Program: static set of statements to perform computational steps of an algorithm Thread:embeds the execution of these steps

Why can't we keep increasing clock rates? 1We are reaching practical limits of pipeline depth 2Wire delays are dominating clock cycles 3Faster clocks more power consumption

•Latency(response time) –time (sec) required to access (find) data in specified location •Bandwidth (BW)(throughput) –transmission rate of bits/bytes (data) Common units: bits/sec (bps) •Access time: latency + (block size)/bandwidth

--

Temporal locality:the property of most programs that if a given memory location is referenced, it is likely to be referenced again, "soon."
Spatial locality:if a given memory location is referenced, those locations near it numerically are likely to be referenced "soon."

--

Vector processors have high-level operations that work on linear arrays of numbers: "vectors"
Fault-Tolerance: the system can provide services even in the presence of faults

CPU Time Texe $Texe=(IC)(CPI)(Tc)$ •IC = Instruction Count # of instructions in program being executed • CPI = Cycles Per Instruction Average # of cycles required to process each instruction •Tc= clock cycle time Amount of time required to execute one clock cycle

$$Speedup=\frac{Exec\_Time_{old}}{Exec\_Time_{new}}$$

$$Speedup=\frac{1}{(1-F)+\frac{F}{Speedup_{Enhanced}}}$$

Cap stores charge Q = $C_{out}V_{DD}$ = CV

Energy stored is $\frac{1}{2}CV^2$

SCALING DYNAMIC POWER

- $P_{dynamic} = \alpha CV_{DD}^2 f$
- C and V_{DD} scale 1/S and f scales S. Hence $P_{dynamic}$ scales like $1/S^2$
- Number of transistor in unit area grow S^2
- Hence power density (power/area) stays constant with scaling

- Power = Energy / time
- So $P = \left[\frac{1}{2}CV^2\right]/T$
- Let α = *activity factor*
- Frequency f = 1/T
- Then, dynamic power equation is
  - $P_{dyn} = \alpha CV_{DD}^2 f$

Dynamic power :

Energy is good metric in battery constrained environments

- Task executed at ½ speed but ¼ power means ½ the energy (2T * ¼ P = ½ E)
- 2X battery life!

Real FET's – small leakage current exists

– I_{DDQ} = **quiescent leakage current**

So $P_{DC} = V_{DD}I_{DDQ}$

Usually, I_{DDQ} is very small (~ 1 pA per gate) so

A system **fails** when it cannot meet its promises (specifications)

An **error** is part of a system state that may lead to a failure

A **fault** is the cause of the error

Module reliability

- Mean time to failure (MTTF)
- Mean time to repair (MTTR)
- Mean time between failures (MTBF) = MTTF + MTTR
- Availability = MTTF / MTBF

Failures in Time (FIT)

- Rate of failures per *billion* hours
- MTTF = 10^9/FIT

R(t) = probability that component *survives* up to time t

$R(t) = N_s(t)/N$, where

- $N_s(t)$ = number of components that survived up to time t
- N = total number of components

Q(t) = probability that component *fails* up to time t = 1 – R(t)

MIPS is Big Endian

Big Endian

Byte Address	0	1	2	3
Data	AB	CD	12	34

Little Endian

Byte Address	0	1	2	3
Data	34	12	CD	AB

We are given a code sequence that will have 4 data bits (original information), with 3 check bits added, to form the following 7-bit sequence: $a_1a_2a_3a_4a_5a_6a_7$

- Bits a_1, a_2, a_4 are the *check bits* (powers of 2)
- The other bits are the *data bits*

$a_1a_2a_3a_4a_5a_6a_7$

The check bit is set so that the total number of 1's in the bit sequence is **even** (even parity):

$a_1 = a_3 \oplus a_5 \oplus a_7$
$a_2 = a_3 \oplus a_6 \oplus a_7$
$a_4 = a_5 \oplus a_6 \oplus a_7$

If we send data bits 1101 ($a_3a_5a_6a_7$), then the check bits are

$a_1 = 1 \oplus 1 \oplus 1 = 1$
$a_2 = 1 \oplus 0 \oplus 1 = 0$
$a_4 = 1 \oplus 0 \oplus 1 = 0$

So transmitted message is 1010101

When message is received, the check bits are examined as follows:

$e_1 = a_1 \oplus a_3 \oplus a_5 \oplus a_7$
$e_2 = a_2 \oplus a_3 \oplus a_6 \oplus a_7$
$e_4 = a_4 \oplus a_5 \oplus a_6 \oplus a_7$

The bit that is in error is found by $4e_4 + 2e_2 + e_1$

Assume that we receive our previous message as 1011101

$e_1 = 1 \oplus 1 \oplus 1 \oplus 1 = 0$
$e_2 = 0 \oplus 1 \oplus 0 \oplus 1 = 0$
$e_4 = 1 \oplus 1 \oplus 0 \oplus 1 = 1$

bit = 4 + 0 + 0 = 4, so bit 4 is incorrect – change from 1 to 0

Character (8 bits)Half word (16 bits)Word (32 bits)Single-precision floating point (32 bits)Double-precision floating point (64 bits)
Memory size 210= 1 K (kilo) 2^20= 1 M (mega) 2^30= 1 G (giga) 2^40= 1 T (tera) 2^50 = 1 P (peta) 1 B (byte) = 8 b (bits)
CPU word size = w Typical: w = 16 or 32 bits •Memory word size = s Typical: s = 8 bits (memory is byte-addressable)

C code:

```
f = (g + h) - (i + j);
```

– f, …, j in $s0, …, $s4

Compiled MIPS code:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

C code:

```
g = h + A[8];
```

– g in $s1, h in $s2, base address of A in $s3

Compiled MIPS code:

– Index 8 requires offset of 32
  • 4 bytes per word

```
lw  $t0, 32($s3)    # load word
add $s1, $s2, $t0
```

C code:

```
A[12] = h + A[8];
```

– h in $s2, base address of A in $s3

Compiled MIPS code:

– Index 8 requires offset of 32

```
lw  $t0, 32($s3)    # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)    # store word
```

Unsigned Binary Integers Given an n-bit number Range: 0 to +2^n – 1 Using 32 bits : 0 to +4,294,967,295
Signed Given an n-bit number Range: – 2n – 1to +2n – 1 – 1 Using 32 bits – 2,147,483,648 to +2,147,483,647
Some specific numbers 0:0000 0000 … 0000 –1:1111 1111 … 1111 Most-negative:1000 0000 … 0000 Most-positive:0111 1111 … 1111
Signed Negation Complement and add 1
Sign Extension +2: 0000 0010 => 0000 00000000 0010 –2: 1111 1110 => 1111 11111111 1110
Register numbers $t0 –$t7 are reg's 8 –15 $t8 –$t9 are reg's 24 –25 $s0 –$s7 are reg's 16 –23

MIPS R-format Instructions

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Instruction fields
- op: operation code (opcode)
- rs: first source register number
- rt: second source register number
- rd: destination register number
- shamt: shift amount (00000 for now)
- funct: function code (extends opcode)

R-format Example

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

`add $t0, $s1, $s2`

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$00000100011001001000000000100000_2 = 02324020_{16}$

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

Example: eca8 6420
- 1110 1100 1010 1000 0110 0100 0010 0000

MIPS I-format Instructions

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

Immediate arithmetic and load/store instructions
- rt: destination or source register number
- Constant: -2^{15} to $+2^{15} - 1$
- Address: offset added to base address in rs

Shift Operations

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

shamt: how many positions to shift

Shift left logical
- Shift left and fill with 0 bits
- `sll` by *i* bits multiplies by 2^i

Shift right logical
- Shift right and fill with 0 bits
- `srl` by *i* bits divides by 2^i (unsigned only)

NOT Operations
- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0
- MIPS has NOR 3-operand instruction
  - a NOR b == NOT (a OR b)

`nor $t0, $t1, $zero` ← Register 0: always read as zero

$t1 0000 0000 0000 0000 0011 1100 0000 0000

$t0 1111 1111 1111 1111 1100 0011 1111 1111

Procedure Calling

Conditional Operations

Branch to a labeled instruction if a condition is true
- Otherwise, continue sequentially

`beq rs, rt, L1`
- if (rs == rt) branch to instruction labeled L1;

`bne rs, rt, L1`
- if (rs != rt) branch to instruction labeled L1;

`j L1`
- unconditional jump to instruction labeled L1

Steps required
1. Place parameters in registers
2. Transfer control to procedure
3. Acquire storage for procedure
4. Perform procedure's operations
5. Place result in register for caller
6. Return to place of call

Register Usage
- $a0 – $a3: arguments (reg's 4 – 7)
- $v0, $v1: result values (reg's 2 and 3)
- $t0 – $t9: temporaries
  - Can be overwritten by callee
- $s0 – $s7: saved
  - Must be saved/restored by callee
- $gp: global pointer for static data (reg 28)
- $sp: stack pointer (reg 29)
- $fp: frame pointer (reg 30)
- $ra: return address (reg 31)

PCSrc : 0 : PC <= PC + 4 1 : PC = PC + 4 + offset
RegWrite: 0 none 1: R[Write register] <= Write data
ALUSrc : 0 : 2nd Alu <= Read data 2 else Sign-extend constant
MemWrite : 0: none else Mem[address] <= Write Data
MemRead : 0: none else read Data <= Mem[address]
MemtoReg : 1 Write data <= Read data else Write data <= Alu result

lw $t1,OFFSET($t2)
Read Register 1 rs = $t2
Write Register rt = $t1
Control signal RegDst= 0result stored in register specified by rt
ALUSrc = 1 second ALU input is Sign-Extended address from instruction
ALU Operation = 0010add
ALU Result = Read Data 1 + sign-extended OFFSET value
MemRead = 1Read data from Data Memory
MemtoReg= 1ALU Result becomes Data Memory address. Read Data passes back to Write Data in registers.
RegWrite= 1write ALU Result to register specified by rt

beq $t1, $t2, OFFSET Read Register 1 rs = $t1 Read Register 2 rt = $t2
ALUSrc = 0second ALU input is Read Register 2
ALU Operation = 0110sub
Add ALU gets PC + Sign-extended OFFSET shifted left by 2
If R[t1] = R[t2], ALU Zero = 1 and PCSrc = 1
PC Add ALU result
Else,PCSrc = 0 PC PC + 4

Recall that execution time = (instruction count)(CPI)(clock cycle time)

Also, clock rate = 1/(clock cycle time

We are given clock rate = 2 GHz = 2 x 10^9 Hz = 2 x 10^9 cycles/sec

CPI (cycles/instruction) = 2

Instruction count (IC) = 10^9

Thus

$$Execution\ time = \frac{(IC)(CPI)}{clock\ rate} = \frac{(10^9)(2)}{2 \times 10^9} = 1\ sec$$

It takes 5 days = 5 (24 hours/day) = 120 hours to get the system running again = MTTR

$$Availability = \frac{MTTF}{MTTF + MTTR} = \frac{10^6}{10^6 + 120} = 0.999880$$

$$P_{dynamic} = \alpha C V_{dd}^2 f = (0.25)(200 \times 10^{-15})(5)^2(4 \times 10^9) = (0.25)(4)(200)(25)(10^{-15})(10^9)$$
$$= (200)(25)(10^{-6}) = (5 \times 10^3)(10^{-6}) = 5 \times 10^{-3} = \mathbf{5\ mW}$$

bne instruction is located at address 100CH in memory. After it is fetched, the program counter is incremented by 4 (PC = 100CH + 4H = 1010H)
Target address = PC + 4(offset) Target address = 1000H (address of LOOP label)
PC = 1010H So, offset = (target address – PC) / 4 = (1000H – 1010H)/4 = -16/4 = **-4**