Lecture 15: Recap

- Today's topics:
  - Recap for mid-term
- Reminders:
  - no class Thursday
  - office hours on Monday (10am-4pm)
  - mid-term Tuesday (arrive early, questions will be handed out at 9am, open-notes-slides-textbook-assignments)

Modern Trends

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

In the future, bullet-2 will help little and bullet-3 will not help much for a single core!

	Pentium	P-Pro	P-II	P-III	P-4	Itanium	Montecito
Year	1993	95	97	99	2000	2002	2005
Transistors	3.1M	5.5M	7.5M	9.5M	42M	300M	1720M
Clock Speed	60M	200M	300M	500M	1500M	800M	1800M

Moore's Law in action

At this point, adding transistors to a core yields little benefit

Power Consumption Trends

- 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;
```

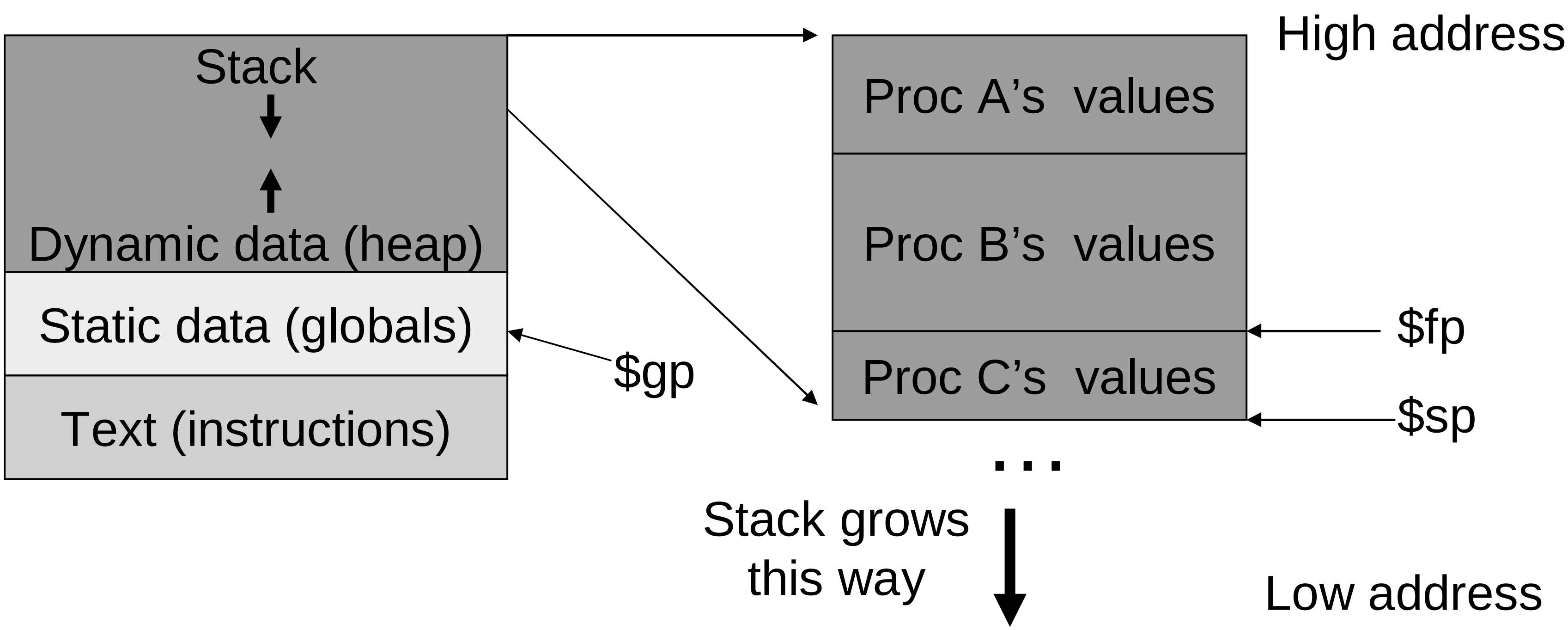
i and k are in $s3 and $s5 and base of array save[] is in $s6

```
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 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

- Callee saves:
  - $s0

- As every element is saved on stack, the stack pointer is decremented
- If the callee's values cannot remain in registers, they will also be spilled into the stack (don't have to create space for them at the start of the proc)

```
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
```

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

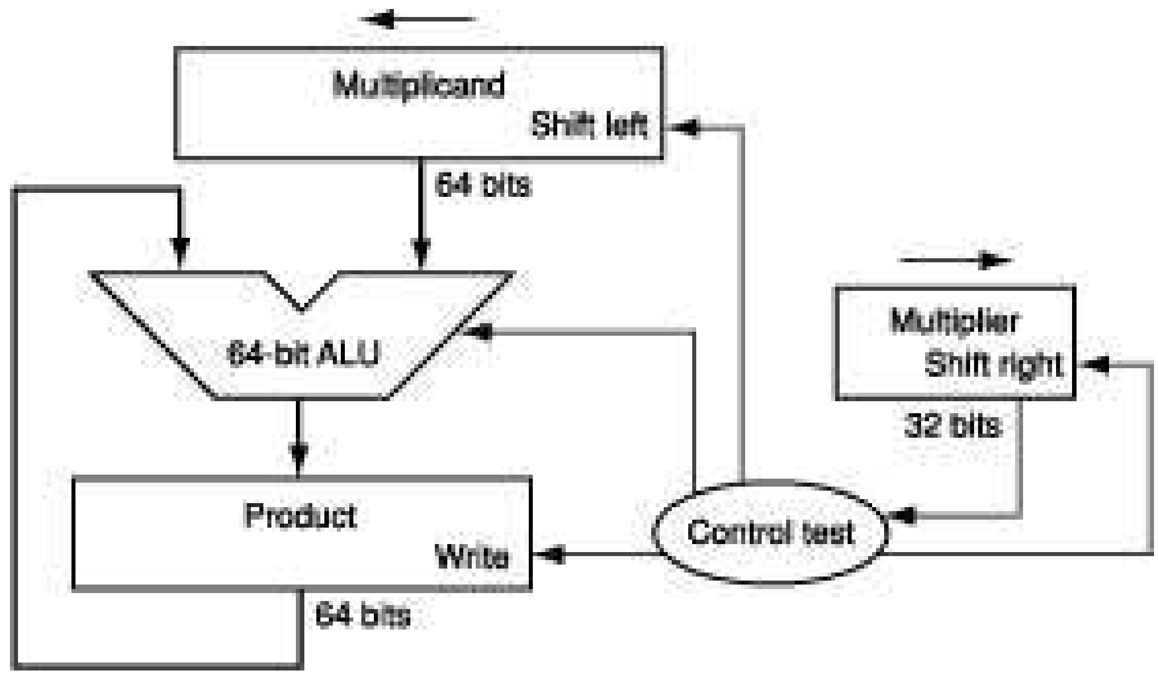
	1000
	0000
	0000
	1000

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

HW Algorithm



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

Quotient: 1001_{ten}

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

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

Quotient 1001_{ten}, Divisor 1000_{ten}, Dividend 1001010_{ten}, Remainder 10_{ten}

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

Quotient: 1001_{ten}

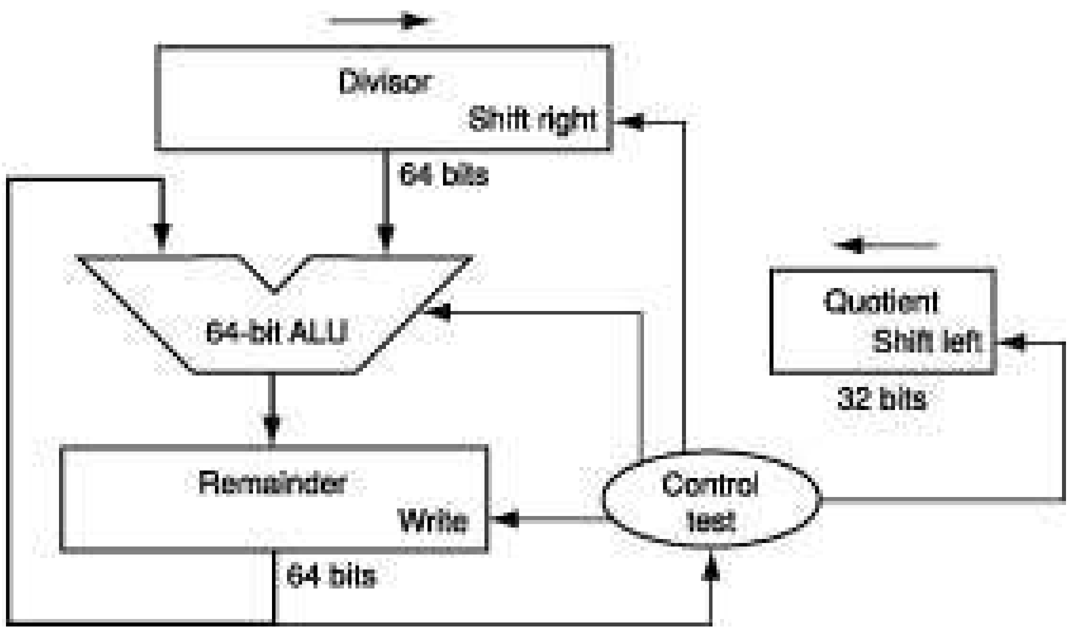
Divisor 1000_{ten} | 1001010_{ten} Dividend

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

Hardware for Division



A comparison requires a subtract; the sign of the result is examined; if the result is negative, the divisor must be added back

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

- 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

- Consider the following decimal example (can maintain only 4 decimal digits and 2 exponent digits)

$9.999 \times 10^1 + 1.610 \times 10^{-1}$

Convert to the larger exponent:

$9.999 \times 10^1 + 0.016 \times 10^1$

Add

10.015×10^1

Normalize

1.0015×10^2

Check for overflow/underflow

Round

1.002×10^2

Re-normalize

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

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 $a_i + b_i = 1$

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

  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

Title

- Bullet