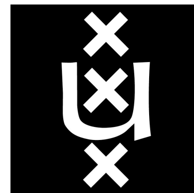
Computer Architecture

R. Poss
Computer Systems Architecture group (UvA)
e-mail: r.c.poss@uva.nl



Performance

Processor performance equations

```
Performance = Results / second
            = Results / Instructions
            x Instructions / Cycles
            x Cycles / Second
```

IPC (→ Instructions / Cycles)

frequency (Hz) (→ Cycles / Second)

```
Execution time = Seconds / Result
               = Instructions / Result
               x Cycles / Instruction
               x Second / Cycle
```

CPI (→ Cycles / Instruction)

- Performance and execution time are related - how?
  - Hint: think throughput vs latency, 1 result vs many

Who controls what?

- Results vs instructions: <u>software</u>
  task of programmer, compiler, instruction set
- Instructions vs cycles: <u>micro-architecture</u>
  task of processor designer, instruction set, partly compiler
- Cycles vs seconds: <u>technology</u>
  task of circuit designer, manufacturer

Performance comparison

- "X has a speedup of n relative to reference Y"
- "X is n times faster than Y"
  - `n = perf(X) / perf(Y)` ← in general case
    `= exectime(Y) / exectime(X)` ← for 1 program
- NB: performance encompasses software + hardware
  - Can't compare performance of sw alone without specifying which hw is used to measure

Power

Two forms of power

- Dynamic power: cost of changing the state of transistors
  - Traditionally dominant
- Static power: cost of keeping the state of transistors
  - Becoming important as dynamic power decreases

Dynamic power

$Power_{dynamic}$ = CapacitiveLoad x Voltage^2 x SwitchFrequency

- Capacitive load is a function of the number of transistors involved in the computation: more transistors = more power
- Dropping voltage reduces dynamic power, however frequency dependent on voltage so frequency must be decreased too
- Most circuits now disable clock of inactive components (set switch frequency to 0) to save dynamic power

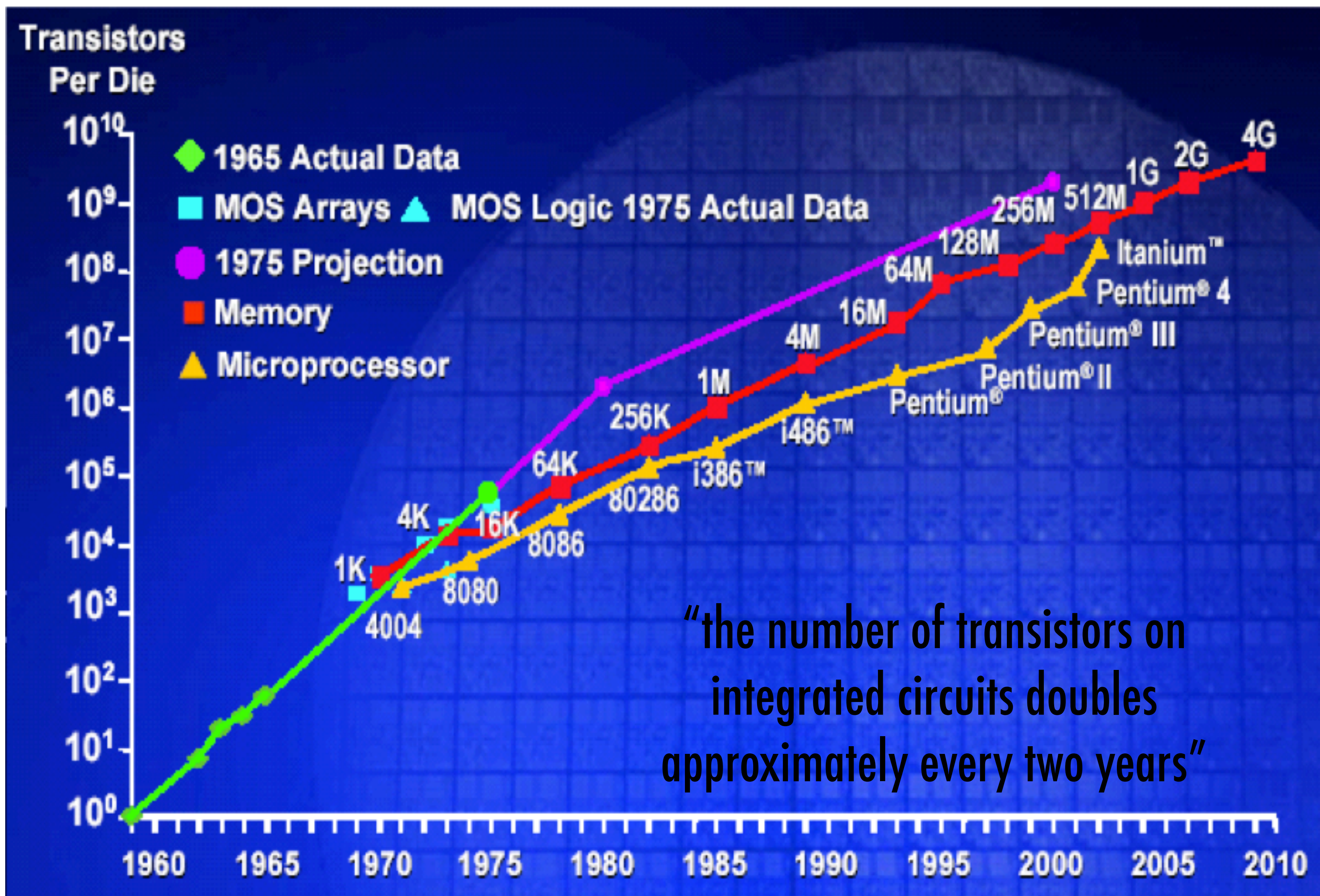
Static power

$Power_{static}$ = LeakageCurrent x Voltage

- Leakage current increases as transistor size decreases
- Leakage current exists even when transistors do not switch
- Low power circuits now disable voltage of inactive components to save on static power
  - However restoring voltage is costly in time

Trends

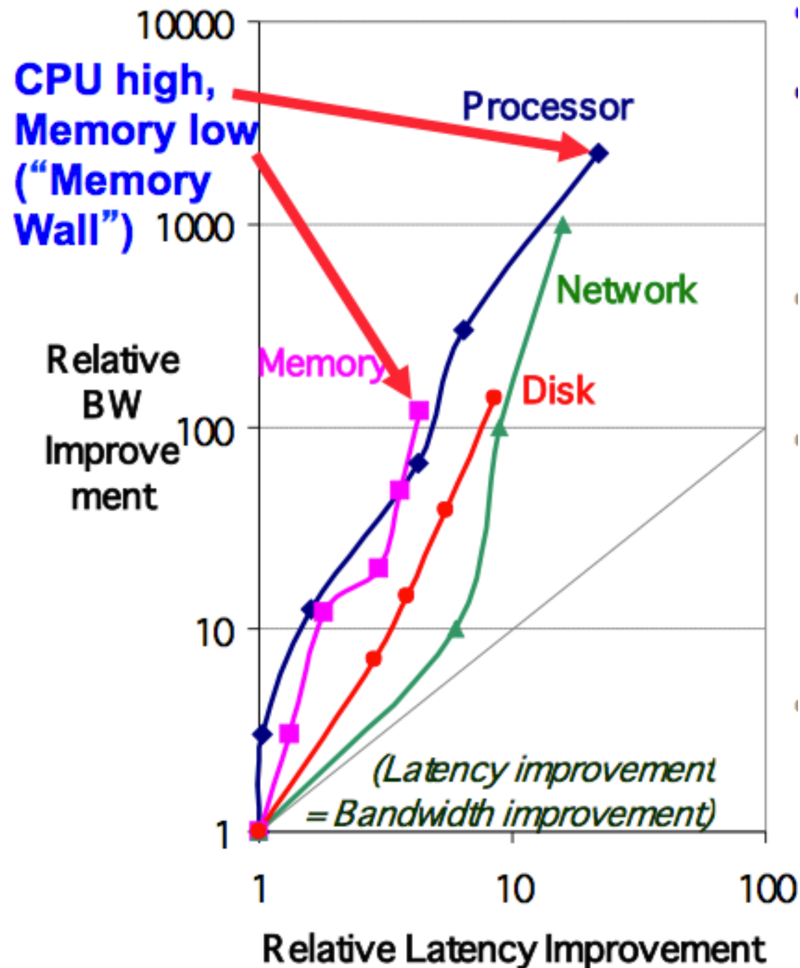
Moore’s law



Moore's law

- Why/how:
  - CMOS: logic based on semiconductor gates in silicon, DRAM: single-gate memory cells
  - laser photolithography to sculpt gates at atomic scale
- Fundamental limits:
  - can't make CMOS smaller than atoms in silicon
  - difficult to increase precision of lasers in manufacturing
- Probable evolutions:
  - number of transistors per unit of area in silicon will stabilize
  - likely: larger chips + 3D designs with multiple layers of silicon (more area)

Latency lags bandwidth



Improvements over ca. 20 years

	Latency	Bandwidth	Transistors
Processors	/30	x3000	x10000+
Networks	/20	x1000	
Memory	/4	x200	x100000+
Disks	/10	x200	

Latency lags bandwidth

- Moore’s law helps bandwidth more than latency
  - More transistors + more pins = more bandwidth
- Distance limits latency, storage capacity increases distance
  - More transistors = relatively longer lines
  - We will study this later in the context of memories
- Market bias: bandwidth easier to sell, so more investment there

Latency lags bandwith

- Latency helps bandwidth, but not the other way around
  - eg: faster disk spin rate: shorter access times, more requests by second
  - but: more disks in parallel = more bandwidth, same latency
- Bandwidth hurts latency
  - Queues help bandwidth, hurts latency (queuing theory)
  - adding parallelism actually increases latency (cf later lecture)

Latency lags bandwidth

- Summary:
  - For 1 component,
    bandwidth increases by square of latency decrease
  - Parallelism allows to scale bandwidth arbitrarily,
    but keeps latency constant or increases
  - Similar ratios for performance vs execution time
  - These trends are there to stay

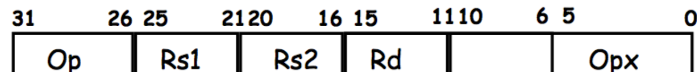
Processors:
RISC pipelines

Instruction set architecture

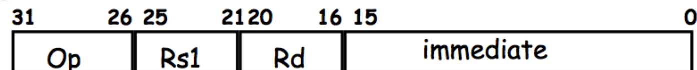
- **ISA = instruction set + operational semantics**
- Instruction set = all possible instruction encodings
  - Described with instruction formats and decode logic
  - Defines how operands and operations are derived from the instruction codes
- Operational semantics = “what instruction do”
  - Described with pseudo-code, also called Register Transfer Language (RTL)
  - Defines how results are produced from operands

Example instruction set: MIPS

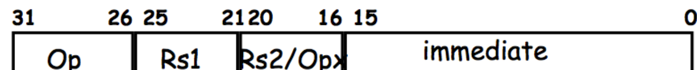
Register-Register



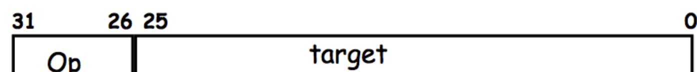
Register-Immediate



Branch



Jump / Call



- First bits determine op and encoding of rest
- More regularity = simpler decode logic

Example encodings

Op	Format	Opx	Insn
0	R-R	0x20	add
0	R-R	0x21	addu
0	R-R	0x22	sub
0	R-R	0x23	subu
8	R-I	-	addi
9	R-I	-	addiu
0x23	R-I	-	lw
0x2B	R-I	-	sw
4	B (R-I)	-	beq
3	J	-	jal

Example RTL: ALU

```
IR <= mem[PC]
PC <= PC + 4

A <= reg[IR_rs1]
B <= reg[IR_rs2]

res <= A op_IRop B

WB <= res
Reg[IR_rd] <= WB
```

Simple names (`IR, PC...`) designate buffers: 1 value

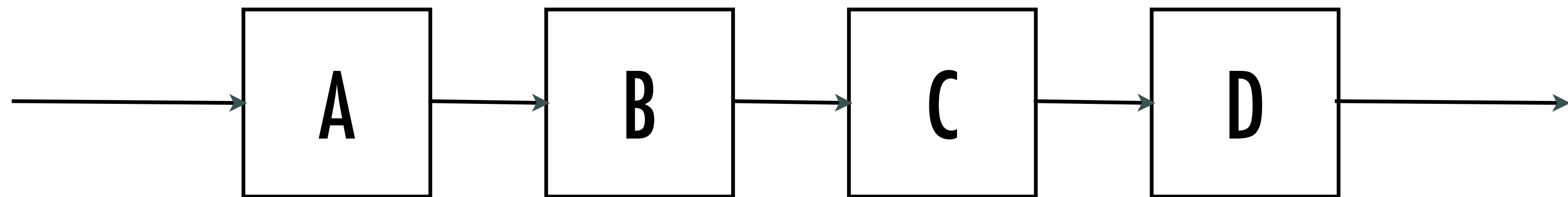
Names with brackets designate memories (address in, data out)

These parts are common to all instructions

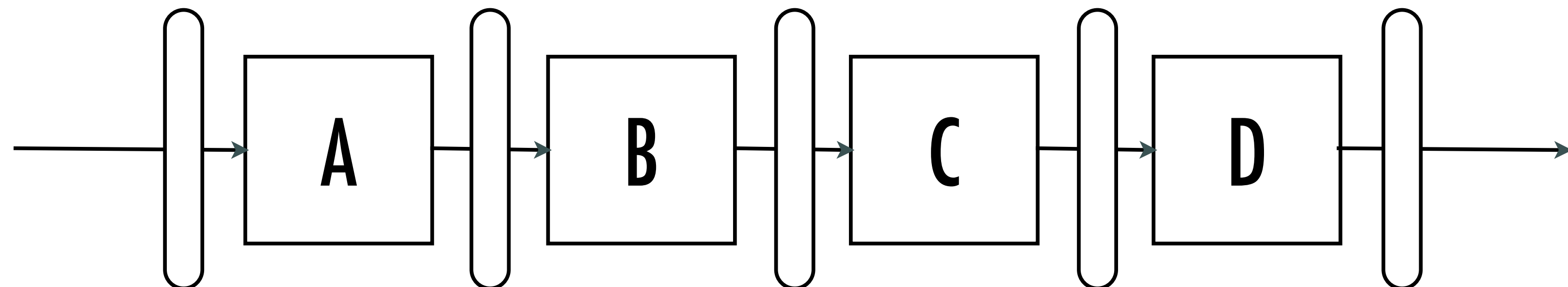
This is specific

Pipelines

- Each instruction / operation can be decomposed in sub-tasks:
  Op = A ; B ; C ; D



- Considering an instruction stream [Op_1; Op_2; ...]
  at each cycle n we can run in parallel: A_{n+3} || B_{n+2} || C_{n+1} || D_n



at start of cycle n: input A_{n+3} input B_{n+2} input C_{n+1}

© Chris Jesshope 2008-2011, Raphael Poss 2011

Origins of pipelining

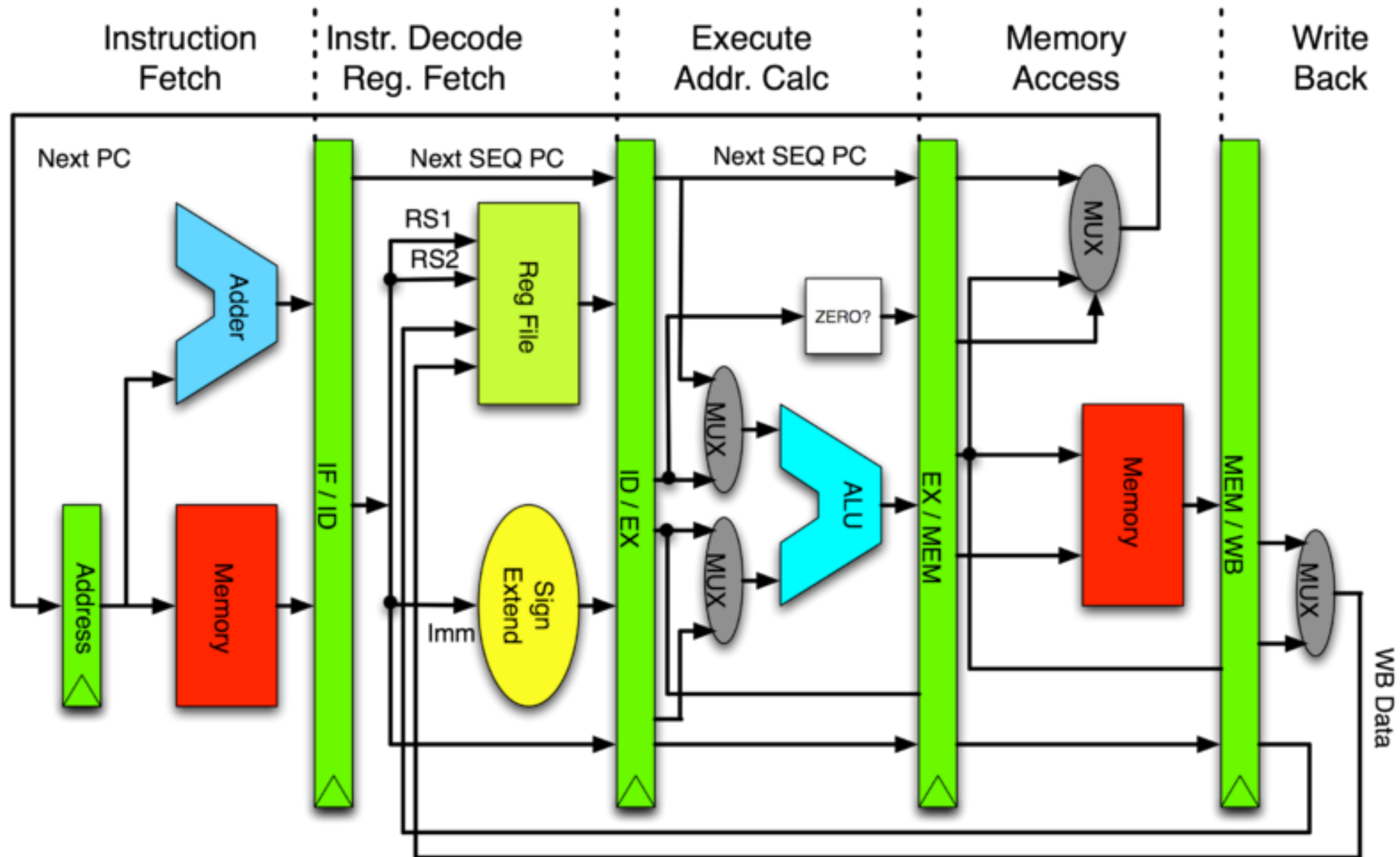
- Idea: “assembly line”
- Different phases of two instructions can occur at the same time
  - eg. read operand of one instruction while the next is being fetched
- More steps in RTL = potentially more stages in pipeline

Processor from scratch

- Start with an ALU: multiple functional units
  - Inputs: A, B, opsel ("which operation to perform")
- Make A and B come from a register file
  - Needs register addresses #A and #B
- Add a decode stage: compute #A, #B, opsel from instruction
  - Needs instruction to decode
- Add a fetch stage: get instruction from memory, using PC
  - Need logic to increase PC after each instruction
- Arithmetic results: Need to store final value C in register file
  - Need register address #C, also decoded from instruction
- Memory: some operations access memory (loads, stores)

"Basic blocks":
- data paths
- buffers
- functional units
- multiplexers
- memories

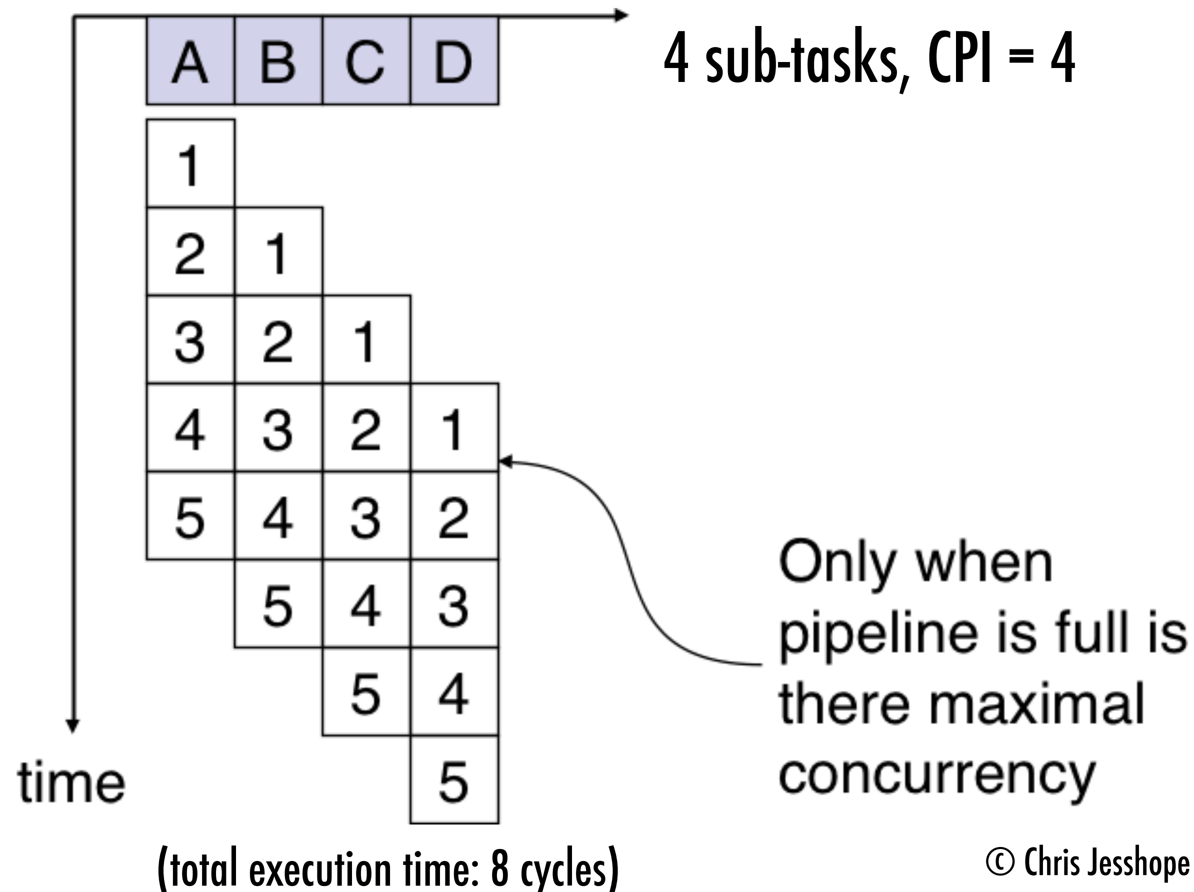
MIPS Pipeline



Dynamic behavior of pipelines

Dynamic behavior

- 1 program, 5 operations:



4 sub-tasks, CPI = 4

(total execution time: 8 cycles)

© Chris Jesshope 2008-2011, Raphael Poss 2011

Pipeline performance

- This pipeline has a length of 4 subtasks, assume each sub-task takes t seconds
  - for a single instruction we get no speedup; it takes 4t seconds to complete all of the subtasks
  - this is the same as performing each sub task in sequence on the same hardware
- In the general case – for n instructions – it takes 4t seconds to produce the first result and t seconds for each subsequent result

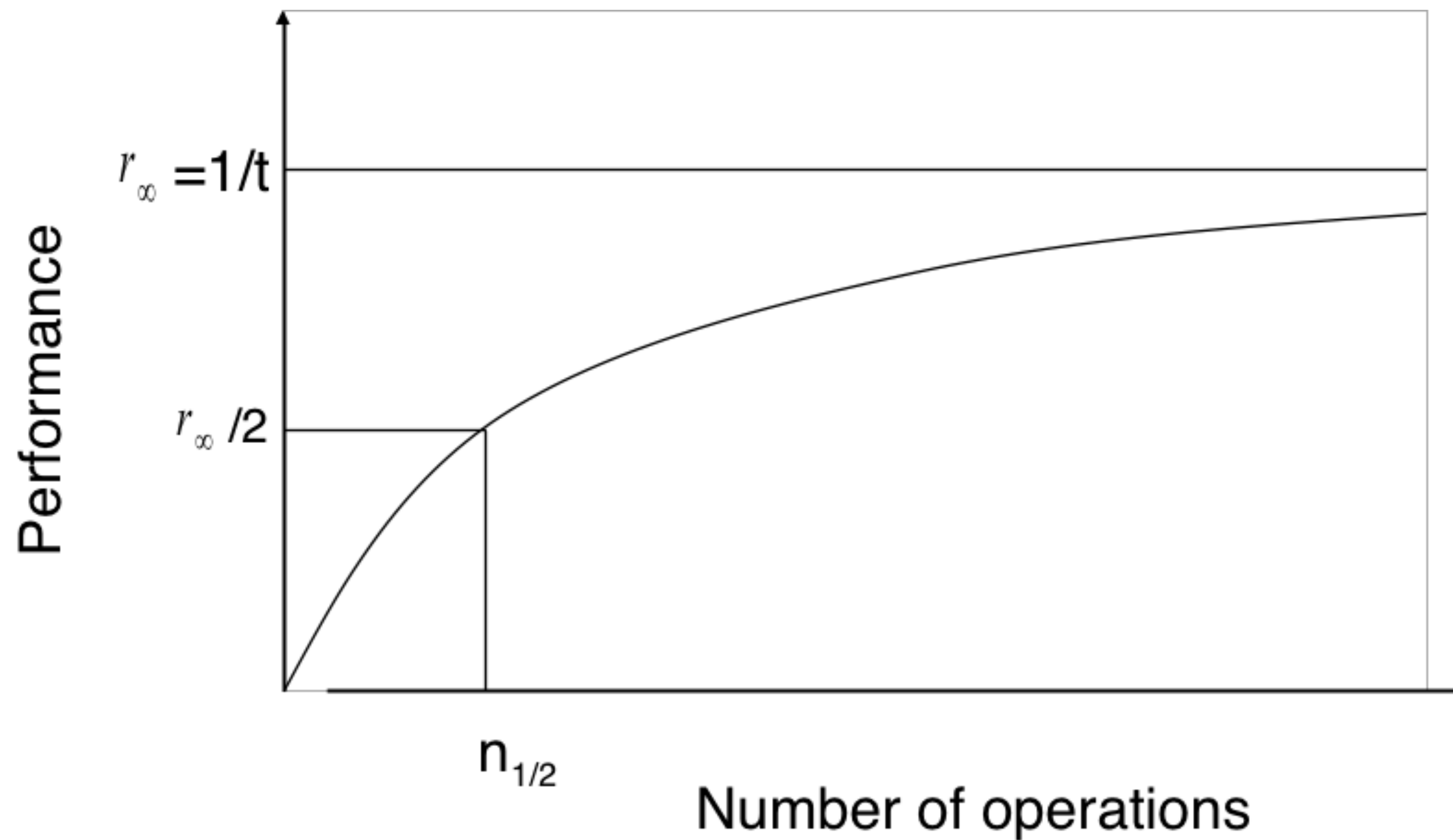
© Chris Jesshope 2008-2011, Raphael Poss 2011

Pipeline performance

- For a pipeline of length L and cycle time t, the time T it takes to process n operations is:
  $\mathbf{T(n) = L \cdot t + (n-1) \cdot t = (L-1) \cdot t + n \cdot t}$
- We can characterise all pipelines by two parameters:
  - **startup time**: $S = (L-1) \cdot t$ (unit: seconds)
  - **maximum rate**: $r_\infty = 1/t$ (unit: instructions per second)
- Also used:
  - **half-performance vector length** $n_{½}$ which verifies $T(n_{½})/2 = n_{½} \cdot r_\infty^{-1}$
    $\mathbf{n_{½} = L-1 = S/t}$ (unit: number of operations)

© Chris Jesshope 2008-2011, Raphael Poss 2011

Asymptotic performance



© Chris Jesshope 2008-2011, Raphael Poss 2011

Optimization strategies

- Long instruction sequences suggest IPC = 1
- However there are problems with this:
  - some instructions require less sub-tasks than others
  - hazards: dependencies and branches
  - long-latency operations: can’t fit the pipeline model
- What to do about these? The rest of the lecture covers this.

© Chris Jesshope 2008-2011, Raphael Poss 2011

Pipeline trade-offs

- Observations:
  - **more complexity** per sub-task requires **more time per cycle**
  - conversely, as the sub-tasks become simpler the cycle time can be reduced
  - so **to increase the clock rate** instructions must be **broken down into smaller sub-tasks**
  - ...but operations have a fixed complexity
  - **smaller sub-tasks mean deeper pipelines** = more stages $\Rightarrow$ more instructions need to be executed to fill the pipeline

© Chris Jesshope 2008-2011, Raphael Poss 2011