Disclaimer

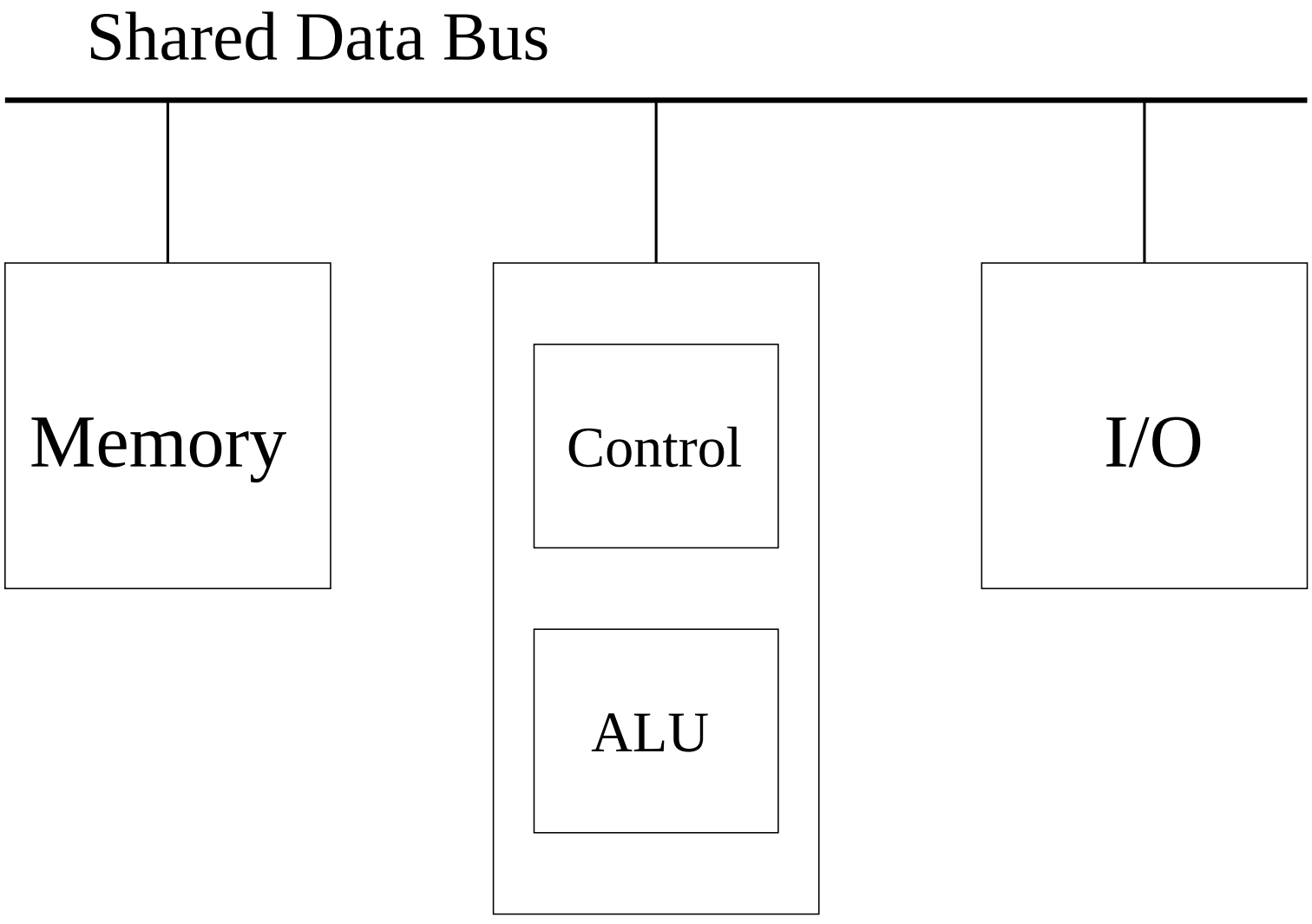
Disclaimer

Various images taken from Wikipedia or other online sources.
The bad ones are ones that I have done myself using xfig.

Von Neumann Architecture

- Classic architecture for computers
- Defined by John Von Neumann in two papers in 1945 and 1946
- The program code is treated exactly the same as the data it acts upon
- Both are stored in the same memory and accessed by the same CPU
- Modern computers don't quite follow it strictly
  - Interrupts, multiple busses, separate caches

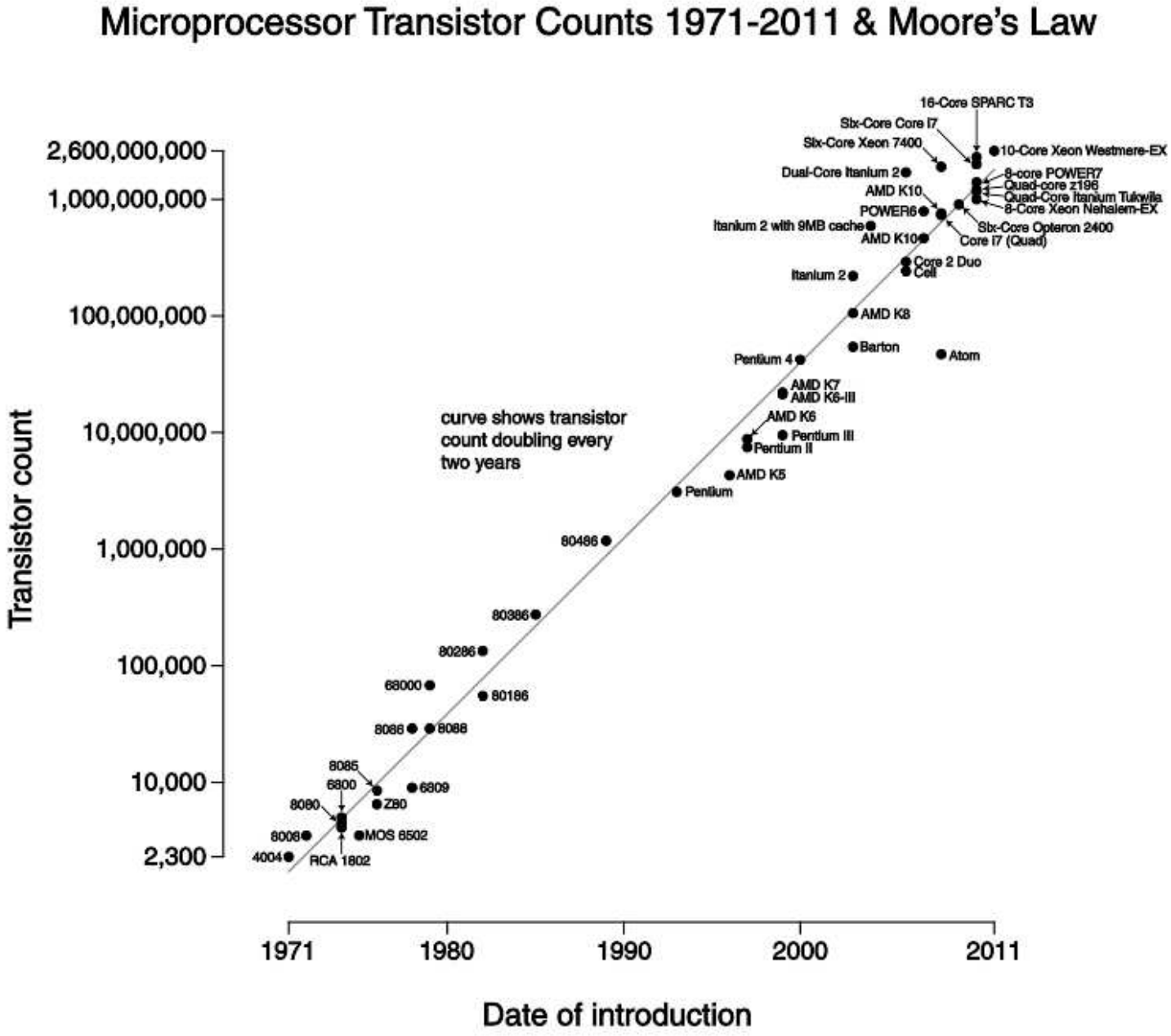
Von Neumann Architecture



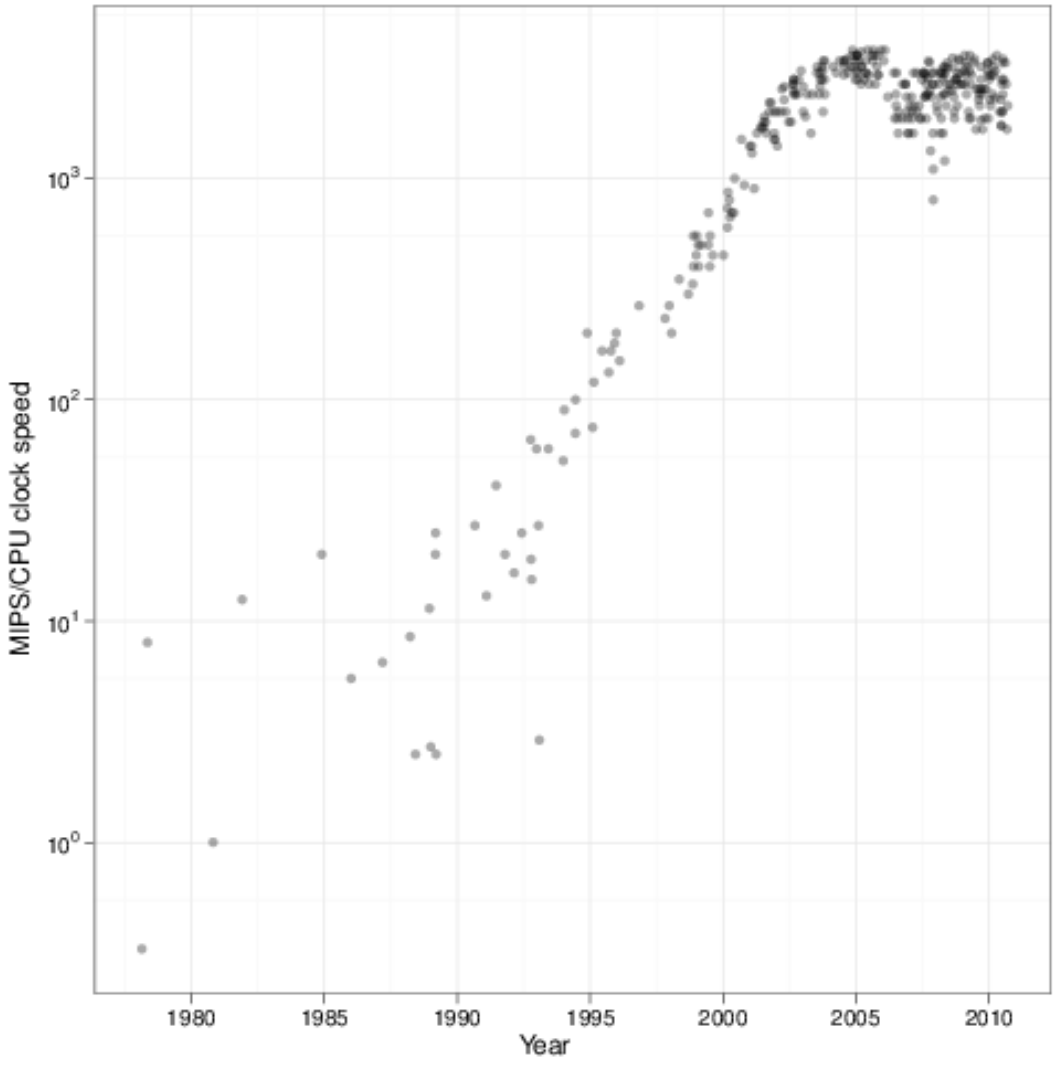
Von Neumann Problems

- Processor improvements have kept users happy
- Memory access time has not kept pace
- Von Neumann's architecture now has a serious bottleneck
- First identified by John Backus in 1977
- Luckily, processor performance is limited by the speed of light and quantum mechanics

Transistor Count



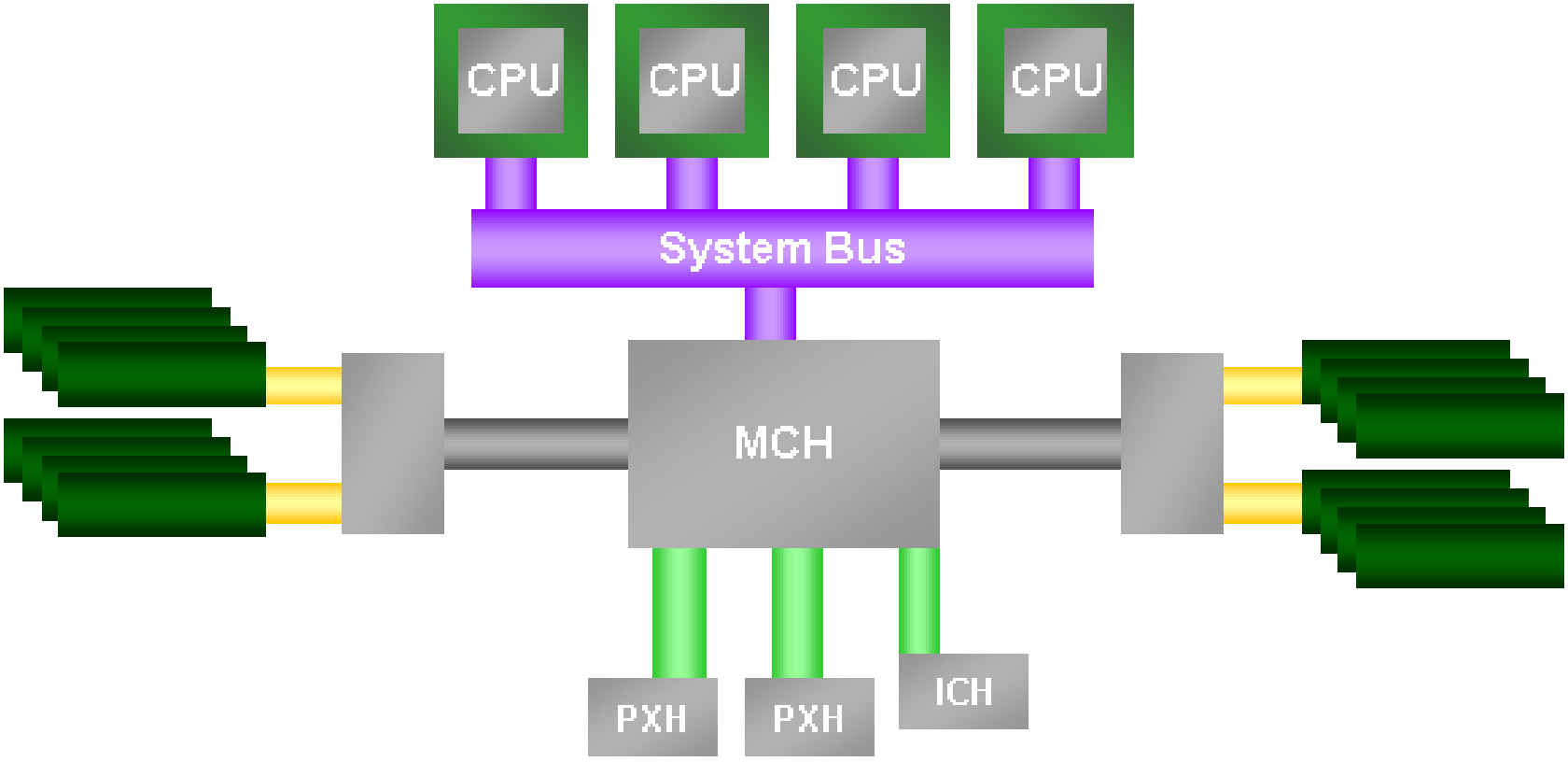
CPU Clock Speed



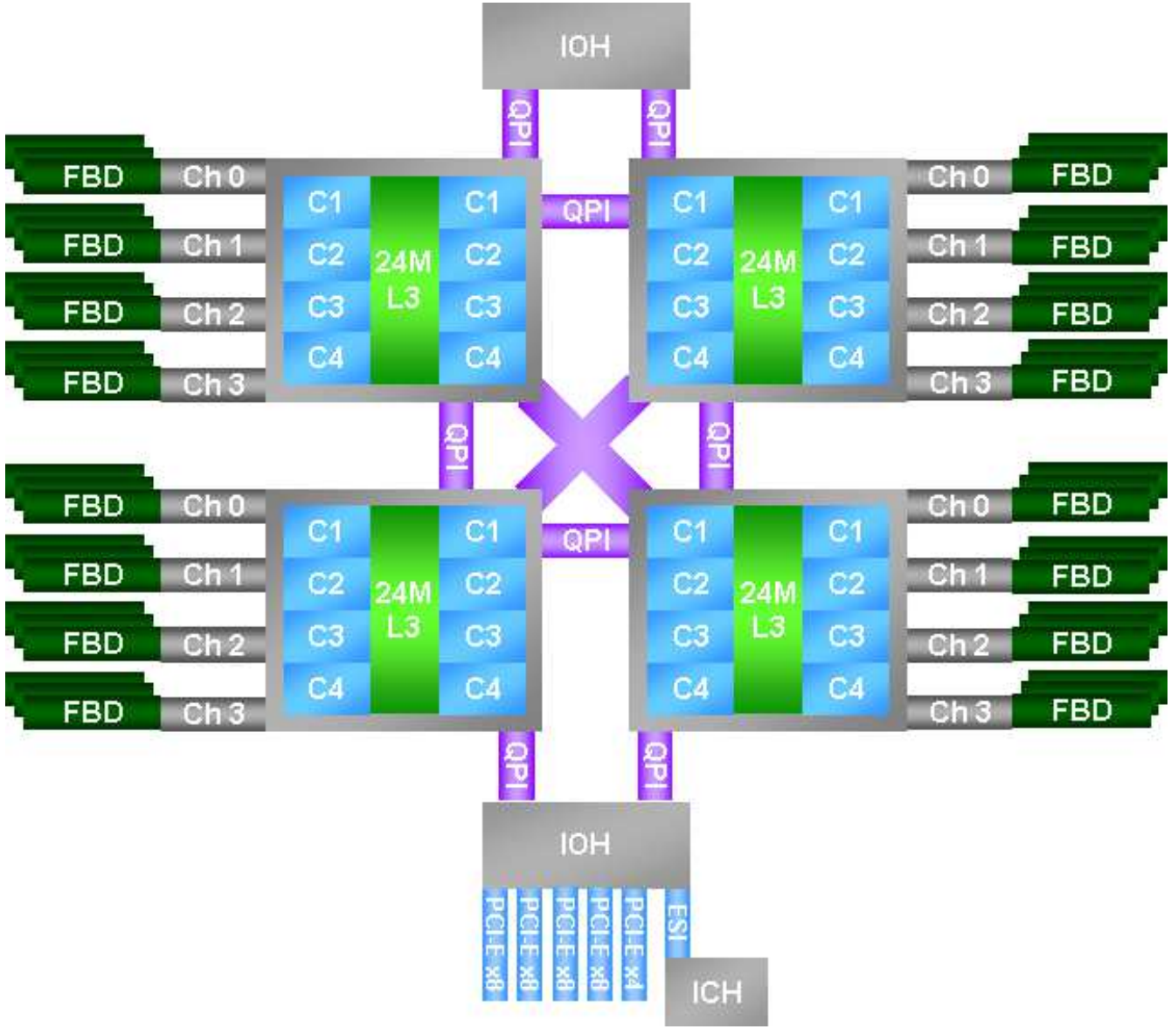
Tackling Von Neumann Problems

- Structure your code to reuse the same memory locations
- Different paths for instructions and data
- Provide a memory heirarchy with caches — NUMA

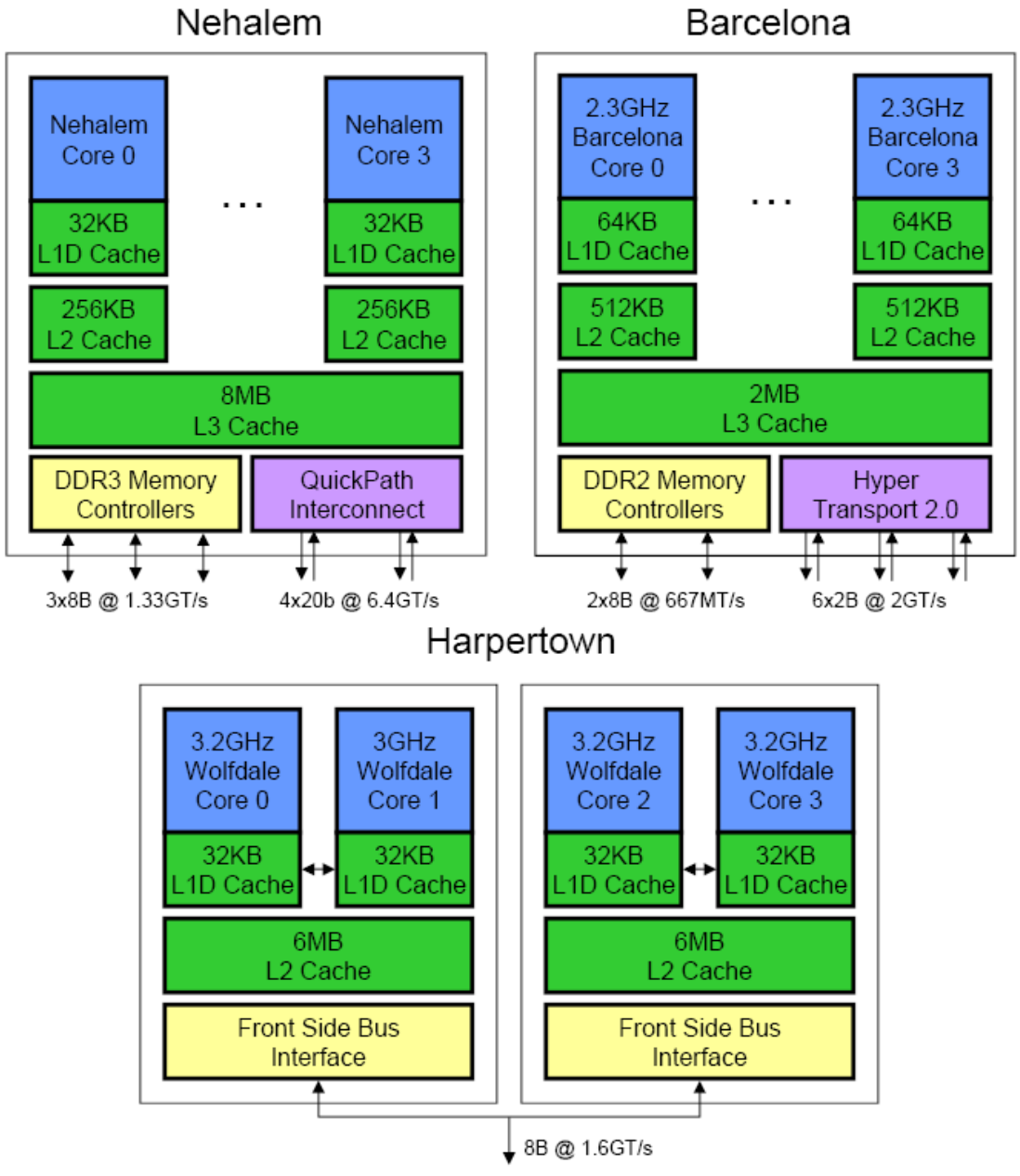
Older Architectures



Modern Architectures



Modern Architectures



Evolving Microarchitectures

- Intel arrange their microarchitectures in a two step (tick-tock) approach
- First iteration shrink the die size
- Second iteration introduce the new microarchitectures
- Now introducing a third optimization step to the cycle
- AMD follow a less aggressive roadmap - currently Excavator uses 28nm tech
- Zen microarchitecture due in 2017 at 14nm

	Name	Year	Size
Tock	Nehalem	2008	45nm
Tick	Westmere	2010	32nm
Tock	Sandy Bridge	2011	32nm
Tick	Ivy Bridge	2012	22nm
Tock	Haswell	2014	22nm
Tick	Broadwell	2015	14 nm

Instruction Sets

- An instruction set is the group of basic operations that a processor executes
- Sometimes referred to as primitives
- Represented by an op-code
- Can see the op-codes from your code by using gcc -S
- Each processor family has its own particular set of instructions
- Historically divided into CISC and RISC
- This boundary is quite blurred now

Sample opcodes

x86-64	ia64
func:	func
pushl %ebp	.prologue 2, 2
movl %esp, %ebp	.vframe r2
subl $52, %esp	mov r2 = r12
movl $0, -4(%ebp)	adds r12 = -48, r12
jmp .L2	.body
.L3:	;;
movl -4(%ebp), %edx	mov r14 = r2
movl -4(%ebp), %eax	;;
movl %eax, -44(%ebp,%edx,4)	st4 [r14] = r0
addl $1, -4(%ebp)	.L2:
.L2:	mov r14 = r2
cmpl $9, -4(%ebp)	;;
jle .L3	ld4 r14 = [r14]
leave	;;
ret	cmp4.ge p6, p7 = 9, r14

CISC Overview

- Complex Instruction Set Computer
- Small amounts of computer memory
- Bad compilers
- Assembly language was cool
- People liked complex instructions
- Complicated op-codes saved space
- User defined op-codes available through micro-code in the processor

CISC — Good or Evil?

- Lots of fancy instructions
- Lots of addressing modes for instructions
- Microcoded instructions
- Intuative as primatives available for most desired operations
- Fun to write assembly language for
- Hard to write good compilers for CISC
- Very complicated to design and implement in silicon
- Impossible to squeeze onto one chip — need co-processors
- 75% of instructions only used 5% of the time

RISC Overview

- Reduced Instruction Set Computer
- Lots of small, fast instructions will outperform a single, slow, complex one
- Even if it doesn't, most of the complex instructions aren't being used anyway
- Microcoding in CISCs was making pipelining hard to implement
- Improved compiler technology
- Chip design becomes much simpler allowing more real estate on chip for data registers

RISC Overview

- Uniform instruction length
  - SPARC — all 4 byte instructions
  - VAX — all sizes up to 56 byte instructions
- Load/Store architecture — no more addressing modes
  - No indirect addressing, using one memory location to get to another
  - No operations that combine memory with arithmetic

RISC Overview

- Hardwired instructions — no more microcode
- Good compilers with lots of optimisations
- Larger program sizes (due to lack of complex instructions)
- Space for on-chip floating point units and caches

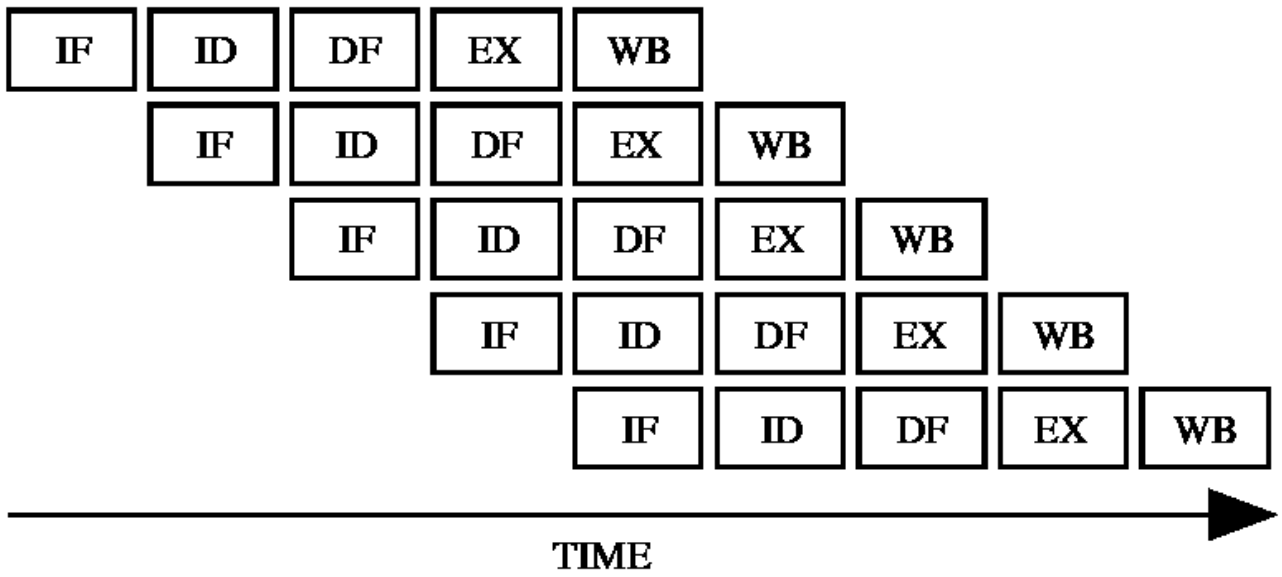
Processor Optimisations

- Pipelining
- Delayed Branch
- Branch Prediction
- Super-pipelining
- Speculative Computation
- Super-scalar

Pipelining

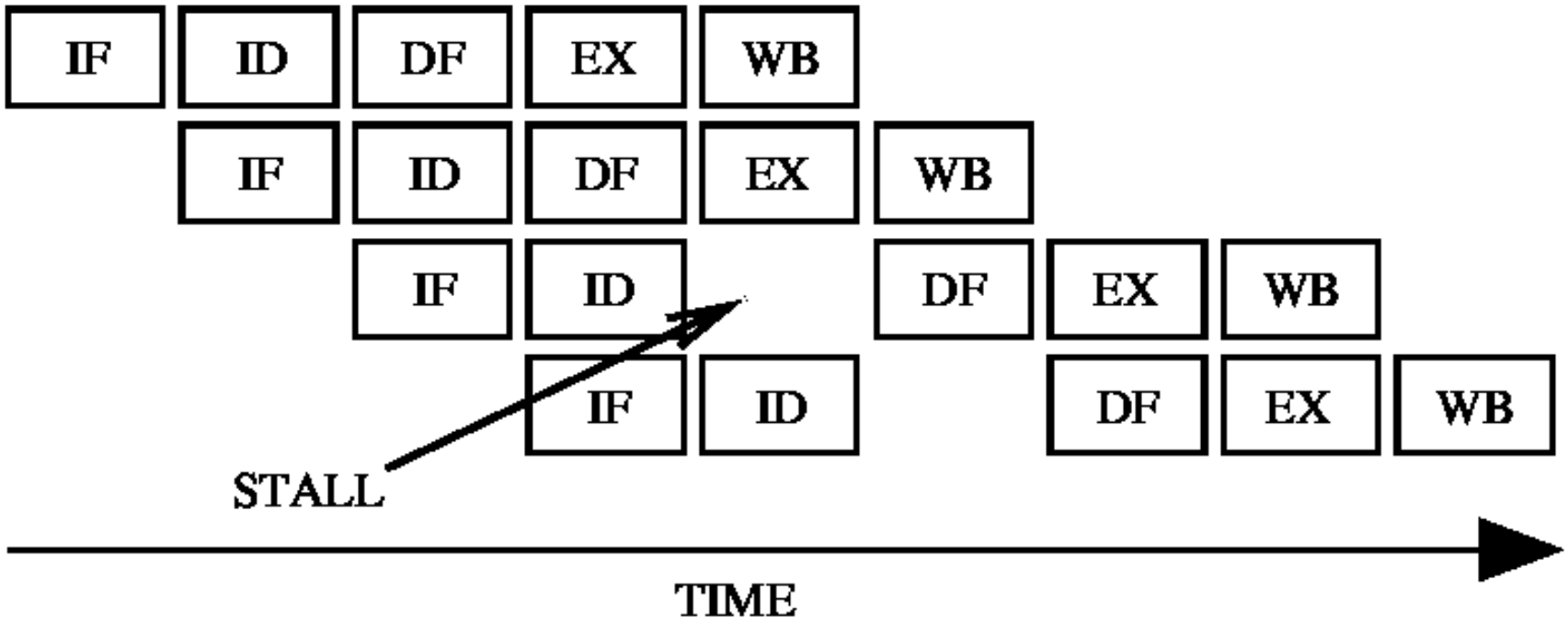
- Overlap of stages of execution of instructions
- Allows completion of one instruction every clock cycle
- The most common pipeline is the instruction pipeline
- Typically consists of five stages
  - Instruction Fetch (IF)
  - Instruction Decode (ID)
  - Data Fetch (DF)
  - Execute (EX)
  - Write Back (WB)

Pipelining


IF
ID
DF
EX
WB
IF
ID
DF
EX
WB
IF
ID
DF
EX
WB
IF
ID
DF
EX
WB
IF
ID
DF
EX
WB
TIME

Pipeline Stalls

- Stalls can occur due to jumps in execution or memory access
- Example: $A = B + C; D = B - C; E = A + B; C = D + B;$



Avoiding Stalls

- To avoid stalls the compiler will often re-order execution of instructions
- Must not change the overall result of the calculation
- Draw a data dependancy graph
- Read off new ordering of instructions which should minimize number of stalls
- Given a three stage pipeline (Fetch, Execute, Writeback) can we re-order the following instructions in a more efficient way?
- $x = y+z; y = p+q; p = x+z; z = p+q; x = x+y; q = y+p$

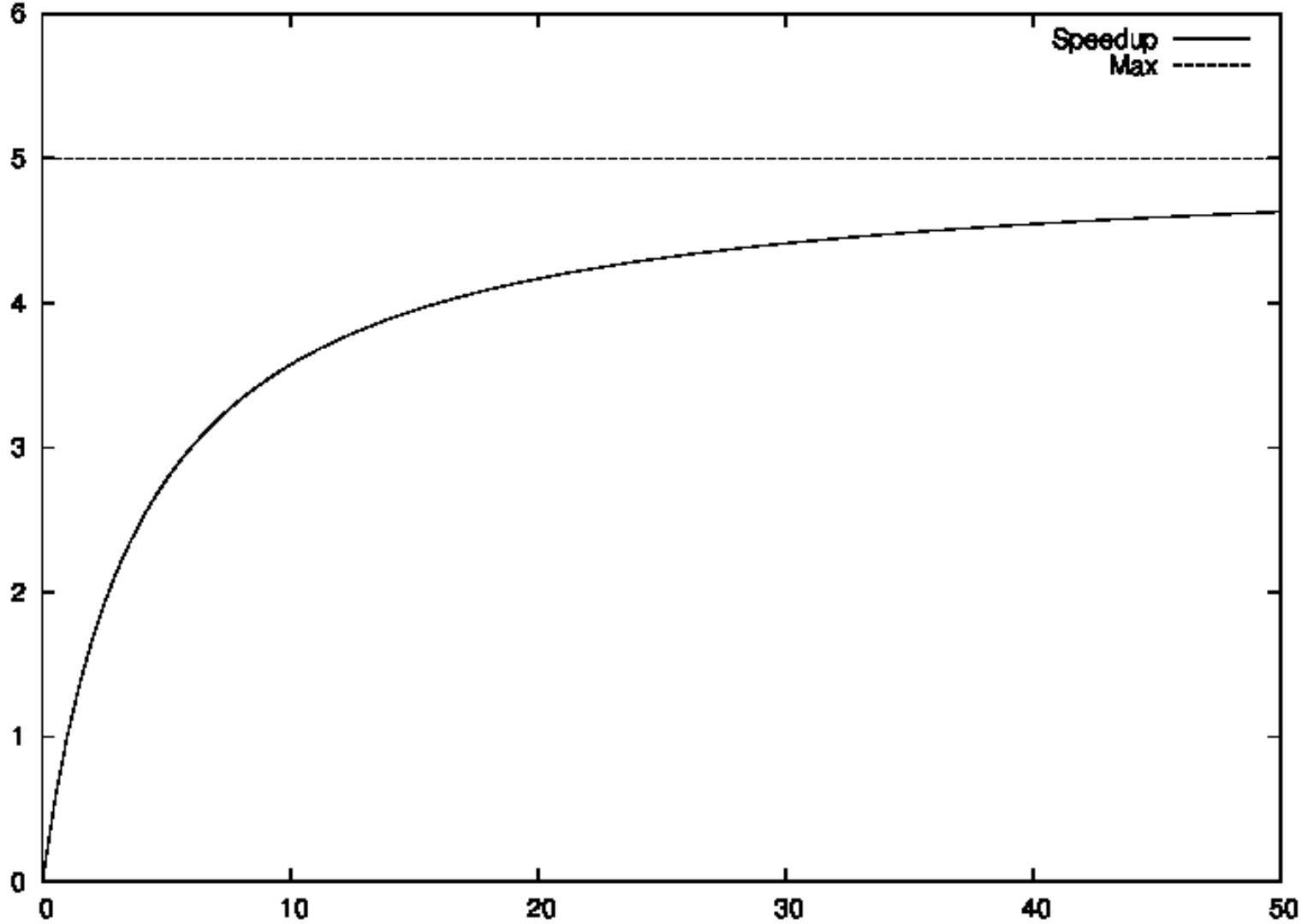
Data Hazards

- In pipelining hazards occur when sequential instructions cannot execute ever clock cycle
  - Data dependency
  - Antidependency
  - Output dependency
  - Procedural dependency
  - Resource conflicts
- Re-ordering or bubbling are the most usual solutions

Pipelining

- More complex operations are difficult to pipeline
- Ideally a pipeline can handle n tasks in $k + (n - 1)$ cycles
- A non-pipelined processor would take nk cycles
- This gives a substantial speed up

Pipeline Speed Up



Delayed Branch

- Instruction following a branch is always executed regardless of branch direction
- Used to avoid pipeline stalls
- Without optimisation compiler just inserts `nop`
- Not terrible effective with multiple execution units

```
.LL2:
        ld      [%fp-20], %g1
        cmp     %g1, 9
        bg      .LL3
        nop
        ld      [%fp-20], %g1
        add     %g1, 1, %g1
        st      %g1, [%fp-20]
        b       .LL2
        nop
.LL3:
        mov     %g1, %i0
        ret
        restore
```

```
        cmp     %g1, 9
        bg      .LL6
        add     %g1, 1, %g1
.LL13:
        cmp     %g1, 9
        ble     .LL13
        add     %g1, 1, %g1
.LL6:
        retl
```

Branch Prediction

- Use past history to predict which path execution will take
- Based on structure of programs — lots of loops
- Three simple schemes often used
  1. Always taken
  2. Never taken
  3. Backwards taken, forward not taken
- Prediction correct about 90% of the time so many stalls are avoided

```
sum = 0;
for(i=0;i<100;i++)
            sum += a[i]*b[i];
```

Superpipelining

- Superpipelining is when some stages in the pipeline are themselves further split
- Needs extremely high clocking speeds to be efficient
- Pipelines take longer to fill
- Not much use for general purpose execution
- Mainly found in older vector processors (Cray etc)

Speculative Computation

- Processing units take a long time to carry out certain operations (eg floating point division)
- If the unit has nothing else to do and the input data won't be changed then start performing the operation early
- If a branch is met and the divide path is taken then the result is ready early
- If the other path is chosen the result is thrown away but no compute time is really lost

Superscalars

- Superscalar processors simply have multiple processing units
- If data is independent then multiple instructions can be executed simultaneously
- Sometimes termed instruction-level parallelism