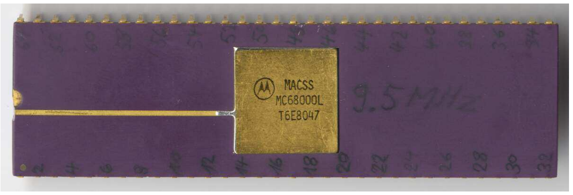
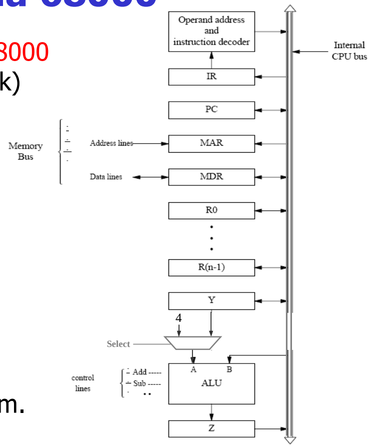
CPU Architectures

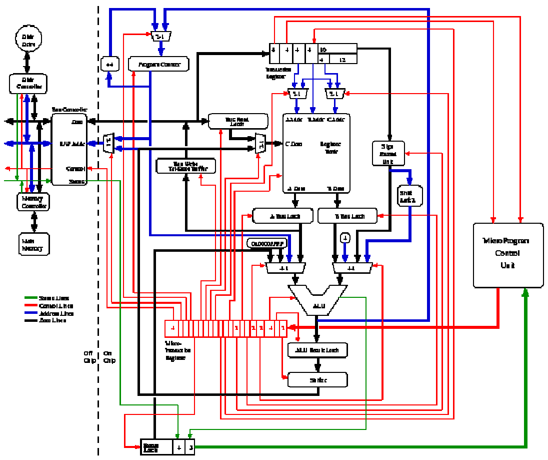
- Several CPU architectures exist currently:
  - Motorola
  - Intel
  - AMD (Advanced Micro Devices)
  - PowerPC
- Pick one to study; others will be variations on this.



Motorola 68000

- Arbitrary pick: Motorola 68000 (or m68k)
- CISC processor
  - translates instructions into microcode, and executes a sequence of micro-instructions on a RISC architecture.
- 32-bit architecture
  - all data is 32 bits long.
  - addresses are 24-bits, stored in 32-bit values.
- Datapath shown in diagram.





M68000 Data

- Data is stored in "big endian" format
  - most significant byte is stored in least significant segment of address. "Little endian" stores the least significant byte in the least significant portion of specified address.
  - Mnemonic: "big end in".
  - Example: Data value `4A3B2C1D` at memory location `0x1000`
    - Big endian:

1000	1001	1002	1003
4A	3B	2C	1D

    - Little endian:

1000	1001	1002	1003
1D	2C	3B	4A

- A word in M68000 is 16 bits (2 bytes), and a long word is 32 bits (4 bytes).
  - Most data and address accesses are long word operations.

Other M68000 Updates

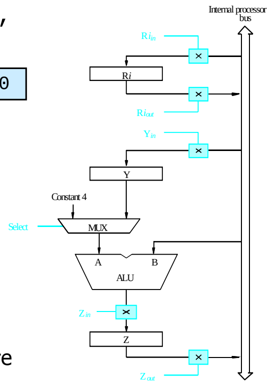
- MAR and MDR pretty much the same.
  - uses signal called MFC (Memory Function Completed) which is sent to the CPU after a memory operation is done.
- 8 data registers (R0 to R7) and 8 address registers (A0 to A7)
  - A7 is also known as SP, since it stores the stack pointer.
- Access to the components of the CPU (i.e. the arrows in the diagram) is controlled by a latched input.
  - The instruction decoder controls which latches are activated.
  - The operation performed by the ALU is also controlled by the instruction decoder.
  - The result of the ALU's operations (e.g. `C`, `V`, `N`) are conveyed back to the instruction decoder through a status register, usually located in the decoder itself.

How It All Works

- Basic operation of the M68000 involves performing one or more of the following functions:
  1. Fetching contents of a memory location and storing these contents in a CPU register.
  2. Storing the contents of a CPU register into a location in memory.
  3. Transferring the contents of one CPU register to another.
  4. Performing an arithmetic or logical operation and storing the result in a CPU register.
- Let's look at how to do some of these things…

Basic Control Signals

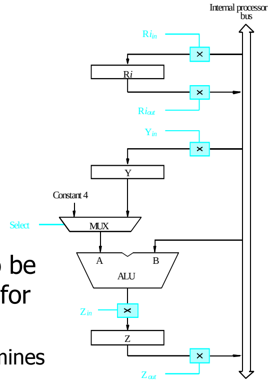
- "Add the contents of R0 and R1, and store the result in R0."
- Instruction form: `Add    R1, R0`



- Need to activate addition mode in the ALU, and get the contents of R0 and R1 into inputs A & B at the same time.
  - Problem: Single-bus architecture only allows one value on the bus.
  - Solution: Use the Y register to store a value temporarily.

Basic Control Signals

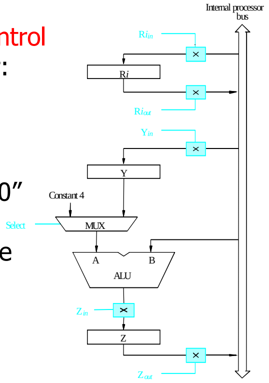
- To perform add operation, perform the following steps:
  1. get R1 (or R0) into Y
  2. get R0 (or R1) onto bus
  3. tell ALU to perform addition
  4. store result in Z
  5. put contents of Z onto bus
  6. store contents of bus in R0



- Of these steps, which have to be done simultaneously in order for the operation to succeed?
  - number of grouped steps determines the number of clock cycles per instruction.

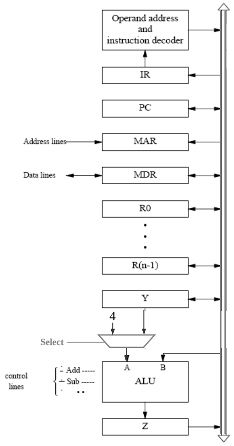
Basic Control Signals

- Translate grouped steps into control signals from instruction decoder:
  - Step 1: $R1_{out}$, Y_{in}
  - Step 2: $R0_{out}$, Select, Add, Z_{in}
  - Step 3: Z_{out}, $R0_{in}$
- Note: The operation "Add R1, R0" is like pseudocode for CPUs. We will use this for the next little while, but assembly language is typically used for implementing these instructions in real-life programs
  - e.g. `ADD.W D1, D0`



Basic Control Signals

- "Fetch the next instruction"
  - Required part of every instruction.
  - Two sets of steps needed…
- Operation #1: Fetch instruction
  1. Put PC value into MAR
  2. Fetch instruction from memory
  3. Store retrieved instruction in IR
- Operation #2: Increment PC
  1. Increment PC by 4
  2. Store incremented value back in PC
- Can these two operations be performed in parallel?



Basic Control Signals

- Translate operations into control signals:
  - Fetch instruction
    1. PC_{out}, MAR_{in}, Read, WMFC ← **PC** (original)
    2. MDR_{out}, IR_{in} ← **IR** (updated)
  - Increment PC
    1. PC_{out}, Add, Z_{in} ← **PC** (original)
    2. Z_{out}, PC_{in} ← **PC** (updated)
- To determine how these could be done in parallel, examine what signals are on the bus in each step…

1. PC_{out}, MAR_{in}, Read, Add, Z_{in}
2. Z_{out}, PC_{in}, WMFC
3. MDR_{out}, IR_{in}

Instruction Retrieval FAQ

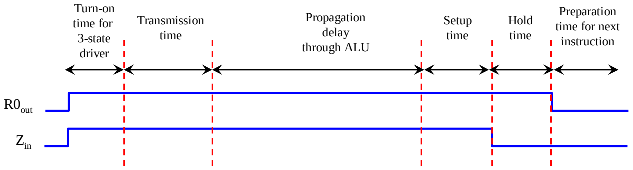
- What is the "Read" signal, exactly?
  - Sends signal to memory, requesting the data value stored at the MAR's address. When complete, the MDR will be storing the retrieved data.
  - "Write" is the only other memory operation, which stores the MDR's value into the memory location specified by the MAR.
- Why does WMFC get moved to the second step?
  - WMFC suspends CPU activity until the memory access is complete. Typically, this is much larger than any single CPU instruction, so get everything else done before you WMFC.
- How does the PC get incremented?
  - The mux either adds the Y value to the bus (if Select is high) or just adds 4 (if Select is low). Instructions are 32 bits long, and each memory location stores 1 byte. You do the math.

Instruction Retrieval Shortcuts

- Since this instruction retrieval step is so common, designers try to find ways to shorten the steps.
  - Memory access is always horribly slow, but that can't be helped. Advances in semiconductor devices and memory technology are needed for this (see Moore's Law).
  - PC increments offer potential design enhancements:
    - CPU designs sometimes create a dedicated increment circuit for the PC, so that it doesn't use the bus or involve the ALU.
    - If the PC stores the instruction address divided by 4, the PC can increment by one instead of 4 for each new instruction. The PC will then be multiplied by 4 before being copied to the MAR.
    - The PC and IR can have a direct connection to memory, much like the MAR and MDR.
    - Instruction memory can be made separate from main memory.

Impact on Timing

- The structure of the datapath has a direct influence on the clock speed of the CPU.
- Consider the delays involved in the Add operation:
  - Activation time for tri-state buffer on $R0_{out}$
  - Transmission time for $R0_{out}$ onto bus
  - Propagation delay through ALU
  - Setup time for loading value into Z register
  - Hold time for Z before lowering $R0_{out}$.


Turn-on time for 3-state driver
Transmission time
Propagation delay through ALU
Setup time
Hold time
Preparation time for next instruction
$R0_{out}$
Z_{in}

Addressing Modes

- While we're talking about the Add operation, how many ways are there to specify operands?
  - Register direct mode = what we've seen so far.

Add R6, R0

  - Register indirect mode = the register stores the memory address of the operand.

Add (A0), R0

Addressing Modes

- Autoincrement mode = a version of register indirect mode, where the address stored in the register is incremented by 1, 2 or 4 after the memory access, depending on the length of the data being accessed.

Add (A0)+, R0

- Autodecrement mode = same as autoincrement, only the address is decremented before the memory access.

Add -(A0), R0

Addressing Modes

- Immediate mode = the value of the operand is given in the instruction itself. The '#' character denotes an immediate value.

```
Add    #200, R0
```

- Absolute mode = the absolute address to be used in the operation is found in the instruction itself. The '$' character denotes that the address is in hexadecimal notation (0x…).
  - Absolute value can be either short or long. If long, then 24 bits are given. If short, 16 bits are given and then sign extended.

```
Jmp    $1200
```

Addressing Modes

- Basic index mode = a 16-bit signed offset and an address register are specified in the instruction. The sum of the offset and the register's contents specifies the operand location.

```
Add    42(A0), R0
```

- Full index mode = an 8-bit signed offset, an address register and a data register are specified in the instruction. The sum of the offset and the two registers specifies the operand location.

```
Add    53(A0, R1), R0
```

Addressing Modes

- Basic relative mode = same as basic index mode, only using the program counter instead of an address register.

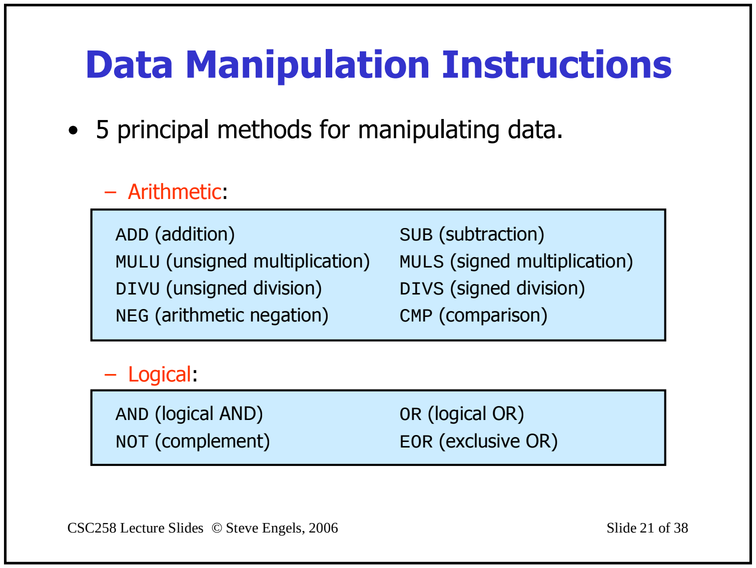
```
Add    42(PC), R0
```

- Full relative mode = same as full index mode, only using the program counter instead of an address register.

```
Add    53(PC, R1), R0
```

Instruction Types

- Now that we've specified the different forms that operands can take, what about the operations?
- Operations for the M68000 fall into one of three major categories:
  - Data manipulation (arithmetic, shifting, copying)
  - Control flow (branching, jumping, subroutines)



Data Manipulation Instructions

- 5 principal methods for manipulating data.
  - Arithmetic:

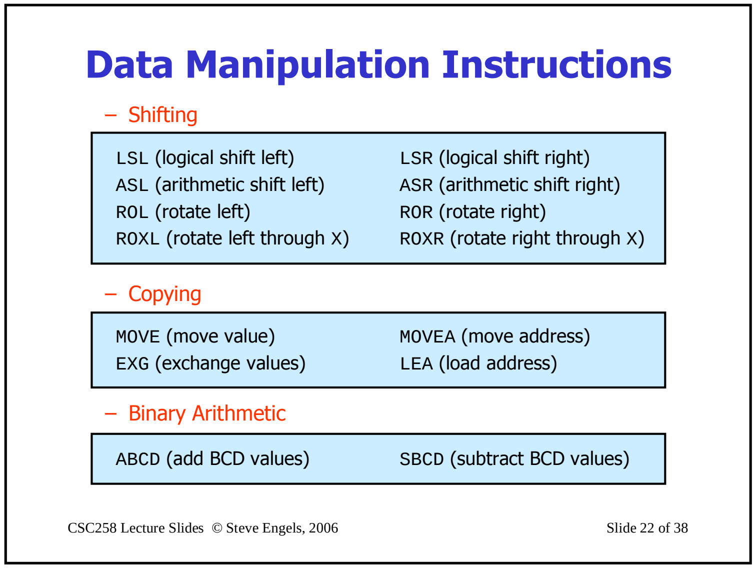
ADD (addition)
MULU (unsigned multiplication)
DIVU (unsigned division)
NEG (arithmetic negation)

SUB (subtraction)
MULS (signed multiplication)
DIVS (signed division)
CMP (comparison)

  - Logical:

AND (logical AND)
NOT (complement)

OR (logical OR)
EOR (exclusive OR)



Data Manipulation Instructions

- Shifting

LSL (logical shift left)
ASL (arithmetic shift left)
ROL (rotate left)
ROXL (rotate left through X)

LSR (logical shift right)
ASR (arithmetic shift right)
ROR (rotate right)
ROXR (rotate right through X)

- Copying

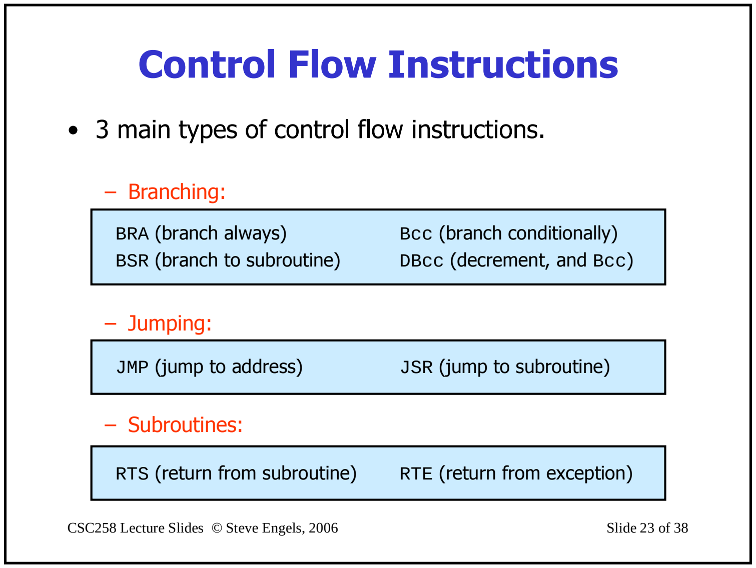
MOVE (move value)
EXG (exchange values)

MOVEA (move address)
LEA (load address)

- Binary Arithmetic

ABCD (add BCD values)

SBCD (subtract BCD values)



Control Flow Instructions

- 3 main types of control flow instructions.
  - Branching:

BRA (branch always)
BSR (branch to subroutine)

Bcc (branch conditionally)
DBcc (decrement, and Bcc)

  - Jumping:

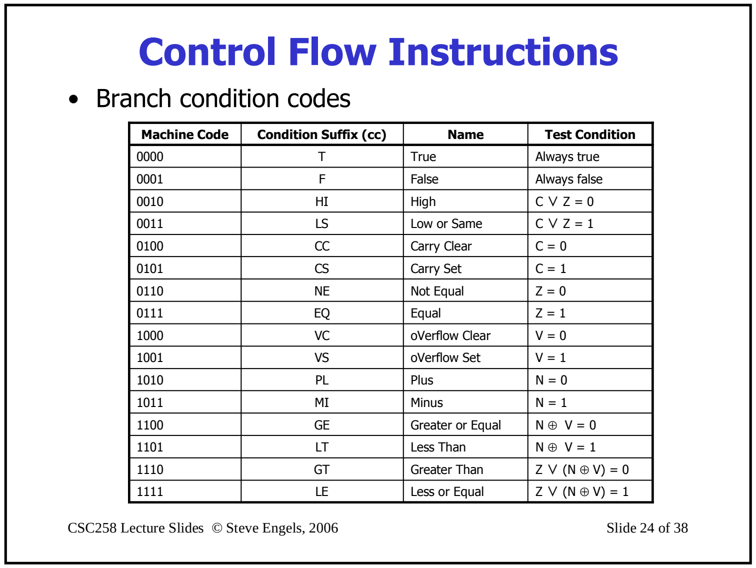
JMP (jump to address)

JSR (jump to subroutine)

  - Subroutines:

RTS (return from subroutine)

RTE (return from exception)



Control Flow Instructions

- Branch condition codes

Machine Code	Condition Suffix (cc)	Name	Test Condition
0000	T	True	Always true
0001	F	False	Always false
0010	HI	High	C ∨ Z = 0
0011	LS	Low or Same	C ∨ Z = 1
0100	CC	Carry Clear	C = 0
0101	CS	Carry Set	C = 1
0110	NE	Not Equal	Z = 0
0111	EQ	Equal	Z = 1
1000	VC	oVerflow Clear	V = 0
1001	VS	oVerflow Set	V = 1
1010	PL	Plus	N = 0
1011	MI	Minus	N = 1
1100	GE	Greater or Equal	N ⊕ V = 0
1101	LT	Less Than	N ⊕ V = 1
1110	GT	Greater Than	Z ∨ (N ⊕ V) = 0
1111	LE	Less or Equal	Z ∨ (N ⊕ V) = 1

Data Details

- When moving or modifying data, it's important to specify the length of the data being moved or modified.
- Example: `move.b (a0), r1`
  - This moves the byte at `a0`'s location into the least significant byte of `r1`.
  - It's important to consider the endian of the system you're using (see slide 4).
- Also, register names can vary from system to system.
  - M68000 often uses d0-d7 to represent data registers, e.g.
  - Don't get confused.

Other Important Commands

- There are many more commands than the ones listed here, but most are rather obscure and rarely used.
- A few important ones commands that were missed:

  - `dc` (declare constant) = allocated space for a constant value, which can be referred to in place of a data value.
    - Example:

```
value:      dc.b      42
            move.w    value, r0
```

    - In this case, `value` is like a global variable to the program.

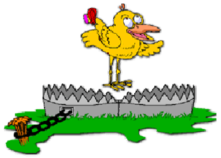
  - `org` (program origin) = set a location in memory for the following assembly instructions should be written.
    - Example: `org $1000`
    - This starts the following code at location `1000` (hex) in memory.

Other Important Commands

- `trap` (software interrupt) = call an interrupt subroutine, such as printing to the screen, or receiving data from standard input.
  - Example: `trap #14`
  - This calls the subroutine that goes to trap vector #14, which loads whatever address is at memory location 14, and jumps the program counter to that address (usually in read-only memory).
  - Often, the program that resides at the address in vector #14 is responsible for printing characters to the screen (m68k only).
  - Before jumping, the current PC and SP values are pushed onto the stack. The `RTE` instruction pops these value back.
  - Other trap locations will be responsible for other operations (e.g. handling misaligned word addresses, reading from keyboard, etc.)

Other `Trap` Details

- The first 256 long words in memory store the memory addresses of vital processor subroutines.
  - address 0 contains the address of the initial startup routine.
- This is also called the exception vector table.
- Locations 32-47 of this table are allocated to the trap vectors, and the contents of these entries can be set by the operating system (which you create).
- Calling `Trap #14` would then look up entry 46 in the exception table, go to the address specified in that entry, and execute the subroutine at that address.



Memory-Mapped I/O

- The trap entries in the exception table contain the address of the exception code. How does that exception code actually read or write from external devices?
- The trap code assumes that the information to write is in a particular register, or that the information to read should be stored in a particular data register.
  - In the case of reading or writing strings, an address register is used instead to store the initial address of the string. A null character is used to indicate that the string has terminated.
- The actual address of the device's IO location can vary, depending on the hardware implementation.

Memory-Mapped I/O

- When performing read or write operations, we're typically interested in the parallel or serial ports.
  - Printer/monitor → parallel
  - Keyboard/internet → serial
- To perform these actions, the M68xxx systems either map specific devices to location in memory, or will implement a multi-function peripheral (MFP) interface.
- The MFP interface is a series of registers in memory that are responsible for the various aspects of I/O operations:
  - Data direction
  - Control register
  - Receiver/transmitter status register
  - Data register

Multi-Function Peripherals

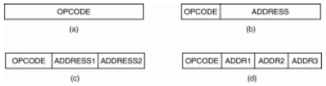
- Each register in the MFP interface is assumed to be an odd offset from a given location in memory.
  - Let's assume that it's at `$E70000` (various sources give different values for this starting address)
    - General-Purpose I/O Register (GPIP) → offset of 1
    - Universal Synchronous/Asynchronous Receiver/Transmitter (USART) Control Register (USC) → offset of 0x29
    - Receiver Status Register (RSR) → offset of 0x2B
    - Transmitter Status Register (TSR) → offset of 0x2D
    - USART Data Register (USR) → offset of 0x2F
  - To perform an I/O operation, use the control registers to set up the nature of the operation (synchronous, byte/word/long), and then use the data and status registers to send information and verify it was received.

Opcodes

- When an assembly language program is compiled by the assembler, it is translated into binary instructions that can be recorded in memory and understood by the instruction decoder.
  - Also called opcodes (short for operation codes)
- How does it do this translation? What do these binary instructions look like?
- Each instruction can be broken down into parts,
  - to see what assembly language instruction it corresponds to
  - to see how the instruction decoder understands its contents

Opcodes

- Note: See Appedix C in the text and the M68000 reference manual on the web page for opcode tables and more opcode information.



- The opcode is broken down into sections differently, depending on the type of instruction.
- In order to know how to break down the instruction, the first section of the instruction contains the identifier for the instruction type.
  - most significant 4 bits (15-12) of a 16-bit instruction.

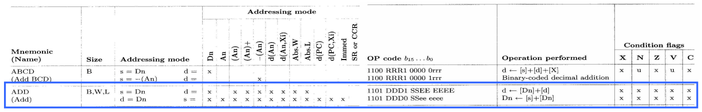
Opcodes

- The instruction decoder reads the first 4 bits, and knows more about how to handle the instruction from there.
  - Example: 0010 → Move Long
- Sometimes, a fifth bit is needed to distinguish instructions within the general categories.
  - Example: 1011
    - if bit 8 is 0 → CMP
    - if bit 8 is 1 → EOR

Bits 15-12	Operation
0000	Bit manipulation/MOVEP/Immed
0001	Move Byte
0010	Move Long
0011	Move Word
0100	Miscellaneous
0101	ADDQ/SUBQ/Scc/DBcc/TRAP
0110	Bcc/BSR/BRA
0111	MOVEQ
1000	OR/DIV/SBCD
1001	SUB/SUBX
1010	(Unassigned, Reserved)
1011	CMP/EOR
1100	AND/MUL/ABCD/EXG
1101	ADD/ADDX
1110	Shift/Rotate/Bit Field
1111	MC68040 & CPU32 Extensions

Opcodes

- After the instruction has been identified, the remaining 16 bits can be interpreted according to the tables in Appendix C from the text.
- Example: ADD



			Addressing mode																		
																	Condition flags				
Mnemonic (Name)	Size	Addressing mode	Dn	An	(An)	(An)+	−(An)	d(An)	d(An,Xi)	Abs.W	Abs.L	d(PC)	d(PC,Xi)	Immed	SR or CCR	OP code $b_{15} \ldots b_0$	Operation performed	X	N	Z	V	C
ABCD	B	s = Dn d =	x													1100 RRR1 0000 0rrr	d ← [s]+[d]+[X]	x	u	x	u	x
(Add BCD)		s = −(An) d =					x									1100 RRR1 0000 1rrr	Binary-coded decimal addition					
ADD	B,W,L	s = Dn d =	x		x	x	x	x	x	x	x					1101 DDD1 SSEE EEEE	d ← [Dn]+[d]	x	x	x	x	x
(Add)		d = Dn s =	x	x	x	x	x	x	x	x	x	x	x	x		1101 DDD0 SSee eeee	Dn ← [s]+[Dn]	x	x	x	x	x

- This table entry specifies the operand types and sizes for this instruction, the format for the opcode, the underlying semantics of the instruction, and the status register codes after the instruction takes place.

Opcodes

- Consider the first opcode format for this instruction:

```
1101 DDD1 SS EEEEEE
```

- This instruction corresponds to: `ADD Dn, X`
  - `Dn` is a data register (`n` is a value from 0 to 7)
  - `X` is the other operand, which also acts as the destination
- What do the letter fields mean?
  - `DDD` = Data register number (i.e. `Dn`)
  - `SS` = Size (`00` → byte, `01` → word, `10` → long word)
  - `EEEEEE` = Effective address of destination operand (i.e. `X`)
    - First three bits of `EEEEEE` → addressing mode
    - Second three bits of `EEEEEE` → register number (if applicable)

Opcodes

- So given the format
  `1101 DDD1 SS EEEEEE`
  how would you specify the instruction
  `ADD.W   R1, (A6)`
  in opcode format?
- Specify parts:
  - DDD = 001
  - SS = 01
  - EEEEEE:
    - Address mode = 010
    - Register no. = 110
- Result? ⟶ **`1101 0011 01 010110`**

Addressing Mode	Mode Field	Register Field
Data register direct	000	Register Number
Address register direct	001	Register Number
Address register indirect	010	Register Number
Autoincrement	011	Register Number
Autodecrement	100	Register Number
Indexed basic	101	Register Number
Indexed full	110	Register Number
Absolute short	111	000
Absolute long	111	001
Relative basic	111	010
Relative full	111	011
Immediate or SR	111	100

Special Opcode Cases

- Branching instructions
  - Branch instructions require an offset to tell them how far forward or backward to branch.
  - Keep in mind that the PC will have incremented already (slides 10 & 11), so the address in the program counter will be one instruction larger than expected.
- Supplemental information
  - Some instructions (such as JMP) will involve an address or other information that is too big to fit into the instruction itself.
  - When this happens, the instruction decoder understands that the necessary information can be found in the 16 or 32 bits following the previous instruction, and will load that information into a data register instead of the IR.