Lecture 27: Review Session

- Disk, reliability wrap-up
- Review Session

- Exam reminders / Thursday
- Class evals, TAs

Role of Disks

- Activities external to the CPU/memory are typically orders of magnitude slower
- Example: while CPU performance has improved by 50% per year, disk latencies have improved by 10% every year
- Typical strategy on I/O: switch contexts and work on something else
- Other metrics, such as bandwidth, reliability, availability, and capacity, often receive more attention than performance

Magnetic Disks

- A magnetic disk consists of 1-12 *platters* (metal or glass disk covered with magnetic recording material on both sides), with diameters between 1-3.5 inches

- Each platter is comprised of concentric *tracks* (5-30K) and each track is divided into *sectors* (100 – 500 per track, each about 512 bytes)

- A movable arm holds the read/write heads for each disk surface and moves them all in tandem – a *cylinder* of data is accessible at a time

Disk Latency

- To read/write data, the arm has to be placed on the correct track – this *seek time* usually takes 5 to 12 ms on average – can take less if there is spatial locality

- *Rotational latency* is the time taken to rotate the correct sector under the head – average is typically more than 2 ms (15,000 RPM)

- *Transfer time* is the time taken to transfer a block of bits out of the disk and is typically 3 – 65 MB/second

- A disk controller maintains a disk cache (spatial locality can be exploited) and sets up the transfer on the bus (*controller overhead*)

Defining Reliability and Availability

- A system toggles between
  - Service accomplishment: service matches specifications
  - Service interruption: service deviates from specs

- The toggle is caused by *failures* and *restorations*

- Reliability measures continuous service accomplishment and is usually expressed as mean time to failure (MTTF)

- Availability measures fraction of time that service matches specifications, expressed as MTTF / (MTTF + MTTR)

RAID

- Reliability and availability are important metrics for disks
- RAID: redundant array of inexpensive (independent) disks
- Redundancy can deal with one or more failures
- Each sector of a disk records check information (CRC) that allows it to determine if the disk has an error or not (in other words, redundancy already exists within a disk)
- When the disk read flags an error, we turn elsewhere for correct data

RAID 0 and RAID 1

- RAID 0 has no additional redundancy (misnomer) – it uses an array of disks and stripes (interleaves) data across the arrays to improve parallelism and throughput
- RAID 1 mirrors or shadows every disk – every write happens to two disks
- Reads to the mirror may happen only when the primary disk fails – or, you may try to read both together and the quicker response is accepted
- Expensive solution: high reliability at twice the cost

RAID 3

- Data is bit-interleaved across several disks and a separate disk maintains parity information for a set of bits
- For example: with 8 disks, bit 0 is in disk-0, bit 1 is in disk-1, …, bit 7 is in disk-7; disk-8 maintains parity for all 8 bits
- For any read, 8 disks must be accessed (as we usually read more than a byte at a time) and for any write, 9 disks must be accessed as parity has to be re-calculated
- High throughput for a single request, low cost for redundancy (overhead: 12.5%), low task-level parallelism

RAID 4 and RAID 5

- Data is block interleaved – this allows us to get all our data from a single disk on a read – in case of a disk error, read all 9 disks

- Block interleaving reduces thruput for a single request (as only a single disk drive is servicing the request), but improves task-level parallelism as other disk drives are free to service other requests

- On a write, we access the disk that stores the data and the parity disk – parity information can be updated simply by checking if the new data differs from the old data

RAID 5

- If we have a single disk for parity, multiple writes can not happen in parallel (as all writes must update parity info)
- RAID 5 distributes the parity block to allow simultaneous writes

RAID Summary

- RAID 1-5 can tolerate a single fault – mirroring (RAID 1) has a 100% overhead, while parity (RAID 3, 4, 5) has modest overhead

- Can tolerate multiple faults by having multiple check functions – each additional check can cost an additional disk (RAID 6)

- RAID 6 and RAID 2 (memory-style ECC) are not commercially employed

Memory Protection

- Most common approach: SECDED – single error correction, double error detection – an 8-bit code for every 64-bit word -- can correct a single error in any 64-bit word – also used in caches

- Extends a 64-bit memory channel to a 72-bit channel and requires ECC DIMMs (e.g., a word is fetched from 9 chips instead of 8)

- Chipkill is a form of error protection where failures in an entire memory chip can be corrected

Computation Errors

- Errors in ALUs and cores are typically handled by performing the computation n times and voting for the correct answer
- n=3 is common and is referred to as triple modular redundancy

3810 Review Session

Spring 2025

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

Example Problem

- A 1 GHz processor takes 100 seconds to execute a CPU-bound program, while consuming 70 W of dynamic power and 30 W of leakage power. Does the program consume less energy in Turbo boost mode when the frequency is increased to 1.2 GHz?

Normal mode energy = 100 W x 100 s = 10,000 J
Turbo mode energy = (70 x 1.2 + 30) x 100/1.2 = 9,500 J

Note:
Frequency only impacts dynamic power, not leakage power.
We assume that the program's CPI is unchanged when frequency is changed, i.e., exec time varies linearly with cycle time for CPU-bound programs.

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;
```

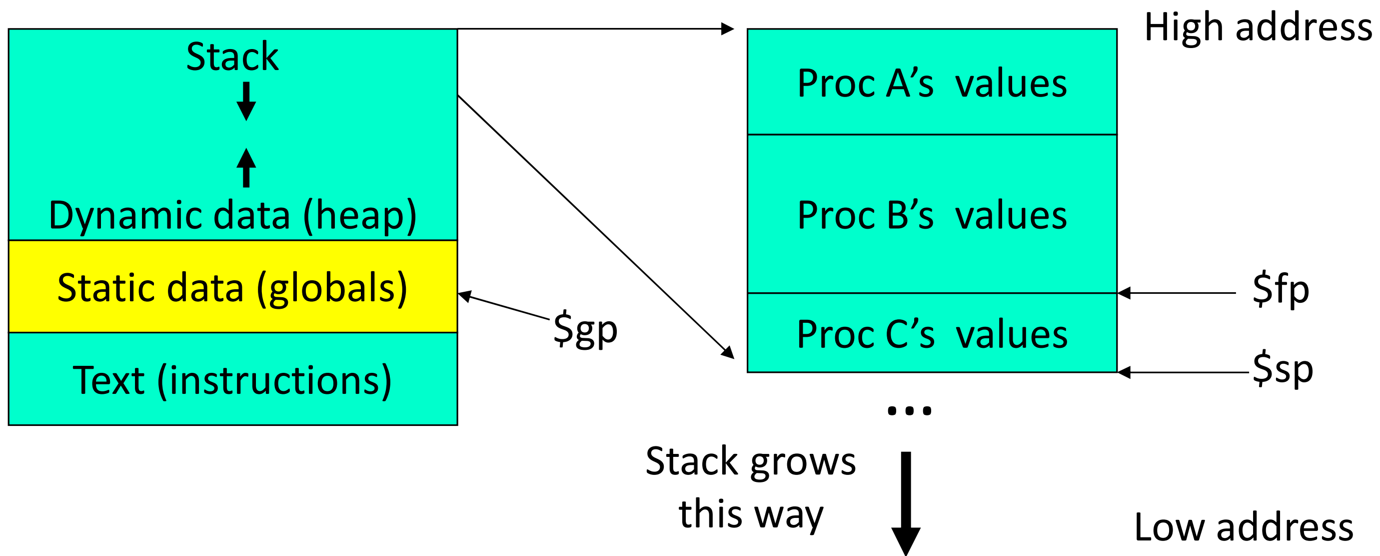
i and k are in $s3 and $s5 and base of array save[] is in $s6

```
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));
}
```

```
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

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

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

```
                                 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

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

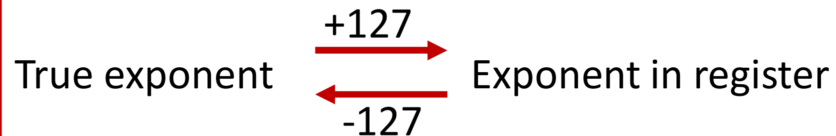
Final representation: $(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$

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

Single: (1 + 8 + 23)

Double: (1 + 11 + 52)

Remember:



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

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)
  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

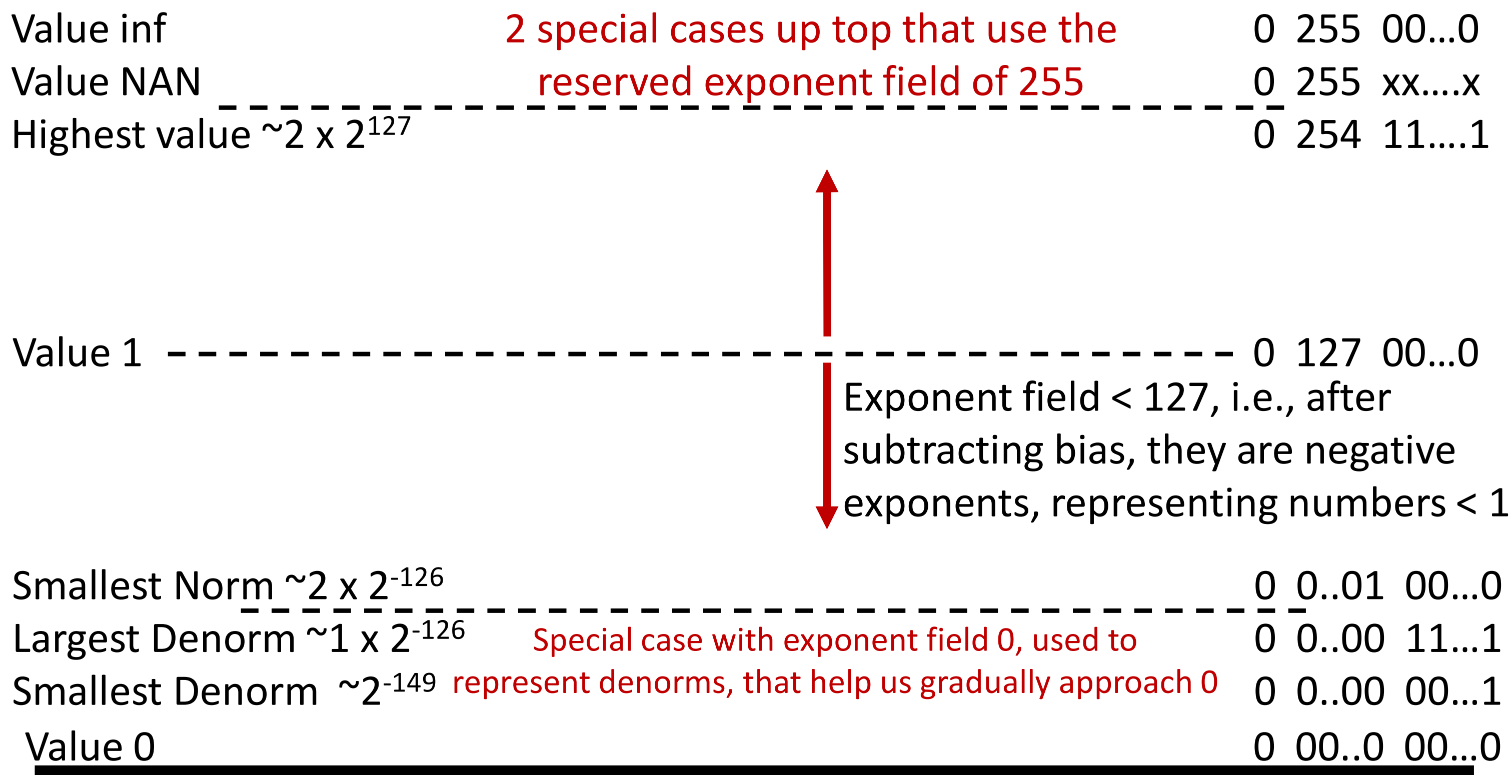
Example 2

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

We've calculated that 36.90625_{ten} = 100100.1110100…0 in binary
Normalized form = $1.001001110100…0 \times 2^5$
(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 ~2 x 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 ~2 x 2^-126
0 0..01 00…0
Largest Denorm ~1 x 2^-126
0 0..00 11…1
Special case with exponent field 0, used to represent denorms, that help us gradually approach 0
Smallest Denorm ~2^-149
0 0..00 00…1
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 $a_i + b_i = 1$

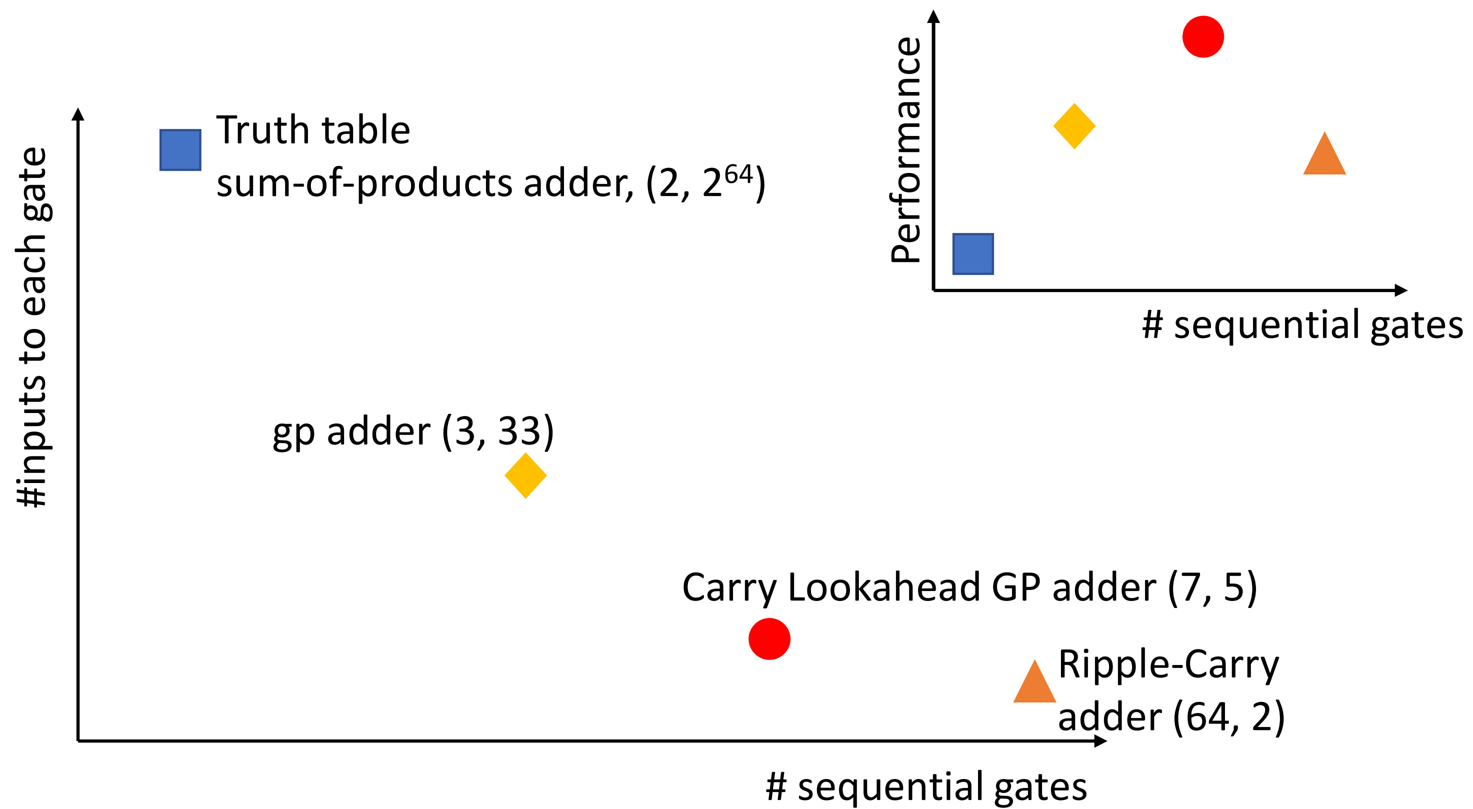
$C_{i+1} = g_i + p_i \, . \, C_i$

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:

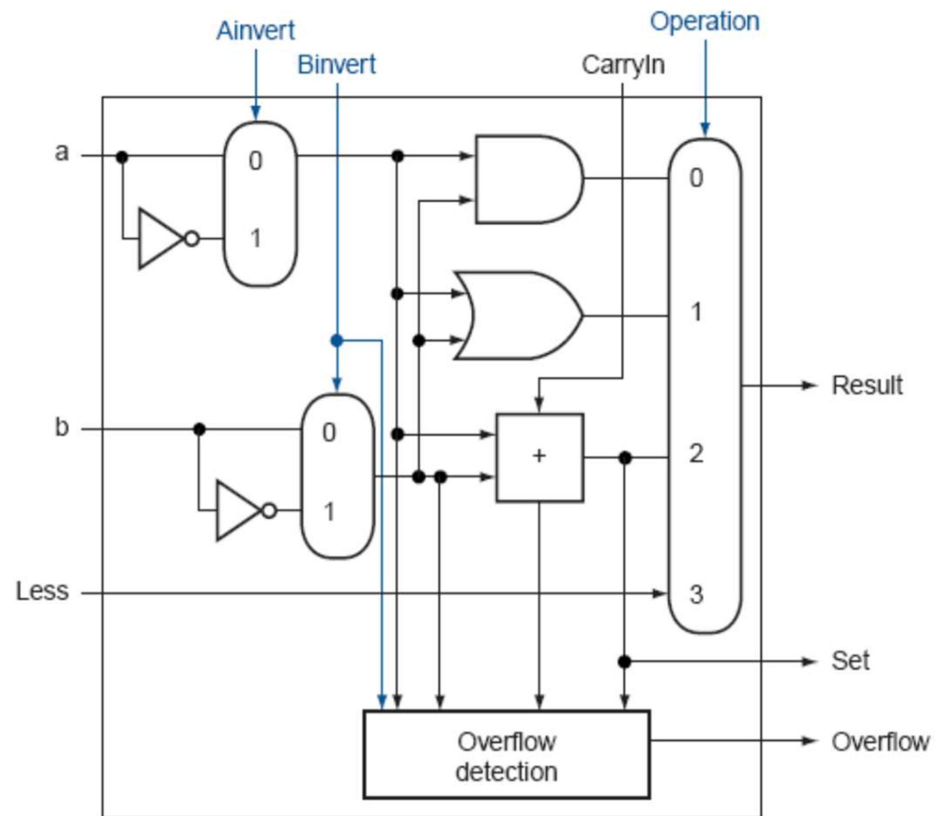
$C_4 = G_3 + G_2.P_3 + G_1.P_2.P_3 + G_0.P_1.P_2.P_3 + C_0.P_0.P_1.P_2.P_3$

Each of the sub-terms is also a similar expression

Trade-Off Curve



32-bit ALU

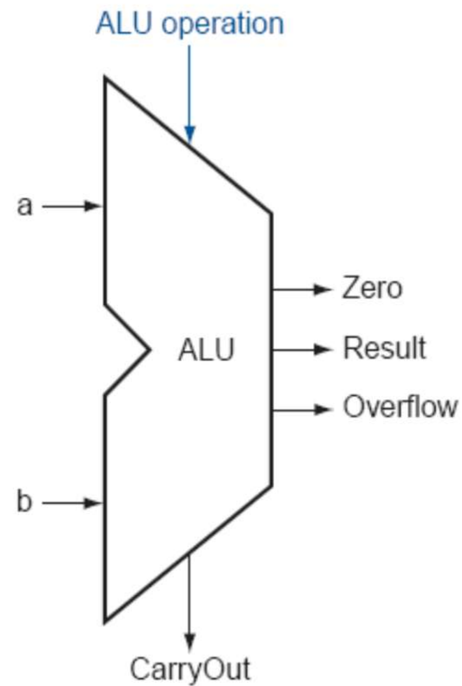


Source: H&P textbook

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

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.

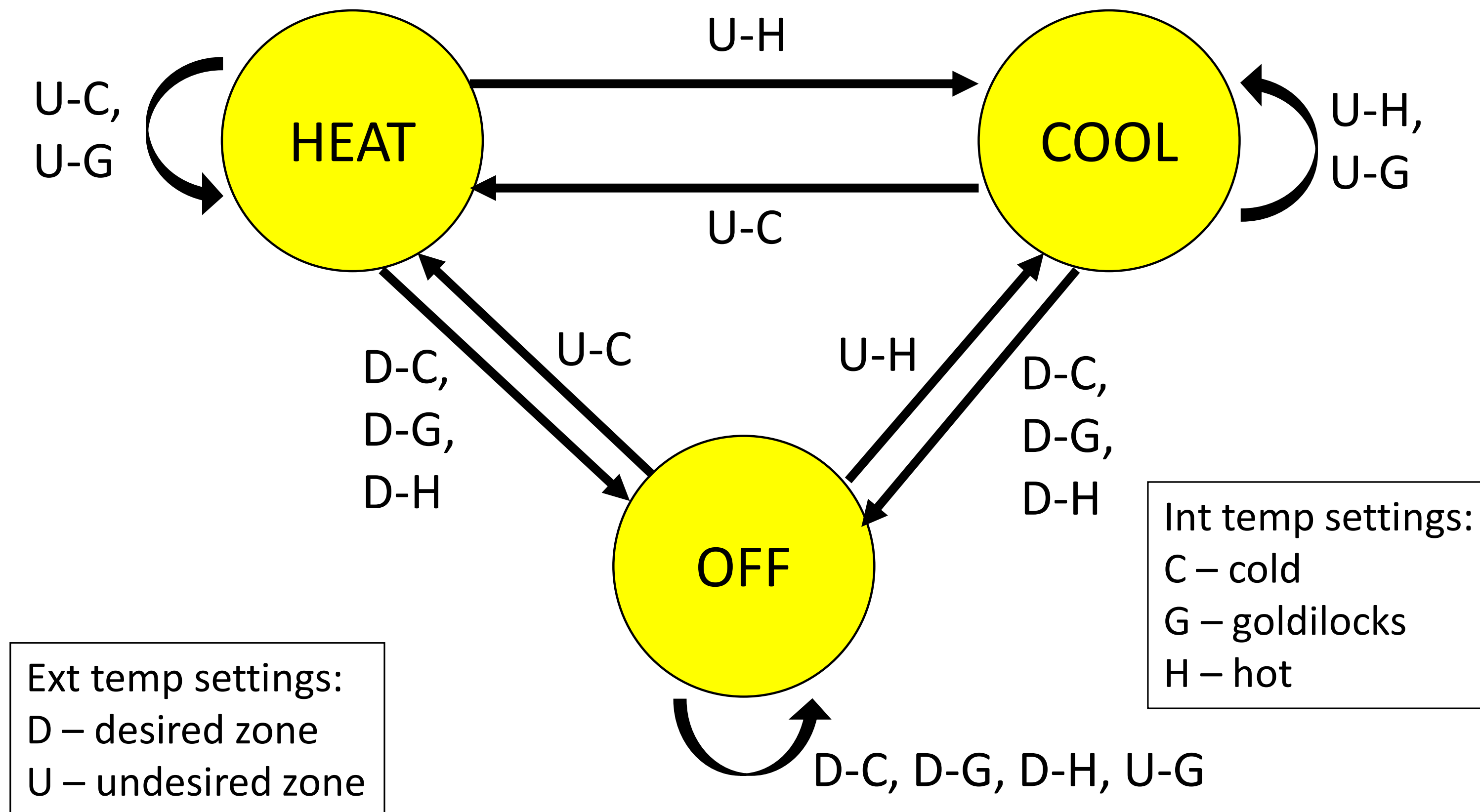
Example – Residential Thermostat

- 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



Pipeline Performance

Unpipelined processor
CPI:
Clock speed:
Throughput:

Pipelined processor
CPI:
Clock speed:
Throughput:

Circuit Assumptions
Length of full circuit:
Length of each stage:
No hazards

No Bypassing
(for the 5-stage pipeline)
Point of production: always RW middle
Point of consumption: always D/R middle

```
                                        * PoP
I1  add:    IF   DR    AL    DM    RW
I2  add:         IF    DR    DR    DR    AL  DM  RW
                                        * PoC
```

Bypassing

Point of production:
  add, sub, etc.: end of ALU
  lw: end of DM

Point of consumption:
  add, sub, lw: start of ALU
  sw $1, 8($2): start of ALU for $2,
                start of DM for $1

```
                                  * PoP
I1  add:    IF   DR    AL    DM    RW
I2  add:         IF    DR    AL    DM  RW
                             * PoC
```

Data Hazards

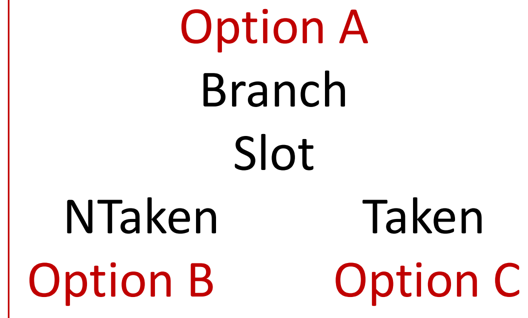
Assumptions

100 instructions
20 branches
14 Not-Taken, 6 Taken
Branch resolved in 6th cycle (penalty of 5)

Approach 3: Branch Delay Slot

Option A: always useful
Option B: useful when the branch goes along common fork
Option C: useful when the branch goes along uncommon fork
Option D: no-op, always non-useful

Approach 1: Panic and wait

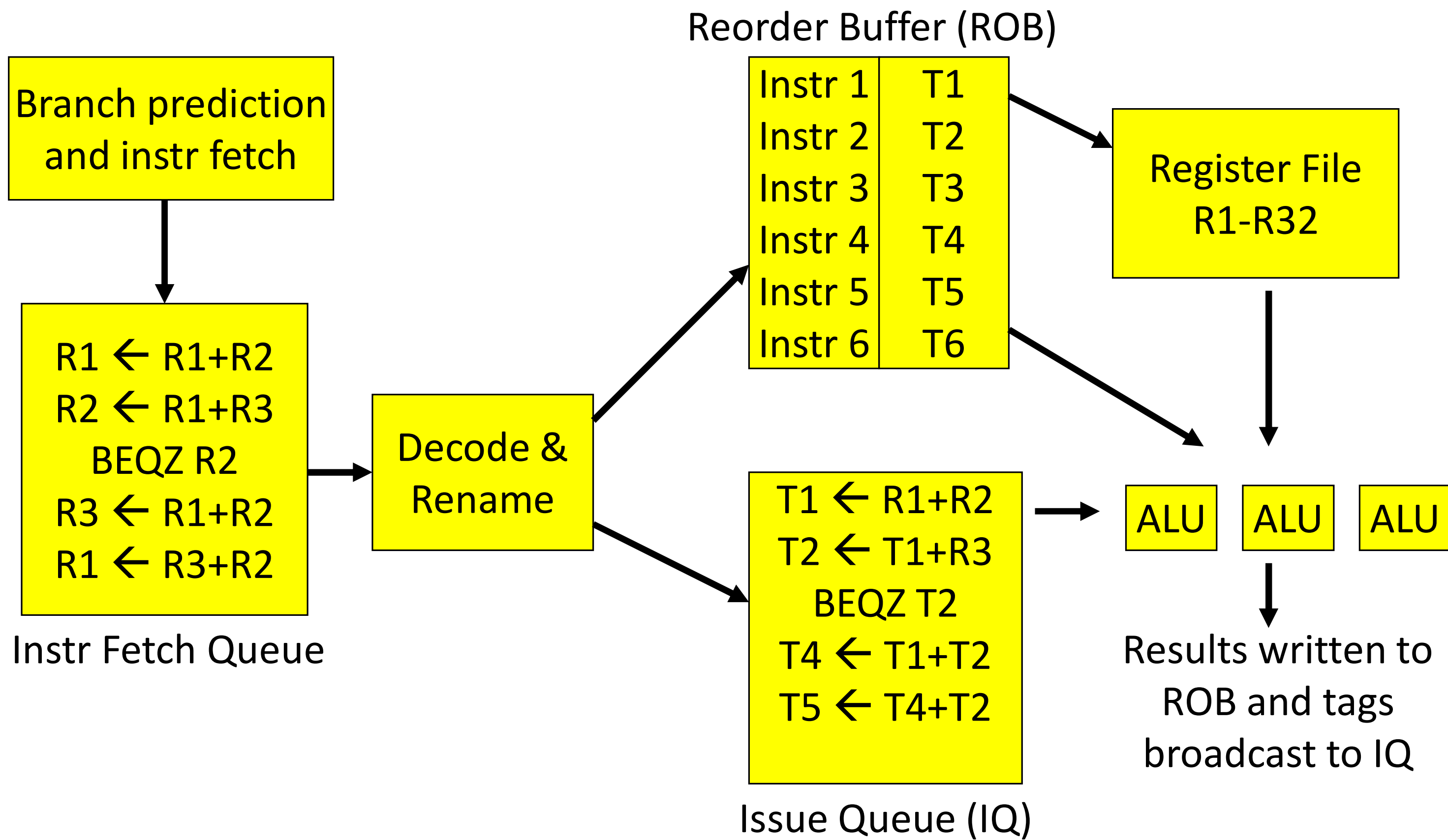


Approach 2: Fetch-next-instr

Approach 4: Branch predictor

Accuracy of 90%

Control Hazards



Out of Order Processor

Assumptions

1000 instructions, 1000 cycles, no stalls with L1 hits
loads/stores:
% of loads/stores that show up at L2:
% of loads/stores that show up at L3:
% of loads/stores that show up at mem:
L2 acc = 10 cyc,   L3 acc = 25 cyc,   mem acc = 200 cyc

Cache Latency

Assumptions

512KB cache, 8-way set-associative, 64-byte blocks, 32-bit addresses

Data array size = #sets x #ways x blocksize
Tag array size = #sets x #ways x tagsize
Offset bits = log(blocksize)
Index bits = log(#sets)
Tag bits + index bits + offset bits = addresswidth

Cache Size

Assumptions

16 sets, 1 way, 32-byte blocks

Access pattern: 4 40 400 480 512 520 1032 1540

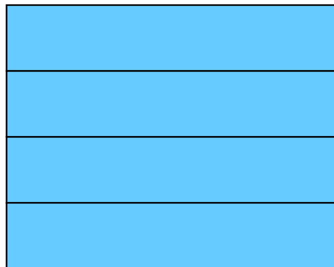
Offset = address % 32 (address modulo 32, extract last 5)
Index = address/32 % 16 (shift right by 5, extract last 4)
Tag = address/512 (shift address right by 9)

	32-bit address				
	23 bits tag	4 bits index	5 bits offset	H/M	Evicted address
4:	0	0	4	M	Inv
40:	0	1	8	M	Inv
400:	0	12	16	M	Inv
480:	0	15	0	M	Inv
512:	1	0	0	M	0
520:	1	0	8	H	-
1032:	2	0	8	M	512
1540:	3	0	4	M	1024

Cache Hits/Misses

Example 0b

Show how the following addresses map to the cache and yield hits or misses.
The cache is direct-mapped, has 16 sets, and a 64-byte block size.
Addresses: 8, 96, 32, 480, 976, 1040, 1096



.
.
.

Offset = address % 64 (address modulo 64, extract last 6)
Index = address/64 % 16 (shift right by 6, extract last 4)
Tag = address/1024 (shift address right by 10)

32-bit address

	22 bits tag	4 bits index	6 bits offset	
8:	0	0	8	M
96:	0	1	32	M
32:	0	0	32	H
480:	0	7	32	M
976:	0	15	16	M
1040:	1	0	16	M
1096:	1	1	8	M

Consider a 3-processor multiprocessor connected with a shared bus that has the following properties:

(i) centralized shared memory accessible with the bus, (ii) snooping-based MSI cache coherence protocol, (iii) write-invalidate policy. Also assume that the caches have a writeback policy. Initially, the caches all have invalid data. The processors issue the following three requests, one after the other. Similar to slide 17 of lecture 25, fill in the following table to indicate what happens for every request. Also indicate if/when memory writeback is performed. (8 points)

P2: Read X
P1: Read X
P2: Write X
P3: Read X

Request	Cache Hit/Miss	Request on the bus	Who responds	State in Cache 1	State in Cache 2	State in Cache 3	State in Cache 4
				Inv	Inv	Inv	Inv
P2: Rd X							
P1: Rd X							
P2: Wr X							
P3: Rd X							

Questions to ask yourself:

How does Meltdown work?

How does Spectre work?

How can you force a footprint? (the relevant code sequence)

How can you examine footprints? (exploiting the side channel)

How can you defend against these attacks?

Security

Questions to ask yourself:
What does the programmer/compiler deal with?
What does the OS deal with?
How is translation done efficiently?

Virtual Memory

Questions to ask yourself:

Why do multiprocs need to deal with prog. models, coherence, synchronization, consistency?

What are race conditions?

What is an example synchronization primitive and how is it implemented?

What consistency model is assumed by a programmer?

Why is it slow?

How do I make life easier for the programmer and provide high performance?

Synchronization, Consistency

Questions to ask yourself:

What are the central philosophies in a GPU?

In what ways does the GPU design differ from a CPU?

What are the different ways that disks provide high reliability?

Can you explain how parity is used to recover lost data?

GPUs, Disks

Disk Basics

- Disk access remains very slow – mechanical head that has to move to the correct "ring" of data – order of milli-seconds – high enough that a context-switch is best
- Focus on other metrics, especially reliability
- A sector on the disk is associated with a cyclic redundancy code (CRC) – a hash that tells us if the read data is correct or not – it is simply an error detector, not an error corrector
- To correct the error, RAID is commonly used
- Reliability measures continuous service accomplishment and is usually expressed as mean time to failure (MTTF)
- Availability is measured as MTTF/(MTTF+MTTRecovery)

RAID

- RAID 0: no redundancy
- RAID 1: mirroring
- RAID 2 and 6: memory-style ECC and rarely deployed
- RAID 3: bit-interleaved, lower cost, but no query-level parallelism
- RAID 4: block-interleaved, lower cost, query-level parallelism, but write bottleneck
- RAID 5: block-interleaved, lower cost, query-level parallelism, write parallelism
- Parity and XOR!