Lecture 14: Recap

- Today's topics:
  - Recap for mid-term
  - No electronic devices in midterm

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;
```

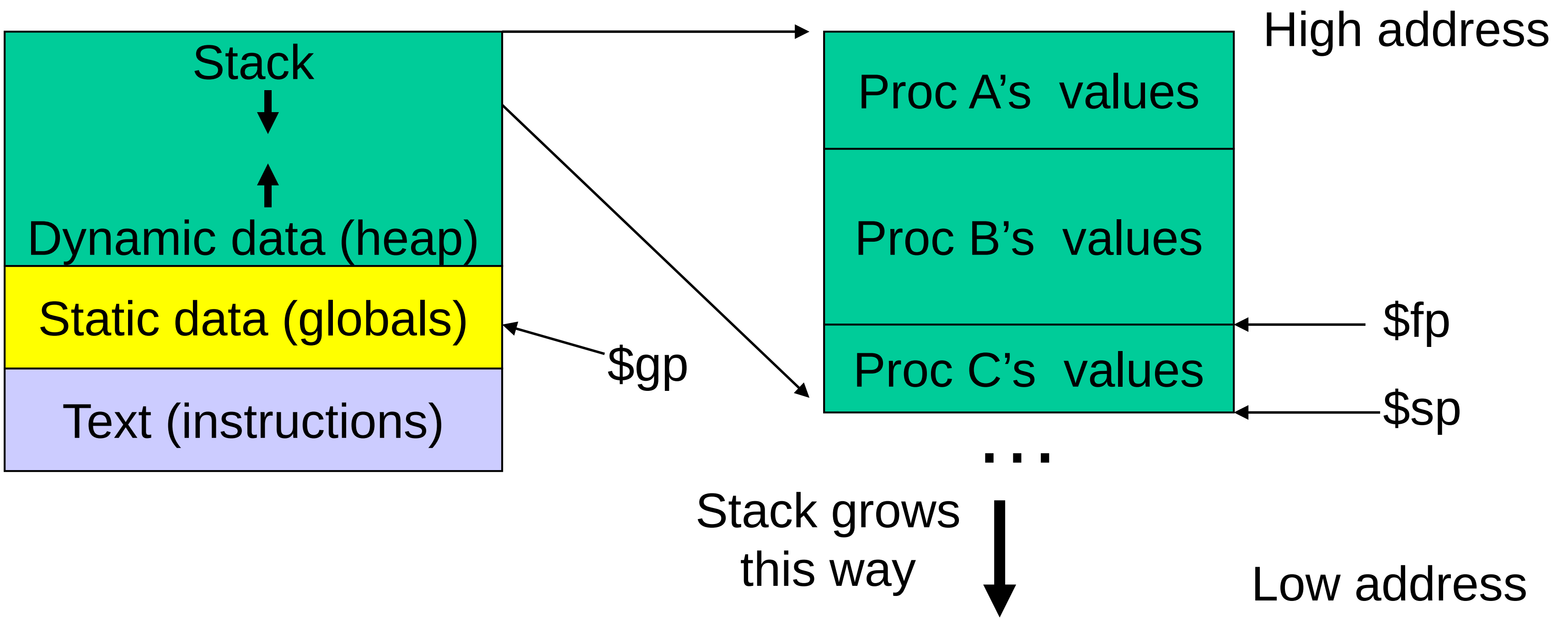
i and k are in $s3 and $s5 and base of array save[] is in $s6

```
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 = …;
    call procB(j);
   … = j;
}
```

```
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));
}
```

```
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
```

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

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

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

Multiplication Example

```
Multiplicand              1000ten
Multiplier            x   1001ten
                      ---------------
                            1000
                           0000
                          0000
                         1000
                      ----------------
Product                  1001000ten
```

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

1001_{ten} Quotient

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

-1000

10

101

1010

-1000

10_{ten} 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

IEEE 754 Format

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

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

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

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

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

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

- 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 $a_i + b_i = 1$

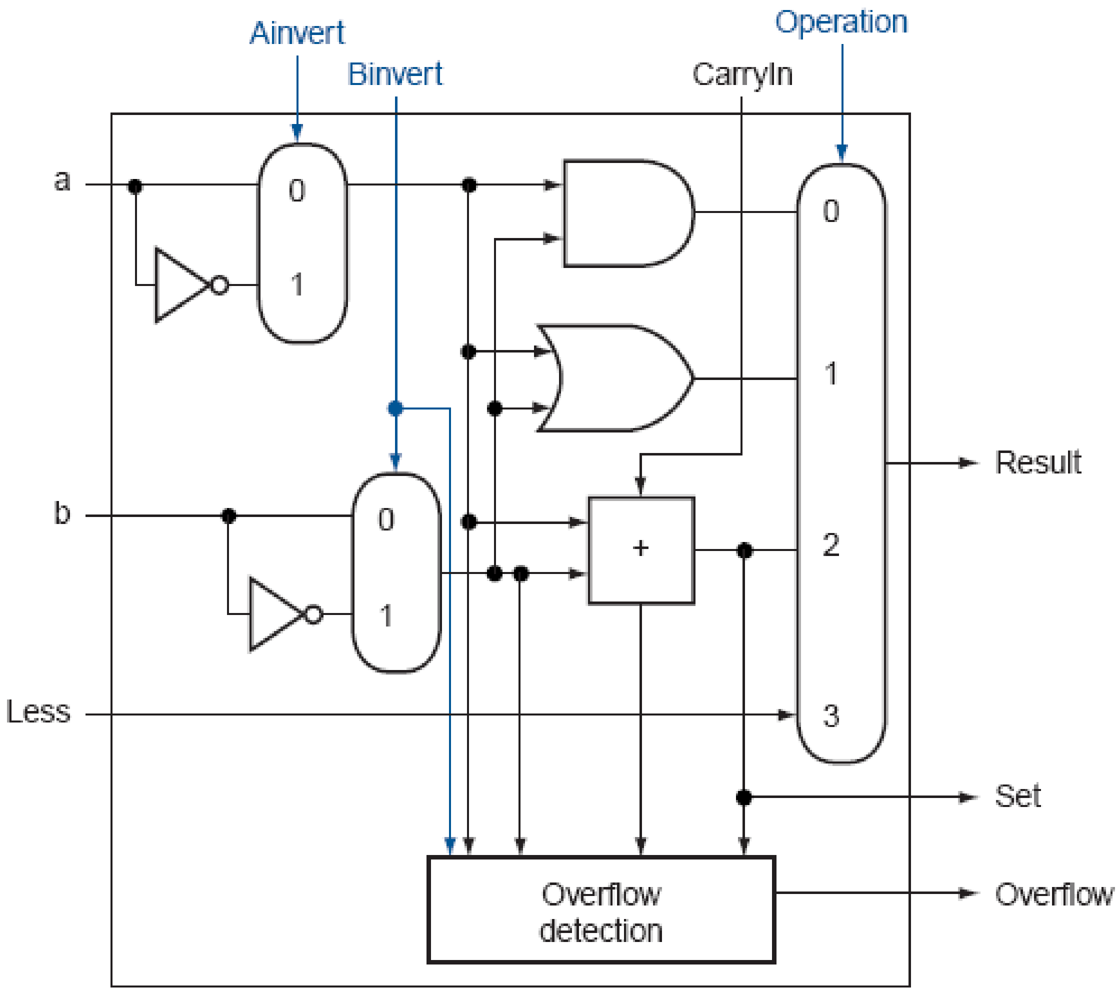
  $C_{i+1} = g_i + p_i \, . \, C_i$

  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 64th 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

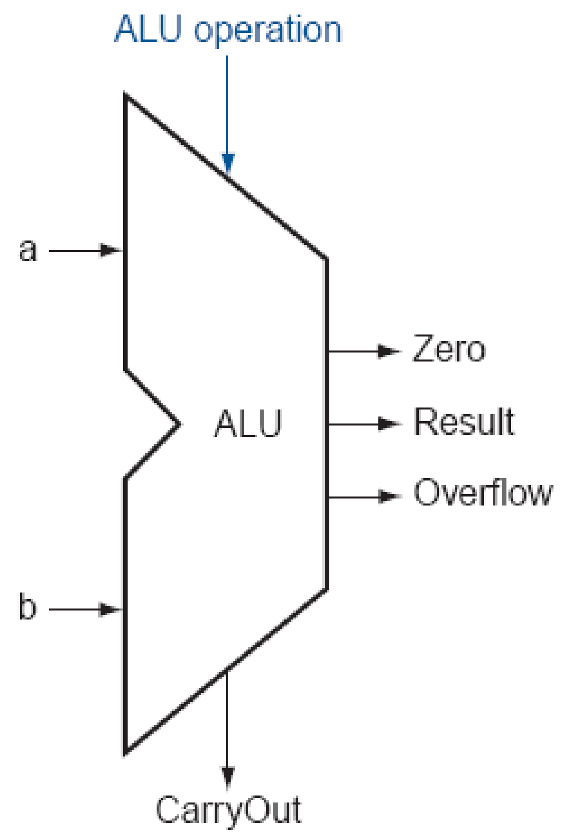


Source: H&P textbook

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
SLT	0	1	11
NOR	1	1	00



Source: H&P textbook

State Transition Table

- Problem description: A traffic light with only green and red; either the North-South road has green or the East-West road has green (both can't be red); there are detectors on the roads to indicate if a car is on the road; the lights are updated every 30 seconds; a light must change only if a car is waiting on the other road

State Transition Table:

CurrState	InputEW	InputNS	NextState=Output
N	0	0	N
N	0	1	N
N	1	0	E
N	1	1	E
E	0	0	E
E	0	1	N
E	1	0	E
E	1	1	N

Midterm Question

Thermostat can be OFF, HEAT, COOL
Two input sensors
Ext temp within 5 degrees of desired temp → OFF
Int temp within 1 degree of desired temp → no change
Int temp < desired temp -1 → HEAT
Int temp > desired temp +1 → COOL

Title