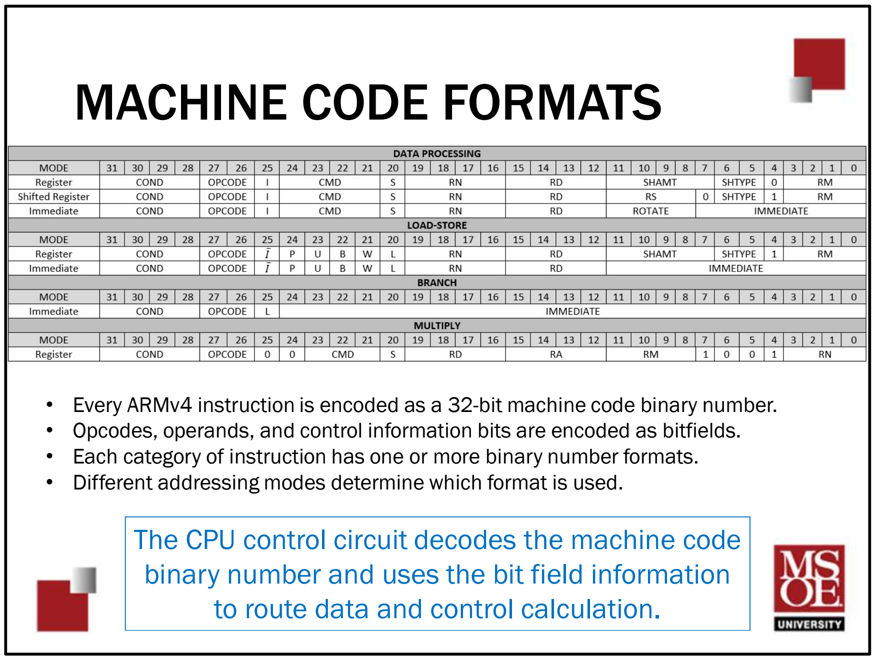
ARMv4 ISA
Machine Code
Data Processing Formats

Dr. Russ Meier
Milwaukee School of Engineering



MACHINE CODE FORMATS

DATA PROCESSING																																
MODE	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
Register	COND				OPCODE		I	CMD				S	RN				RD				SHAMT					SHTYPE		0	RM			
Shifted Register	COND				OPCODE		I	CMD				S	RN				RD				RS				0	SHTYPE		1	RM			
Immediate	COND				OPCODE		I	CMD				S	RN				RD				ROTATE				IMMEDIATE							
LOAD-STORE																																
MODE	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
Register	COND				OPCODE		$\bar{I}$	P	U	B	W	L	RN				RD				SHAMT					SHTYPE		1	RM			
Immediate	COND				OPCODE		$\bar{I}$	P	U	B	W	L	RN				RD				IMMEDIATE											
BRANCH																																
MODE	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
Immediate	COND				OPCODE		L	IMMEDIATE																								
MULTIPLY																																
MODE	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
Register	COND				OPCODE		0	0	CMD			S	RD				RA				RM				1	0	0	1	RN			

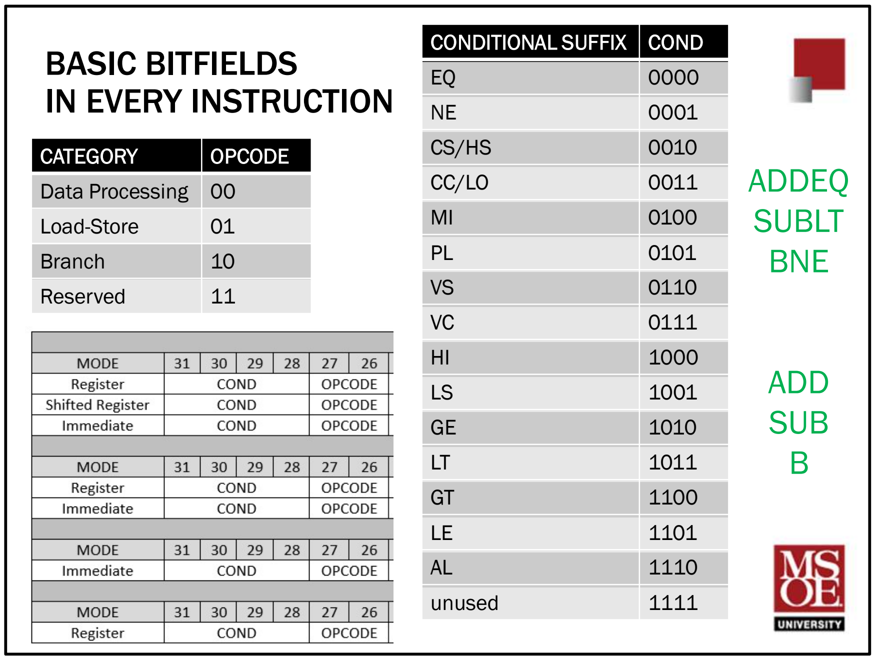
- Every ARMv4 instruction is encoded as a 32-bit machine code binary number.
- Opcodes, operands, and control information bits are encoded as bitfields.
- Each category of instruction has one or more binary number formats.
- Different addressing modes determine which format is used.

The CPU control circuit decodes the machine code binary number and uses the bit field information to route data and control calculation.

ARMv4 instructions are encoded as 32-bit binary machine code numbers stored in instruction memory.

- Each instruction category has a set of **machine code binary number formats** based on addressing modes.
- The **addressing mode** determines how the second ALU operand is found in the memory pyramid.
- Different addressing modes provide different information to the control circuitry though **machine code bit fields**.
- Some bit fields exist in all instruction formats.
- Most bit fields are fixed at constant bit locations within the binary number. One exception is notable. The multiply instruction formats reorder the location of register addresses Rd, Rn, and Rm when compared to all other data processing instruction formats.
- The **CE1921 ARMv4 quick reference card** and these lecture slides should be consulted when creating machine code for assembly instructions. This table from the quick reference card shows all instruction formats studied in CE1921.
- ARMv4 does define a few more instruction format that can be found in the online ARM reference manual for ARMv4 or in the appendix of the course textbook.

The key point to remember is that computer programs are binary numbers stored in instruction memory. The CPU control circuit decodes the machine code binary numbers and uses the bit field information to route data and control the calculation.



BASIC BITFIELDS IN EVERY INSTRUCTION

CATEGORY	OPCODE
Data Processing	00
Load-Store	01
Branch	10
Reserved	11

MODE	31	30	29	28	27	26
Register	COND				OPCODE	
Shifted Register	COND				OPCODE	
Immediate	COND				OPCODE	

MODE	31	30	29	28	27	26
Register	COND				OPCODE	
Immediate	COND				OPCODE	

MODE	31	30	29	28	27	26
Immediate	COND				OPCODE	

MODE	31	30	29	28	27	26
Register	COND				OPCODE	

CONDITIONAL SUFFIX	COND
EQ	0000
NE	0001
CS/HS	0010
CC/LO	0011
MI	0100
PL	0101
VS	0110
VC	0111
HI	1000
LS	1001
GE	1010
LT	1011
GT	1100
LE	1101
AL	1110
unused	1111

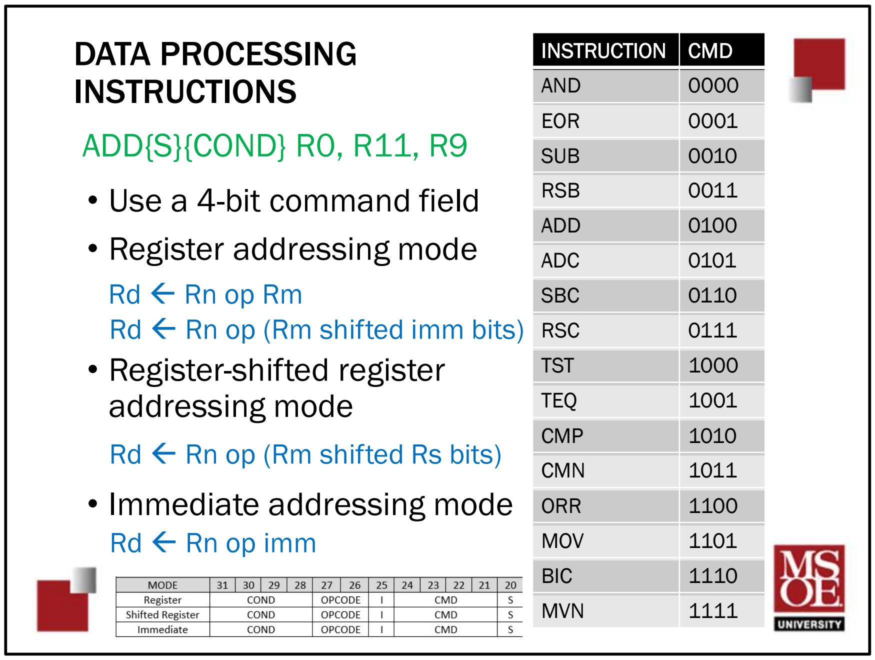
ADDEQ
SUBLT
BNE

ADD
SUB
B

All machine code formats include conditional execution and opcode information.

- The **conditional execution suffix is encoded in the most significant nibble** of every machine code binary number. The conditional execution suffix commands the control circuit to only execute the instruction if the condition code nibble from the ALU, composed of the C, V, N, and Z flags, matches the specified condition in the suffix. Examples of conditional execution suffixes are shown in green instructions on this slide. The final three instructions use the **always suffix, AL –** a suffix that is traditionally omitted from the assembly language mnemonic.

- The **opcode** is a 2-bit field at bit positions 27 and 26 in the machine code binary number that identifies the category of the instruction to the control circuit. Data processing instructions use opcode 0, while load-store instructions use opcode 1 and branch instructions use opcode 2.



DATA PROCESSING INSTRUCTIONS

ADD{S}{COND} R0, R11, R9

- Use a 4-bit command field
- Register addressing mode

  Rd ← Rn op Rm

  Rd ← Rn op (Rm shifted imm bits)
- Register-shifted register addressing mode

  Rd ← Rn op (Rm shifted Rs bits)
- Immediate addressing mode

  Rd ← Rn op imm

MODE	31	30	29	28	27	26	25	24	23	22	21	20
Register	COND				OPCODE		I	CMD				S
Shifted Register	COND				OPCODE		I	CMD				S
Immediate	COND				OPCODE		I	CMD				S

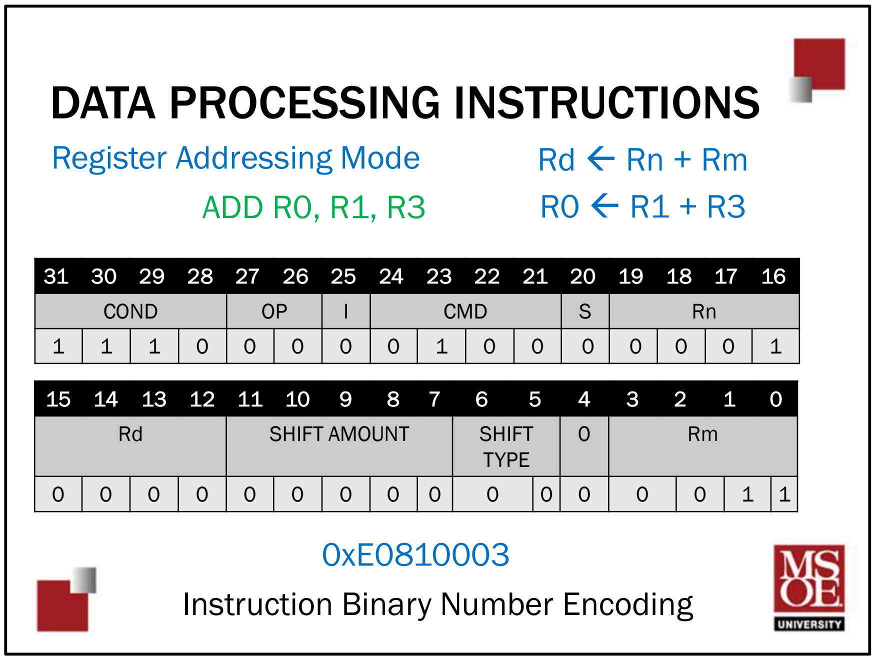
INSTRUCTION	CMD
AND	0000
EOR	0001
SUB	0010
RSB	0011
ADD	0100
ADC	0101
SBC	0110
RSC	0111
TST	1000
TEQ	1001
CMP	1010
CMN	1011
ORR	1100
MOV	1101
BIC	1110
MVN	1111

MSOE UNIVERSITY

This presentation focuses on the data processing machine code formats.

- **Data processing instructions** complete arithmetic or logic operations.
- All data processing instructions use a **command field** to inform the control circuit of the requested operation.
- Multiply uses a three-bit command field documented later in this presentation.
- All other data processing instructions use a **command nibble** located at bit positions 24 down to 21 in the machine code binary number.
- The addressing mode is identified to the control circuit using the immediate control bit located in bit position 25 within the machine code binary number. Immediate addressing mode is encoded with the **I-BIT** set to logic-1. Register addressing modes clear the I-BIT to logic-0.
- Data processing instructions can command the control circuit to sample the **condition code nibble** and store it in the current program status register (CPSR) by setting the **S-BIT** to logic-1. The compare and test instructions (CMP, CMN, TST, and TEQ) always encode S=1. All other instructions encode S=0 unless the S suffix is added to the assembly language mnemonic.
- As the green example instruction shows, the ARMv4 ISA specifies that the optional conditional execution suffix should be the last suffix appended to the instruction

mnemonic.



DATA PROCESSING INSTRUCTIONS

Register Addressing Mode   Rd ← Rn + Rm

ADD R0, R1, R3   R0 ← R1 + R3

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
COND				OP		I	CMD				S	Rn			
1	1	1	0	0	0	0	0	1	0	0	0	0	0	0	1

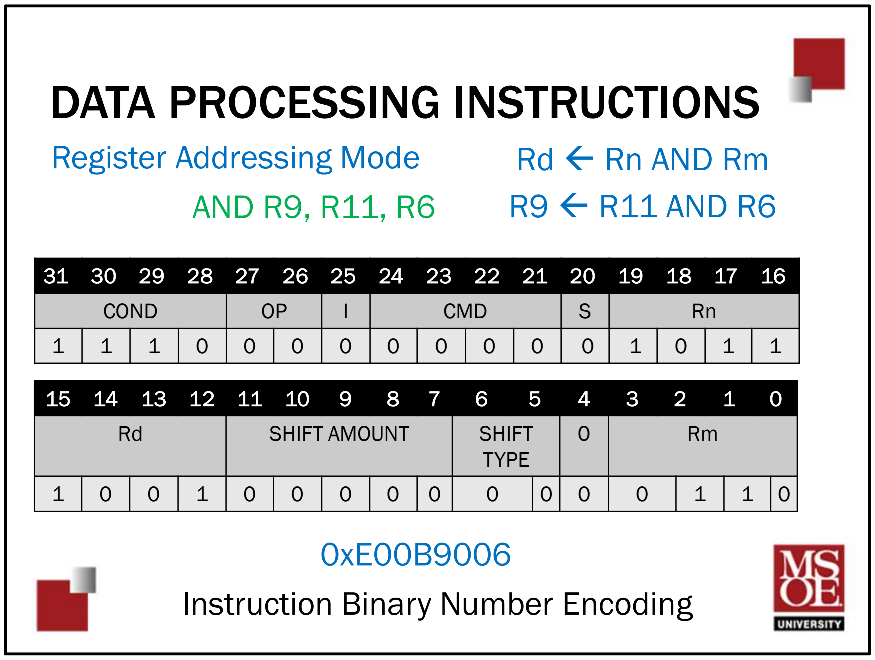
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rd				SHIFT AMOUNT					SHIFT TYPE		0	Rm			
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

0xE0810003

Instruction Binary Number Encoding

Let's start a series of machine code examples with the instruction ADD R0, R1, R3. Hand assembly of instructions into machine code binary numbers follows a set process:

- **Identify** the addressing mode of data processing instructions by looking at the second ALU operand. In this case, the second operand is R3 – a register. This is **register addressing mode**.
- **Write** the register transfer level (RTL) equations in abstract and register-name forms.
- **Draw** the correct machine code format as a table of bits and bit field names.
- **Encode** the addressing mode. Register mode is **not immediate mode** and thus **I=0**.
- **Encode** suffix information. In this case, there is no S suffix and the conditional execution suffix is the omitted **always** suffix. This results in **S=0** and **COND= always=1110**.
- **Encode** the command field. From the table of commands, **CMD=4=0100**.
- **Encode** the operands. **Rn=1=0001, Rd=0=0000, Rm=3=0011**.
- **Encode** the shift information. There is no shift information and all shift information bits default to 0.
- **Write** the final machine code binary number in hexadecimal.



DATA PROCESSING INSTRUCTIONS

Register Addressing Mode Rd ← Rn AND Rm

AND R9, R11, R6 R9 ← R11 AND R6

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
COND				OP		I	CMD				S	Rn			
1	1	1	0	0	0	0	0	0	0	0	0	1	0	1	1

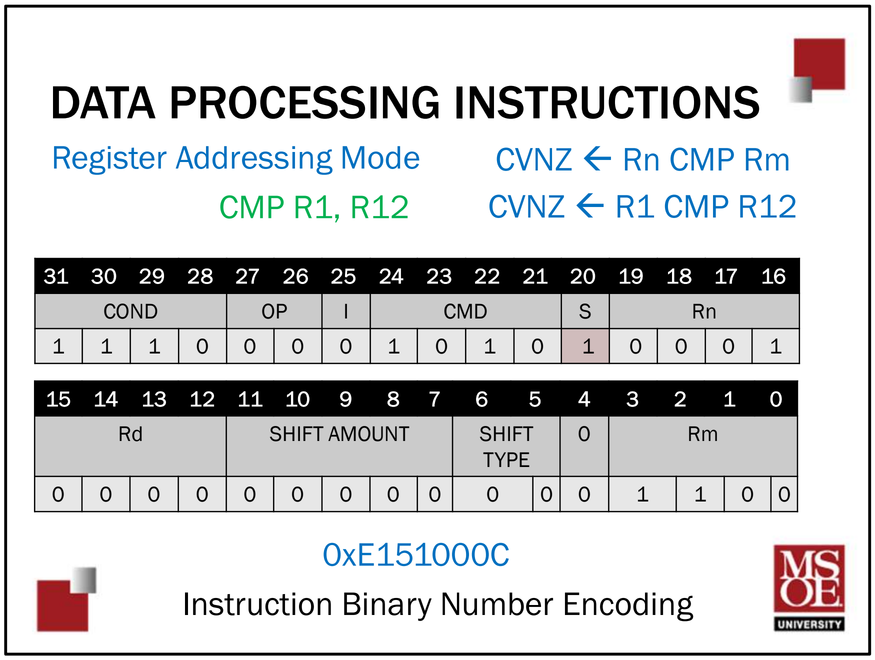
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rd				SHIFT AMOUNT					SHIFT TYPE		0	Rm			
1	0	0	1	0	0	0	0	0	0	0	0	0	1	1	0

0xE00B9006

Instruction Binary Number Encoding

Encode the instruction AND R9, R11, R6 into its machine code binary number.

- **Identify** the addressing mode of data processing instructions by looking at the second ALU operand. In this case, the second operand is R6 – a register. This is **register addressing mode**.
- **Write** the register transfer level (RTL) equations in abstract and register-name forms.
- **Draw** the correct machine code format as a table of bits and bit field names.
- **Encode** the addressing mode. Register mode is **not immediate mode** and thus **I=0**.
- **Encode** suffix information. In this case, there is no S suffix and the conditional execution suffix is the omitted **always** suffix. This results in **S=0** and **COND= always=1110**.
- **Encode** the command field. From the table of commands, **CMD=0=0000**.
- **Encode** the operands. **Rn=11=1101, Rd=9=1001, Rm=6=0110**.
- **Encode** the shift information. There is no shift information and all shift information bits default to 0.
- **Write** the final machine code binary number in hexadecimal.



DATA PROCESSING INSTRUCTIONS

Register Addressing Mode   CVNZ ← Rn CMP Rm

CMP R1, R12   CVNZ ← R1 CMP R12

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
COND				OP		I	CMD				S	Rn			
1	1	1	0	0	0	0	1	0	1	0	1	0	0	0	1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rd				SHIFT AMOUNT					SHIFT TYPE		0	Rm			
0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0

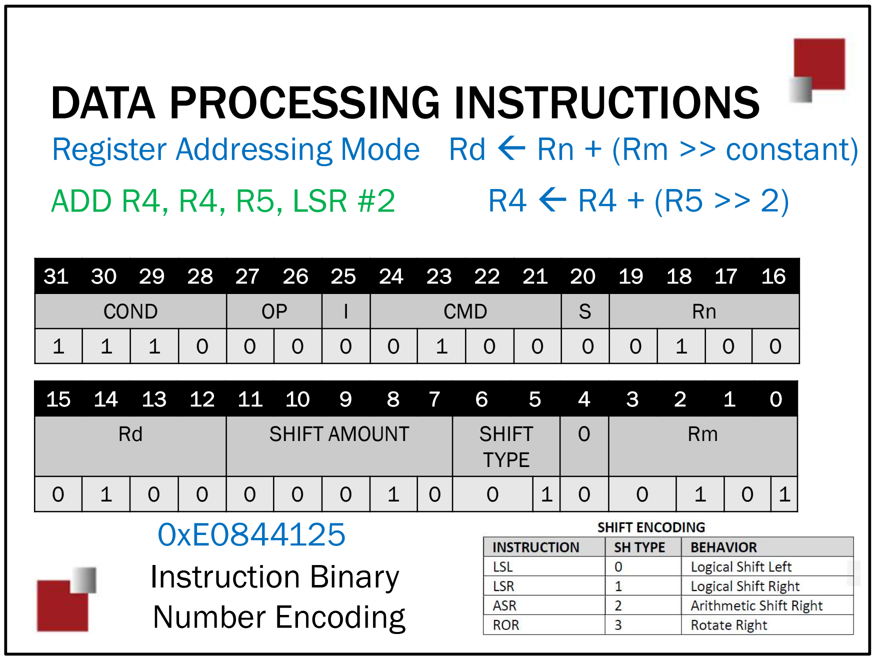
0xE151000C

Instruction Binary Number Encoding

Encode the instruction CMP R1, R12 into its machine code binary number.

- **Identify** the addressing mode of data processing instructions by looking at the second ALU operand. In this case, the second operand is R12 – a register. This is **register addressing mode**.
- **Write** the register transfer level (RTL) equations in abstract and register-name forms. The compare instruction subtracts R1 – R12 and updates the condition code nibble. Rather than write subtraction in the RTL equation, use CMP to remind yourself that the subtraction result is not being stored but the condition code nibble is.
- **Draw** the correct machine code format as a table of bits and bit field names.
- **Encode** the addressing mode. Register mode is **not immediate mode** and thus **I=0**.
- **Encode** suffix information. In this case, there is no S suffix but compare and test instructions always set the condition code nibble. This results in **S=1**. The conditional execution suffix is the omitted **always** suffix resulting in **COND= always=1110**.
- **Encode** the command field. From the table of commands, **CMD=10=1010**.
- **Encode** the operands. **Rn=1=0001, Rd=unused=defaulted to 0000, Rm=12=1100**.
- **Encode** the shift information. There is no shift information and all shift information

bits default to 0.
- **Write** the final machine code binary number in hexadecimal.



DATA PROCESSING INSTRUCTIONS

Register Addressing Mode Rd ← Rn + (Rm >> constant)

ADD R4, R4, R5, LSR #2 R4 ← R4 + (R5 >> 2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
COND				OP		I	CMD				S	Rn			
1	1	1	0	0	0	0	0	1	0	0	0	0	1	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rd				SHIFT AMOUNT					SHIFT TYPE		0	Rm			
0	1	0	0	0	0	0	1	0	0	1	0	0	1	0	1

0xE0844125

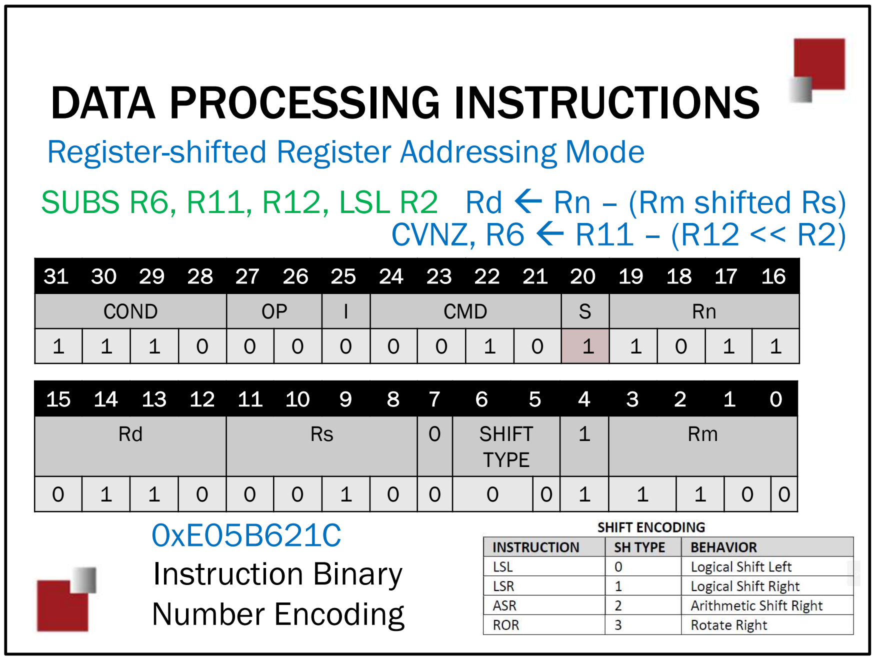
Instruction Binary Number Encoding

SHIFT ENCODING

INSTRUCTION	SH TYPE	BEHAVIOR
LSL	0	Logical Shift Left
LSR	1	Logical Shift Right
ASR	2	Arithmetic Shift Right
ROR	3	Rotate Right

Encode the instruction ADD R4, R4, R5, LSR #2 into its machine code binary number.

- **Identify** the addressing mode of data processing instructions by looking at the second ALU operand. In this case, the second operand is R5 – a register – shifted right 2 positions. This is **register addressing mode** because the shifted value is stated as a constant and not stated as a value held in another register.
- **Write** the register transfer level (RTL) equations in abstract and register-name forms.
- **Draw** the correct machine code format as a table of bits and bit field names.
- **Encode** the addressing mode. Register mode is **not immediate mode** and thus **I=0**.
- **Encode** suffix information. In this case, there is no S suffix and the conditional execution suffix is the omitted **always** suffix. This results in **S=0** and **COND= always=1110**.
- **Encode** the command field. From the table of commands, **CMD=4=0100**.
- **Encode** the operands. **Rn=4=0100, Rd=4=0100, Rm=5=0101**.
- **Encode** the shift information. **SHIFT AMOUNT=2=00010** and **SHIFT TYPE=LSR=01**.
- **Write** the final machine code binary number in hexadecimal.



DATA PROCESSING INSTRUCTIONS

Register-shifted Register Addressing Mode

SUBS R6, R11, R12, LSL R2 Rd ← Rn – (Rm shifted Rs)

CVNZ, R6 ← R11 – (R12 << R2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
COND				OP		I	CMD				S	Rn			
1	1	1	0	0	0	0	0	0	1	0	1	1	0	1	1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rd				Rs				0	SHIFT TYPE		1	Rm			
0	1	1	0	0	0	1	0	0	0	0	1	1	1	0	0

0xE05B621C

Instruction Binary Number Encoding

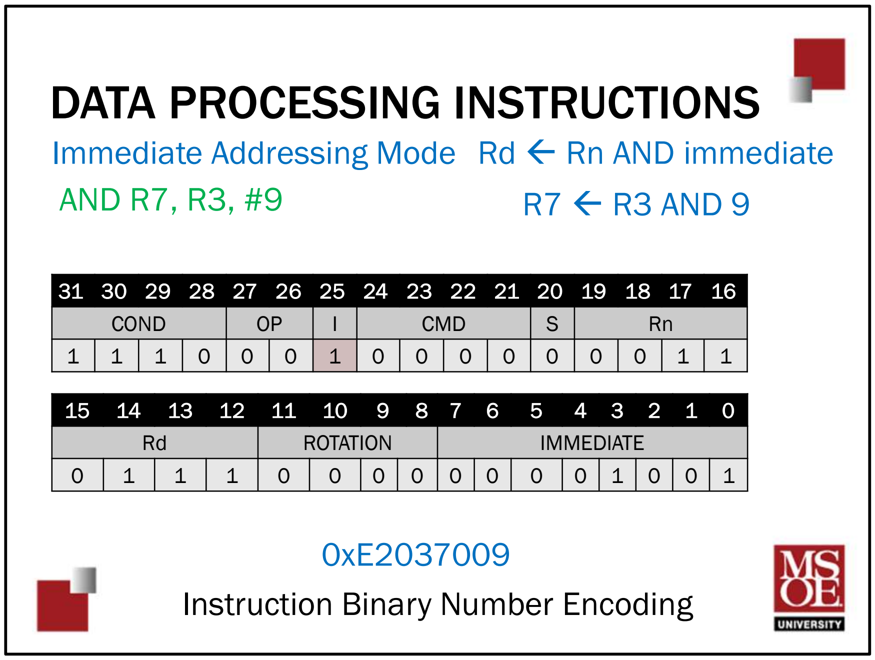
SHIFT ENCODING

INSTRUCTION	SH TYPE	BEHAVIOR
LSL	0	Logical Shift Left
LSR	1	Logical Shift Right
ASR	2	Arithmetic Shift Right
ROR	3	Rotate Right

Encode the instruction SUBS R6, R11, R12, LSR R2 into its machine code binary number.

- **Identify** the addressing mode of data processing instructions by looking at the second ALU operand. In this case, the second operand is R12 – a register – shifted left the number of times specified in the lower byte of R2. This is **register-shifted register addressing mode** because the shifted value is stored in a fourth register.
- **Write** the register transfer level (RTL) equations in abstract and register-name forms. Because the S suffix is present, this instruction will set the condition code nibble and store the calculated result in register R6.
- **Draw** the correct machine code format as a table of bits and bit field names.
- **Encode** the addressing mode. Register mode is **not immediate mode** and thus **I=0**.
- **Encode** suffix information. In this case, there is an S suffix and the conditional execution suffix is the omitted **always** suffix. This results in **S=1** and **COND= always=1110**.
- **Encode** the command field. From the table of commands, **CMD=2=0010**.
- **Encode** the operands. **Rn=11=1011, Rd=6=0110, Rm=12=1100**, and the shift value register **Rs=2=0010**.
- **Encode** the shift information. **SHIFT TYPE=LSL=00**.

- **Write** the final machine code binary number in hexadecimal.



DATA PROCESSING INSTRUCTIONS

Immediate Addressing Mode Rd ← Rn AND immediate

AND R7, R3, #9 R7 ← R3 AND 9

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
COND				OP		I	CMD				S	Rn			
1	1	1	0	0	0	1	0	0	0	0	0	0	0	1	1

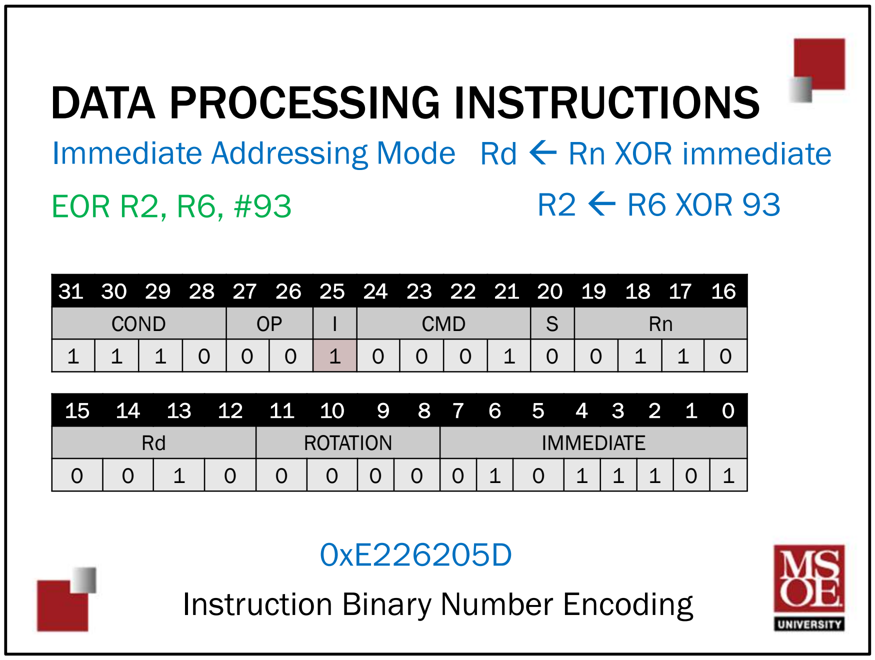
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rd				ROTATION				IMMEDIATE							
0	1	1	1	0	0	0	0	0	0	0	0	1	0	0	1

0xE2037009

Instruction Binary Number Encoding

Encode the instruction AND R7, R3, #9 into its machine code binary number.

- **Identify** the addressing mode of data processing instructions by looking at the second ALU operand. In this case, the second operand is a constant. This is **immediate addressing mode**.
- **Write** the register transfer level (RTL) equations in abstract and register-name forms.
- **Draw** the correct machine code format as a table of bits and bit field names.
- **Encode** the addressing mode: **I=1**.
- **Encode** suffix information. In this case, there is no S suffix and the conditional execution suffix is the omitted **always** suffix. This results in **S=0** and **COND= always=1110**.
- **Encode** the command field. From the table of commands, **CMD=0=0100**.
- **Encode** the register operands. **Rn=3=0011, Rd=7=0111.**
- **Encode** the immediate constant and rotation fields. Since the constant fits on the 8-bit number line, simply encode **ROTATE=0** and **IMMEDIATE=9=00001001.**
- **Write** the final machine code binary number in hexadecimal.



DATA PROCESSING INSTRUCTIONS

Immediate Addressing Mode Rd ← Rn XOR immediate

EOR R2, R6, #93 R2 ← R6 XOR 93

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
COND				OP		I	CMD				S	Rn			
1	1	1	0	0	0	1	0	0	0	1	0	0	1	1	0

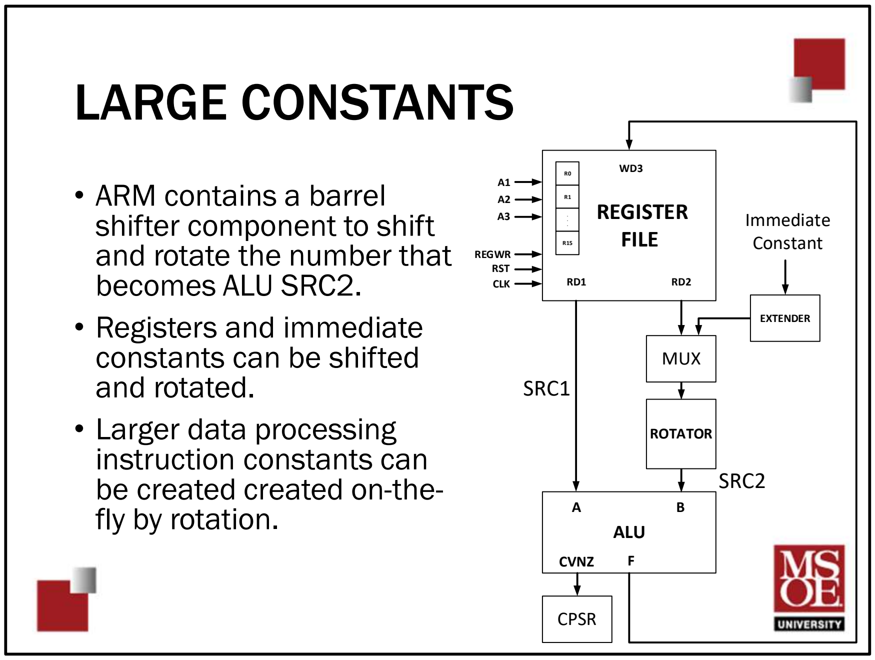
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rd				ROTATION				IMMEDIATE							
0	0	1	0	0	0	0	0	0	1	0	1	1	1	0	1

0xE226205D

Instruction Binary Number Encoding

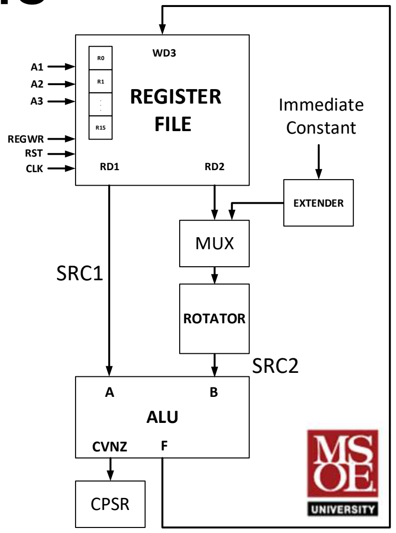
Encode the instruction EOR R2, R6, #93 into its machine code binary number.

- **Identify** the addressing mode of data processing instructions by looking at the second ALU operand. In this case, the second operand is a constant. This is **immediate addressing mode**.
- **Write** the register transfer level (RTL) equations in abstract and register-name forms.
- **Draw** the correct machine code format as a table of bits and bit field names.
- **Encode** the addressing mode: **I=1**.
- **Encode** suffix information. In this case, there is no S suffix and the conditional execution suffix is the omitted **always** suffix. This results in **S=0** and **COND= always=1110**.
- **Encode** the command field. From the table of commands, **CMD=1=0001**.
- **Encode** the register operands. **Rn=6=0110, Rd=2=0010**.
- **Encode** the immediate constant and rotation fields. Since the constant fits on the 8-bit number line, simply encode **ROTATE=0** and **IMMEDIATE=93=01011101.**
- **Write** the final machine code binary number in hexadecimal.



LARGE CONSTANTS

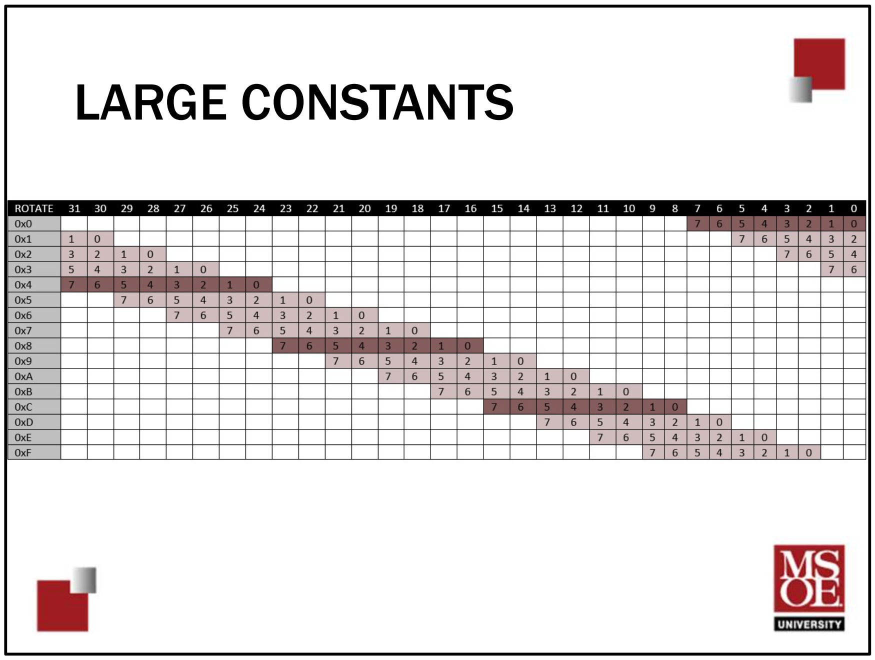
- ARM contains a barrel shifter component to shift and rotate the number that becomes ALU SRC2.
- Registers and immediate constants can be shifted and rotated.
- Larger data processing instruction constants can be created created on-the-fly by rotation.



ARMv4 includes a shift and rotate component called a **barrel shifter** in the operand circuitry for SRC2. In this diagram, the barrel shifter is labeled as a **rotator** because it is the rotation functionality that allows constants much larger than 8-bits to be specified in the small rotation and immediate bit fields defined in the data processing machine code format.

- Remember that the whole point of having an “immediate constant” in the machine code binary number is speed – avoiding an additional data memory load.
- Many 32-bit RISC architectures developed at the same time as the ARMv4 ISA use 16-bit immediate bit fields. ARMv4 doesn’t have space for a 16-bit immediate bit field because it allows conditional execution for most instructions. This requires a conditional execution suffix. When this suffix, the opcode, the command bits, control bits I and S, and the register operands Rd and Rn are encoded, there are only 12-bits remaining for an immediate – leaving a smaller immediate number line than the competitors.
- ARMv4 may have been at a disadvantage commercially if it couldn’t form larger constants directly from information encoded in the machine code number.
- The architecture mitigates this disadvantage by using one nibble of the 12-bits to specify a **rotation amount** used to rotate an 8-bit immediate into position

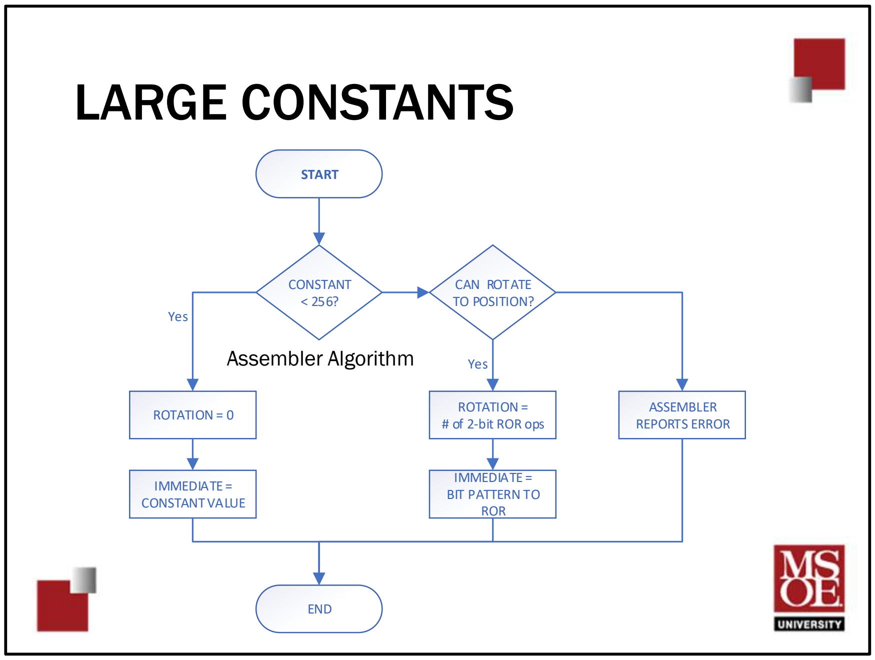
somewhere else within 32-bits. This provides a much more diverse set of allowed constants than the 0 to 4095 number line specified by a fixed 12-bit immediate field.



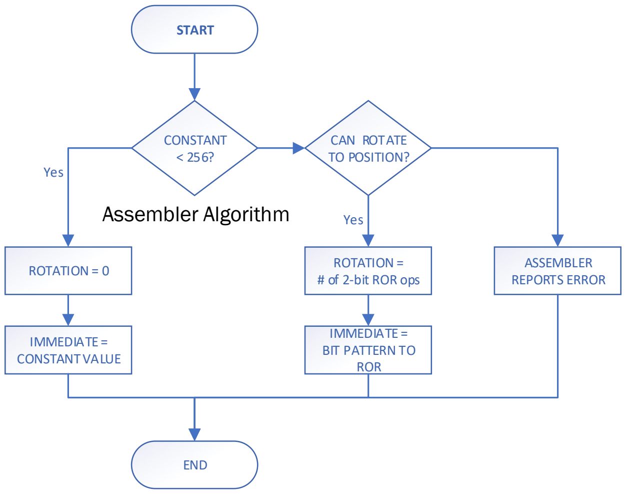
LARGE CONSTANTS

ROTATE	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
0x0																									7	6	5	4	3	2	1	0
0x1	1	0																									7	6	5	4	3	2
0x2	3	2	1	0																									7	6	5	4
0x3	5	4	3	2	1	0																									7	6
0x4	7	6	5	4	3	2	1	0																								
0x5			7	6	5	4	3	2	1	0																						
0x6					7	6	5	4	3	2	1	0																				
0x7							7	6	5	4	3	2	1	0																		
0x8									7	6	5	4	3	2	1	0																
0x9											7	6	5	4	3	2	1	0														
0xA													7	6	5	4	3	2	1	0												
0xB															7	6	5	4	3	2	1	0										
0xC																	7	6	5	4	3	2	1	0								
0xD																			7	6	5	4	3	2	1	0						
0xE																					7	6	5	4	3	2	1	0				
0xF																							7	6	5	4	3	2	1	0		

The rotation nibble in the data processing machine code format specifies how many rotate-right steps should be completed. Each rotate-right moves the immediate byte two positions. This provides a rich set of possible constants but not a complete 32-bit number line. In this diagram, white squares are filled with zeros and shaded squares represent the bits of the machine code immediate byte. The darker shading shows that the machine code immediate byte can be aligned into any byte of the 32-bit number that arrives at the ALU. A quick inspection will also show that it can be aligned to any nibble within the 32-bit number that arrives at the ALU.

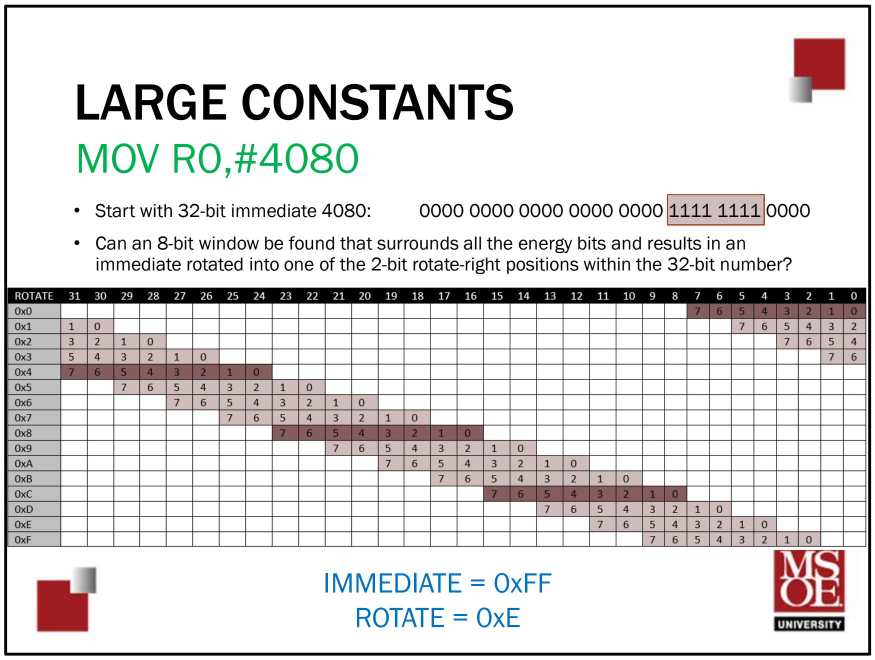


LARGE CONSTANTS



Assembler Algorithm

A human completing hand assembly, or a software assembler must determine if the desired large immediate value can be formed from an 8-bit immediate rotated into position. This flowchart outlines a basic algorithm. There are three outcomes: a constant that naturally fits on the 8-bit number line, a rotated immediate that creates the desired constant, and an assembler error if the constant cannot be formed. Pause this presentation and study the algorithm a bit.



LARGE CONSTANTS

MOV R0,#4080

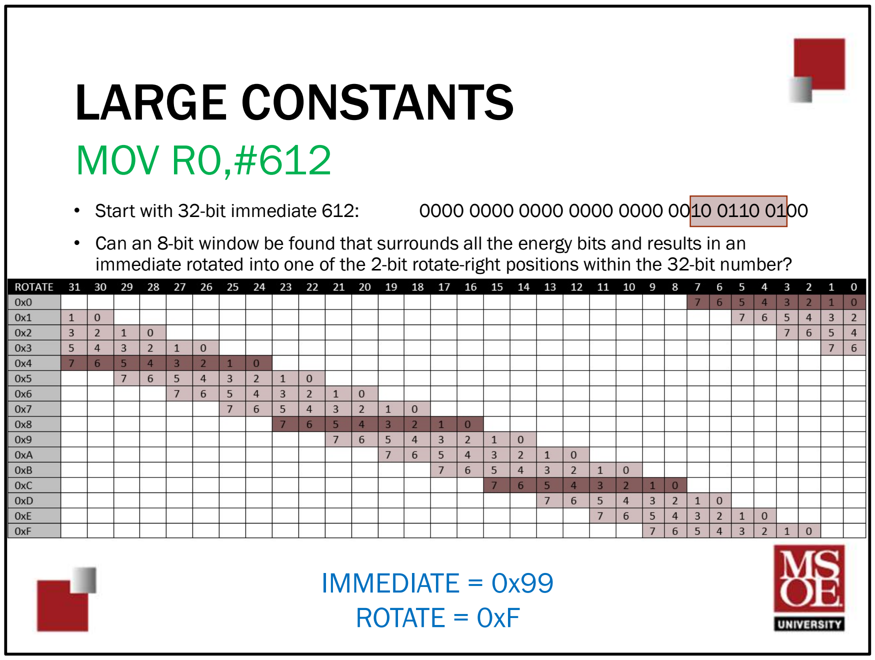
- Start with 32-bit immediate 4080: 0000 0000 0000 0000 0000 1111 1111 0000
- Can an 8-bit window be found that surrounds all the energy bits and results in an immediate rotated into one of the 2-bit rotate-right positions within the 32-bit number?

ROTATE	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
0x0																									7	6	5	4	3	2	1	0
0x1	1	0																									7	6	5	4	3	2
0x2	3	2	1	0																									7	6	5	4
0x3	5	4	3	2	1	0																									7	6
0x4	7	6	5	4	3	2	1	0																								
0x5			7	6	5	4	3	2	1	0																						
0x6					7	6	5	4	3	2	1	0																				
0x7							7	6	5	4	3	2	1	0																		
0x8									7	6	5	4	3	2	1	0																
0x9											7	6	5	4	3	2	1	0														
0xA													7	6	5	4	3	2	1	0												
0xB															7	6	5	4	3	2	1	0										
0xC																	7	6	5	4	3	2	1	0								
0xD																			7	6	5	4	3	2	1	0						
0xE																					7	6	5	4	3	2	1	0				
0xF																							7	6	5	4	3	2	1	0		

IMMEDIATE = 0xFF
ROTATE = 0xE

Determine the immediate and rotation bit fields required to create large constant 4080.

- Write the 32-bit binary equivalent of 4080.
- Compare the window against positions within the rotation table.
- This is immediate bitfield 0xFF and rotate bitfield 0xE.



LARGE CONSTANTS

MOV R0,#612

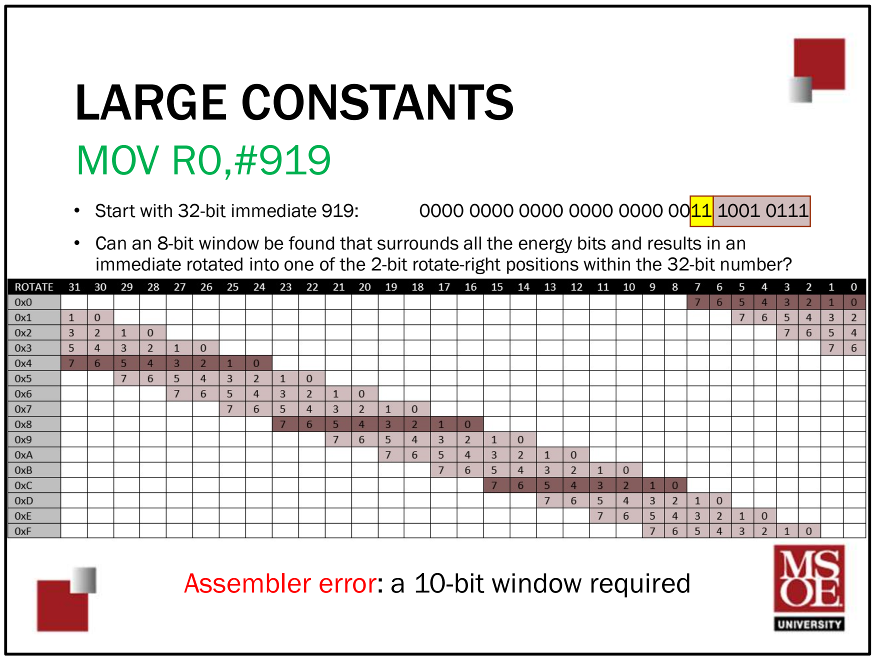
- Start with 32-bit immediate 612: 0000 0000 0000 0000 0000 0010 0110 0100
- Can an 8-bit window be found that surrounds all the energy bits and results in an immediate rotated into one of the 2-bit rotate-right positions within the 32-bit number?

ROTATE	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
0x0																									7	6	5	4	3	2	1	0
0x1	1	0																									7	6	5	4	3	2
0x2	3	2	1	0																									7	6	5	4
0x3	5	4	3	2	1	0																									7	6
0x4	7	6	5	4	3	2	1	0																								
0x5			7	6	5	4	3	2	1	0																						
0x6					7	6	5	4	3	2	1	0																				
0x7							7	6	5	4	3	2	1	0																		
0x8									7	6	5	4	3	2	1	0																
0x9											7	6	5	4	3	2	1	0														
0xA													7	6	5	4	3	2	1	0												
0xB															7	6	5	4	3	2	1	0										
0xC																	7	6	5	4	3	2	1	0								
0xD																			7	6	5	4	3	2	1	0						
0xE																					7	6	5	4	3	2	1	0				
0xF																							7	6	5	4	3	2	1	0		

IMMEDIATE = 0x99
ROTATE = 0xF

Determine the immediate and rotation bit fields required to create large constant 612.

- Write the 32-bit binary equivalent of 612.
- Compare the window against the positions within the rotation table.
- This is immediate bitfield 0x99 and rotate bitfield 0xF.



LARGE CONSTANTS

MOV R0,#919

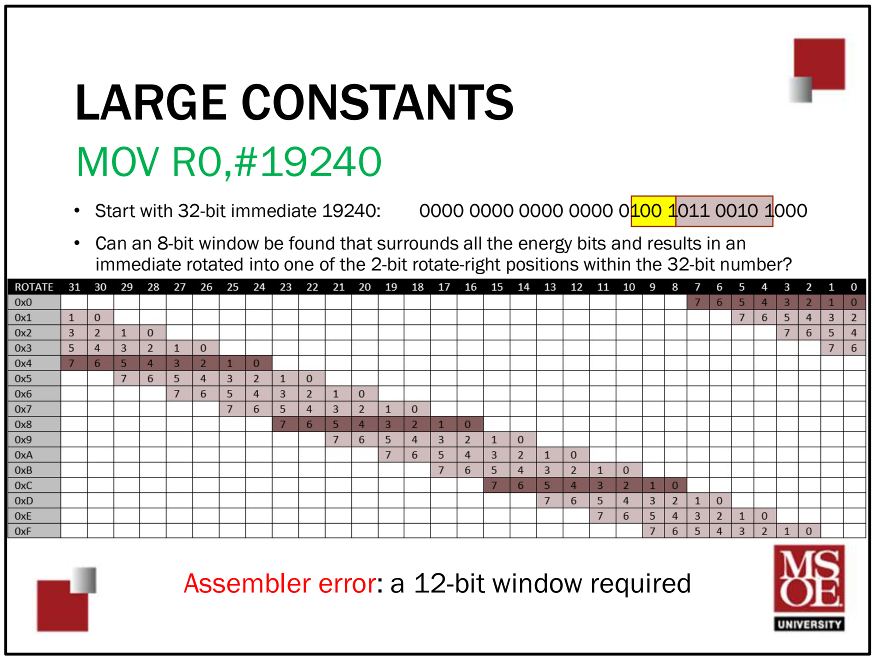
- Start with 32-bit immediate 919: 0000 0000 0000 0000 0000 0011 1001 0111
- Can an 8-bit window be found that surrounds all the energy bits and results in an immediate rotated into one of the 2-bit rotate-right positions within the 32-bit number?

ROTATE	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
0x0																									7	6	5	4	3	2	1	0
0x1	1	0																									7	6	5	4	3	2
0x2	3	2	1	0																									7	6	5	4
0x3	5	4	3	2	1	0																									7	6
0x4	7	6	5	4	3	2	1	0																								
0x5			7	6	5	4	3	2	1	0																						
0x6					7	6	5	4	3	2	1	0																				
0x7							7	6	5	4	3	2	1	0																		
0x8									7	6	5	4	3	2	1	0																
0x9											7	6	5	4	3	2	1	0														
0xA													7	6	5	4	3	2	1	0												
0xB															7	6	5	4	3	2	1	0										
0xC																	7	6	5	4	3	2	1	0								
0xD																			7	6	5	4	3	2	1	0						
0xE																					7	6	5	4	3	2	1	0				
0xF																							7	6	5	4	3	2	1	0		

Assembler error: a 10-bit window required

Determine the immediate and rotation bit fields required to create large constant 612.

- Write the 32-bit binary equivalent of 612.
- Compare the window against the positions within the rotation table.
- The required window to enclose all energy bits is 10-bits wide. This cannot fit in the 8-bit immediate field.
- This constant cannot be inserted in the machine code binary number.
- The assembler reports an error and the programmer must instead use a load from data memory.



LARGE CONSTANTS

MOV R0,#19240

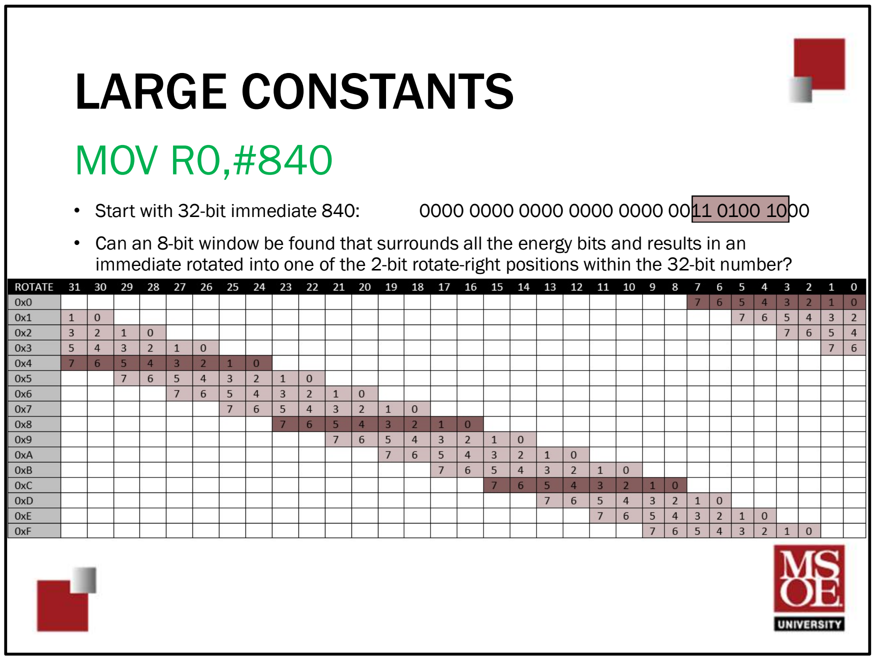
- Start with 32-bit immediate 19240: 0000 0000 0000 0000 0100 1011 0010 1000
- Can an 8-bit window be found that surrounds all the energy bits and results in an immediate rotated into one of the 2-bit rotate-right positions within the 32-bit number?

ROTATE	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
0x0																									7	6	5	4	3	2	1	0
0x1	1	0																									7	6	5	4	3	2
0x2	3	2	1	0																									7	6	5	4
0x3	5	4	3	2	1	0																									7	6
0x4	7	6	5	4	3	2	1	0																								
0x5			7	6	5	4	3	2	1	0																						
0x6					7	6	5	4	3	2	1	0																				
0x7							7	6	5	4	3	2	1	0																		
0x8									7	6	5	4	3	2	1	0																
0x9											7	6	5	4	3	2	1	0														
0xA													7	6	5	4	3	2	1	0												
0xB															7	6	5	4	3	2	1	0										
0xC																	7	6	5	4	3	2	1	0								
0xD																			7	6	5	4	3	2	1	0						
0xE																					7	6	5	4	3	2	1	0				
0xF																							7	6	5	4	3	2	1	0		

Assembler error: a 12-bit window required

Determine the immediate and rotation bit fields required to create large constant 612.

- Write the 32-bit binary equivalent of 612.
- Compare the window against the positions within the rotation table.
- The required window to enclose all energy bits is 12-bits wide. This cannot fit in the 8-bit immediate field.
- This constant cannot be inserted in the machine code binary number.
- The assembler reports an error and the programmer must instead use a load from data memory.



LARGE CONSTANTS

MOV R0,#840

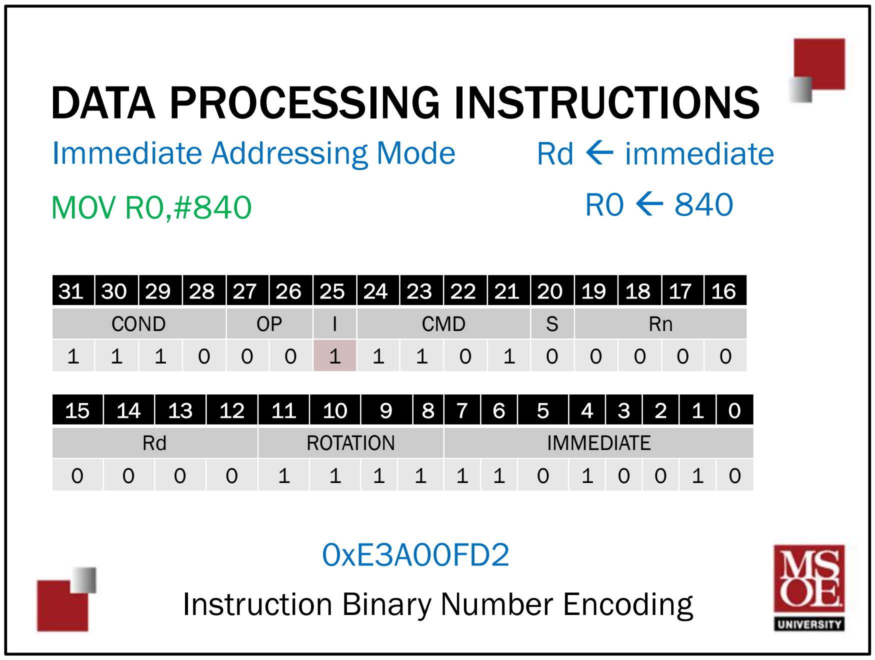
- Start with 32-bit immediate 840: 0000 0000 0000 0000 0000 0011 0100 1000
- Can an 8-bit window be found that surrounds all the energy bits and results in an immediate rotated into one of the 2-bit rotate-right positions within the 32-bit number?

ROTATE	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
0x0																									7	6	5	4	3	2	1	0
0x1	1	0																									7	6	5	4	3	2
0x2	3	2	1	0																									7	6	5	4
0x3	5	4	3	2	1	0																									7	6
0x4	7	6	5	4	3	2	1	0																								
0x5			7	6	5	4	3	2	1	0																						
0x6					7	6	5	4	3	2	1	0																				
0x7							7	6	5	4	3	2	1	0																		
0x8									7	6	5	4	3	2	1	0																
0x9											7	6	5	4	3	2	1	0														
0xA													7	6	5	4	3	2	1	0												
0xB															7	6	5	4	3	2	1	0										
0xC																	7	6	5	4	3	2	1	0								
0xD																			7	6	5	4	3	2	1	0						
0xE																					7	6	5	4	3	2	1	0				
0xF																							7	6	5	4	3	2	1	0		

MSOE UNIVERSITY

Determine the immediate and rotation bit fields required to create large constant 612.

- Write the 32-bit binary equivalent of 840.
- Compare the window against the positions within the rotation table.
- This is immediate bitfield 0xD2 and rotate bitfield 0xF.



DATA PROCESSING INSTRUCTIONS

Immediate Addressing Mode Rd ← immediate

MOV R0,#840 R0 ← 840

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
COND				OP		I	CMD				S	Rn			
1	1	1	0	0	0	1	1	1	0	1	0	0	0	0	0

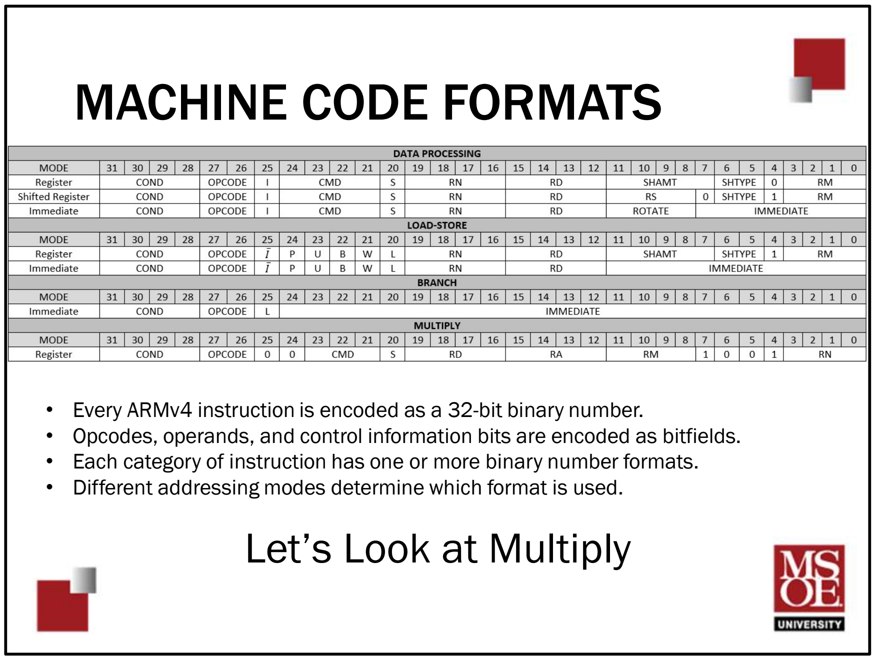
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rd				ROTATION				IMMEDIATE							
0	0	0	0	1	1	1	1	1	1	0	1	0	0	1	0

0xE3A00FD2

Instruction Binary Number Encoding

Encode the instruction MOV R0, #840 into its machine code binary number.

- **Identify** the addressing mode of data processing instructions by looking at the second ALU operand. In this case, the second operand is a constant. This is **immediate addressing mode**.
- **Write** the register transfer level (RTL) equations in abstract and register-name forms.
- **Draw** the correct machine code format as a table of bits and bit field names.
- **Encode** the addressing mode: **I=1**.
- **Encode** suffix information. In this case, there is no S suffix and the conditional execution suffix is the omitted **always** suffix. This results in **S=0** and **COND= always=1110**.
- **Encode** the command field. From the table of commands, **CMD=13=1101**.
- **Encode** the register operands. **Rn=unused=defaults to 0000**, **Rd=0=0000**.
- **Encode** the immediate constant and rotation fields. **ROTATE=0** and **IMMEDIATE=0xD2=11010010.**
- **Write** the final machine code binary number in hexadecimal.



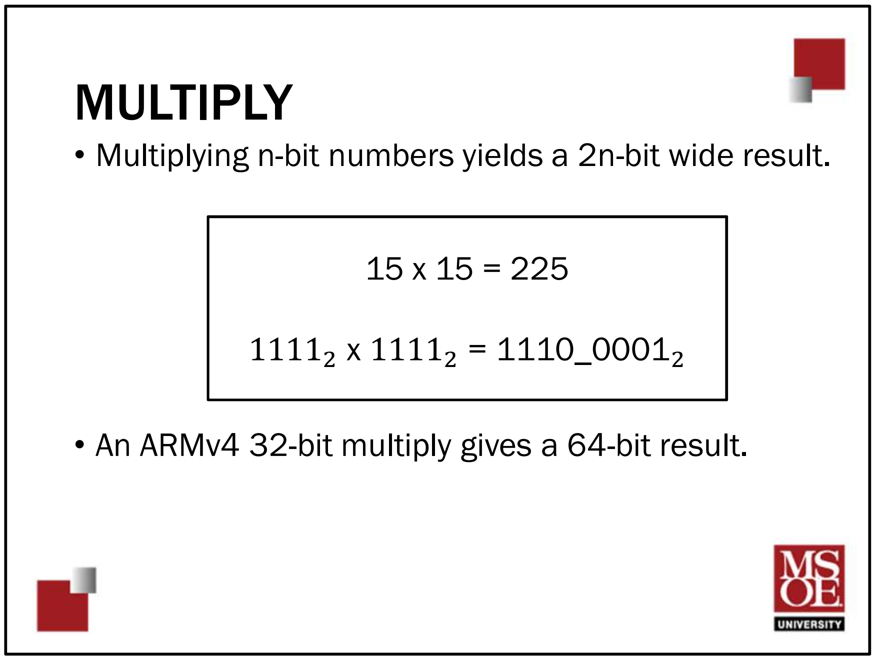
MACHINE CODE FORMATS

DATA PROCESSING																																
MODE	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
Register	COND				OPCODE		I	CMD				S	RN				RD				SHAMT					SHTYPE		0	RM			
Shifted Register	COND				OPCODE		I	CMD				S	RN				RD				RS				0	SHTYPE		1	RM			
Immediate	COND				OPCODE		I	CMD				S	RN				RD				ROTATE				IMMEDIATE							
LOAD-STORE																																
MODE	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
Register	COND				OPCODE		$\bar{I}$	P	U	B	W	L	RN				RD				SHAMT					SHTYPE		1	RM			
Immediate	COND				OPCODE		$\bar{I}$	P	U	B	W	L	RN				RD				IMMEDIATE											
BRANCH																																
MODE	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
Immediate	COND				OPCODE		L	IMMEDIATE																								
MULTIPLY																																
MODE	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
Register	COND				OPCODE		0	0	CMD			S	RD				RA				RM				1	0	0	1	RN			

- Every ARMv4 instruction is encoded as a 32-bit binary number.
- Opcodes, operands, and control information bits are encoded as bitfields.
- Each category of instruction has one or more binary number formats.
- Different addressing modes determine which format is used.

Let’s Look at Multiply

ARMv4 does not provide divide instructions because divide is the most expensive arithmetic operation in terms of circuit silicon space and speed of calculation. It does provide a set of data processing instructions for multiplication. These instructions do not use the machine code formats of the other data processing instructions. Instead, a special format just for multiplication instructions is used. It is not clear why this format rearranged the location of the register operands, but as you can see all register operands have moved to new positions in this format. Let’s look at multiply and practice encoding some example instructions.



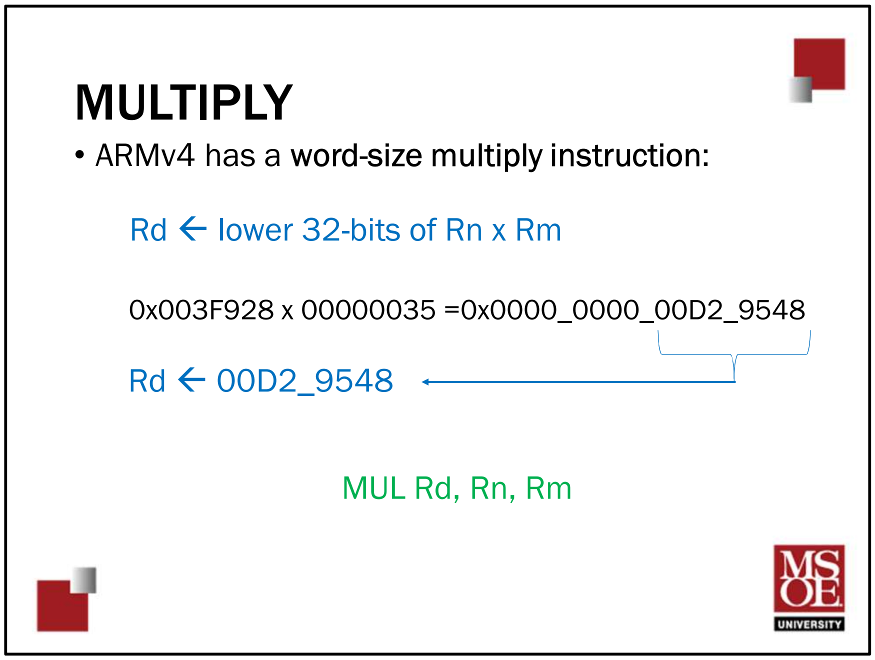
MULTIPLY

- Multiplying n-bit numbers yields a 2n-bit wide result.

$$15 \times 15 = 225$$

$$1111_2 \times 1111_2 = 1110\_0001_2$$

- An ARMv4 32-bit multiply gives a 64-bit result.



MULTIPLY

- ARMv4 has a **word-size multiply instruction:**

Rd ← lower 32-bits of Rn x Rm

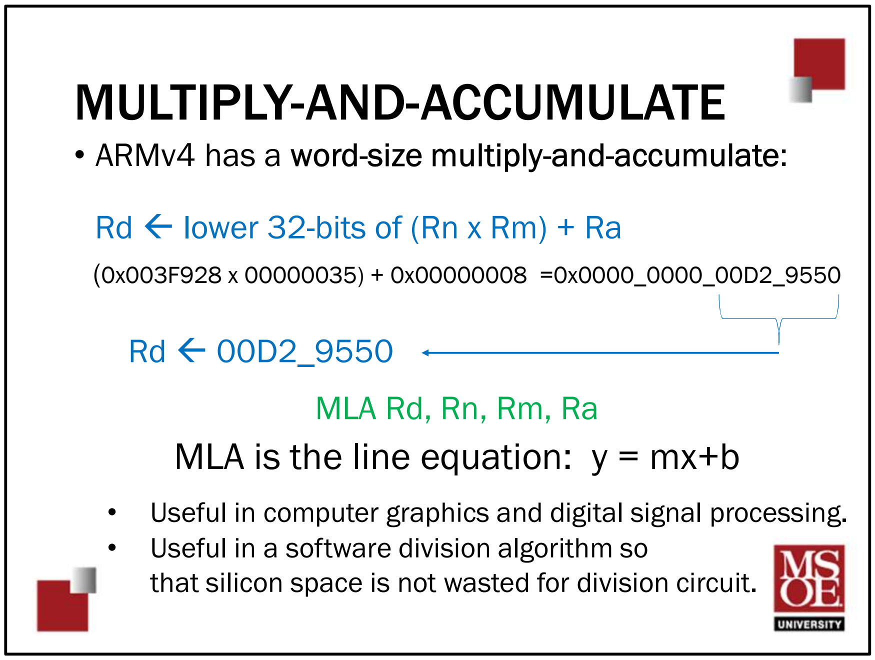
$0x003F928 \times 00000035 = 0x0000\_0000\_\underbrace{00D2\_9548}$

Rd ← 00D2_9548

MUL Rd, Rn, Rm

The size of most arithmetic operations on ARM is 32-bits. This is the **word size** of the processor. ARMv4 defines the **word size instruction** shown in green. The blue register transfer level notation and the example provided illustrate capturing the least significant word of the 64-bit result for placement in Rd.

Instructions that store all 64-bits of the result. This set of instructions are **long-word size instructions**.



MULTIPLY-AND-ACCUMULATE

- ARMv4 has a **word-size multiply-and-accumulate:**

Rd ← lower 32-bits of (Rn x Rm) + Ra

$(0x003F928 \times 00000035) + 0x00000008 = 0x0000\_0000\_\underbrace{00D2\_9550}$

Rd ← 00D2_9550

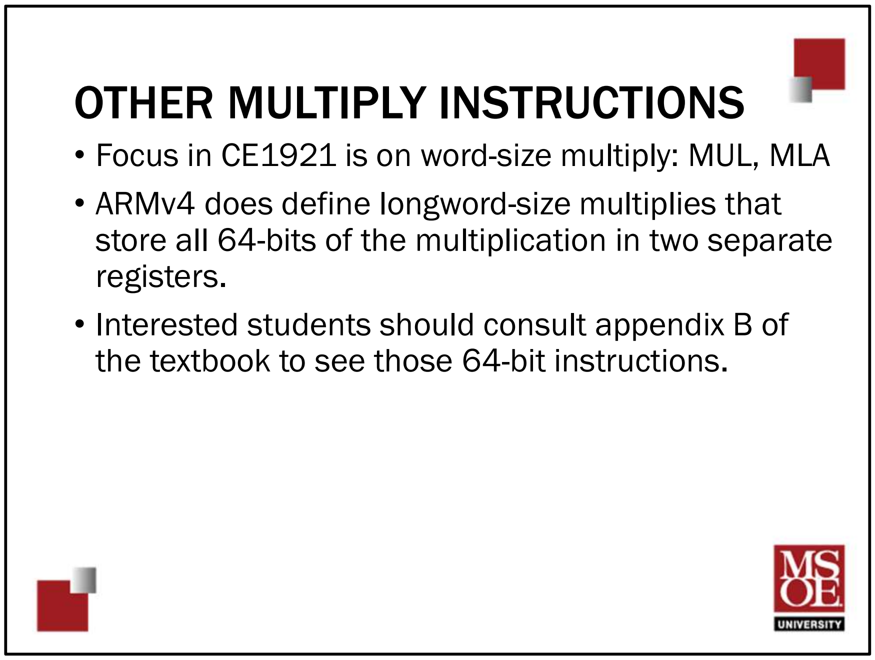
MLA Rd, Rn, Rm, Ra

MLA is the line equation: y = mx+b

- Useful in computer graphics and digital signal processing.
- Useful in a software division algorithm so that silicon space is not wasted for division circuit.

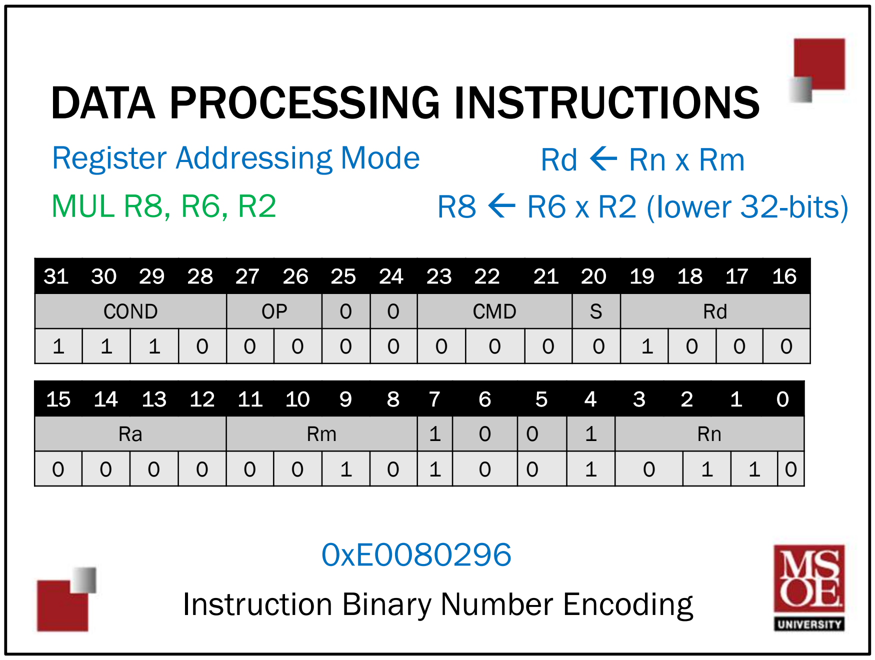
The size of most arithmetic operations on ARM is 32-bits. This is the **word size** of the processor. ARMv4 defines the **word size instruction** shown in green. The blue register transfer level notation and the example provided illustrate capturing the least significant word of the 64-bit result for placement in Rd. This is an incredibly powerful instruction that has useful application in computer graphics, division, and digital signal processing applications.

Instructions that store all 64-bits of the result. This set of instructions are **long-word size instructions**.



OTHER MULTIPLY INSTRUCTIONS

- Focus in CE1921 is on word-size multiply: MUL, MLA
- ARMv4 does define longword-size multiplies that store all 64-bits of the multiplication in two separate registers.
- Interested students should consult appendix B of the textbook to see those 64-bit instructions.



DATA PROCESSING INSTRUCTIONS

Register Addressing Mode $Rd \leftarrow Rn \times Rm$

MUL R8, R6, R2 $R8 \leftarrow R6 \times R2$ (lower 32-bits)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
COND				OP		0	0	CMD			S	Rd			
1	1	1	0	0	0	0	0	0	0	0	0	1	0	0	0

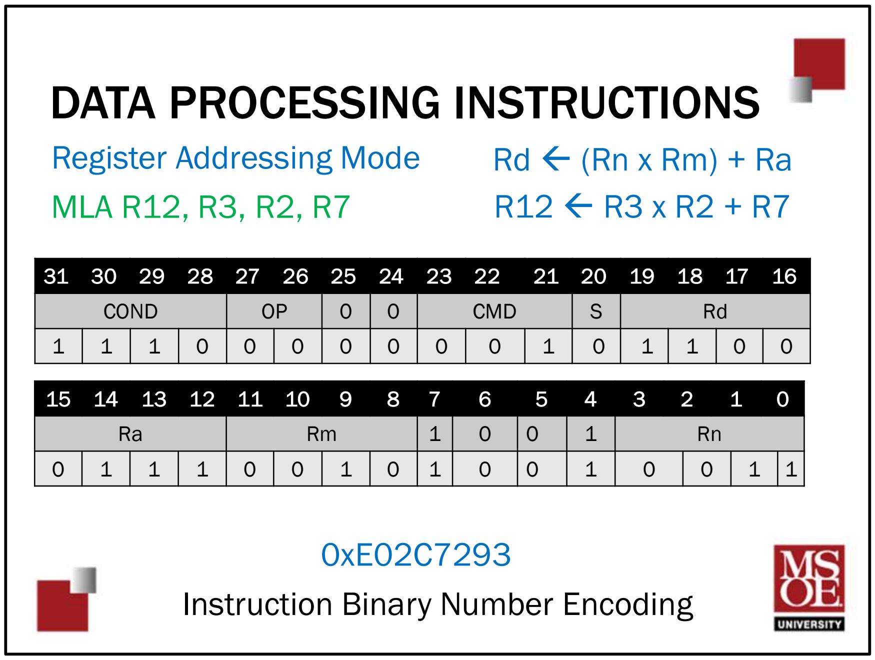
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Ra				Rm				1	0	0	1	Rn			
0	0	0	0	0	0	1	0	1	0	0	1	0	1	1	0

0xE0080296

Instruction Binary Number Encoding

Encode the instruction MUL R8, R6, R2 into its machine code binary number.

- **Identify** the addressing mode of data processing instructions by looking at the second ALU operand. In this case, the second operand is R2 – a register. This is **register addressing mode**.
- **Write** the register transfer level (RTL) equations in abstract and register-name forms.
- **Draw** the correct machine code format as a table of bits and bit field names.
- **Encode** suffix information. In this case, there is no S suffix and the conditional execution suffix is the omitted **always** suffix. This results in **S=0** and **COND= always=1110**.
- **Encode** the command field. For MUL, **CMD=0=000**.
- **Encode** the operands. **Rd=8=1000, Ra=unused=0000, Rm=2=0010**, **Rn=6=0110**.
- **Write** the final machine code binary number in hexadecimal.



DATA PROCESSING INSTRUCTIONS

Register Addressing Mode

Rd $\leftarrow$ (Rn x Rm) + Ra

MLA R12, R3, R2, R7

R12 $\leftarrow$ R3 x R2 + R7

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
COND				OP		0	0	CMD			S	Rd			
1	1	1	0	0	0	0	0	0	0	1	0	1	1	0	0

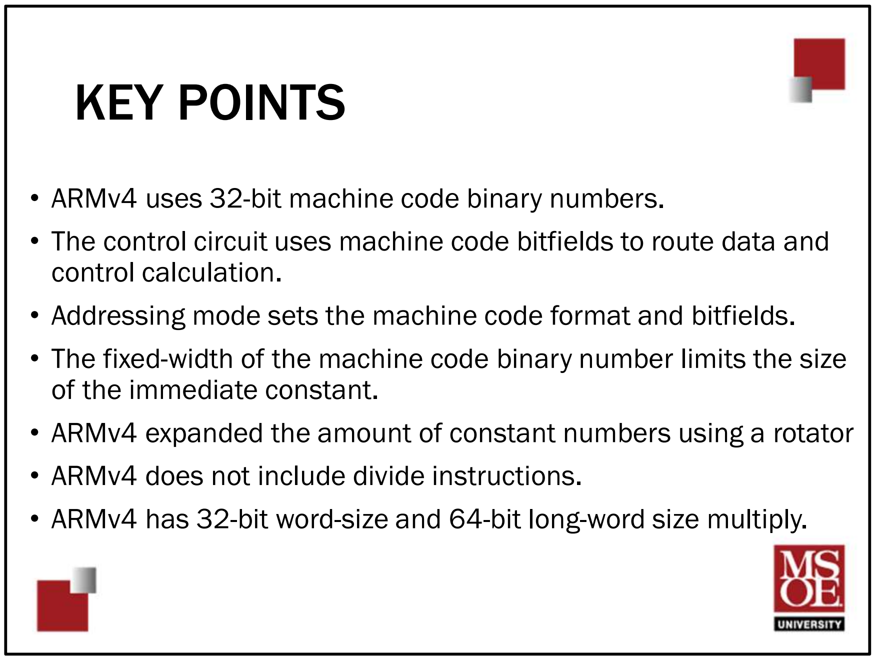
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Ra				Rm				1	0	0	1	Rn			
0	1	1	1	0	0	1	0	1	0	0	1	0	0	1	1

0xE02C7293

Instruction Binary Number Encoding

Encode the instruction MLA R12, R3, R2, R7 into its machine code binary number.

- **Identify** the addressing mode of data processing instructions by looking at the second ALU operand. In this case, the second operand is R2 – a register. This is **register addressing mode**.
- **Write** the register transfer level (RTL) equations in abstract and register-name forms.
- **Draw** the correct machine code format as a table of bits and bit field names.
- **Encode** suffix information. In this case, there is no S suffix and the conditional execution suffix is the omitted **always** suffix. This results in **S=0** and **COND= always=1110**.
- **Encode** the command field. For MLA, **CMD=1=001**.
- **Encode** the operands. **Rd=12=1100, Ra=7=0111, Rm=2=0010**, **Rn=3=0011**.
- **Write** the final machine code binary number in hexadecimal.



KEY POINTS

- ARMv4 uses 32-bit machine code binary numbers.
- The control circuit uses machine code bitfields to route data and control calculation.
- Addressing mode sets the machine code format and bitfields.
- The fixed-width of the machine code binary number limits the size of the immediate constant.
- ARMv4 expanded the amount of constant numbers using a rotator
- ARMv4 does not include divide instructions.
- ARMv4 has 32-bit word-size and 64-bit long-word size multiply.

MSOE UNIVERSITY

This summary slide notes the key points of this presentation. Continue to review this video when needed as your study the ARM instructions and use them to write assembly language programs.