B	0
C	A	D	0

State	Next $w = 0$	Next $w = 1$	z
D	C	B	1

- Now we need to assign states
  - There are 4 states, so we need 2 flip-flops y_2, y_1
  - Choose state codes: $A = 00, B = 01, C = 10, D = 11$
- Now make the state-assigned table:

y_2y_1	Y_2Y_1 for $w = 0$	Y_2Y_1 for $w = 1$	z
00	00	01	0
01	10	01	0
10	00	11	0
11	10	01	1

- Now we basically have a truth table, we can synthesize the circuit:
  - $Y_1 = w$
  - $Y_2 = \bar{w}y_1 + wy_2\bar{y}_1$ (can be found through a 3-variable K-map with w, y_1, y_2)
  - $z = y_2y_1$

Lecture 16, Oct 17, 2022

Alternative Design for Example FSM

- State machine designs are not unique
- From this state diagram we can make a state table:

State	Next $w = 0$	Next $w = 1$	z
A	A	B	0
B	C	B	0
C	A	D	0
D	C	B	1

- Instead of using 2 flip flops we can use 4 flip flops $y_4y_3y_2y_1$
  - Assign state codes as $A = 0001, B = 0010, C = 0100, D = 1000$
  - This is a *one-hot encoding*, where for each state there is only 1 bit that's 1
- State-assigned table:

$y_4y_4y_2y_1$	$Y_4Y_3Y_2Y_1$ for $w = 0$	$Y_4Y_3Y_2Y_1$ for $w = 1$	z
0001	0001	0010	0
0010	0100	0010	0
0100	0001	1000	0
1000	0100	0010	1

- We can synthesize the circuit simply by inspecting the state diagram
  - Y_4 is 1 in state D; transition to state D is from state C with $w = 1$, so $Y_4 = y_3w$
  - Similar for Y_3: transitions are from state D with $w = 0$ and from state B with $w = 0$ so $Y_3 = (y_4 + y_2)\bar{w}$
  - For Y_2: transitions are from A with $w = 1$, B with $w = 1$, D with $w = 1$ so $Y_2 = (y_1 + y_2 + y_4)w$
  - For Y_1: transitions are from A with $w = 0$, C with $w = 0$ so $Y_1 = (y_1 + y_3)\bar{w}$

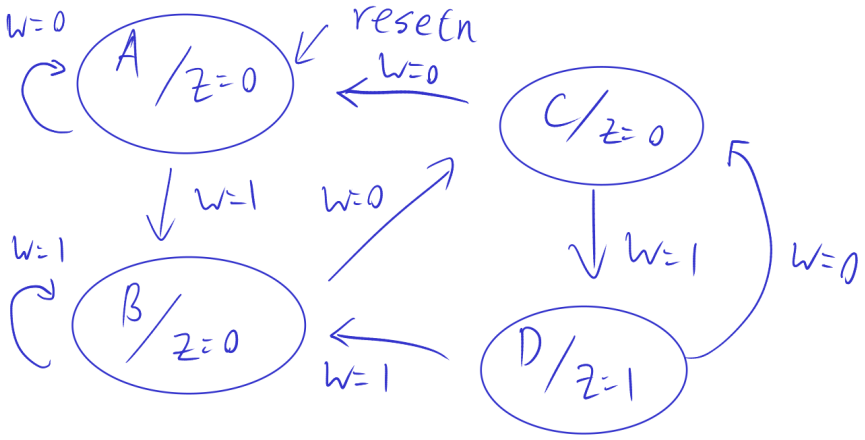


Figure 18: State diagram for the example problem

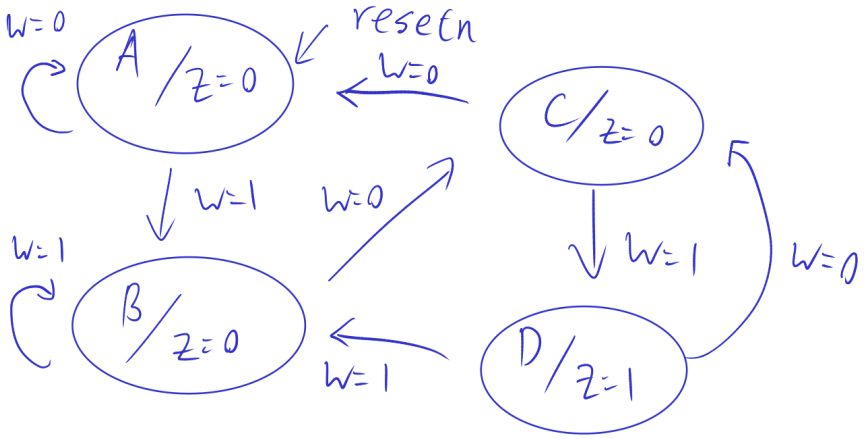


Figure 19: State diagram for the example problem

  - Finally $Z = y_4$
- This different circuit will produce the same behaviour as the one from the previous lecture
- We can do another alternative design in which each state represents the last 3 values of w
  - This simply produces a 3-bit shift register with a comparator
  - A shift register consists of flip-flops chained together

Verilog Code for Example FSM

- All we need to do in Verilog is to specify the state table (using a `case` statement)
- Separate the code into 3 sections:
  1. Flip-flops (reset + update to new value)
  2. State table
  3. Output

```
module fsm(input logic w, clock, resetn,
           output logic z);
    // Define states
    typedef enum logic [1:0] (A, B, C, D) statetype;
    statetype ps, ns;
    // 1. Flip flops
    always_ff @(posedge clock, negedge resetn)
        if (!resetn)
            // Make present state A on reset
            ps <= A;
        else
            // Otherwise, make the present state take the next state's value
            ps <= ns;
    // 2. State table
    always_comb
        // Determine next state from the present state
        case (ps)
            A: ns = w ? B : A;
            B: ns = w ? B : C;
            C: ns = w ? D : A;
            D: ns = w ? B : C;
        endcase
    // 3. Assign output
    assign z = (ps == D);
endmodule
```

- `enum` defines an *enumeration*, which makes the compiler encode the state for us
  - The type after `enum` is the underlying type, in this case a 2-bit logic value
  - The symbols in brackets after are the possible values for the enum; in this case we can refer to the states now as A, B, C, D
  - By default, these are in numerical order, but the compiler sees this only as a suggestion (if it can optimize by changing the encodings, then it will do so)
- `typedef <type> <name>` defines a type alias, in this case `statetype` for the enum type

Lecture 17, Oct 18, 2022

Verilog For Shift Register State Machine

```
module fsm_shift(input logic lo, clock, resetn,
                 output logic z);
    // State of the 3 flip flops
```

```
    logic [3:1] y;
    always_ff @(posedge clk, negedge resetn)
        if (!resetn)
            y <= 3'b000;
        else
        begin
            y[3] = w;
            y[2] = y[3];
            y[1] = y[2];

            // Equivalently:
            // y <= y >> 1;
            // y[3] <= w;
        end
    assign z = y[3] & ~y[2] & y[1];
endmodule
```

- Note again `<=` is a nonblocking assignment, which all happen at the same time
  - This important because otherwise when `y[2] = y[3];` happens, `y[3]` will already have been updated!

Example: Traffic Light Controller

- On reset, Light A is green, Light B is red
- Every clock cycle, traffic sensors T_A, T_B make a decision:
  - If traffic on T_A, light A stays green, otherwise light A goes to yellow and then red; light B stays red and then transitions to green
  - Same scenario for T_B and light B
- State diagram:

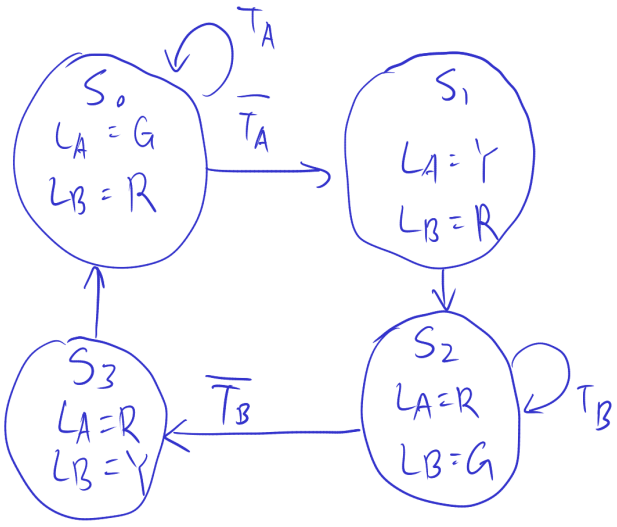


Figure 20: State diagram for traffic light controller

- State table:

Present	00	01	10	11	L_A	L_B
S_0	S_1	S_1	S_0	S_0	G	R
S_1	S_2	S_2	S_2	S_2	Y	R

Present	00	01	10	11	L_A	L_B
S_2	S_3	S_2	S_3	S_2	R	G
S_3	S_0	S_0	S_0	S_0	G	R

- State assignment:
  - $S_0 = 00, S_1 = 01, S_2 = 10, S_3 = 11$
  - Output encoding: $G = 00, Y = 01, R = 10$
- State assigned table:

y_2y_1	00	01	10	11	L_A	L_B
00	01	01	00	00	G	R
01	10	10	10	10	Y	R
10	11	10	11	10	R	G
11	00	00	00	00	G	R

- Derive logic expressions:
  - Use a 4-variable K-map
  - $Y_2 = \bar{y}_2 y_1 + y_2 \bar{y}_1 = y_2 \oplus y_1$
  - $Y_1 = \bar{y}_2 \bar{y}_1 \bar{T}_A + y_2 \bar{y}_1 \bar{T}_B$
  - Output: $L_{A,1} = y_2, L_{A,0} = \bar{y}_2 y_1, L_{B,1} = \bar{y}_2, L_{B,0} = \bar{y}_2 \bar{y}_1$

Key Take-Aways

- FSM consists of 2 blocks of combinational logic: calculating the next state from the current state, and calculating the output from the current state
- On each clock edge the FSM advances to the next state, computed based only on the input and present state
- There are multiple ways to specify/assign states; some are more efficient than others
  - Minimal encoding: Minimum number of flip-flops
  - One-hot encoding: One flip-flop for each possible state
- In Verilog, code is divided into 3 sections: the flip flops, the state table, and the output assignment

Lecture 18, Oct 20, 2022

Two's Complement System

- Two's complement is a way to represent negative numbers in binary
- Example: if we have a 4 bit number, if we add 1111_2 to it, this turns out to reduce its value by 1, so in 4 bits, 1111_2 represents -1
- Given a number k represented with n bits, $-k$ is represented as $2^n - k$
  - We can check for correctness by verifying that $k + -k = 1$ (after truncating the carry out)
  - Example:
    - $n = 4, k = 1 \implies k = 0001_2, -k = 1111_2$
    - $n = 4, k = 6 \implies k = 0110_2, -k = 1010_2$
      - Check: $0110_2 + 1010_2 = 10000_2 = 0$ when carry out is dropped
    - $n = 5, k = 13 \implies k = 01101_2, -k = 10011_2$
      - Check: $01101_2 + 10011_2 = 100000_2 = 0$
- Shortcut: Invert all the bits and add 1
  - This is because inverting all the bits is equivalent to calculating $(2^n - 1) - k$, so adding 1 to it is equal to $2^n - k$
  - We can simply do this again to get from $-k$ back to k!

- With two's complement, the maximum value representable with n bits is reduced by half to make room for the negative numbers
  - e.g. for 4 bits we used to be able to represent up to $1111_2 = 15$, with two's complement we can only go up to $0111_2 = 7$, since 1111_2 will now represent -1
    * However now we can represent down to $1000_2 = -8$
- The MSB is 0 for positive numbers and 1 for negative numbers; we can use it as a sign indicator (the sign bit)
- Sign extension: if we want to represent the same number with more bits, we simply duplicate the sign bit into the new MSB bit positions
  - Example:
    * -1 represented with 4 bits is 1111_2, if we want to expand it to 8 bits, we duplicate the sign bit as 11111111_2
    * 1 represented with 4 bits is 0001_2, extend to 8 bits is 00000001_2
- Two's complement allows us to do subtraction with the same hardware as addition – just add the negative!
  - XOR all the input bits by 1 to invert, and then utilize the carry-in to the full adder to add 1 in order to convert the operand to its negative

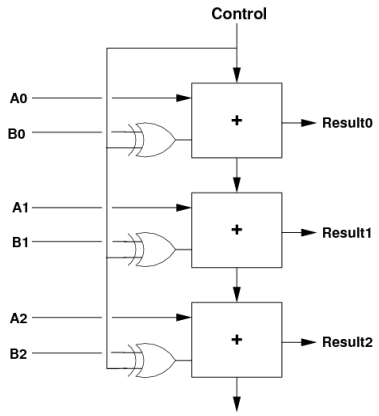


Figure 21: Add/subtract ALU

Overflow

- If we add two numbers that are too large, we get an arithmetic overflow
  - e.g. $0111_2 + 0001_2$ is $7 + 1$, but the sum is 1000_2, which in 4-bit two's complement is -8; this is because 8 can't be represented in a signed 4-bit number
- When our result overflows the range of the output
- How do we detect overflow?
  - Positive number plus positive number should have positive result
    * Positive plus positive always has a carry out of 0
    * When positive plus positive overflows, MSB (sign bit) would be 1 (since the result will have the wrong sign)
  - Negative number plus negative number should have negative result
    * Negative plus negative always has a carry out of 1
    * When negative plus negative overflows, sign bit would be 0
  - Positive number plus negative number will never overflow
- To detect overflow, consider the MSB (sign bit) of the result and the carry-out; overflow occurs when these have different values
  - $o = s_3 \oplus c_{out}$

Lecture 19, Oct 24, 2022

Processors

- Processes are logic circuits at their core, consisting of both combinational and sequential logic, controlled by an FSM that dictates what operation it does
- Includes a set of n-bit general purpose registers to hold values to use
  - RISC-V has 32 registers, each 32 bits (`x0` to `x31`)
  - Registers are very fast to access, but they can only hold a relatively small amount of information
- Includes an ALU
  - Can add, subtract, multiply, AND, OR, XOR, shift, rotate, ...
- Has an external interface (peripherals)
  - Memory to hold data that the registers can't store (DRAM)
  - I/O devices, e.g. keyboard, network, display, etc
  - Address: Who to communicate with
  - Data: What to communicate
  - Control: Whether we're reading or writing

Simplified Processor Example

- 8 bit bus
- 4 8-bit registers, with enables $R_{i,in}$ and inputs connected to the bus, and the outputs connected to tri-state buffers to the bus with control $R_{i,out}$
  - A tri-state buffer is used to disconnect an input from its output
  - If we want to load a value into a register, we write the value to the bus, and enable the register we want
  - If we want to use the value from a register, we enable the correct tri-state buffer
  - These hold the values we want to use in a computation or the result of a computation

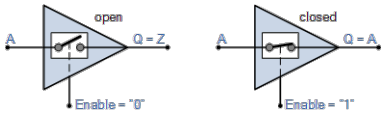


Figure 22: Tri-state buffer

- ALU with addition, subtraction operations
  - Since we can only read one 8-bit value from a register per clock cycle, we need a register to hold the inputs to the ALU
  - 8-bit A register with enable A_{in}, input from bus and output to the ALU input; input B comes directly from the bus
  - Output is written to another register G with enable G_{in}, output connected via a tri-state buffer back to the bus with control G_{out}
- An external interface, a tri-state buffer connected to the main bus
- A control FSM, which produces the right values for all the enables – $R_{i,in}, R_{i,out}, A_{in}, \cdots$
  - Inputs w, function, where w tells it to start an operation, and function is the operation it should do
  - An output done is asserted when operation is complete

Lecture 20, Oct 25, 2022

Simple Processor Continued

- Instructions:
  - `load Rx, Data`

        * Load `Data` into `Rx`, where `Data` comes from the external signal
        * Need to enable the external signal input tri-state buffer, and enable a write to the correct register
    - `move Rx, Ry`
        * Copies data from `Ry` into `Rx`
        * Need to enable the tri-state buffer on the output of `Ry`, and enable a write to `Rx`
    - `add Rx, Ry`
        * Store `Rx + Ry` into `Rx`
        * First enable the tri-state on output `Rx`, and enable the temporary `A` register for the ALU
        * On the second clock cycle, enable the tri-state on output `Ry`, the ALU does the computation and stores into `G`
        * On the third clock cycle, store `G` into `Rx`
    - `sub Rx, Ry`
        * Store `Rx - Ry` into `Ry`
        * Same thing as the add instruction but subtraction
- Tradeoff between instruction usability and complexity
    - e.g. old processors used to have very complex instructions that compilers could not always take advantage of
- Each instruction and register has an encoding
    - 00 for load, 01 for move, 10 for add, 11 for sub
    - These are referred to as "opcodes", in this case 2-bit opcodes
    - Registers are encoded as the register number (register index)
    - e.g. `add R1, R2` is encoded as `10, 01, 10`
- When designing the processor we need to know how many steps (clock cycles) can instructions take
    - The longest instruction is add/sub at 3 steps
    - Therefore we need a 2-bit counter to count which step we're currently on
        * Counter with a clock, clear, and produces Q_1, Q_0
- Now turn those 2 bits into a 1-hot code
    - 1-hot codes make the control FSM logic much easier to derive
- For the control FSM:
    - Need a function register, taking in f_1, f_0, the opcode, 2-bit values for R_x and R_y, an input FR_{in} which is used to indicate when we're loading a new instruction
    - First decoder decodes the opcode into a 1-hot code for each instruction
    - Second and third decoders decode the register inputs into 1-hot codes
- We then derive control signals for each step
    - $A_{in} = (I_2 + I_3)T_1$
    - $G_{in} = (I_2 + I_3)T_2$
    - $G_{out} = (I_2 + I_3)T_3$
    - extern $= I_0T_1$
    - Done $= (I_0 + I_1)T_1 + (I_2 + I_3)T_3$
    - $\text{FR}_{in} = wT_0$
    - Clear $=$ Done $+ \bar{w}T_0$
        * $\bar{w}T_0$ means if we're in state 0 and we aren't starting an operation, we stay in state 0

Lecture 21, Oct 27, 2022

Introduction to Assembly Language

- A human-readable form of the processor's native language
- Many different flavours, e.g. x86, ARM, RISC-V, etc
- Assembly is translated into binary machine code by an assembler
    - High-level languages are translated into assembly and then compiled into machine code
    - While high level languages such as C don't care about the underlying processor, assembly is targeted for a specific architecture

- Each instruction specifies two things: the operation and operands
  - Operands can come from registers, memory, or constants in the instruction itself
- Assembly instructions are encoded as a word and stored in memory
  - In RISC-V these are 32-bit words

Introduction to Computer Organization

- Processor communicates to memory via an address bus and data bus (bidirectional or two unidirectional buses)
- Control signals such as read and write are on another bus
- I/O devices are connected to the same data bus, control signal bus, etc
- The memory and I/O ports are each assigned a range of addresses called *memory maps*
  - This way we can identify whether a read/write is to memory or I/O or something else
  - Referred to as *memory-mapped I/O*
  - e.g. Memory can be mapped to addresses `0x0 - 0x3FFF'FFFF`, LED can be mapped to addresses `0xFF20'0000 - 0xFF20000F`
    * In this case an address of e.g. `0x10000000` is in memory

Memory Architecture

- Registers are small, so memory is used to store large amounts of data
- Memory can be thought of as a 2D array that you can index into
  - e.g. at address 0 is word 0, at address 4 is word 1, etc
    * This is because words are 4 bytes but memory is byte-addressable
- With a k bit address (k address lines or wires), we can address $A = 2^k$ bytes or 2^{k-2} words
  - The first $k-2$ bits select the "row", or the word, and the last 2 bits select the "column", or the byte within the word

Notes on Lab 6

- Most non-trivial circuits are separated into 2 functions
  - The datapath (where the data moves), with e.g. ALUs, registers, etc
  - The control path (manipulates the signals in the datapath), with e.g. mux select signals, register enables, etc
- Given a datapath that computes $A^2 + B$, compute $Ax^2 + Bx + C$
  - Registers holding values for A and B; enables on the datapath
    * Inputs are muxed, with both register inputs coming either from the data input or from the ALU output
  - ALU
    * Inputs are muxed, allowing either A or B to go to both inputs
    * 2 operations: 0 adds, 1 multiplies
  - Result register for the ALU
  - To do the operation, we need to first compute A^2, store it somewhere, and then add B to it

Lecture 22, Oct 31, 2022

Introduction to RISC-V

- One of the many instruction set architectures (ISAs)
- A newer, open source instruction set
  - Designed recently so it's less bloated and cleaner
- Different ISAs have different instructions, but some primitives are common across all of them
- The instruction set doesn't define the underlying hardware – it exists as an interface between hardware and software, but the hardware can be implemented in many different ways

- RISC-V comes in different flavours
  - We will be using RISC-V 32I (32-bit integer)
- Instructions define operation and operands
  - Operands can be registers, memory, constants, etc
  - RISC-V has 32 registers, each 32 bits

RISC-V Instructions

- Arithmetic instructions
  - e.g. an add operation:
    - In C: `a = b + c`
    - In assembly: `add s0, s1, s2`
      - `s0` holds `a`, `s1` holds `b`, `s2` holds c
      - `s1, s2` are source operands, `s0` is the destination operand
  - A subtraction would be `sub s0, s1, s2`
  - e.g. `a = b + c - d` is `add t0, s1, s2` and then `sub s0, t0, s3`
- Design principle: make the common case faster
  - Use multiple simple instructions rather than one complex instruction, since simpler instructions are faster in hardware
- Registers
  - Internal to a processor; much faster to access than main memory, but there is a limited number
  - In RISC-V the register set is `x0` to `x31`, but there are special names:
    - `zero` always holds the constant value 0
    - `s0` to `s11`, `t0` to `t6` are the "general purpose" registers, generally used to store variables
    - `ra`, `a0` to `a7` are used for function calls
    - `sp`, `gp`, `fp` are the stack pointer, global pointer, and frame pointer (more on this later)
- Constants ("immediate values")
  - These values are immediately available as part of the instruction (no fetching from memory necessary)
  - Use `addi` instruction: `addi s0, s0, 4` performs `a = a + 4`
    - Note there is no `subi` instruction, but we can use `addi` with a negative number
  - We can also initialize values using immediates, by using an `addi` with the zero register
    - e.g. `addi s4, zero, -78` initializes `s4` to -78
  - Use `0x` prefix for a hexadecimal number, `0b` for a binary number
  - Immediates can only be up to 12 bit two's complement numbers since we need to use the other 20 bits for the instruction
    - The numbers are sign-extended to 32 bits
  - If the numbers are bigger than 12 bits:
    - Use `lui`, load upper immediate, followed by an `addi`
      - `lui` allows specification of a 20-bit value, which is loaded into the most significant 20 bits of the instruction and sets the rest to 0
      - The `addi` can add in the other 12 bits
      - e.g. if we want `a = 0xABCDE123` we can do `lui s2, 0xABCDE` followed by `addi s2, s2, 0x123`
    - Alternatively we can use a pseudo-instruction `li`, load 32-bit immediate, and just do `li s2, 0xABCDE123`
      - The assembler converts the `li` into `lui` and `addi`
      - Pseudo-instructions make our lives easier; they are not real instructions but are converted into real instructions by the assembler

Lecture 23, Nov 1, 2022

RISC-V Instructions Continued

- Arithmetic operations can only access registers and immediates, not main memory
  - We need a memory instruction to first retrieve a value from memory before it can be used, and then write back a value if needed
  - This is known as a *load-store* architecture – we can only access memory via loads and stores
- Memory operations
  - Load word instruction reads a data word from memory into a register (reads 4 bytes at once)
    - e.g. `lw s0, 8(zero)` performs `a = mem[2];`
      - `8` is the *offset address*, `(zero)` is the *base address*
      - We are using the zero register to start at address 0 and offset by 8, so we're accessing address `0x00000008`
      - Note memory is byte addressable, so address 8 is the third word
    - Load word requires the address to be *word-aligned*, that is, a multiple of 4
      - Can't load a word that's split up into two places in memory
  - Store word instruction writes a data word from a register into memory (writes 4 bytes at once)
    - e.g. `sw s0, 12(zero)` performs `mem[3] = a;`

Basic Assembly Program

```
.data # Global data section - stores data used by the whole program

# LIST is a label, which we can use to refer to the data later
# These 4 words could be stored anywhere, but they are guaranteed to be contiguous
LIST: .word 1, 2, 3, 4 ; Declare 4 words, initialize to 1, 2, 3, 4

.text # Program instructions

_start: # The entry point of the program; another label
    la s1, LIST # Load address of LIST into s1
    lw s2, 0(s1) # s2 = mem[LIST + 0]; s2 is now 1
    lw s3, 4(s1) # s3 = mem[LIST + 4]; s3 is now 2
    add s2, s2, s3 # s2 = 1 + 2; s2 is now 3
    lw s3, 8(s1) # s3 = mem[LIST + 8]; s3 is now 3
    add s2, s2, s3 # s2 = 3 + 3; s2 is now 6
    lw s3, 12(s1) # s3 = mem[LIST + 12]; s3 is now 4
    add s2, s2, s3 # s2 = 6 + 4; s2 is now 10

END: ebreak # Transfer control over to the debugger
# Without the ebreak, the processor keeps executing whatever is in memory
```

- `.data`, `.global`, `.text` are *assembler directives* – not instructions, but tell the assembler about what it should do
  - `.data` declares the global data section
    - We can use this to store data used by the whole program
    - In this example, it's an array
  - `.text` declares the section for the program itself
  - `.global` declares something to be visible outside the file (for a multi-file program)
  - `.word` declares the things that come next should take up an entire word of memory
- `la` is the load-address psuedo-instruction, which loads the address of some global data into a register

More Instructions

- Logic instructions
  - Bitwise operations that operate on 2 source registers
  - `and s0, s1, s2` puts the bitwise AND of `s1` and `s2` into `s0`
  - Similarly for `or s0, s1, s2` and `xor s0, s1, s2`
  - `not s0, s1` puts the bitwise NOT of `s1` into `s0`
    - Actually a pseudo-instruction, compiles to `xori s0, s1, -1`
  - Also have immediate versions `andi`, `ori`, `xori`

Lecture 24, Nov 3, 2022

More RISC-V Instructions

- 4 types of shift instructions:
  - Shift left logical `sll`
    - Fill the LSBs with zero
  - Shift right logical `srl`
    - Fill the MSBs with zero
  - Shift right arithmetic `sra`
    - Fill the MSBs with the sign bit
    - This keeps the sign of a two's complement negative integer
  - Also have immediate versions `slli`, `srli`, `srai`
    - The immediates are 5 bits, since we only ever need to shift something by a maximum of 32 bits
  - Left-shifting by N is equal to multiplying by 2^N
  - Right shifting by N is equal to dividing by 2^N (and truncating the remainder)
- Using shifts we can extract or assemble bit fields
  - If `s6 = 0x1234ABCD`:
  - `srli s6, s7, 8 # s6 = 0x001234AB`
  - `andi s6, s6, 0xFF # s6 = 0x000000AB`
- Accessing bytes or half-words
  - `lb` and `lbu` – load byte and load byte unsigned
    - Since we're taking 8-bits from memory and putting it into a 32-bit register we need to know what to do with the rest of the bits
    - `lb` fills the rest with the sign bit (sign extension), `lbu` fills with zero
    - Does not have to be aligned
  - `lh` and `lhu` – load half word and load half word unsigned
    - Has to be half-word aligned (address divisible by 2)
  - `sb` and `sh` – store byte and store half word
    - Both store the least significant byte and half-word of the register
    - This keeps the other bytes in the word the same, so there is no question of sign extension

Program Flow

- Like data, the program itself is also stored in memory
  - Each instruction is 32-bits so each consecutive instruction is a difference in memory address by 4
- The *program counter* (PC) holds the address of the current instruction the processor is executing
  - When an instruction completes, it's automatically incremented by 4
  - By changing the program counter, we can make the program jump around, thus accomplishing nonlinear program flow
- We have conditional branch instructions that modify the program counter based on some condition, so we can accomplish structures such as conditionals and loops
  - There are many flavours of conditional branch instructions, but they al compare 2 source registers

    * `beq s1, s2, LABEL` – branch if equal, will branch if `s1` and `s2` are equal and set the program counter to `LABEL`
    * `bne` – branch if not equal
    * `blt`, `bge` – branch less than, branch greater than or equal to
        - Unsigned versions `bltu, bgeu`
    * Pseudo-instructions:
        - `beqz`, `bnez` – branch if equal to zero/not equal to zero
        - `bnez`, `blez`, `bgtz`, `bltz`, etc
        - `bgt`, `ble`

Lecture 25, Nov 14, 2022

Program Flow (Continued)

- Jump instructions (unconditional branches): `j LABEL`
    - Jump and link `jal`, jump register `jr` relate to subroutines (function calls)
- In a loop, we jump back to the beginning of the loop if we want to keep looping
- In an if/else statement, we jump over the "if" code if the condition is not true

Examples

- Continuously decrement `s8` until it is zero:

```
LOOP1:
    addi s8, s8, -1 # Decrement s8
    bnez s8, LOOP1 # Jump back to the label if s8 is not zero
```

- Converting from C code:

```
if (s8 > s9) {
    // THEN code
}
else {
    // ELSE code
}
// AFTER code
```

```
    ble s8, s9, ELSE1
THEN1:
    # THEN code
    j AFTER1
ELSE1:
    # ELSE code
AFTER1:
    # AFTER code
```

- Note the conditional jump instructions can only compare against registers, not immediates, so we have to load an immediate into a register first if we want to compare against a constant value
- For loop example:

```
for (s8 = 1; s8 < 5; s8++) {
    s9 = s9 + s10;
}
```

```
    addi s8, zero, 1
    addi t0, zero, 5
LOOP3:
```

```
    bge s8, t0, DONE
    add s9, s9, s10
    addi s8, s8, 1
    j LOOP3
DONE:
    # Code after
```

Machine Code

- Assembly language is human readable, but ultimately compiled to machine code
- All instructions are encoded into 32 bits (even if they may not need as many), because regularity supports simplicity, which improves performance
- RISC-V has 4 min instruction formats:
  - R-type (register type): Instructions that use two register source operands, e.g. `add`
    * Bits 31-25 (7) are the function code `func7`
      - These are used if the instruction needs more bits than just the opcode to specify their behaviour
    * Bits 24-20 (5) represent source register #2 `rs2`
    * Bits 19-15 (5) represent source register #1 `rs1`
    * Bits 14-12 (3) are 3 more function bits `func3`
    * Bits 11-7 (5) represent the destination register `rd`
    * Bits 6-0 (7) represent the opcode `op`
      - These identify the operation
  - I-type (immediate type): Instructions that use a register and an immediate, e.g. `addi`
    * Bits 31-20 (12) are the immediate value `imm12`
    * Bits 19-15 (5) represent source register #1 `rs1`
    * Bits 14-12 (3) are 3 function bits `func3`
    * Bits 11-7 (5) represent the destination register `rd`
    * Bits 6-0 (7) represent the opcode `op`
    * Notice the regularity of how the 3 function bits, source register 1, destination register, and opcode are in the same bits as in R-type
  - S/B-type (store/branch type): Storing into memory or branching
  - U/J-type (upper immediate/jump type): Load upper immediate or jump
  - The type of instruction is part of the opcode
- Examples:
  - `add s2, s3, s4` (R-type)
    * Opcode for `add` is 51, `0b0110011`
    * Both `func7` and `func3` are 0
    * `s2` is `x18`, `0b10010`
    * `s3` is `x19`, `0b10011`
    * `s4` is `x20`, `0b10100`
    * The final encoded instruction is `0000000'10100'10011'000'10010'0110011` or `0x01498933`
  - `addi s0, s1, 15`
    * Opcode for `addi` is 19, function bits all 0
    * `s0` is `x8`, `s1` is `x9`
    * The final encoded instruction is `000000001111'01001'000'01000'0010011` or `0x00F48413`

Lecture 26, Nov 15, 2022

Subroutines (Functions)

- Allows code modularization and reuse
- Subroutines have input arguments and return values

- To invoke a subroutine we need to branch into it, but we need to branch back when the subroutine is done, so branching is different here
- The jump and link instruction `jal` is used (the "link" part remembers how to get back)
  - `jal LABEL` jumps to the label, and saves the program counter of the instruction after it into the return address register, `ra`
- To return from a subroutine use the jump register instruction `jr`
  - `jr ra` jumps back into the address in `ra`
  - There is only one `ra` register, so if we want to call subroutines inside subroutines we need to use the stack

Passing Arguments and Returning Values

- Calling subroutines involve a *calling convention*, an agreement between caller and callee
  - The caller and callee need to agree on where the arguments and return values are stored
  - The callee must also not interfere with the behaviour of the caller
    - The subroutine must not change any registers that the caller are using
- In RISC-V the 8 registers `a0` to `a7` are used for function arguments, from left to right
  - If we have more than 8 arguments, we need to use the stack
- The return value is stored in `a0`
- Example:

```c
int main() {
    add6(11, 22, 33, 44, 55, 66);
}

int add6(int a, int b, int c, int d, int e, int f) {
    return a + b + c + d + e + f;
}
```

```
_start:
    # Load all the arguments into registers
    addi a0, zero, 11
    addi a1, zero, 22
    addi a2, zero, 33
    addi a3, zero, 44
    addi a4, zero, 55
    addi a5, zero, 66
    # Call subroutine
    jal ADD6
END:
    ebreak

ADD6:
    # Add all the values together
    add s1, a0, a1
    add s2, a2, a3
    add s3, a3, a5
    add s1, s1, s2
    # Set a0 to return value
    add a0, s1, s3
    # Jump back
    jr ra
```

- Note problem with this: generally we don't want to use the `s` registers in the subroutine because the caller may be using these!
  - Solution is to use the stack

Lecture 27, Nov 17, 2022

Using the Stack

- A region of memory used for temporary storage of data
  - LIFO structure
- Starts at a large address offset, grows downward (i.e. to lower addresses)
- The *stack pointer* (in the `sp` register) points to the element at the top of the stack
  - Adding a word to the stack decrements the stack pointer by 4
- The stack is important for subroutine calls since we can use it to save and restore registers
  - By saving registers onto the stack, we can make sure a subroutine does not trample on the caller
- In RISC-V there are preserved registers and nonpreserved registers
  - Preserved registers `s0` to `s11` and `sp` must take on the same values before and after a subroutine call (i.e. subroutines must save these)
  - Non-preserved registers `t0` to `t6` can be changed by subroutines (i.e. subroutines are free to modify these)
    - Registers `a0` to `a7` are also non-preserved
  - In the example from the previous lecture, in order to respect the calling convention we need to push the `s` registers onto the stack, or use the `t` registers
  - Note this is only a convention and not enforced in hardware
- Example: pushing 3 registers onto the stack:

```
addi sp, sp, -12
sw s1, 8(sp)
sw s2, 4(sp)
sw s3, 0(sp)
```

- To restore the registers back:

```
lw s3, 0(sp)
sw s2, 4(sp)
lw s1, 8(sp)
addi sp, sp, 12
```

Nested Subroutines

- To call a subroutine from another subroutine, we need to save the `ra` register onto the stack
- Example:

```
int main() {
    add6(11, 22, 33, 44, 55, 66);
}

int add6(int a, int b, int c, int d, int e, int f) {
    return add3(a, b, c) + add3(d, e, f);
}

int add3(int x, int y, int z) {
    return x + y + z;
}
```

```
_start:
    # Load all the arguments into registers
    addi a0, zero, 11
    addi a1, zero, 22
    addi a2, zero, 33
    addi a3, zero, 44
```

```
    addi a4, zero, 55
    addi a5, zero, 66
    # Call subroutine
    jal add6
END:
    ebreak

add6:
    # Push the return address register onto the stack
    addi sp, sp, -4
    sw ra, 0(sp)
    # Call add3, which makes a0 = a0 + a1 + a2
    # This will overwrite ra
    jal add3
    # Save a0 temporarily
    addi t0, zero, a0
    # Load the arguments and call add3 again
    addi a0, zero, a3
    addi a1, zero, a4
    addi a2, zero, a5
    jal add3
    # Add the 2 results
    add a0, a0, t0
    # Return, but first pop ra off the stack
    lw ra, 0(sp)
    addi sp, sp, 4
    jr ra

add3:
    add a0, a0, a1
    add a0, a0, a2
    jr ra
```

- Using the stack we can push additional arguments onto it if we need more than 8 arguments
  - Freeing these arguments is the responsibility of the caller – the callee does not restore the stack pointer
- Caller save: `t0` to `t7`, `a0` to `a7`, `sp` if necessary
- Callee save: `s0` to `s11`, saved and restored before the callee returns

Lecture 28, Nov 21, 2022

Subroutine and Stack Example

```
int FINDSUM(int N) {
    int sum = 0;
    while (N != 0) {
        sum = sum + N;
        N = N - 1;
    }
    return sum;
}

int main() {
    FINDSUM(5);
```

```
}
```

```
.data
N: .word 5
.text
.global _start

_start:
    la s4, N
    lw a0, 0(s4)
    jal FINDSUM
    ebreak

FINDSUM:
    addi t0, zero, 0
WHILE:
    beqz a0, ENDLOOP
    add t0, t0, a0
    addi a0, a0, -1
    j WHILE
ENDLOOP:
    add a0, zero, t0
    jr ra
```

- Recursive version:

```
int FINDSUM(int N) {
    if (N == 0)
        return 0;
    else
        return N + FINDSUM(N - 1);
}
```

```
FINDSUM:
    bnez a0, PUSH
    jr ra
PUSH:
    addi sp, sp, -8
    sw a0, 4(sp)
    sw ra, 0(sp)

    addi a0, a0, -1
    jal FINDSUM

    lw t0, 4(sp)
    add a0, a0, t0

    lw ra, 0(sp)
    addi sp, sp, 8
    jr ra
```

Lecture 29, Nov 22, 2022

Input/Output

- Memory mapped I/O – we can manipulate I/O devices though loads and stores to specific memory addresses
  - Devices sit at the memory locations for certain addresses and respond to those addresses
  - Consequently real memory ignores those addresses
  - These are known as *address spaces*
- The address bus goes into an address decoder, which outputs enable signals to different memory or I/O devices depending on which device's memory map the address is in
  - This enable signal controls which device the data on the bus is written to
  - The address decoder also controls a mux of all the device outputs to select which device's output goes on the data input bus to the CPU
- Example: I/O device 1 is on memory address `0x20001000`; write the value 7 to this device and read its output
  - When we read data from the device, this may or may not be the same data we sent; it could also be e.g. an ack or some processed form of data

```
li s1, 0x20001000 # Load the device address
addi s0, zero, 7
sw s0, 0(s1)      # Write the value to the device
lw s0, 0(s1)      # Read back a value from the device
```

- Often we might need a delay loop to intentionally slow down the CPU to match the speed of the I/O device
  - e.g. using a delay when updating a counter connected to a hex display to make the numbers readable

Lecture 30, Nov 24, 2022

Polling

- Repeatedly checks to see if a device is ready or if there has been an event, e.g. checking for a button input
- Example: system with 4 keys turning on LEDs, keys are at the lowest 4 bits of `0xFF200050`, LEDs at the lowest 4 bits of `0xFF200000`

```
_start:
    li s0, 0xff200050 # Load address of keys
    li s1, 0xff200000 # Load address of LEDs

# Repeatedly check the keys to see if they have been pressed
POLL:
    lw s2, 0(s0)
    beqz s2, POLL # If no keys are pressed, poll again
WAIT:
    lw s3, 0(s0)
    bnez s3, WAIT # Wait until keys are released
    li s3, 1 # If key 0 is pressed, then s3 is 0b0001
    bne s2, s3, CHECK_1
    li s4, 0 # Turn on 0 LEDs if key 0 is pressed
    j UPDATE_LED
CHECK_1:
    li s3, 2
    bne s2, s3, CHECK_2
```

```
    li s4, 1
    j UPDATE_LED
CHECK_2:
    li s3, 4
    bne s2, s3, CHECK_3
    li s4, 3
    j UPDATE_LED
CHECK_3:
    li s4, 7 # No need to do another check, this is the only scenario left
UPDATE_LED:
    s2 s4, 0(s1) # Actually update the LEDs
    j POLL
```

Interrupts

- Polling is inefficient, since the processor is constantly checking for events and so cannot do other work
- Instead we can use interrupts: the processor executes code normally, and when an event occurs the code execution is interrupted so the processor can handle the event
- The CPU first needs to be configured to accept interrupts
- Since interrupts can occur at any point during code execution, in order to return to execution after handling an interrupt, the CPU needs to save the state (this is done automatically)
  - Similar to calling subroutines, except all registers are saved
  - The CPU then jumps to an interrupt handler, and when the interrupt is done, it restores the registers and goes back to the old
- Unlike polling, interrupts can happen anytime (once set up), but is more difficult to do
  - Polling is good when the wait is short
  - Interrupts are better for medium to long wait events
- Examples of events handled by interrupts:
  - External devices such as UARTs, USBs, network adapters, etc
  - OS timer
  - Disk I/O
  - Debugging breakpoints
  - Program errors (e.g. misaligned memory access, divide by zero, segfaults)
    * These are also known as exceptions and always arise within the CPU
- Interrupts from external devices come from an IRQ (interrupt request) line when the interrupt is acknowledged, the CPU sends back a signal to the device via an IACK (interrupt acknowledge) line
  - When the external device receives the ack, it will de-assert the IRQ line
- Interrupt handling uses Control and Status Registers (CSRs), which are special registers that monitor system state
  - Cannot be repurposed and are not interchangeable
  - Can be read and written to, but need special instructions
  - `ustatus`, `uie` (interrupt enable), `utvec` (trap/interrupt handler base address), `uepc` (exception program counter)
  - CSRs that start with `u` are "user" registers
- System instructions:
  - `ebreak` - pause execution (breakpoint)
  - `uret` - return from interrupt handler
  - `csrrw` - read/write CSR
  - `csrrsi` - read and set bits in CSR (immediate)

Lecture 31, Nov 28, 2022

Using Interrupts

- When an interrupt occurs:
  1. The CPU saves the pc of the interrupted instruction to `uepc`
  2. The CPU jumps to the interrupt handler address, as specified by the interrupt vector table `utvec`
  3. The interrupt handler code executes
  4. Execution returns to interrupted instruction
- To enable interrupts:
  1. Configure the specific device being used as the interrupt source
  2. Set `utvec` to the address of the interrupt handler
  3. Set interrupt enable bit in `ustatus`
  4. Set interrupt bit for interrupt source in `uie`
- The interrupt handler is like a subroutine, except it cannot use any registers without saving them, as interrupts can occur at any time
- Example: timer interrupt
  - `0xFFFF0018` and `0xFFFF001C` hold the current time count
  - `0xFFFF0020` and `0xFFFF0024` hold the timer comparison value (interrupt on hitting this value)

```
.data
curr_time:  .word 0xffff0018
time_cmp:   .word 0xffff0020
counter:    .word 0

.text
.global _start
_start:
    # Set up timer
    la s0, time_cmp
    li s1, 1000
    sw s1, 0(s0)
    # Enable interrupts
    la t0, timer_handler
    csrrw zero, utvec, t0
    csrrsi zero, utstatus, 1
    csrrsi zero, uie, 0x10
LOOP:
    # The processor can now do anything
    j LOOP

timer_handler:
    # Save registers as necessary
    addi sp, sp -12
    sw t0, 0(sp)
    sw t1, 4(sp)
    sw t2, 8(sp)
    # Increment counter
    la t1, counter
    lw t2, 0(t1)
    addi t2, t2, 1
    sw t2, 0(t1)
    # Set a new timer comparision value (TODO)
    lw t0, 0(sp)
    lw t1, 4(sp)
```

```
    lw t2, 8(sp)
    addi sp, sp, 12
    uret
```

Lecture 32, Dec 1, 2022

Timing Analysis

- So far we've assumed that our gates operate with zero delay, but this is not true
- Capacitance of wires and gates introduce delay, which determines the clock speed
- Consider:

```
module muxDFF(input logic w, r, L, E, clock,
              output logic Q);
    always_ff @(posedge clock)
        if (L)
            Q <= r;
        else if (E)
            Q <= w;
endmodule

module shift4(input logic [3:0] SW, KEY,
              output logic [3:0] LEDR);
    logic [3:0] Q;
    muxDFF u3(KEY[0], SW[3], KEY[2], KEY[1], KEY[3], Q[3]);
    muxDFF u2(Q[3], SW[2], KEY[2], KEY[1], KEY[3], Q[2]);
    muxDFF u1(Q[2], SW[1], KEY[2], KEY[1], KEY[3], Q[1]);
    muxDFF u0(Q[1], SW[0], KEY[2], KEY[1], KEY[3], Q[0]);
    assign LEDR = Q;
endmodule
```

- What is the maximum clock frequency at which this circuit can operate?
  - t_{CQ} is the "clock to Q" propagation delay, time delay from clock change to output observed
  - There are also delays through the muxes before we're ready to load D again
- Timing analysis for the flip flops:
  - *Setup time* (t_{su}): D has to have been stable for some amount of time before the clock edge
  - *Hold time* (t_h): D has to be held stable for some amount of time after the clock edge
- The minimum clock period $T_{min} = t_{CQ} + t_{mux} + t_{mux} + t_{su}$
  - The signal must propagate through the first flip flop and then two muxes, and then be stable for some amount of time
  - $F_{max} = \frac{1}{T_{min}}$
- When doing the analysis we look at flip-flop to flip-flop paths, from a source to a sink, and we find the longest path

Clock Skew

- It is possible for the clock edge to not arrive at the same time to all the flip flops
  - On a large chip there can be significant delay in the wires
- On a diagram this is shown as a box with a delta in it, to show the possibility of skew
- We measure the skew time as t_{skew} = arrival time of clock at the sink - arrival time of clock at the source
  - The sink takes its input from the output of the source
- Skew can be positive or negative

  - Positive skew allows us to operate at a higher frequency, since this gives more time between the rising edge of the source clock to the next rising edge of the sink clock
  - Negative skew gives a lower maximum frequency
  - The effective period is reduced/elongated by the skew time

Set-Up Time Violation

- Find all paths between flip flops, and look for the longest such path, including setup time
- Another way to think about the setup-time violation is that the logic is too slow for the data to arrive at the input of the sink before the clock changes
- The constraint is $t_{min} \geq t_{CQ} + t_{\text{logic,max}} + t_{su} + \Delta$

Lecture 33, Dec 5, 2022

Hold Time Violation

- We also want to calculate for hold time violations, which are dictated by the *shortest* path between flip flops
  - Whereas the previous one did not consider hold time, this does not consider setup time
- In this case the data tries to race ahead of the clock (race through) and change D of the sink during the hold time of the previous signal
- Constraint: $t_h + \Delta \leq t_{CQ} + t_{\text{logic,min}}$
  - $t_h + \Delta$ is the amount of time required for D of the sink to be stable, from the rising edge of the source clock
  - $t_{CQ} + t_{\text{logic,min}}$ is the time between rising edge of the source clock and the signal propagating to D of the sink
- In order to prevent this, we need to delay the change of D, sometimes by inserting additional logic to increase $t_{\text{logic,min}}$
- Note this is unrelated to clock frequency