INTRODUCTION

Computer systems use technology to simulate the human world

<table>
<thead>
<tr>
<th>Human thought processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>High level languages</td>
</tr>
<tr>
<td>Assembly language</td>
</tr>
<tr>
<td>Machine code</td>
</tr>
<tr>
<td>011000001010111</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>native data types</td>
</tr>
<tr>
<td>instruction set</td>
</tr>
<tr>
<td>registers</td>
</tr>
<tr>
<td>ISA</td>
</tr>
<tr>
<td>exception handling</td>
</tr>
<tr>
<td>I/O handling</td>
</tr>
<tr>
<td>addressing modes</td>
</tr>
<tr>
<td>cycles per instruction</td>
</tr>
<tr>
<td>physical registers</td>
</tr>
<tr>
<td>Gates</td>
</tr>
<tr>
<td>Transistors</td>
</tr>
</tbody>
</table>
A GENERIC COMPUTER

MEMORY HIERARCHY

MEMORY HIERARCHY

The IC Industry

Very large market
Very few products
High rate of development
Long development times
Multiple generations in simultaneous development
Discontinuous technological change

Producing the Wafers
The MIPS Computer

a popular microprocessor
(a billion sold?)
RISC architecture

Architecture

Machine Language

Instruction Set

Compilers

Design Goal

The Add Instruction

The MIPS computer has a 3-address architecture

```
add a, b, c  # a = b + c
sub a, b, c  # a = b - c
add a, b, c  # a = b + c
add a, a, d  # a = a + d
add a, a, e  # a = a + e
# a contains the sum of b, c, d, & e

move $8, $19  # r8 ← r19  - desired behaviour
add $8, $0, $19  # r8 ← 0 + $19  - actual implementation
```
The ISA

Expression Trees And Evaluation Order

```
+   +
  a  c  
  +  +
  e  b  d
     +
    a  
    +
    f  

Register Name(s) Use
0  $zero
1
2-3  $v0-$v1
4-7  $a0-$a3
8-15  $t0-$t17
16-23  $s0-$s7
24-25  $t8-$t19
26-27
28  $gp
29  $sp
30  $fp
31  $ra
```

The Registers

$0, $1, ..., $31

address calculation, stack pointers as well as data storage

RISC Designs Favour Simplicity

32-bit instructions
two types: R(egister) –type & I(mmediate)-type

<table>
<thead>
<tr>
<th>R-type</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 b</td>
<td>5 b</td>
<td>5 b</td>
<td>5 b</td>
<td>5 b</td>
<td>5 b</td>
<td>5 b</td>
</tr>
<tr>
<td>6 b</td>
<td>5 b</td>
<td>5 b</td>
<td>5 b</td>
<td>5 b</td>
<td>5 b</td>
<td>5 b</td>
</tr>
</tbody>
</table>
RISC Designs Favour Simplicity

32-bit instructions
two types: R(register) & I(immediate)-type

<table>
<thead>
<tr>
<th>R-type</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-type</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>constant or address</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

have a constant operand or access memory

(Yeah, right)

let's say x is in register called $t0 in the assembler, actual reg. 8
h is in register called $s2 in the assembler, actual reg.18

array a starts at the location contained in reg. $s3, actual reg.19

x = h + a[8]

load $t0 with the word in $s2 plus the word 8 up from the address in $s3
lw $t0, 32($s3)
add $t0, $s2, $t0

(psst; there's a third type as well; J-type, for jump instructions)

regularity is an ideal, but good compromises must sometimes be made

COMPILATION

f = (g+h) – (i+j)

name register

expression tree

variable-register associations

+ $t0 + $t1

$19 contains start, then

lw $8, 12($19)
MORE ABOUT MEMORY

Compiler also allocates

\[ 2^{12} \text{ bytes} = 4,294,967,296 \]
\[ 2^{30} \text{ words} = 1,073,741,824 \]

Large address space means access times

Compiler tries to keeps spills

BRANCHES

Computers must have mostly, that's just that's why

sometimes it needs to that's why the PC is the choice depends on

beq $1, $2, L1  # branch to L1 if $1 and $2 are equal

PC is loaded with
PC is incremented

bne $1, $2, L1  # branch to L1 if $1 and $2 not equal

TESTS USING INEQUALITY

SLT (Set on Less Than)
compares
SLT $r1, $r2, $r3

C Equivalent:

BLT (Branch on Less Than)
would need simpler & more regular to use

SLTUI (Set on Less Than Unsigned Immediate)

SLTU (Set on Less Than Unsigned)

C Equivalent:
### IF Statements

if (a == b)  
c = d + e;  
else  
c = d - e

**Register allocations**  
c d e a b  
$16$ $17$ $18$ $19$ $20$

### Loops

while loop  
while (this[i] == k)  
i = i + j

**Register allocations**  
i j k 4 (constant)  
$19$, $20$, $21$ $10$

### Subroutine Calls

Another variety of saves  
jal procAddress  

jr $31$

For nested procedure calls,  
stack is spilled into memory  
$sp$ contains  
stack lives at top end of memory, & grows downwards  

Parameters passing uses registers  
for a nested subroutine call,  
caller save: called subroutine can then use any registers  
callee save: calling subroutine doesn’t have to restore registers

### Immediate Instructions To Operate On Constants

```
addi $29$, $29$, 4     # sp = sp - 1!
```

```
lui <reg>, <16-bit const>
```

```
lui     $8     255
00011100000001000000000011111111
```

```
addi $8$, $8$, 96
```

```
0010000100010000000000000110000
```

```
0000000011111111000000000110000 r8
```
**Design Principles**

Smaller is faster
- more registers → greater area → slower clock

Simplicity favours regularity
- decoding is faster with

Good design demands good compromise
- R-type, I–type, and J-type instructions are all

Make the common case fast
- immediate instructions don’t often involve big constants
  - so 16-bit constants are OK, with `lui` only needed occasionally

**Performance Metrics**

- **throughput**
  - total work accomplished in a given time

- **execution time**
  - time for a given job

  - performance (rate or speed) =

    \[
    \text{if } \text{performance}_X > \text{performance}_Y \text{ then}
    \]

**CPU Time, I/O Time, and Wall Clock Time**

CPU time is

\[
\text{access time} = \text{access times of CPU} + \text{access times of I/O} + \text{elapsed time}
\]

**Factors Influencing Performance**

- **hardware-related factors**
  - ISA implementation
  - CPU cycle time
  - bus cycle time
  - caching
  - parallelism
  - pipelining

- **software-related factors**
  - user algorithm
  - operating system
  - compilers
**Performance Measures; The Clock Cycle**

- **Clock Cycle**
  - E.g. 10ns

- **Clock Rate**
  - E.g. 4GHz

- No. of clock cycles per instruction is
  - CPI – Cycles Per Instruction – also a factor

- Time = No. of instructions $\times$ CPI

**Performance Components**

- CPU clock cycles = $\sum_{i=1}^{n} CPI_i \times$ No. of instructions

- Time = $\frac{\text{CPU clock cycles}}{\text{CPI}}$ x No. of instructions

- No one of these is a full measure of performance

**MFLOPS**

- Millions of Floating Point Operations per Second
- Much scientific, graphic and engineering computing involves fast floating point arithmetic
- Many computers have

  - The same caveat applies to MFLOPS as to MIPS

**Examples**

<table>
<thead>
<tr>
<th>Compiler</th>
<th>CPI</th>
<th>Instruction 1</th>
<th>Instruction 2</th>
<th>Instruction 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler 1</td>
<td>1</td>
<td>10</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>Compiler 2</td>
<td>2</td>
<td>10</td>
<td>2</td>
<td>15</td>
</tr>
</tbody>
</table>

**Computation**

- MIPS = $\frac{10 + 10 + 15}{3}$
BENCHMARKS

too many variables, too much hype

benchmarks are standard programs whose
e.g. Livermore loops

e.g. SPEC benchmark suites

manufacturers can to achieve good stats
gives an unrealistic impression of

PARALLELISM

some tasks can be
systems that can be divided into
e.g. the atmosphere (weather prediction)
ideally in practice,

PARALLELISM - AMDALH’S LAW

how do we measure if a task takes 100s in one configuration and 80s in another, what's the speedup?
speed₁ = 1 task/100s =
speed₂ = 1 task/80s =
speedup = 0.0125 / 0.01 =

most code is a mixture of serial processing component is & limits

T = max speedup → if tₚ can be reduced to 0

if we have P processors working perfectly in parallel then we reduce the time for the parallel section of the code by a factor of P so the total task time in the parallel configuration Tₚ = speedup S =

PARALLELISM - AMDALH’S LAW

parallelism has two flavours
independent tasks; dependant tasks;

many tasks in computers involve
cf. assembly line for producing goods that undergo same operation sequence
**Pipeline Performance**

![Diagram of pipeline](image)

- If each task in a pipeline of length $L$ takes $t$ seconds, a single task takes $t$.
- But for $n$ tasks:
  - Delay before 1st output =
  - Delay between subsequent outputs =
  - $T(n) = (L + n - 1)t$

**Pipeline**

**Pipeline Characteristics**

- Pipeline rate, $r_w =$
- Pipeline startup time, $s =$
- Half performance vector length, $n_{1/2}$

\[(L-1)t + n_{1/2}t = n_{1/2} = \]

**Numbers and the Datapath**

- Datapath
  - ALU
- Number representation: 2's complement
- Arithmetic operations: +, *, /, shift
- Used in nearly all microprocessors
- Most significant bit
- Other bits
**INT  E GERS A ND  ADD R E SSES**

Integers are... addresses are...

No type information included with the data.

Note: unsigned addresses are not used in all computers.

Transputer addresses are 2's complement, in the range $-2^{31}$ to $2^{31}-1$.

---

**2's Complement**

For $n$ bit numbers:

- $2^{n-1}$ positive numbers, starting at 0
- $2^{n-1}$ negative numbers, starting at $-1$

\[ \text{max -ve no} = -2^{n-1} \]

\[ x + -x = 0 \]

\[ -x = \text{two's complement of } x \]

Operation overflows:

- $-5$ 1011
- $-3$ 1101
- $-8$ 1000

---

**OVERFLOW DETECTION**

Easy to detect: when an overflow occurs.

Address of overflowing instruction saved.

Interrupt handler is called and

After interrupt code finishes, instruction...

---

**1-bit ALU**

Bit-slice architecture

Design an ALU component that handles

Logical operations require

We could extend this design by adding more functional blocks, e.g. multipliers.

Carry propagation, carry lookahead...
can we detect when less significant bit slices will generate a carry output?

set up modules that allow modules generally incorporate

conventional full adder involves we could design a 32-bit adder as 64 inputs \(2^{64}\) terms; too big is there a less expensive way?

1-bit ALU – Adding Carry Lookahead

Boolean expressions for

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>cin</th>
<th>cout</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Gi =
Pj =
carry input to the next phase:

\(c_{i+1} = \begin{cases} 0, & \text{carry kill} \\ 1, & \text{carry propagate} \end{cases} \)

\(c_{i} = \begin{cases} 0, & \text{carry propagate} \\ 1, & \text{carry generate} \end{cases} \)

all can be calculated in parallel from data inputs and \(c_0\)

1-bit ALU – Adding Carry Lookahead

carry lookahead circuit works on CL units are usually associated with cascaded to make up

logical operations

e.g. AND

bit-for-bit logical operation on a pair of words

shift operations

e.g. sll & srl (Shift Left Logical & Shift Right Logical)

shift information in a register by a specified no. of bits

combination of logical and shift operations to extract parts of a word

create a 32-bit mask with desired bits set to 1 (e.g. 8 bits for a character)

AND

shift result by
1-bit ALU – Overflow Detection

When overflow occurs, carry-in and carry-out of sign bit differ. Connect an EOR gate to \( c_{in} \) and \( c_{out} \) of most significant adder.

```
invert b
operation
a b result
```

1-bit ALU - SLT

Result =

If \( a < b \), is equivalent to set \( invert_b \) when performing slt operation to -ve values. Positive values have a sign bit.

Feed o/p of \( invert_b \) back to make the o/p MUX in all the ALUs.

```
operation invert_b
and 0 -
or 1 -
add 2 0
sub 2 1
slt 3 1
```

Branch Instructions – BEQ and BNE

Equality test for beq and bne instructions also relies on

If \( a = b \), then

```
zero-detect circuit controls
```

BNE and BEQ also use

```
bne 2 1
beq 2 1
```

Shift

5-bit shamt field specifies too slow to shift in barrel shifter shifts. E.g., to shift 5 bits, shamt is

```
1's bit 2's bit 4's bit
```
DATAPATH LAYOUT

ALU layout is very regular
layout on silicon is also highly structured
control flows and data flows are orthogonal
minimises complexity and communication times

MULTIPLIER

signed multiplication

problem:
takes multiple clock cycles

MULTIPLIER - FASTER

Booth noticed that, when counting in binary,
a string of
will be
with a
next time the
so the string of can be rewritten as instead of
what's the benefit of that?
a simple multiplier uses to multiply by string of x 1s
but if the multiplier can handle
the algorithm looking for
still have to
**MIPS Multiplication**

The product of a 32-bit multiplications `mult` and `multu` occupies 64 bits.

<table>
<thead>
<tr>
<th>hi</th>
<th>lo</th>
</tr>
</thead>
</table>

`hi` and `lo` registers are not.

`mflo $1` (move from `lo`) puts

`mfhi $1` (move from `hi`) to check that

`mult` ASM instruction is a generates

generates

**Division**

Mathematically (ideally), division is the inverse of multiplication.

\[
\begin{align*}
\text{if} & \quad a = b/c \\
\text{then} & \quad b = \\
\text{and if } b & = 1 \\
\text{then} &
\end{align*}
\]

but with finite precision arithmetic, occur

**Division By Repeated Subtraction**

When we multiplied `m` by `n` to produce a product `p` we generated.

When we divide `p` by `m` we're finding out.

We can divide `p` by `m` by repeatedly `p` till

**Why Are FP Numbers Special?**

We need a way to represent numbers with fractions, e.g., 3.14159

Very small numbers, e.g., 0.0000000001

Very large numbers, e.g., 178478 x 10^9

Representation:

- Sign, exponent, significand:
- More bits for mantissa ➔
- More bits for exponent ➔

IEEE 754 floating point standard:

- Single precision uses
- Double precision uses
IEEE 754 Floating Point Standard:

- **Exponent**
  - Single precision: 8 bits
  - Double precision: 11 bits

- **Mantissa**
  - Single precision: 23 bits
  - Double precision: 52 bits

- **Special Cases**
  - All 0s: 0
  - All 1s: $\pm\infty$
  - Non-zero: NaN
    - Quiet: NaN
    - Signalling: NaN

- **Rounding Options**
  - Nearest integer
  - Nearest even integer if fraction is exactly 0.5
  - Towards 0
  - Towards $+\infty$
  - Towards $-\infty$

Guard And Round Bits

- Extra bits
- Consider adding two decimal numbers with 3 bits of precision:
  2.56 + 2.34 x 10^2

  - With no extra digits: 2.34
  - With 1 extra digit: 2.340
  - With 2 extra digits: 2.3400

  - Round to correct number of significant digits:
  - This may occasionally ripple through to the msb and generate an unnormalised result.
  - Renormalise again if required.
**Floating Point Numbers**

The "Sticky Bit"

When generating a result, a string of 0s may be followed by a 1 that will be normalised away

1.936×0001 (ignoring the exponent)

Simple rounding to nearest even value based on rounding digit, would produce

keep sticky bit as next bit to help resolve "mid-way" rounding problems

1.93650001

Floating Point Multiplication

Add exponents

both include a bias, so have to subtract 1 bias from the result

Multiply

Normalise result, check for overflow

Round

Renormalise if necessary

Floating Point and MIPS

MIPS instructions to support IEEE single and double precision floating point:

- add.s and add.d
- sub.s and sub.d
- mul.s and mul.d
- div.s and div.d
- c.<x>.s and c.<x>.d
  - `<x>` may be `eq`, `neq`, `lt`, `le`, `gt`, `ge`
  - comparison sets bit to
- bclt and bclf

ASMs (Designing a Controller)

Controller executes an infinite loop

- instructs processor to get an instruction from memory
- identifies instruction that the processor has retrieved
- instructs processor to perform data manipulations required by the instruction

Controller ASM

- Algorithmic State Machine
- Specifies timing of data manipulations
- Receives status information

Architecture
**STATUS INFORMATION (architecture → controller)**

Current instruction
If instruction involves a choice
  e.g. JPZ instruction
then controller examines a status line to determine appropriate action

Other status signals are commonly used
  NEG
  OVFL

---

**CONTROL SIGNALS (controller → architecture)**

<table>
<thead>
<tr>
<th>Architectural building block</th>
<th>Control commands</th>
<th>No of bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUXes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>etc</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**FINITE STATE MACHINES (THE BASIS OF ALGORITHMIC STATE MACHINES)**

A state specifies actions that occur on one clock pulse

- A $\square$ must be present
  - even if it is empty

- Outputs in a $\square$ are TRUE

- Outputs in a $\bigcirc$ are TRUE
  - during that state's clock pulse, if ...

Rectangles contain

Diamonds contain

Roundtangles contain
The state is represented as a number.
At any instant (clock pulse), the ASM asserts if the condition that governs them is fulfilled.
ASM calculates the next state no.

The ASM state transition table (navigation only):

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_b$ $B_p$ $Z$</td>
<td>$A_n$ $B_n$</td>
</tr>
</tbody>
</table>
**General structure of the circuit**

- **External inputs**
- **Status**
- **Combinatorial logic**
- **Outputs**
- **Next state**
- **Register**
- **Present state**
- **Clock**

**The complete ASM state transition table**

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A_p ) ( B_p ) ( Z )</td>
<td>( A_n ) ( B_n ) ( P ) ( Q ) ( R ) ( S )</td>
</tr>
<tr>
<td>0 0 0</td>
<td>1 0</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>0 1 -</td>
<td>0 0</td>
</tr>
<tr>
<td>1 0 -</td>
<td>0 0</td>
</tr>
</tbody>
</table>

**A practical problem:**
initialising the state register

- Consider a 22" naval gun
  - controlled by an ASM
  - autoseeking
  - autofiring

- *At power-up,*
  - if state register contains 0, 1, or 3 \( \checkmark \)
  - if state register contains 2 \( ? \)

**We need a reset-on-powerup circuit**

\[ +5V \rightarrow T \rightarrow 0V \]
SUMMARY: DESIGNING AN ASM

Construct: Shows all inputs and control signals.

Translate: external inputs (if any), status inputs, present state, control commands, next state.

ASM vs. SOFTWARE

ASMs underlie processor instruction sets.

but would you ever use an ASM instead of computer program?

| Software | ASMs in discrete logic | ASMs in FPGA |

Is it easy to understand or modify an ASM circuit?

General format is easily recognisable:

```
inputs -> commands
```

But combinatorial circuitry and high-level flowchart vocabularies differ significantly.

Is it easy to understand or modify an ASM circuit?

Modifications to state sequence require a complete redesign.

Disjunction between ASM diagram and combinatorial circuitry.

Could the circuitry (and the outputs)
Using MUXes as a Lookup Table

If we connect:
the control input of a MUX to the state number
the data inputs to TRUE and FALSE

We'll use 1 MUX for each output (including the bits of the next state number)

Simpler and easier to understand than completely combinatorial system

Using MUXes as a Lookup Table

Outputs are sometimes

Q should be in state 3, if

Consider our prototype ASM

A Lift Controller
A LIFT CONTROLLER
Up button means "Take something upstairs"
  If the lift is downstairs,
  
  If the lift is upstairs,

Down button means "Take something downstairs"
A MULTIPLICATION CIRCUIT

Phase 1

Phase 2

How does “manual” multiplication work?

e.g. $5_{10} \times 11_{10}$

Hardware multiplication works similarly
But multiplier
When the process ends, running total contains
Otherwise we’d have to use

Storage requirements
Multiplier
Multiplicand
Product

Put running total in

For each 1 in multiplier,
add
allowing for

Each time a partial product is added to the running total
significance needs to be
First PP goes into position with
Shift running total
A MULTIPLICATION CIRCUIT: Analysis

Put running total in a shift register $2n+1$-bits wide

For each 1 in multiplier,
add multiplicand into running total
allowing for significance of the 1 bit in the multiplier

1011
  1 bit
  2 bit
  4 bit
  8 bit

Each time a partial product is added to the running total
significance needs to be
First PP goes into position
Shift running total
If multiplicand is large,
Add an extra bit to the most significant shift register

A MULTIPLICATION CIRCUIT: Design of the architecture

A MULTIPLICATION CIRCUIT – ASM Diagram
The Processor = Datapath + Control

datapath can process data as specified in the instructions

but fetch/decode/execute cycle needs

control signals regulate

- output to determine the location of the next instruction
- read specified by the instruction (sometimes 1, sometimes 2)
- perform (memory ref, arithmetic/logical, or branch)

control loop always has

- output
- read
- perform
**COMBINATORIAL vs. SEQUENTIAL LOGIC**

Datapath components developed so far use:
- Outputs
- Output of a given set of inputs
- Delay between

Full datapath is:
- Contains storage elements
- Outputs depend on
- Clock regulates

Controller is a sequential circuit:
- Usually implemented as

**GATED CLOCKS**

Clocking Data Through A Path

Is it worth cutting a slow stage into two?
- Yes, if:
  - \( n_{c} > (n+1)(c-\delta c) \)
  - \( (n+1)\delta c > c \)

Register loads data:
- Register i/p unstable
- Register o/p stable during cycle time

Setup hold

We don't always want to load data on every clock cycle:
- Use a separate write control line to:
  - Clock edge specifies: The data should be loaded
  - Write line specifies: The data should be loaded

**INSTRUCTION SUBSET**

Memory reference instructions:
- \( lw \) and \( sw \)

Arithmetic instructions:
- Add, sub, and, or, slt

Branch instructions:
- Beq and \( j \)

Two phases:
- Single-cycle implementation, combinatorial controller
- Multi-cycle implementation – leads to
### The Program Counter

- **Add:** 4 → PC → Instruction Memory → Instruction

### Register File and R-Format Instructions

- **Register File:** Contains 32 32-bit registers implemented as a fast static RAM with dedicated read and write ports. Addresses correspond to control signals based on current instruction. Specify allows two reads and 1 write on a clock cycle. ALU operates on output from 2 registers, writes result back to register file.

### Memory References

- **LW:** $t1, offset ($t2)
- **SW:** $t1, offset ($t2)

To generate branch destination:
- Dedicated ALU adds offset to:
  - If offset is −ve, sign is in bit 15
  - Need to set bits 16 - 31 to 1

### The BEQ Control Logic

- **BEQ $1, $2, offset**

Branch address is register (PC) + offset:
- PC + 4 (here the units are bytes) is offset is needs unit of offset is words, not bytes; shift it 2 bits to the left to multiply by 4

- **PC + 4**
- **ALU sum** → Branch destination
**The BEQ Control Logic**

beq is a conditional branch instruction
so processor must

ALU
if ALU's zero-detect is TRUE,

pc+4

operation

to branch control logic

regWrite

**The J Instruction**

j address
another instruction that loads the PC – unconditionally this time
26 bit address has 2 0 bits added to
no negative addresses, so no need for
top 4 bits of PC are left unaffected
so j instruction can only access

unchanged

26

00

PC

**Single Clock Cycle Restrictions**

all operations must start and finish in 1 clock cycle
no resources can be shared
multiple operations require
increment PC, calculate address, compare registers all need
however, different instruction types can use the same resource
memory references calculate
register operations calculate
can multiplex

Similarly
**Adding Instruction Memory**

Single-cycle instruction requires separate data and instruction memory
no time to read

**Adding The BEQ Instruction**

controller will handle a subset of the ALU functions

<table>
<thead>
<tr>
<th>Function</th>
<th>ALU Control Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>000</td>
</tr>
<tr>
<td>or</td>
<td>001</td>
</tr>
<tr>
<td>add</td>
<td>010</td>
</tr>
<tr>
<td>sub</td>
<td>110</td>
</tr>
<tr>
<td>set on less than</td>
<td>111</td>
</tr>
</tbody>
</table>
SINGLE-CYCLE ARCHITECTURE

COMBINATORIAL CONTROL UNIT

<table>
<thead>
<tr>
<th>instruction</th>
<th>ALU action</th>
<th>ALUop</th>
<th>function code</th>
<th>ALU control</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>add</td>
<td>00</td>
<td>xxxxxxx</td>
<td>010</td>
<td>lw</td>
</tr>
<tr>
<td>sw</td>
<td>add</td>
<td>00</td>
<td>xxxxxxx</td>
<td>010</td>
<td>sw</td>
</tr>
<tr>
<td>beq</td>
<td>subtract</td>
<td>01</td>
<td>xxxxxxx</td>
<td>110</td>
<td>beq</td>
</tr>
<tr>
<td>add</td>
<td>add</td>
<td>10</td>
<td>100000</td>
<td>010</td>
<td>R-type</td>
</tr>
<tr>
<td>sub</td>
<td>subtract</td>
<td>10</td>
<td>100010</td>
<td>110</td>
<td>R-type</td>
</tr>
<tr>
<td>and</td>
<td>and</td>
<td>10</td>
<td>100110</td>
<td>000</td>
<td>R-type</td>
</tr>
<tr>
<td>or</td>
<td>or</td>
<td>10</td>
<td>100101</td>
<td>001</td>
<td>R-type</td>
</tr>
<tr>
<td>slt</td>
<td>set on less than</td>
<td>10</td>
<td>101010</td>
<td>111</td>
<td>R-type</td>
</tr>
</tbody>
</table>

ALU control

C2 = A0 + A1F1

C2 = A0 + A1F1

C1 = A1 + F2

C1 = A1 + F2
**SINGLE-CYCLE ARCHITECTURE**

### Controller Truth Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>R-type</th>
<th>RegDest</th>
<th>ALUSrc</th>
<th>MemToReg</th>
<th>MemWrite</th>
<th>RegWrite</th>
<th>ALUOp</th>
<th>ALUOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw 100011</td>
<td>0 0 0 0 0 (0)</td>
<td>1 0 0 1 0</td>
<td>1 1 1 1 1</td>
<td>0 0 0 0 0</td>
<td>1 0 0 0 0</td>
<td>0 0 0 0 0</td>
<td>0 0 0 0 0</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>sw 101011</td>
<td>0 0 0 0 0 (35)</td>
<td>1 1 1 1 1</td>
<td>0 0 0 0 0</td>
<td>1 0 0 0 0</td>
<td>0 0 0 0 0</td>
<td>0 0 0 0 0</td>
<td>0 0 0 0 0</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>beq 00010</td>
<td>0 0 0 0 0 (4)</td>
<td>X 1 X 0 0</td>
<td>1 0 0 0 0</td>
<td>1 0 0 0 0</td>
<td>0 0 0 0 0</td>
<td>0 0 0 0 0</td>
<td>1 0 1 0 1</td>
<td>0 0 0 0 0</td>
</tr>
</tbody>
</table>

### SINGLE-CYCLE vs. MULTI-CYCLE IMPLEMENTATION

With single cycle, longest instruction limits speed of whole machine:

- **CPI = 1** looks good, but

- load and store instructions involve
  - 1 memory access
  - 1 memory access
  - separate and memories necessary to

  - we can divide instructions into phases
    - e.g., instruction read; register(s) read; compute phase; register write (R-type instruction)
    - set clock period to length of longest phase of an instruction instead of longest instruction
    - instructions become

  - multi-cycle instructions would be faster for all but longest instruction
  - single memory can be used
  - single ALU can be used for data, PC and address operations
**MULTI-CYCLE ARCHITECTURE**

**TRISTATE OUTPUTS**

Using a multiplexor to select inputs to an ALU means multiple 32-bit-wide data paths.

Alternatively, run a single 32-bit bus past all the sources give the sources tristate outputs with n data sources:

- \( \log n \) multiplexor control inputs →
- 32 x n data wires →

**MULTI-CYCLE ARCHITECTURE**

**DATAPATH**

No longer a single clock cycle with standard sequence of control signals. Write signals required. Temporary registers needed for results because:
- signal is computed on 1 clock cycle and inputs that produced it change implicit control signals used.

**MULTI-CYCLE ARCHITECTURE**

**BREAKING INSTRUCTION INTO CLOCK CYCLES**

Equalise time spent in each clock cycle. Minimise time for whole instruction.

Clock cycle should contain no more than

- 1
- 1
- 1

**CLOCK CYCLE 1 - Instruction Fetch**

Common to all instructions.

Stores instruction in IR so that:
- Load IR and incPC in parallel
  - Both
  - Use
  - Take effect

IR ↔

PC ↔
MULTI-CYCLE ARCHITECTURE

CLOCK CYCLE 2 – Instruction Decode & Register Fetch
still don’t know what the instruction is, tho it’s in the IR
read registers specified by rs and rt fields of instruction into A and B registers
don’t need them for all instructions, but does no harm
also compute branch target address, just in case
save result

A ←
B ←
ALU\textsubscript{out} ←

MULTI-CYCLE ARCHITECTURE

CLOCK CYCLE 3 – Mem Addr Computation, or Branch Completion
in this clock cycle, depends on
memory reference instructions (lw and sw)
ALU\textsubscript{out} ←
R-type instructions (arithmetic-logic)
ALU\textsubscript{out} ←
conditional branch instructions
\text{if (A==B) } PC ←
jump instruction
PC ←

MULTI-CYCLE ARCHITECTURE

CLOCK CYCLE 4 – Mem Access, or R-Type Instruction Completion
memory reference instructions (lw and sw)
MDR ←
or
memory[ALU\textsubscript{out}] ←
R-type instructions (arithmetic-logic)
reg [ IR [11:15 ]] ←

MULTI-CYCLE ARCHITECTURE

CLOCK CYCLE 5 – Memory Read Completion
load instruction
MDR ←
or
memory[ALU\textsubscript{out}] ←
**MULTI-CYCLE ARCHITECTURE**

**STATE MACHINE CONTROLLER**

Standard ASM approach to constructing a controller
see Patterson & Hennessy, pps 330-340

**MICROPROGRAMMING**

look up control signals for (instruction, clock cycle) instead of calculating them

consider a processor with n-bit opcode and no instruction 0

on power-up, fetch loads with jumps to

jump table jumps to ; instruction executes, jumps back to

```
0: jump to fetch
jump table 2\(^n\) entries
```

new microcode can be downloaded

extra memory access for each clock cycle

**MULTI-CYCLE ARCHITECTURE**

```
Instruction fetch
Ir \leftrightarrow \text{memory}[\text{PC}];
\text{PC} \leftarrow \text{PC} + 4;
```

```
instr decode, register fetch
A \leftarrow \text{reg} \left[ \text{IR}[21:25] \];
B \leftarrow \text{reg} \left[ \text{IR}[16:20] \];
\text{ALU}_{out} \leftarrow \text{PC} + (\text{sign-extend}(\text{IR}[0:15])) << 21;
```

```
Memory Address, Computation, or Branch Completion
\text{ALU}_{out} \leftarrow \text{A op B};
\text{ALU}_{out} \leftarrow \text{A} + \text{sign-extend} \left(\text{IR}[0:15]\right);
\text{PC} \leftarrow \text{PC} + \left(\text{sign-extend}(\text{IR}[0:15]) \ll \text{shift}ight);
```

```
Memory Address of R-type completion
\text{ALU}_{out} \leftarrow \text{reg}[\text{IR}[11:15]];
\text{MDR} \leftarrow \text{M(\text{ALU}_{out})} \# \text{sw}
```

```
Memory read completion
load: \text{reg}[\text{IR}[16:20]] \leftarrow \text{MDR};
```

```
branch
jump
```

**MICROPROGRAMMING**

look up control signals for (instruction, clock cycle) instead of calculating them

consider a processor with n-bit opcode and no instruction 0

on power-up, fetch loads with jumps to

jump table jumps to ; instruction executes, jumps back to

```
0: jump to fetch
jump table 2\(^n\) entries
```

new microcode can be downloaded

extra memory access for each clock cycle
EXCEPTIONS AND INTERRUPTS

exceptions are unexpected events
e.g.

interrupts are unexpected events from outside the processor
I/O devices generate interrupts to signal input events; process swapping

terminological confusion
MIPS convention: both types of event are
Intel 32-bit processor convention: both

handling exceptions is time-consuming
may determine overall speed of machine
save address of current instruction
transfer control to
operating system:
OS can then
or

MULTI-CYCLE ARCHITECTURE

COMPLEX MULTI-CYCLE ARCHITECTURES

suitable for
CISC machines can have instructions from 2-3 clock cycles to tens or even hundreds
when data for current instruction moves along datapath,
early parts of datapath

EXCEPTIONS AND INTERRUPTS

exception handling starts
e.g. signals from:
overflow detector, unrecognised opcode (simplified MIPS processor)
external pin on processor (I/O devices)
states in controller ASM where exceptions can occur have jump

ASM loads:
cause register with
EPC with
PC with
(location of OS routine for )

OS:
handles
or

vectored interrupts
when exception occurs, controller
OS routine to handle that exception is

PIPLINING

THE BASIC IDEA

single-cycle (1 CPI) architecture simple but slow
no instruction can run faster than the slowest
multi-cycle architecture reduces
instructions still
but some instructions take fewer than
so

can we combine 1 CPI behaviour with shorter clocks?

A → B → C → D

each stage in datapath acts on data from a separate instruction
D enacts phase 4 of instruction i on data for instruction i
C enacts phase 3 of instruction i-1 on data for instruction i-1
later instructions can't work on data currently being produced by the datapath
resources can't be used at several stages in the datapath
need intermediate registers to keep results available for several clock cycles
### Comparison of Approaches

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Instr Fetch</th>
<th>Reg Read</th>
<th>ALU Op</th>
<th>Data Mem</th>
<th>Reg Write</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>R Format</td>
<td>10ns</td>
<td>5ns</td>
<td>10ns</td>
<td>9ns</td>
<td>30ns</td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td>10ns</td>
<td>5ns</td>
<td>10ns</td>
<td>10ns</td>
<td>5ns</td>
<td>40ns</td>
</tr>
<tr>
<td>sw</td>
<td>10ns</td>
<td>5ns</td>
<td>10ns</td>
<td>10ns</td>
<td>5ns</td>
<td>35ns</td>
</tr>
<tr>
<td>beq</td>
<td>10ns</td>
<td>5ns</td>
<td>10ns</td>
<td>10ns</td>
<td>5ns</td>
<td>25ns</td>
</tr>
</tbody>
</table>

### Speedup

- Single clock cycle instructions start every 40ns
- Multi-clock instructions can start every:
  - 50ns (lw) $\times 0.8$
  - 40ns (sw & R-type) $\times 1$
  - 37.5ns (branch) $\times 1.33$

### Saving of Resources
- (3 ALUs $\rightarrow$ 1 ALU) as well as speedup

### Pipeline Overheads

- Load time
- Flush time
- Unequal stage delays
- Delays in interstage registers

### Are Some Instruction Sets Better Than Others?

- Constant length instructions "fit" the hardware better
- IA32 (Pentium) instructions 1-17 bytes translated into microinstructions that suit pipelining
- Standard format with operands in consistent locations allows register reads to occur before instruction type is known

### Standard Format

<table>
<thead>
<tr>
<th>R-type</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-type</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

constant or address
**Are Some Instruction Sets Better Than Others?**

Constant length instructions "fit" the hardware better. IA32 (Pentium) instructions 1-17 bytes translated into microinstructions that suit pipelining.

Standard format with operands in consistent locations allows register reads to occur before instruction type is known.

<table>
<thead>
<tr>
<th>R-type</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-type</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>constant or address</td>
<td></td>
</tr>
</tbody>
</table>

**Dividing The Datapath Into Pipeline Stages**

Information needed in a later stage must be passed via pipeline registers load, read. Pipeline registers are named after for writeback, the pipeline register.

**Dividing The Datapath Into Pipeline Stages**

Memory access instructions are shorter if all calculations are register-based. Calculation phase can be used for word-aligned operands reduce memory accesses. No operand transfer takes.
**THE LW INSTRUCTION: EXECUTION TRACE**

**PIPELINING**

![Diagram of the LW Instruction Execution Trace]

**Diagram Explanation:**
- **Instruction Fetch:**
  - PC (Program Counter)
  - Instruction Memory
- **Instruction Decode:**
  - Register Read
  - Address Calculation
- **Execute:**
  - ALU (Arithmetic Logic Unit)
  - Zero
  - Shift Left 2
  - Sum
- **Memory Access:**
  - Data Memory
  - Read Address
  - Write Address
- **Write Back:**
  - Write Register
  - Read Register
  - Write Data
  - Data Memory

**Timeline:**
- IF (Instruction Fetch)
- ID (Instruction Decode)
- EX (Execute)
- MEM (Memory Access)
- WB (Write Back)
PIELINEI NG

THE LW INSTRUCTION: EXECUTIO N TRACE

Instruction fetch: read instruction, register read: read data, write data, read address, data memory, write address
Instruction decode, register read: zero, ALU, data
Execute, address calculation: memory access
Memory access: write data, read data
Write back: write register, read register: write data, read data

CONTROLLING THE PIPELINE

Instruction fetch: read address, data memory, write address
Instruction decode, register read: zero, ALU, data
Execute, address calculation: memory access
Memory access: write data, read data
Write back: write register, read register: write data, read data

HAZARDS: WHEN PARTS OF THE PIPELINE STAND IDLE

structural hazards: two instructions need the same resource

consider 1w instructions on a MIPS processor with 1 memory for data and program

solution:
**Hazards: When Parts of the Pipeline Stand Idle**

**Structural Hazards:** Two instructions need the same resource

- Consider `lw` instructions on a MIPS processor with 1 memory for data and program

**Data Hazards:** Instruction needs data before instruction, has finished producing it

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Fetch</th>
<th>ALU</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $s0 $t0 $t1</td>
<td>instr fetch</td>
<td>ALU</td>
<td>data</td>
</tr>
<tr>
<td>sub $t2 $s0 $t3</td>
<td>instr fetch</td>
<td>ALU</td>
<td>data</td>
</tr>
</tbody>
</table>

**Solution:** Forward data to next instruction as soon as it's produced

**Control Hazards:** Control decision depends on

- If (pipelined) instruction starts on clock cycle, instruction_{i+1} starts on cycle_{i+1}, unless instruction is a branch instruction

**Solution:** Controller inserts stalls (aka “bubbles”) into the datapath

---

**Hazards: When Parts of the Pipeline Stand Idle**

**Structural Hazards:** Two instructions need the same resource

- Consider `lw` instructions on a MIPS processor with 1 memory for data and program

**Data Hazards:** Instruction needs data before instruction, has finished producing it

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Fetch</th>
<th>ALU</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $s0 $t0 $t1</td>
<td>instr fetch</td>
<td>ALU</td>
<td>data</td>
</tr>
<tr>
<td>sub $t2 $s0 $t3</td>
<td>instr fetch</td>
<td>ALU</td>
<td>data</td>
</tr>
</tbody>
</table>

**Solution:** Controller inserts stalls (aka “bubbles”) into the datapath

**Control Hazards:** Control decision depends on result of an incomplete instruction

- If (pipelined) instruction starts on clock cycle, instruction_{i+1} starts on cycle_{i+1}, unless instruction is a branch instruction

**Solution:** Controller inserts stalls (aka “bubbles”) into the datapath
HAZARDS: WHEN PARTS OF THE PIPELINE STAND IDLE

control hazards: control decision depends on result of an incomplete instruction
if (pipelined) instruction, starts on clock cycle, instruction \( n \) starts on cycle \( n+1 \)
unless instruction is a branch instruction; destination not known for 4 more clock cycles

can just
put in extra tests, calculates \( n \)
&
still have 1 clock-cycle stall
or take one alternative anyway

improvements:
assume
load next instruction
still lose a clock cycle if

or compiler
or
to use the stall time (MIPS)
on the basis of record

SCHEDULING THE BRANCH DELAY SLOT

from before          from target          from fall through

\[ \text{sub } $s1, $s2, $s3 \]
\[ \text{add } $s1, $s2, $s3 \]
if \( s2 = 0 \)

\[ \text{delay slot} \]

\[ \text{add } $s1, $s2, $s3 \]
\[ \text{if } s1 = 0 \]

\[ \text{sub } $s1, $s2, $s3 \]

\[ \text{if } s1 = 0 \]

branch prediction

small memory indexed by
each location has 1 bit - set a bit
will sometimes be set
prediction unrelated to
but will contain
but consider performance of a loop branch
taken times, not taken once
misprediction
but after last iteration, prediction is
so misprediction at
branch taken 90% of time, correct prediction 80% of time

use 2-bit branch prediction memory
must be wrong twice before prediction changes
copes with repeated loops

DATA HAZARDS - CATEGORISATION

in each of the hazards below, instruction \( i \) starts executing before instruction \( i+1 \)
hazard name refers to what should happen, not what goes wrong (!)

RAW – Read After Write

WAR – Write After Read

WAW – Write After Write
**RAW Hazards**

four situations – 2 problematic, 2 not

```
LW R1, 45, (R2)
DADD R5, R6, R7
DSUB R8, R6, R7
OR R9, R6, R7
```

nothing in following 3 instructions depends on R1
(4th following instruction will be in IF when 1st is in WB)

```
LW R1, 45, (R2)
DADD R5, R1, R7
DSUB R8, R6, R7
OR R9, R6, R7
```

hardware detects phase1, R1 write, phase2 R1 read
stalls DADD instruction's EX phase
(& following instructions)

```
LW R1, 45, (R2)
DADD R5, R6, R7
DSUB R8, R1, R7
OR R9, R6, R7
```

hardware detects
forwards

```
LW R1, 45, (R2)
DADD R5, R6, R7
DSUB R8, R6, R7
OR R9, R1, R7
```

no action required. Write of R1 occurs during 1st half
of DSUB's ID phase, and read occurs in 2nd half

---

**The Ideal, The Reality, & The Solution**

The ideal
- indefinite memory capacity
- random access
- any word instantly available

The reality
- limited memory capacity
- finite speeds
- high speeds → high costs
- high capacity → low speed

The solution
- hierarchy of memories, with processor registers at the top
each step down has more capacity but slower access

---

**When To Forward In EX Phase**

<table>
<thead>
<tr>
<th>source instruction</th>
<th>destination instruction</th>
<th>forward if: destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>R-type</td>
<td>EX/MEM[rd] = ID/EX[rs]</td>
</tr>
<tr>
<td></td>
<td>I-type, lw, sw, bra</td>
<td>MEM/WB[rd] = ID/EX[rt]</td>
</tr>
<tr>
<td></td>
<td>R-type</td>
<td>MEM/WB[rd] = ID/EX[rt]</td>
</tr>
<tr>
<td></td>
<td>R-type</td>
<td>MEM/WB[rd] = ID/EX[rt]</td>
</tr>
<tr>
<td></td>
<td>R-type</td>
<td>MEM/WB[rd] = ID/EX[rt]</td>
</tr>
<tr>
<td></td>
<td>R-type</td>
<td>MEM/WB[rd] = ID/EX[rt]</td>
</tr>
<tr>
<td></td>
<td>R-type</td>
<td>MEM/WB[rd] = ID/EX[rt]</td>
</tr>
<tr>
<td>lw</td>
<td>lw</td>
<td>MEM/WB[rt] = ID/EX[rt]</td>
</tr>
<tr>
<td>lw</td>
<td>lw</td>
<td>MEM/WB[rt] = ID/EX[rt]</td>
</tr>
</tbody>
</table>

---

**The Solution**

Memory management blurs the distinctions to make memory seem
as possible
as possible
as possible

Virtual Memory
auto archival
and file retrieval

processor
registrs
main
(Primary)
memory
backing
(Secondary)
store
archive
**Alternative View**

- CPU
- Increasing distance from the CPU in access time
- Levels in the memory hierarchy
- Size of the memory at each level

**Principle(s) of Locality**

- Only a small proportion is of interest at any time
- Principle of is liable to be followed
- Principle of is liable to be followed by (special case of preceding principle)

- On memory reference:
  - Bring item from to SRAM: fast, but expensive and thus small
  - From which (and ask it)

**Aims**

- Aims: to make memory behave
  - As as
  - As as

- Technique:
  - On a hit,
  - On a miss,
  - May need to transfer from to

- Hit ratio:
- Miss ratio:
- Miss penalty:
- Hit time:

**Cache**

- Small, fast between registers and
- How do we map the large DRAM address space onto the small SRAM?

- Direct-mapped cache
  - Address in cache with $2^n$ locations is just

- Diagram of memory levels and cache mapping.
ACCESSING A WORD IN CACHE – the Write Back strategy

MEMORY MANAGEMENT - cache

ACCESSING A WORD IN CACHE

MEMORY MANAGEMENT - cache

OTHER FLAVOURS OF CACHE (Guava, Loganberry, Snail)

ASSessing CACHE UPDATE ALGorithms

changed bit is not always used
when a cache location is overwritten
its impossible to tell
always write cache data back to memory when
Simple Swap strategy

OR Write-Through (even simpler)
always write data to cache & back to memory when
not as inefficient as you might think;
buffer queue stores data & address so
Block Transfers

temporal locality supported by

spatial locality requires

Handling a Cache Miss

what happens to the current instruction when a cache miss occurs?

consider a miss when loading an instruction

instruction register contains

we'll have to when the value has loaded

PC ← (because )

initiate

write ; reset

Intrinsity FastMATH Processor Caches

Instruction miss rate:
data miss rate:
### Transferring Blocks Efficiently

**Memory Management - Cache**

**Memory Speed** puts lower limit on latency of access to 1st word of a block.

Wider memory & bus allow no reduction overall transfer rate.

<table>
<thead>
<tr>
<th>Bus and Memory Width</th>
<th>Cache Block Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 word</td>
<td>4 words</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Bus Clock Cycles (10 longer than processor cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>to send address</td>
</tr>
<tr>
<td>to access data</td>
</tr>
<tr>
<td>to send data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Miss Penalty</th>
<th>Bandwidth Achieved</th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
<td>= 0.25 words/cycle</td>
</tr>
</tbody>
</table>

**Bandwidth Achieved** = \( \frac{20}{4 \times 4} = 0.25 \) words/cycle

---

### Transferring Blocks Efficiently - Wider Memory And Bus

Memory speed puts lower limit on latency of access to 1st word of a block.

Wider memory & bus allow parallel transfer of complete block.

No reduction in latency of 1st word.

Overall transfer rate higher.

<table>
<thead>
<tr>
<th>Bus and Memory Width</th>
<th>Cache Block Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 word</td>
<td>4 words</td>
</tr>
<tr>
<td>2 words</td>
<td>4 words</td>
</tr>
<tr>
<td>4 words</td>
<td>4 words</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Bus Clock Cycles (10 longer than processor cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>to send address</td>
</tr>
<tr>
<td>to access data</td>
</tr>
<tr>
<td>to send data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Miss Penalty</th>
<th>Bandwidth Achieved</th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
<td>= 0.25 words/cycle</td>
</tr>
<tr>
<td>33</td>
<td>= 0.48 words/cycle</td>
</tr>
<tr>
<td>17</td>
<td>= 0.94 words/cycle</td>
</tr>
</tbody>
</table>

**Bandwidth Achieved** = \( \frac{20}{4 \times 4} \times 2 = 0.48 \) words/cycle

---

### Transferring Blocks Efficiently - Interleaved Memory

**Cache Design Issues**

- **Cache size**
- **Memory update policy**
- **Placement policy**
- **Replacement policy**

1-word wide memory

Multi-word wide memory

Interleaved memories

All memories can read in parallel.

<table>
<thead>
<tr>
<th>Single-word bus cycle count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 to send address</td>
</tr>
<tr>
<td>15 to access memory</td>
</tr>
<tr>
<td>4 x 1 to send data bandwidth</td>
</tr>
</tbody>
</table>

\( = \frac{(4 \times 4)}{20} = 0.8 \) words/cycle
MEASURING AND IMPROVING CACHE PERFORMANCE

Consider a machine with separate instruction and data caches.

Processor CPI (without memory stalls): 2

Miss penalty (all misses): 100 cycles

For a particular program:

Instructions executed: I

Loads and stores as % of total instructions: 36%

Data cache miss rate: 4%

Instruction cache miss rate: 2%

How much faster would the program be if we eliminated misses?

Total miss cycles = +

CPI including memory stalls = + =

Now, speed is IPC = 1/CPI

Speed increase =

Speed increase = 2.72

What about improving the processor performance?

Let's try reducing the Cycles Per Instruction by 50%.

Speed increase = 1.18

Remember Amdahl's Law?

Max speedup = total time/time that can be reduced to 0
MEASURING AND IMPROVING CACHE PERFORMANCE

how much faster would the program be if we eliminated misses?
speed increase = 2.72

what about improving the processor performance?
let's try reducing the Cycles Per Instruction by 50%
speed increase = 1.18
let's try doubling the clock rate (memory read/write times don't change)
miss penalty = 200 clock cycles (previously 100)
total miss cycles = instruction misses + data misses
= I x 0.02 x 200 + I x 0.36 x 0.04 x 200 = 6.88 I
CPI = 2 + 6.88 = 8.88

speed increase = \( \frac{\text{execution time}_{\text{fast clock}}}{\text{execution time}_{\text{slow clock}}} = \frac{I \times CPI \times \text{clock cycle}_{\text{1}}}{I \times CPI_{2} \times 0.5 \times \text{clock cycle}_{\text{1}}} \)
= 5.44 / (8.88 x 0.5) = 1.23

cache misses (& Amdahl) reduce impact of other improvements
increasing clock rate AND decreasing CPI incurs a double hit
when processor clock speeds increase, memory clock speeds don’t
improve. Miss penalty high clock speed processor > miss penalty low clock speed processor

good cache design helps performance as much as increasing processor speed

ALTERNATIVE PLACEMENT POLICIES

there are various schemes for placing blocks in cache
intended to reduce cache misses

direct mapping
cache index bits are subset of memory address,
each block of memory locations has only one possible cache destination

set-associative mapping
n-way SA cache has n blocks
each memory block maps to any element of a unique subset (>=2) of the cache blocks
mapping to set is direct, mapping within set is associative

fully associative mapping
memory blocks go anywhere in cache, source address is stored with them
cache access involves comparing address tag & cache tag at every cache location
multiple comparators ➔ expensive hardware
suitable for L1 (only a few words) caches only

INDICATIVE DIAGRAMS

direct mapping

set associative mappings

fully associative mapping
**FORMAL SPECIFICATIONS - Direct Mapping**

Consider a system with memory and cache both organised into blocks.

- blocksize, \( b = 2^w \) words, for some \( w \)
- lines in cache, \( L = 2^k \) \( \Rightarrow \) cachesize = \( 2^{k+w} \) words
- blocks in memory, \( B = 2^{k+1} \) \( \Rightarrow \) memorysize = \( 2^{k+w+1} \) words

Hence addresses are \( k_2 + w \) bits long.

If \( L = 4, k_1 = 2 \)
\( B = 8, k_2 = 3 \)
\( b = 4, w = 2 \)

Then these mappings hold:

\[ M_0 \rightarrow C_0 \]
\[ M_4 \rightarrow C_0 \]
\[ M_1 \rightarrow C_1 \]
\[ M_5 \rightarrow C_1 \]

etc.

Block line \( j = i \mod L \) (\( i \) is memory block)

**FORMAL SPECIFICATIONS - Fully Associative Mapping**

Direct mapping

- desired data may be in only one cache location
- though mapping is many-to-one

Fully associative mapping

- desired data may be in several cache locations
- the one which contains the word addressed (if any) must then be identified
- many-to-many mapping

Again, divide memory into \( 2^k \) blocks of \( 2^w \) words.

- \( (k_2 + w) \)-bit addresses
- each cache line contains
  - data field (\( 2^w \) words)
  - a tag field (top \( k_2 \) bits of the block’s address)
  - equality-detect circuit (tag field = top \( k_2 \) bits of address)

At each memory reference:

- if tag matches, cache contains the addressed data
- the equality detect signal acts as a "line select" to allow I/O on the appropriate word of the line

**4-WAY FULLY-ASSOCIATIVE CACHE**

Amalgam of direct and associative schemes

- cache has structure
- lines are grouped into block has \( w \) words - as before
- memory organisation cache has

Mapping works at first

- set \( j = i \mod S \) (\( j \) is a block in main memory)
- Then, after associative search for block
- Then: to find specific word

**SET-ASSOCIATIVE CACHE**
L1, L2, L3

L1 cache
- typically 8KB - 128KB
- part of processor core
- fast technology (SRAM)
- higher processor speed

Some L2 and even L3 caches run at processor speeds.

So what's the point of smaller L1 cache?

L2 cache
- typically 256KB - 1MB
- originally off chip; now often on ½ or ⅛ processor speed

A more sophisticated, more expensive, more efficient memory mapping policy is desirable but only cost-effective on a small scale???

L3 cache
- increasingly common
- typically 16MB - 256MB
- off chip (but sometimes on the same die)
- expense; used in high-end processors
- ½ L2 cache speed

Some L2 and even L3 caches run at processor speeds.

So what's the point of smaller L1 cache?

A more sophisticated, more expensive, more efficient memory mapping policy is desirable but only cost-effective on a small scale???

L1, L2, L3

L1 cache
- typically 8KB - 128KB
- part of processor core
- fast technology (SRAM)
- processor speed

Some L2 and even L3 caches run at processor speeds.

So what's the point of smaller L1 cache?

L2 cache
- typically 256KB - 1MB
- originally off chip; now often on ½ or ¼ processor speed

A more sophisticated, more expensive, more efficient memory mapping policy is desirable but only cost-effective on a small scale???

L3 cache
- increasingly common
- typically 16MB - 256MB
- off chip (but sometimes on the same die)
- expensive; used in high-end processors
- ½ L2 cache speed

Some L2 and even L3 caches run at processor speeds.

So what's the point of smaller L1 cache?

A more sophisticated, more expensive, more efficient memory mapping policy is desirable but only cost-effective on a small scale???
**BLOCK REPLACEMENT POLICY**

- **direct mapping cache**
  - no choice; incoming block can only go into one slot

- **associative cache**
  - any block could go

- **set-associative cache**
  - incoming block can only go into one set
  - any block in selected set could go

**FIFO**

Least Recently Used; significantly better performance than FIFO
- each set has a reference number
- when set with reference no. \( n \) is referenced
  - reference nos < \( n \) are incremented
  - reference no of referenced set is set to 0
- block-to-go is always block with largest reference number
- expensive hardware when sets are large.

**VIRTUAL MEMORY'S RAISONS D'ÊTRE**

Original VM let programs use a memory space larger than physical memory
- programmers had to divide programs into mutually exclusive overlays (code & data)
- program had to control loading of its own overlays
- VM automatically maps program pages onto physical memory addresses

Also allowed multiple programs to run simultaneously
- independent virtual address spaces
- memory protection
- predominant use today

<table>
<thead>
<tr>
<th>installed memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>program 1</td>
</tr>
<tr>
<td>program 2</td>
</tr>
<tr>
<td>program 3</td>
</tr>
<tr>
<td>program 4</td>
</tr>
</tbody>
</table>

| individually smaller, together larger |

**MEMORY MANAGEMENT - virtual memory**

**THE NEXT PHASE**

Memory management blurs the distinctions to make memory seem
- as big as possible
- as fast as possible
- as cheap as possible
- as secure as possible

- **cache**
- **Virtual Memory**
- auto archival and file retrieval

**GENERAL IDEA**

- program code and data are stored as fixed-sized units called pages
  - pages live on disk, have a disk address, are copied into memory when necessary
  - memory operations use (Virtual Page no, page offset)

- **address translation unit**
  - converts VP no. into base address of page in physical memory
  - base address + page offset produces real address of data
  - bits in offset \( \rightarrow \) page size

- program's code (and data) pages don't have to be contiguous
### Pragmatic Decisions

- **Page Fault Costs**: Millions of clock cycles, mostly latency of first word. Later words arrive comparatively rapidly, so make the page size big enough to repay cost of page fault (4KB – 64KB).

- **Reducing Page Faults**: It's worth putting considerable effort into reducing page faults.

- **Page Placement**: Fully associative page placement.

- **Disk Access**: Long disk access allows time for (complex) software to handle page faults.

- **Write Time**: Long write time justifies complexity of write-back over write-through.

- **Fixed Size Pages**: With fixed size pages, (page no., offset) boundary in addresses is transparent. Cf. variable-sized segments, where software manipulates segment no. & offset explicitly.

- **Memory Protection**: Memory protection can use the same mechanisms as virtual memory.

### Page Placement

- **Virtual Pages**: Can go anywhere in memory. Huge miss penalty means it's worth using complex algorithm, & data structures.

- **Process Page Table**: Each process has a page table. Page table lives in memory. Page table register points at start of page table. To perform a process swap, point page table register at a different process's page table (and swap the program counter and processor registers).

### Page Table

- **Virtual Address**: 32-bit virtual address, 30-bit physical address (2 bottom bits = 0).

- **Virtual Memory Space**: 4x larger than physical address space.

- **Page Table Entry**: Page table stored in (32-bit) main memory has 19-bit entries. 13 extra bits (not shown) for page protection information.
**Page Faults**

OS is responsible for handling page faults
hardware detects that valid bit for selected page is FALSE

not all of a program's pages have to be in memory while it's running
only pages that have been referenced since the process was swapped in

but disk space is much more abundant
OS usually reserves enough disk space for all the process's pages
called the swap space

**Page Replacement**

if the OS needs to bring in a page and all pages in the swap space are in use
it must oust a page
which page? swap space is a fully associative store

page-to-go should be about to be unused for as long as possible
predicting the future is a difficult business
prediction: Least Recently Used page will be Furthest Future Use page

strict LRU algorithm would collect stats at every memory reference
too expensive
instead, each page references sets a reference bit for the page
reset periodically, tested after standard delay
any page with reference bit still reset is Not Recently Used – can be paged out

ousted page goes to swap space

**Page Table Sizes**

typically scores or hundreds of processes running at a time
for a machine with 32-bit addresses, 32-bit page table entries and 4KB pages
how many page table entries? \(2^{12} / 2^{12} = 2^{20}\)
how big is the page table? \(2^{20} \times 2^{2} = 2^{22} \text{B} = 4\text{MB}\)
so, maybe 400MB in total

how to minimise memory usage generally and size of page table in particular?
use dynamically-sized page table; only big if program is page-greedy
keep last page register (aka page limit register)
forces page table to grow in 1 direction only
but stack and heap usually grow in opposite directions

for a machine with 32-bit addresses, 32-bit page table entries and 4KB pages
how many page table entries? \(2^{12} / 2^{12} = 2^{20}\)
how big is the page table? \(2^{20} \times 2^{2} = 2^{22} \text{B} = 4\text{MB}\)
so, maybe 400MB in total
**MEMORY MANAGEMENT — virtual memory**

**Page Table Sizes**

how to minimise memory usage generally and size of page table in particular?

- use dynamically-sized page table; only big if program is page-greedy
  - keep last page register (aka page limit register)
  - forces page table to grow in 1 direction only

- but stack and heap usually grow in opposite directions
  - separate page tables, with 2 page limit registers, 1 for up, 1 for down
  - top bit of address differentiates between top & bottom segments of address space

- inverted page table
  - only in-use pages are stored; can’t use address to index page’s entry, must search
to reduce search time, hash the addresses’ entries

- multi-level (tree-structured) page table complex, but suits non-contiguous pages
  - highest order bits address a “segment”; if valid, lower bits address page in segment

- page the page table
  - increases no. of page faults; lock some pages of page table in memory

**Handling Write Operations**

- cache can use writethrough
  - memory only hundreds of times slower than registers
  - small writethrough buffer masks write latency

- VM write has to wait for disk access
  - millions of clock cycles
  - buffer would be impractical
  - writeback used instead
  - disk write only occurs when essential – when OS overwrites a dirty page in memory

**The TLB – Handling The Overhead**

- page table resides in memory
  - ordinary read or write (Instr. Fetch or lw/sw instruction) involves 2 memory accesses
    - 1 to convert virtual address to physical address
    - 1 to access the data

- to reduce memory references
  - principle of locality applies to page table entries too
  - Translation Lookaside Buffer is a translation cache
    - cf. scraps of paper provided by libraries for writing library call no.
    - maintains list of locations of a subset of pages
  - replacement policy difficult
    - software too slow; hardware too expensive for complex policy (e.g. LRU)
    - often randomly choose entry-to-go

**TLB Circuit Outline**

- virtual address
  - 20-bit virtual page number
  - 12-bit page offset

- if no hit in TLB, use page table
  - if no hit there, get page from disk
  - if no cache hit, use memory
**MEMORY MANAGEMENT — Virtual Memory**

### TLB Circuit Outline

- **Virtual address**: 20-bit virtual page number, 12-bit page offset.
- **TLB**:
  - **Tag**, **Physical Page Number**
  - **Physical Page Number**, **Page Offset**

- **Cache Index**: 8
- **Byte Offset**: 4

- **Hit**: if no hit in TLB, use page table; if no hit there, get page from disk; if no cache hit, use memory.

### Using the TLB for a Read/Write Operation

- **Virtual address**
- **TLB access**
  - **TLB hit?**
    - **Yes**: TLB provides physical address
    - **No**: Try to read data from cache
      - **Write?**
        - **No**: Write access bit on? (Yes)
          - **write data into cache**
            - **update the tag**
            - **put data & address into the write buffer**
        - **Yes**: No write access bit on?
          - **write data into cache**
            - **update the tag**
            - **put data & address into the write buffer**
  - **No**: TLB miss exception
    - **write?**
      - **Yes**: No write access bit on?
        - **write data into cache**
          - **update the tag**
          - **put data & address into the write buffer**
      - **Yes**: No write access bit on?
        - **write data into cache**
          - **update the tag**
          - **put data & address into the write buffer**

---

**MEMORY MANAGEMENT — Virtual Memory**

### The Synergy Between VM and Memory Protection

Machines run multiple processes "simultaneously" on single processor machines only one process is active at a time. Simple multitasking swaps between processes when I/O occurs. Preemptive multitasking swaps at short intervals, giving users on multiuser systems the impression of sole access. Allows modern multiprocessor systems to handle hundreds of processes.

**Active Process**

**I/O Request**

**Processes Engaged in I/O**

**I/O Terminates**

**Ready Processes**

**VM Allows Processes**

A process's pages can be.

What if one process addresses outside its own space?
PROTECTION REQUIREMENTS
separate user and OS (supervisor) modes
some instructions available
ability to make certain information read-only for user processes

mechanism for swapping between modes
transfers control to
store in
return from exception

put page tables in
allows OS to
prevents user process from
prevents user process from

IMPLEMENTING A PROCESS SWITCH
consider a switch from P₁ to P₂

on a machine without a TLB, this is comparatively simple
point page table register at

on a machine with a TLB
need to
need to
replacing all of P₁'s entries in TLB can be inefficient if:

problem; virtual address spaces the same
solution: make them different
give each process
OS remembers
put
at each memory access,

SHARING INFORMATION
in general, one process should not be able to
TLB has
that prevents a process from

sometimes processes need to be able to share information
P₁ wants to access information in a page owned by P₂
process P₂ asks OS to create a new page table entry
page table entry goes into P₁'s virtual address space but accesses P₂'s physical page
P₂ can ask OS to set write protection bit in P₁'s page so P₁ can't update the page