Lecture 15: Review Session

- Today's topics:
  - FSM examples
  - Midterm review session
  - Midterm rules:

Students are allowed to bring 3 A4/letter-sized sheets of paper with anything written/printed on both sides. In addition, you may bring the ``green sheet''. You may also bring a phone/calculator that can be used for any numeric calculations (but it's also ok to write a mathematical term, say 1.4/2.2 GHz without doing the calculation). You may of course not use your phone to surf the web or consult with others during the test. You may also not use the MARS simulator or other calculators/tools for numeric conversions. If necessary, make reasonable assumptions and clearly state them. The only clarifications you may ask for during the exam are definitions of terms. You will receive partial credit if you show your steps and explain your line of thinking, so attempt every question even if you can't fully solve it. Complete your answers in the space provided (including the back-side of each page). Confirm that you have 14 questions on 8 pages, followed by a blank page. Turn in your answer sheets before 10:35am. The test is worth 100 points and you have about 90 minutes, so allocate time accordingly.

Tackling FSM Problems

- Three questions worth asking:
  - What are the possible output states? Draw a bubble for each.
  - What are inputs? What values can those inputs take?
  - For each state, what do I do for each possible input value? Draw an arc out of every bubble for every input value.

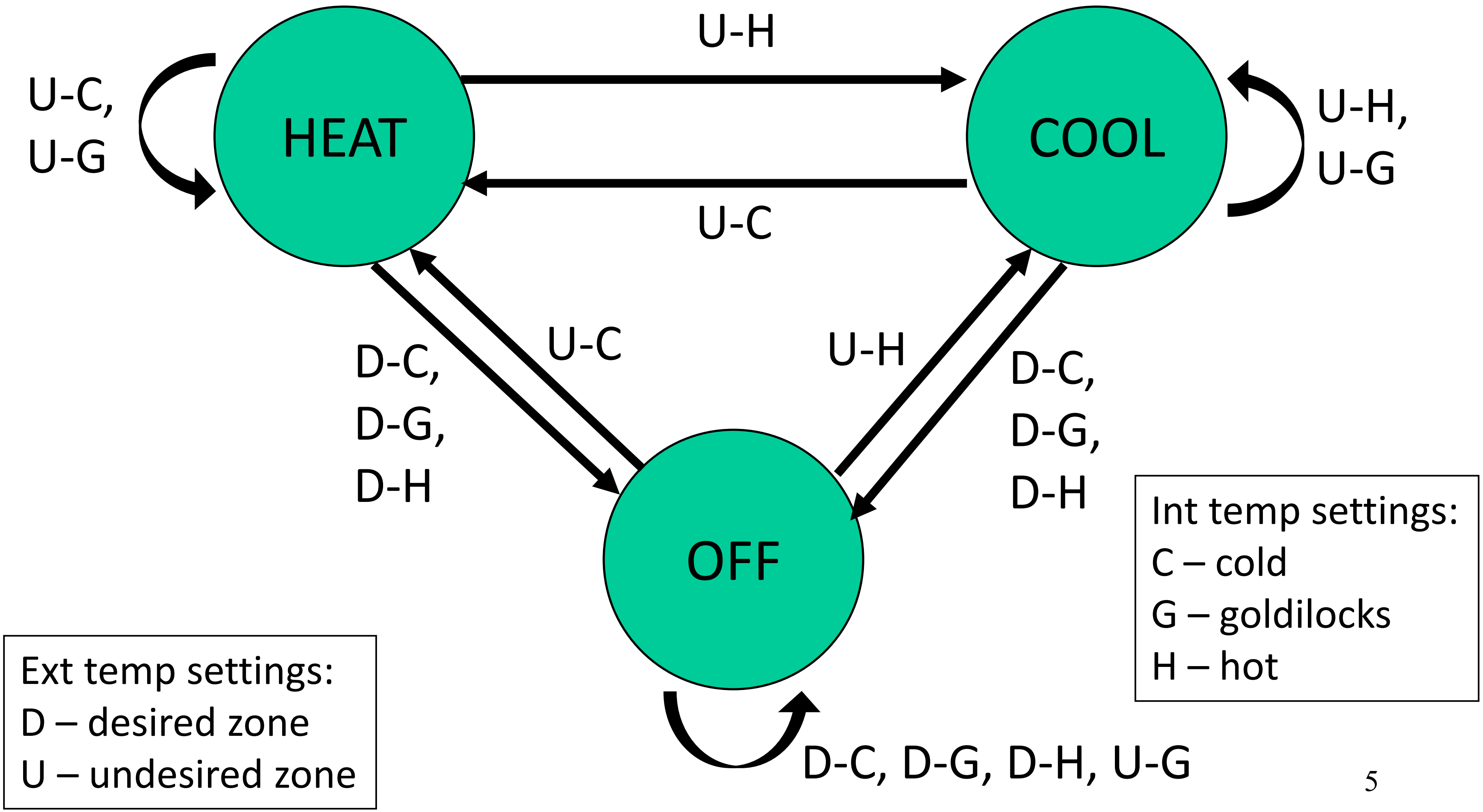
Example – Residential Thermostat

- Two temp sensors: internal and external
- If internal temp is within 1 degree of desired, don't change setting
- If internal temp is > 1 degree higher than desired, turn AC on; if internal temp is < 1 degree lower than desired, turn heater on
- If external temp and desired temp are within 5 degrees, disregard the internal temp, and turn both AC and heater off

Finite State Machine Table

Current State	Input E	Input I	Output State
HEAT	D	C	OFF
HEAT	D	G	OFF
HEAT	D	H	OFF
HEAT	U	C	HEAT
HEAT	U	G	HEAT
HEAT	U	H	COOL
COOL	D	C	OFF
COOL	D	G	OFF
COOL	D	H	OFF
COOL	U	C	HEAT
COOL	U	G	COOL
COOL	U	H	COOL
OFF	D	C	OFF
OFF	D	G	OFF
OFF	D	H	OFF
OFF	U	C	HEAT
OFF	U	G	OFF
OFF	U	H	COOL

Finite State Diagram



Vacuum Robot Example

Consider the following sequential circuit for an automated vacuum cleaning device. The circuit decides if the steering should be kept straight, or if it should be moved right, or moved left. The device has a sensor that determines if the device has hit a wall. If a wall hasn't been hit, the steering is kept straight. If a wall has been hit, the steering is turned to the opposite of its last non-straight position, i.e., if the steering's last non-straight position was right, the steering is now moved to left.

- What are the output states for this finite state machine?
- What are the different input values being received by the finite state machine?
- Construct the finite state transition table for this circuit.

Vacuum Robot Example

At first, the output of the circuit appears to be LEFT, RIGHT, STRAIGHT. But the circuit also needs to remember if the steering's last non-straight setting was LEFT or RIGHT. I could either do this with an internal variable that remembers this state, or I could make this part of the output state. I'll go with the latter approach, so the output states now become LEFT, RIGHT, LSTRAIGHT, RSTRAIGHT.

There's only 1 input, which is the wall-detecting sensor, which is either 1 (wall detected) or 0 (wall not detected).

The finite state table will therefore have 8 entries (4 states, each receiving 2 possible input values).

Vacuum Robot Example

CURR-STATE	INPUT	NEXT-STATE
LEFT	0	LSTRAIGHT
RIGHT	0	RSTRAIGHT
LSTRAIGHT	0	LSTRAIGHT
RSTRAIGHT	0	RSTRAIGHT
LEFT	1	RIGHT
RIGHT	1	LEFT
LSTRAIGHT	1	RIGHT
RSTRAIGHT	1	LEFT

Modern Trends

- Historical contributions to performance:
  - Better processes (faster devices) ~20%
  - Better circuits/pipelines ~15%
  - Better organization/architecture ~15%

Today, annual improvement is closer to 20%; this is primarily because of slowly increasing transistor count and more cores.

Need multi-thread parallelism and accelerators to boost performance every year.

Performance Measures

- Performance = 1 / execution time
- Speedup = ratio of performance
- Performance improvement = speedup -1
- Execution time = clock cycle time x CPI x number of instrs

Program takes 100 seconds on ProcA and 150 seconds on ProcB

Speedup of A over B = 150/100 = 1.5
Performance improvement of A over B = 1.5 – 1 = 0.5 = 50%

Speedup of B over A = 100/150 = 0.66 (speedup less than 1 means performance went down)
Performance improvement of B over A = 0.66 – 1 = -0.33 = -33%
or Performance degradation of B, relative to A = 33%

If multiple programs are executed, the execution times are combined into a single number using AM, weighted AM, or GM

Performance Equations

CPU execution time = CPU clock cycles x Clock cycle time

CPU clock cycles = number of instrs x avg clock cycles per instruction (CPI)

Substituting in previous equation,

Execution time = clock cycle time x number of instrs x avg CPI

If a 2 GHz processor graduates an instruction every third cycle, how many instructions are there in a program that runs for 10 seconds?

Power Consumption

- Dyn power α activity x capacitance x voltage^2 x frequency
- Capacitance per transistor and voltage are decreasing, but number of transistors and frequency are increasing at a faster rate
- Leakage power is also rising and will soon match dynamic power
- Power consumption is already around 100W in some high-performance processors today

Basic MIPS Instructions

- lw $t1, 16($t2)
- add $t3, $t1, $t2
- addi $t3, $t3, 16
- sw $t3, 16($t2)
- beq $t1, $t2, 16
- blt is implemented as slt and bne
- j 64
- jr $t1
- sll $t1, $t1, 2

Convert to assembly:

```
 while   (save[i] == k)
      i += 1;
```

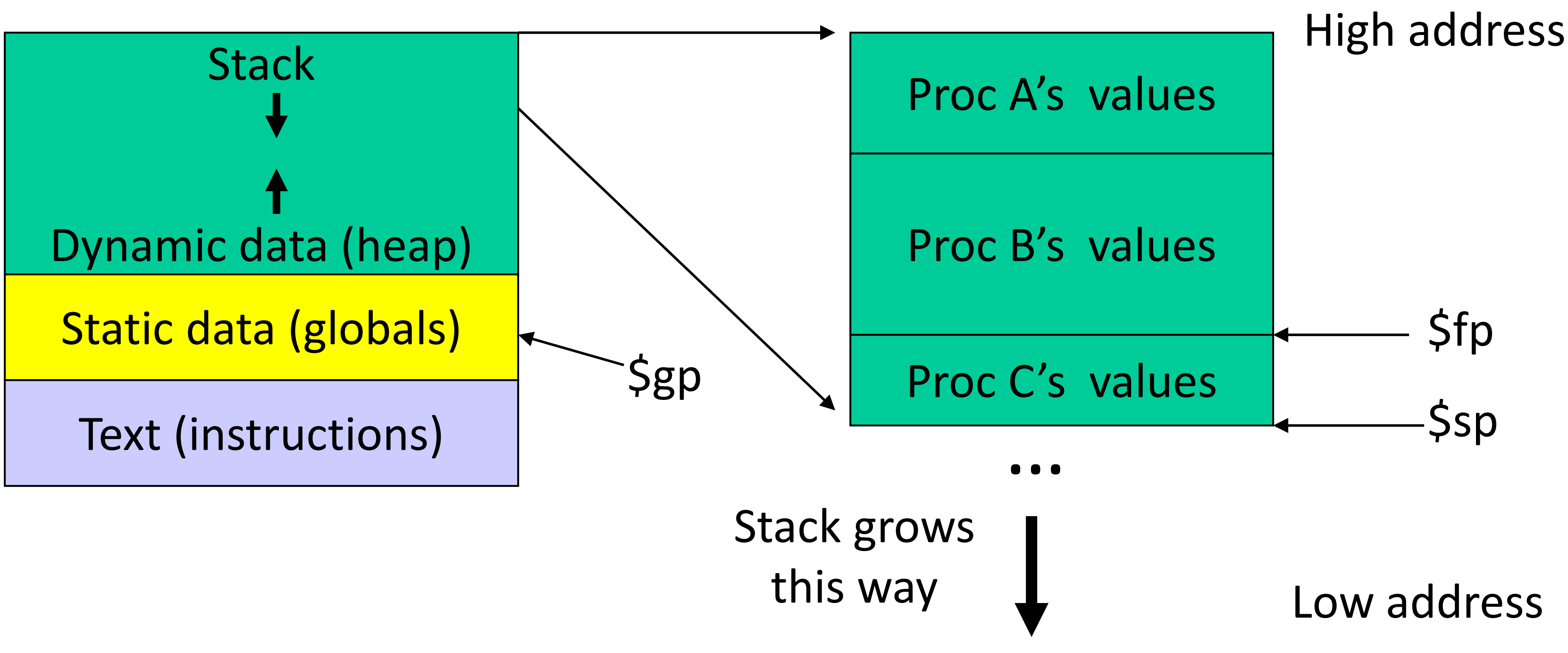
i and k are in $s3 and $s5 and base of array save[] is in $s6

```
Loop:  sll    $t1, $s3, 2
       add    $t1, $t1, $s6
       lw     $t0, 0($t1)
       bne    $t0, $s5, Exit
       addi   $s3, $s3, 1
       j      Loop
Exit:
```

Registers

- The 32 MIPS registers are partitioned as follows:
  - Register 0 : $zero always stores the constant 0
  - Regs 2-3 : $v0, $v1 return values of a procedure
  - Regs 4-7 : $a0-$a3 input arguments to a procedure
  - Regs 8-15 : $t0-$t7 temporaries
  - Regs 16-23: $s0-$s7 variables
  - Regs 24-25: $t8-$t9 more temporaries
  - Reg 28 : $gp global pointer
  - Reg 29 : $sp stack pointer
  - Reg 30 : $fp frame pointer
  - Reg 31 : $ra return address

Memory Organization



Procedure Calls/Returns

```
procA (int i)
{
    int j;
    j = …;
    i = call procB(j);
   … = i;
}
```

```
procB (int j)
{
    int k;
    … = j;
    k = …;
    return k;
}
```

```
procA:
    $s0 = … # value of j
    $t0  = … # some tempval
    $a0 = $s0  # the argument
    …
    jal  procB
    …
    … = $v0
```

```
procB:
    $t0  = … # some tempval
    … = $a0 # using the argument
    $s0 = … # value of k
    $v0 = $s0;
    jr   $ra
```

Saves and Restores

- Caller saves:
  - $ra, $a0, $t0, $fp (if reqd)
- Callee saves:
  - $s0

- As every element is saved on stack, the stack pointer is decremented

```
procA:
    $s0 = … # value of j
    $t0  = … # some tempval
    $a0 = $s0  # the argument
    …
    jal  procB
    …
    … = $v0
```

```
procB:
    $t0  = … # some tempval
    … = $a0 # using the argument
    $s0 = … # value of k
    $v0 = $s0;
    jr   $ra
```

Example 2

```
int   fact  (int n)
{
    if (n < 1)  return (1);
        else return (n * fact(n-1));
}
```

Notes:
The caller saves $a0 and $ra in its stack space.
Temps are never saved.

```
fact:
    addi     $sp, $sp, -8
    sw       $ra, 4($sp)
    sw       $a0, 0($sp)
    slti     $t0, $a0, 1
    beq      $t0, $zero, L1
    addi     $v0, $zero, 1
    addi     $sp, $sp, 8
    jr       $ra
L1:
    addi     $a0, $a0, -1
    jal      fact
    lw       $a0, 0($sp)
    lw       $ra, 4($sp)
    addi     $sp, $sp, 8
    mul      $v0, $a0, $v0
    jr       $ra
```

Recap – Numeric Representations

- Decimal $35_{10} = 3 \times 10^1 + 5 \times 10^0$
- Binary $00100011_2 = 1 \times 2^5 + 1 \times 2^1 + 1 \times 2^0$
- Hexadecimal (compact representation)

  0x 23 or $23_{hex} = 2 \times 16^1 + 3 \times 16^0$

  0-15 (decimal) → 0-9, a-f (hex)

Dec	Binary	Hex	Dec	Binary	Hex	Dec	Binary	Hex	Dec	Binary	Hex
0	0000	00	4	0100	04	8	1000	08	12	1100	0c
1	0001	01	5	0101	05	9	1001	09	13	1101	0d
2	0010	02	6	0110	06	10	1010	0a	14	1110	0e
3	0011	03	7	0111	07	11	1011	0b	15	1111	0f

2’s Complement

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
…
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2^{31}-1$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2^{31}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -(2^{31} - 1)$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -(2^{31} - 2)$
…
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1$

Note that the sum of a number x and its inverted representation x’ always equals a string of 1s (-1).

$x + x' = -1$
$x' + 1 = -x$ … hence, can compute the negative of a number by
$-x = x' + 1$ inverting all bits and adding 1

This format can directly undergo addition without any conversions!

Each number represents the quantity

$x_{31}\ -2^{31} + x_{30}\ 2^{30} + x_{29}\ 2^{29} + \ldots + x_1\ 2^1 + x_0\ 2^0$

Multiplication Example

Multiplicand		1000_{ten}
Multiplier	x	1001_{ten}

```
         1000
        0000
       0000
      1000
```

Product 1001000_{ten}

In every step

- multiplicand is shifted
- next bit of multiplier is examined (also a shifting step)
- if this bit is 1, shifted multiplicand is added to the product

Division

```
                              1001ten          Quotient
Divisor    1000ten   |     1001010ten          Dividend
                          -1000
                              10
                              101
                              1010
                             -1000
                                 10ten         Remainder
```

At every step,

- shift divisor right and compare it with current dividend
- if divisor is larger, shift 0 as the next bit of the quotient
- if divisor is smaller, subtract to get new dividend and shift 1 as the next bit of the quotient

Division

		1001_{ten}	Quotient
Divisor	1000_{ten}	| 1001010_{ten}	Dividend

0001001010 0001001010 0000001010 0000001010

100000000000 → 0001000000→ 0000100000→0000001000

Quo: 0 000001 0000010 000001001

At every step,

- shift divisor right and compare it with current dividend
- if divisor is larger, shift 0 as the next bit of the quotient
- if divisor is smaller, subtract to get new dividend and shift 1 as the next bit of the quotient

Divide Example

- Divide 7_{ten} ($0000\ 0111_{two}$) by 2_{ten} (0010_{two})

Iter	Step	Quot	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	Rem = Rem – Div	0000	0010 0000	1110 0111
	Rem < 0 → +Div, shift 0 into Q	0000	0010 0000	0000 0111
	Shift Div right	0000	0001 0000	0000 0111
2	Same steps as 1	0000	0001 0000	1111 0111
		0000	0001 0000	0000 0111
		0000	0000 1000	0000 0111
3	Same steps as 1	0000	0000 0100	0000 0111
4	Rem = Rem – Div	0000	0000 0100	0000 0011
	Rem >= 0 → shift 1 into Q	0001	0000 0100	0000 0011
	Shift Div right	0001	0000 0010	0000 0011
5	Same steps as 4	0011	0000 0001	0000 0001

Binary FP Numbers

- 20.45 decimal = ? Binary

- 20 decimal = 10100 binary

- 0.45 x 2 = 0.9 (not greater than 1, first bit after binary point is 0)
  0.90 x 2 = 1.8 (greater than 1, second bit is 1, subtract 1 from 1.8)
  0.80 x 2 = 1.6 (greater than 1, third bit is 1, subtract 1 from 1.6)
  0.60 x 2 = 1.2 (greater than 1, fourth bit is 1, subtract 1 from 1.2)
  0.20 x 2 = 0.4 (less than 1, fifth bit is 0)
  0.40 x 2 = 0.8 (less than 1, sixth bit is 0)
  0.80 x 2 = 1.6 (greater than 1, seventh bit is 1, subtract 1 from 1.6)

… and the pattern repeats

10100.011100110011001100…

Normalized form = 1.0100011100110011… x 2^4

Examples

Final representation: $(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$

- Represent -0.75_{ten} in single and double-precision formats

  Single: (1 + 8 + 23)

  Double: (1 + 11 + 52)

> Remember:
>
> True exponent —(+127)→ Exponent in register
>
> Exponent in register —(-127)→ True exponent

- What decimal number is represented by the following single-precision number?

  1 1000 0001 01000…0000

Examples

Final representation: $(-1)^S$ x (1 + Fraction) x $2^{(\text{Exponent} - \text{Bias})}$

- Represent -0.75_{ten} in single and double-precision formats

  Single: (1 + 8 + 23)
  1 0111 1110 1000…000

  Double: (1 + 11 + 52)
  1 0111 1111 110 1000…000

- What decimal number is represented by the following single-precision number?
  1 1000 0001 01000…0000
  -5.0

Example 2

Final representation: $(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$

- Represent 36.90625_{ten} in single-precision format

36 / 2 = 18 rem 0	0.90625 x 2 = 1.81250
18 / 2 = 9 rem 0	0.8125 x 2 = 1.6250
9 / 2 = 4 rem 1	0.625 x 2 = 1.250
4 / 2 = 2 rem 0	0.25 x 2 = 0.50
2 / 2 = 1 rem 0	0.5 x 2 = 1.00
1 / 2 = 0 rem 1	0.0 x 2 = 0.0

36 is 100100

0.90625 is 0.1110100...0

Example 2

Final representation: $(-1)^S$ x (1 + Fraction) x $2^{(\text{Exponent} - \text{Bias})}$

We've calculated that 36.90625_{ten} = 100100.1110100…0 in binary
Normalized form = 1.001001110100…0 x 2^5
(had to shift 5 places to get only one bit left of the point)

The sign bit is 0 (positive number)
The fraction field is 001001110100…0 (the 23 bits after the point)
The exponent field is 5 + 127 (have to add the bias) = 132,
which in binary is 10000100

The IEEE 754 format is 0 10000100 001001110100…..0
sign exponent 23 fraction bits

Value inf — 2 special cases up top that use the reserved exponent field of 255 — 0 255 00…0

Value NAN — 0 255 xx….x

Highest value $\sim 2 \times 2^{127}$ — 0 254 11….1

Value 1 — 0 127 00…0

Exponent field < 127, i.e., after subtracting bias, they are negative exponents, representing numbers < 1

Smallest Norm $\sim 2 \times 2^{-126}$ — 0 0..01 00…0

Largest Denorm $\sim 1 \times 2^{-126}$ — 0 0..00 11…1

Smallest Denorm $\sim 2^{-149}$ — 0 0..00 00…1

Special case with exponent field 0, used to represent denorms, that help us gradually approach 0

Value 0 — 0 00..0 00…0

Same rules as above, but the sign bit is 1

Same magnitudes as above, but negative numbers

FP Addition – Binary Example

- Consider the following binary example

  $1.010 \times 2^1 \quad + \quad 1.100 \times 2^3$

  Convert to the larger exponent:

  $0.0101 \times 2^3 \quad + \quad 1.1000 \times 2^3$

  Add

  1.1101×2^3

  Normalize

  1.1101×2^3

  Check for overflow/underflow

  Round

  Re-normalize

  IEEE 754 format: 0 10000010 11010000000000000000000

Boolean Algebra

- $\overline{A + B} = \overline{A} . \overline{B}$

- $\overline{A . B} = \overline{A} + \overline{B}$

A	B	C	E
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Any truth table can be expressed as a sum of products

$(A . B . \overline{C}) + (A . C . \overline{B}) + (C . B . \overline{A})$

- Can also use "product of sums"
- Any equation can be implemented with an array of ANDs, followed by an array of ORs

Adder Implementations

- Ripple-Carry adder – each 1-bit adder feeds its carry-out to next stage – simple design, but we must wait for the carry to propagate thru all bits

- Carry-Lookahead adder – each bit can be represented by an equation that only involves input bits (a_i, b_i) and initial carry-in (c_0) -- this is a complex equation, so it's broken into sub-parts

For bits a_i, $b_{i,}$, and c_i, a carry is generated if $a_i.b_i = 1$ and a carry is propagated if $a_i + b_i = 1$

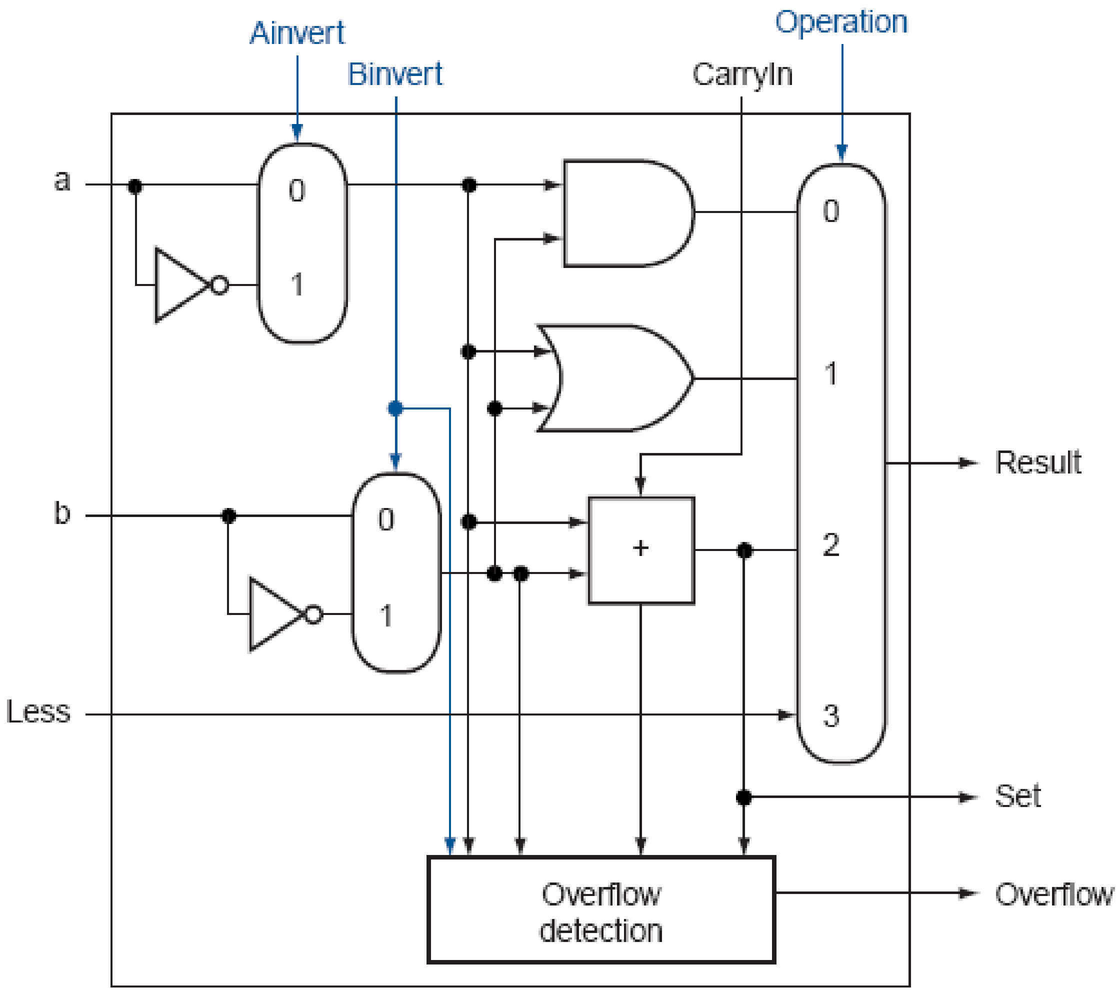
$$C_{i+1} = g_i + p_i \cdot C_i$$

Similarly, compute these values for a block of 4 bits, then for a block of 16 bits, then for a block of 64 bits....Finally, the carry-out for the 64^{th} bit is represented by an equation such as this:

$$C_4 = G_3 + G_2.P_3 + G_1.P_2.P_3 + G_0.P_1.P_2.P_3 + C_0.P_0.P_1.P_2.P_3$$

Each of the sub-terms is also a similar expression

32-bit ALU

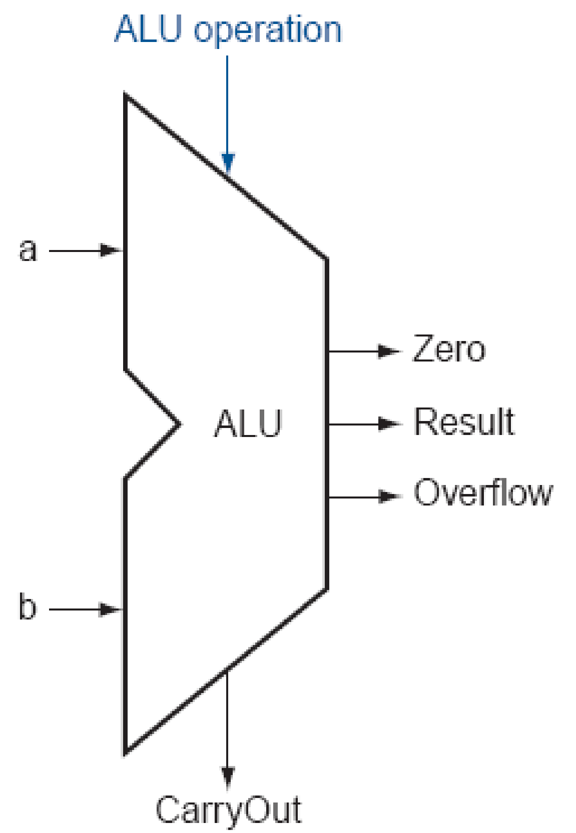


Source: H&P textbook

Control Lines

What are the values of the control lines and what operations do they correspond to?

	Ai	Bn	Op
AND	0	0	00
OR	0	0	01
Add	0	0	10
Sub	0	1	10
NOR	1	1	00
NAND	1	1	01
SLT	0	1	11
BEQ	0	1	10 (xx)



Source: H&P textbook