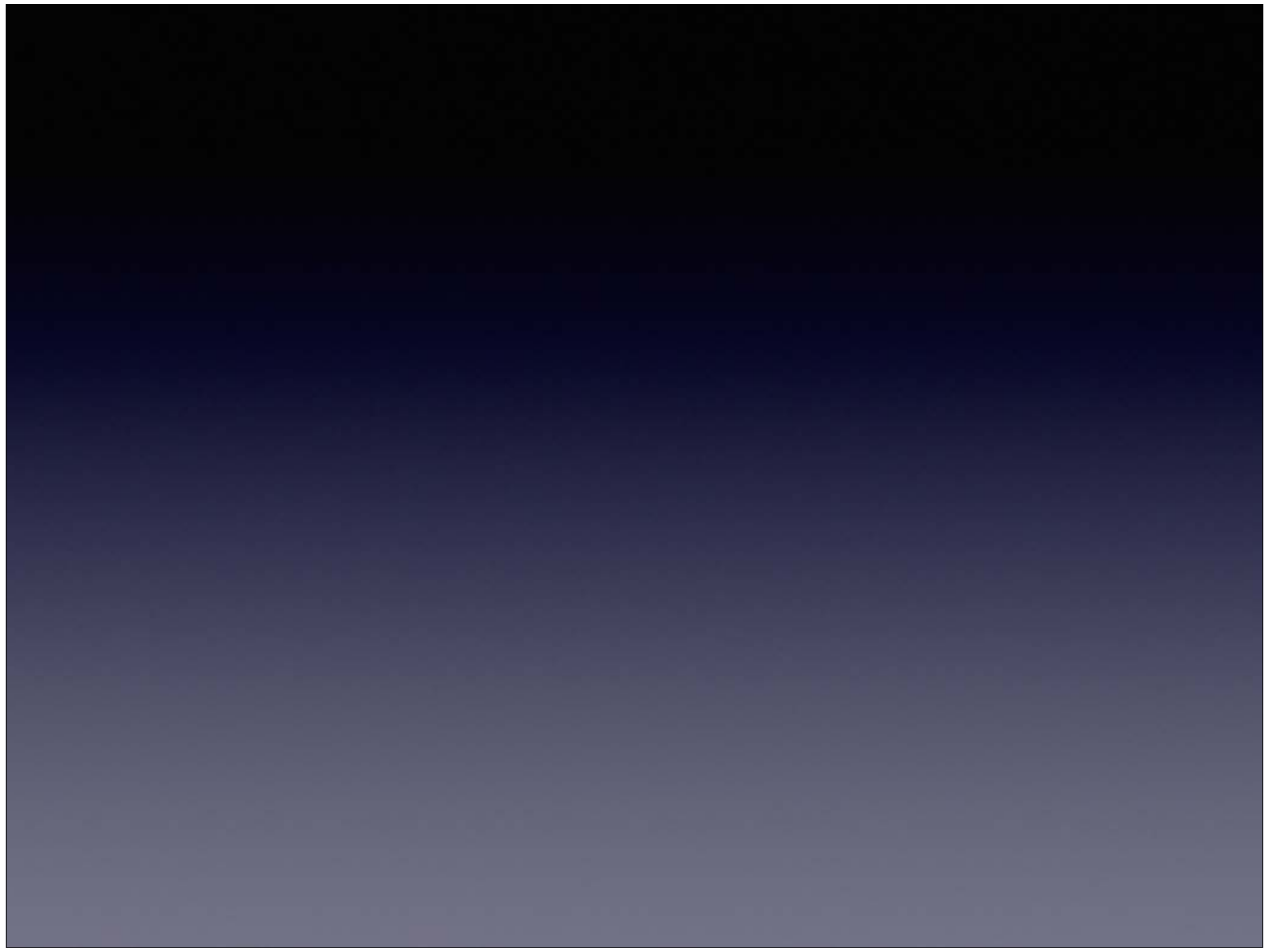
Central Processing Unit

Steven R. Bagley

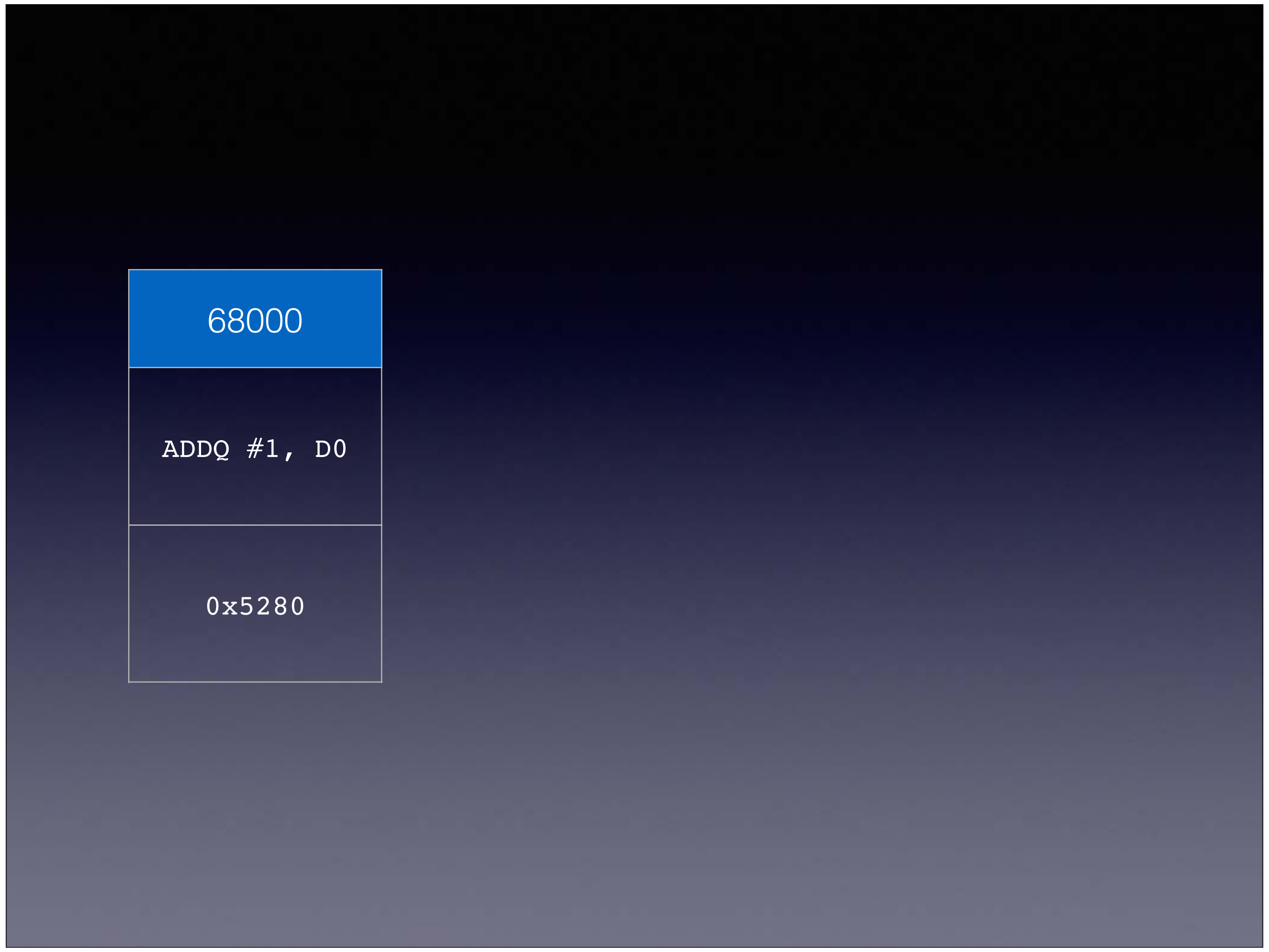
Introduction

- CPU gets instructions from the computer’s memory
- Each instruction is encoded as a binary pattern (an opcode)
- Each CPU family has its own *Instruction Set*
- Which are usually incompatible with other CPU families



4 different CPUs instruction to increment a register. Although in most cases not the only way you can do it.

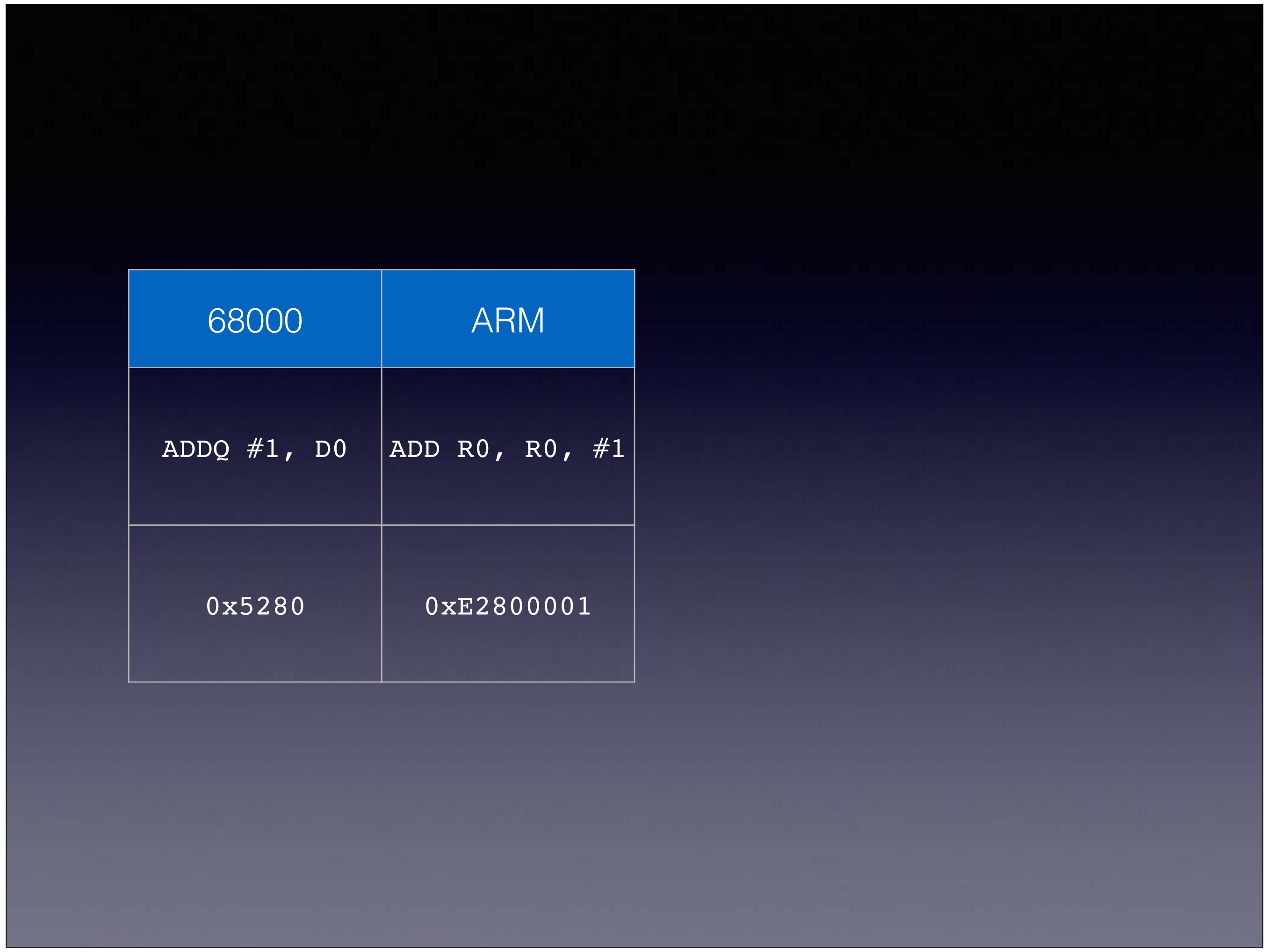
I’m not even going to attempt to work out what the value would be on the x86 — probably around 0x40



68000
`ADDQ #1, D0`
`0x5280`

4 different CPUs instruction to increment a register. Although in most cases not the only way you can do it.

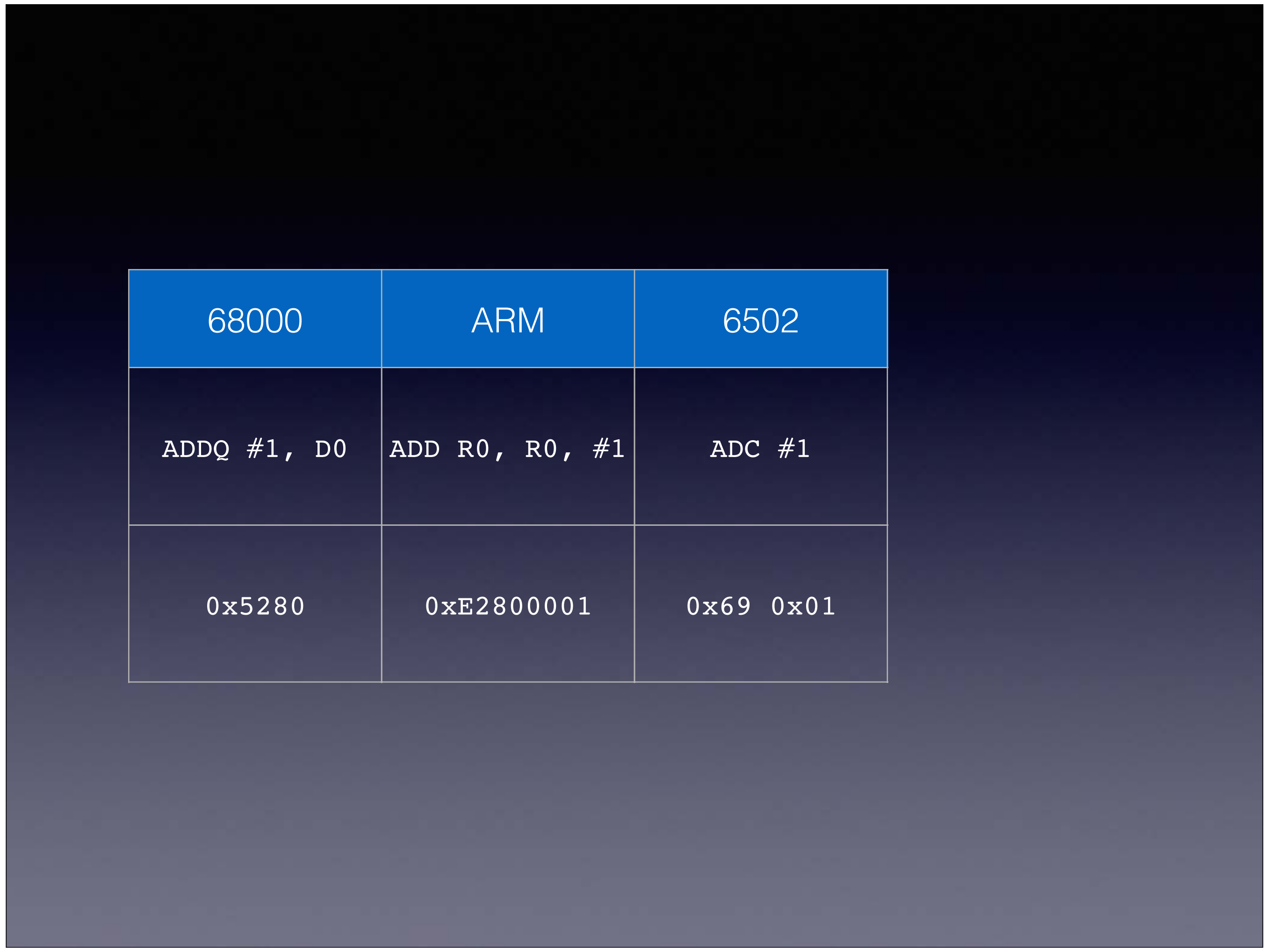
I’m not even going to attempt to work out what the value would be on the x86 — probably around 0x40



68000	ARM
`ADDQ #1, D0`	`ADD R0, R0, #1`
`0x5280`	`0xE2800001`

4 different CPUs instruction to increment a register. Although in most cases not the only way you can do it.

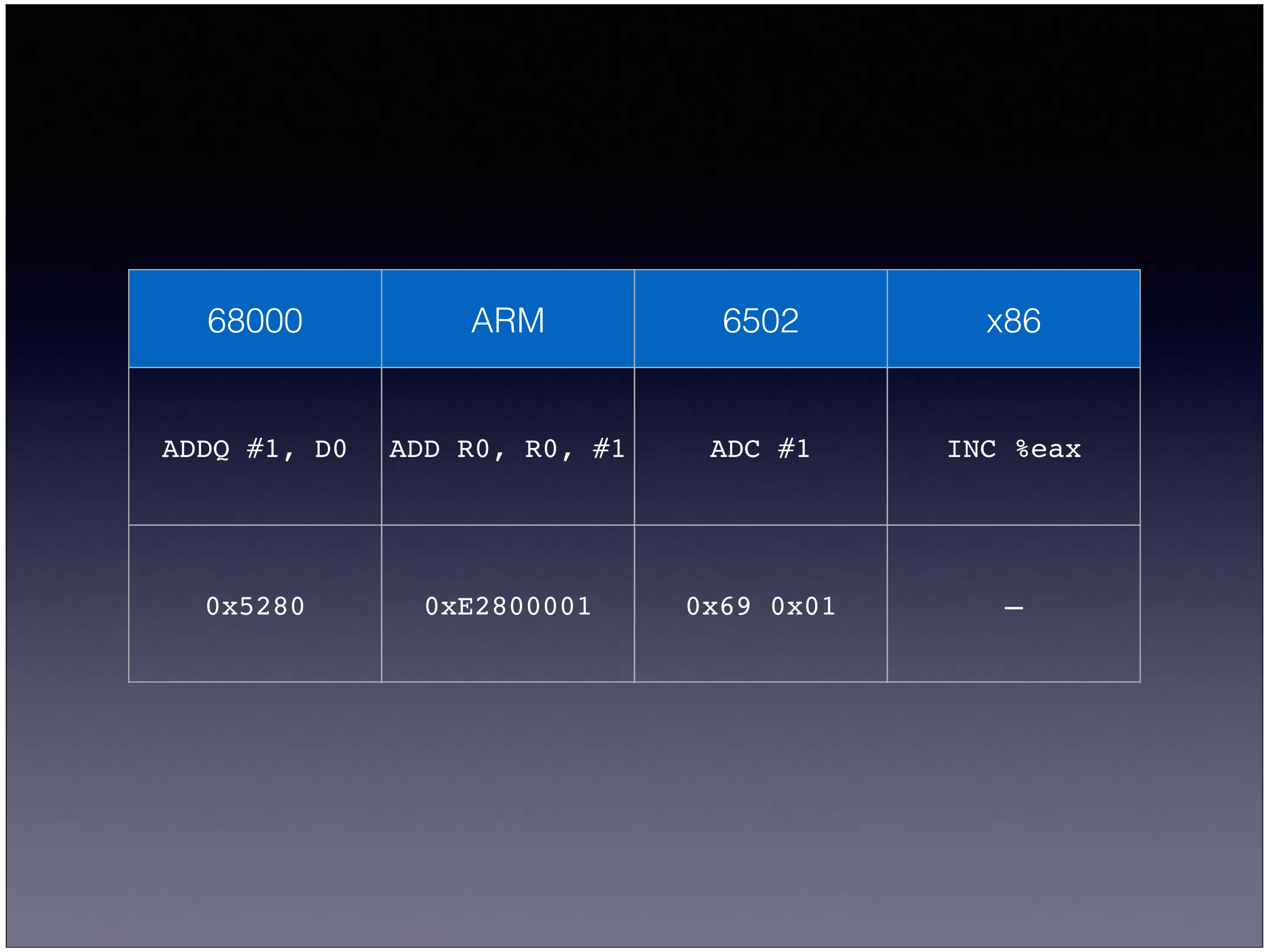
I'm not even going to attempt to work out what the value would be on the x86 — probably around 0x40



68000	ARM	6502
`ADDQ #1, D0`	`ADD R0, R0, #1`	`ADC #1`
`0x5280`	`0xE2800001`	`0x69 0x01`

4 different CPUs instruction to increment a register. Although in most cases not the only way you can do it.

I’m not even going to attempt to work out what the value would be on the x86 — probably around 0x40



68000	ARM	6502	x86
`ADDQ #1, D0`	`ADD R0, R0, #1`	`ADC #1`	`INC %eax`
`0x5280`	`0xE2800001`	`0x69 0x01`	–

4 different CPUs instruction to increment a register. Although in most cases not the only way you can do it.

I'm not even going to attempt to work out what the value would be on the x86 — probably around 0x40

CPU Instructions

- Opcode size varies with different Instruction Sets
- ARM ISA has opcodes that are exactly 32-bits wide
- x86 has a variable instruction width (1 to 16 bytes)
- Fixed instruction width makes it much easier to build a CPU
- x86 doesn’t know how long the instruction is until it starts decoding it…

Fetch - Decode - Execute

- CPU sits in a cycle of:
  - Fetching an instruction
  - Decoding the instruction
  - Executing it
    - Which may involve accessing memory again
- Next instruction then fetched

Program Counter

- Internally, the CPU has a register called the *Program Counter* (PC)
- Contains the address of the current/next instruction
- PC address placed on address bus to fetch instruction
- Then incremented to point at the next…

Registers

- CPUs have a number of registers
- A register stores a single value inside the CPU
- So faster to access than memory
- Most are accessible to the programmer, others are for internal use only…
- Number varies depending on the CPU…
- Some have special meaning (such as PC)
- Stored in the ‘Register File’ within the CPU

Registers

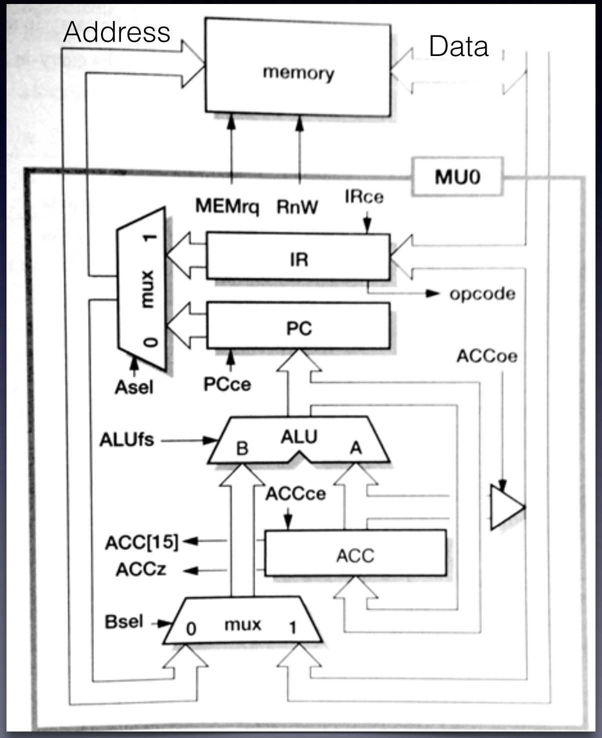
- 6502 had 3 general purpose 8-bit registers (`A`, `X`, `Y`), 2 specialised registers(`SP`, `P`) and a 16-bit `PC`
- 68000 has 17 32-bit registers (8 data registers `D0—7` and 8 address registers `A0—7`), a 16bit status register (`SR`) and a `PC`
- ARM has 31 32-bit registers (`R0—R15`) and a status register `CPSR`
- Some registers have special purposes…

ARM and 68000 both duplicate some registers, with specific versions accessible depending on the mode of the CPU

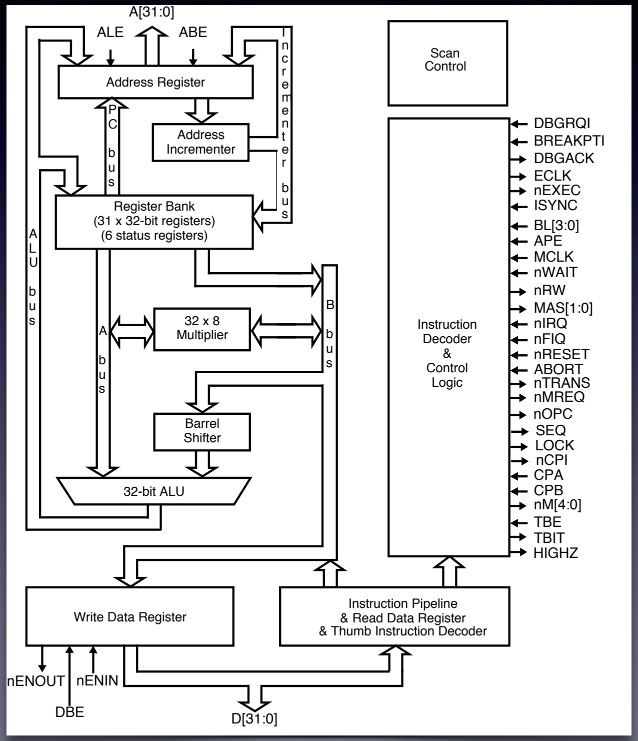
Goodness knows what the x86 has…

Some CPUs allow registers to be paired (e.g to store a 16bit value in two 8bit registers)

MU0 CPU internals



ARM CPU internals



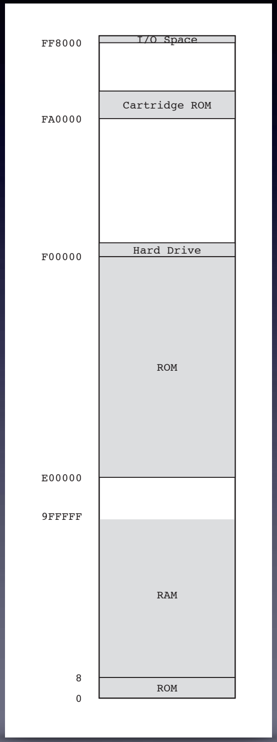
Reset

- CPU needs a defined start up sequence otherwise we wouldn’t be able to build a system
- Varies from CPU to CPU but usually involves starting to execute code at a known address
- Or looking at a specific memory location for the address to start at
- Build system around the CPU so that the correct value is there

Reset

- These days there is usually some ROM mapped into the memory space at that point
- Containing the start up code (e.g. the BIOS/uEFI in a PC)
- But some earlier machines had a set of switches you used to manually enter the boot up code

Example Memory Map



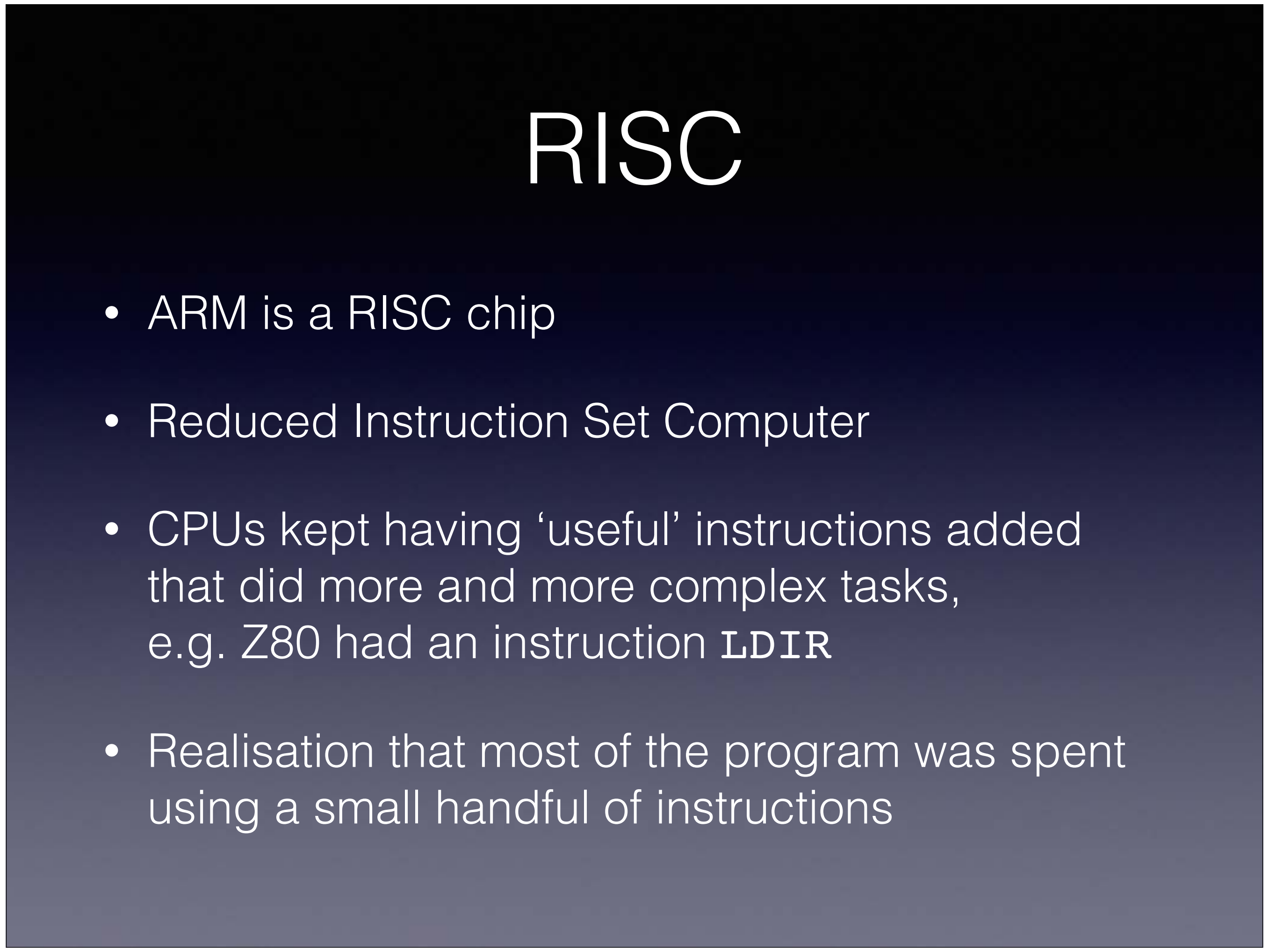
From Atari STe — not to scale!

Machine Code Instructions

- Machine code instructions tend to be very basic
- Load value into register
- Add value to register
- Store register to memory
- Jump to new location etc…

Machine Code Instructions

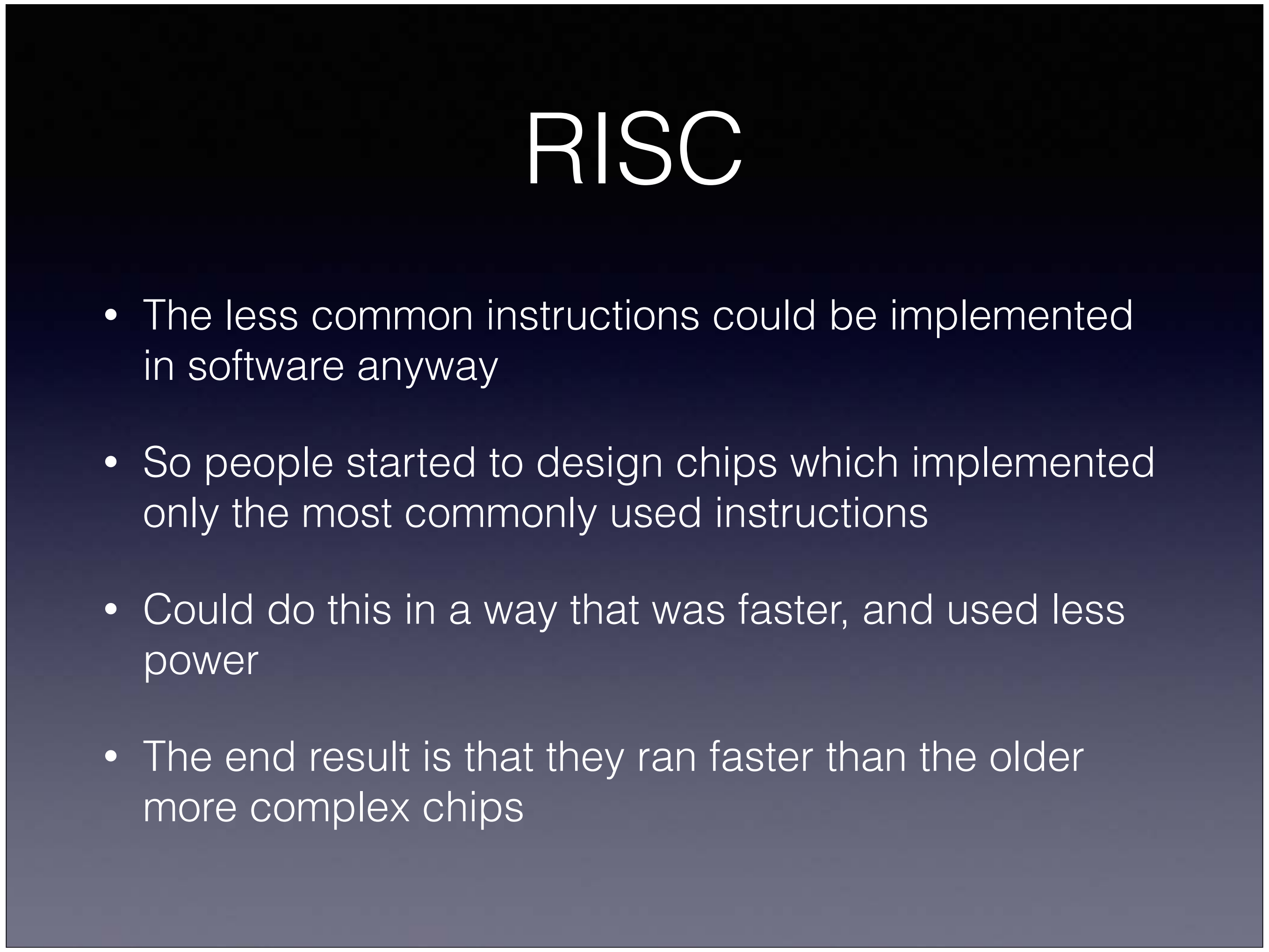
- We are going to look at writing ARM Machine Code
- Acorn Risc Machine was a CPU designed by Acorn in the 1980s
- Spun out from Acorn (with help from Apple) to form ARM (Advanced Risc Machines) in the early-1990s
- ARM chips now power most mobile devices…



RISC

- ARM is a RISC chip
- Reduced Instruction Set Computer
- CPUs kept having ‘useful’ instructions added that did more and more complex tasks, e.g. Z80 had an instruction `LDIR`
- Realisation that most of the program was spent using a small handful of instructions

LDIR (loads, decrements, increment and repeat) could copy a block of memory from one location to another. Useful, but also implementable in software

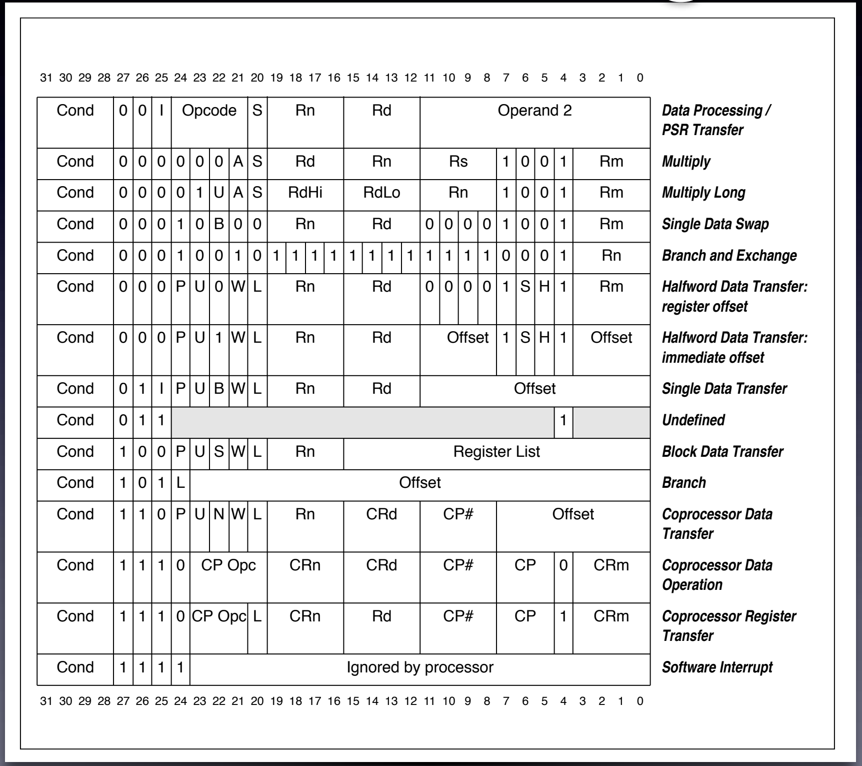


RISC

- The less common instructions could be implemented in software anyway
- So people started to design chips which implemented only the most commonly used instructions
- Could do this in a way that was faster, and used less power
- The end result is that they ran faster than the older more complex chips

Even though some tasks required many instructions

ARM instruction groups



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Cond				0	0	I	Opcode				S	Rn				Rd				Operand 2												*Data Processing / PSR Transfer*
Cond				0	0	0	0	0	0	A	S	Rd				Rn				Rs				1	0	0	1	Rm				*Multiply*
Cond				0	0	0	0	1	U	A	S	RdHi				RdLo				Rn				1	0	0	1	Rm				*Multiply Long*
Cond				0	0	0	1	0	B	0	0	Rn				Rd				0	0	0	0	1	0	0	1	Rm				*Single Data Swap*
Cond				0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	Rn				*Branch and Exchange*
Cond				0	0	0	P	U	0	W	L	Rn				Rd				0	0	0	0	1	S	H	1	Rm				*Halfword Data Transfer: register offset*
Cond				0	0	0	P	U	1	W	L	Rn				Rd				Offset				1	S	H	1	Offset				*Halfword Data Transfer: immediate offset*
Cond				0	1	I	P	U	B	W	L	Rn				Rd				Offset												*Single Data Transfer*
Cond				0	1	1	▨																				1	▨				*Undefined*
Cond				1	0	0	P	U	S	W	L	Rn				Register List																*Block Data Transfer*
Cond				1	0	1	L	Offset																								*Branch*
Cond				1	1	0	P	U	N	W	L	Rn				CRd				CP#				Offset								*Coprocessor Data Transfer*
Cond				1	1	1	0	CP Opc				CRn				CRd				CP#				CP			0	CRm				*Coprocessor Data Operation*
Cond				1	1	1	0	CP Opc			L	CRn				Rd				CP#				CP			1	CRm				*Coprocessor Register Transfer*
Cond				1	1	1	1	Ignored by processor																								*Software Interrupt*
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	

Shows the different groups of instructions offered by the ARM cpu. Also shows how they are mapped into the 32bit opcodes.

Assembly Language

- Hard to remember the exact bit-patterns that form each opcode
- Tend to use mnemonics that the human understands instead
- Called Assembly Language
- Then let an assembler convert the assembly into the relevant opcodes…

Assembly Example

```
        B fred

four    DEFW    4
success DEFB    "R0 has reached the value of \0"

        ALIGN

fred    LDR R1, four
        MOV R0, #1
next    CMP R0, R1
        BNE skip
        ADR R0, success
        SWI 3
        MOV R0,R1
        SWI 4
        MOV R0, #10
        SWI 0
        SWI 2

skip    ADD R0, R0, #1
        B next
```