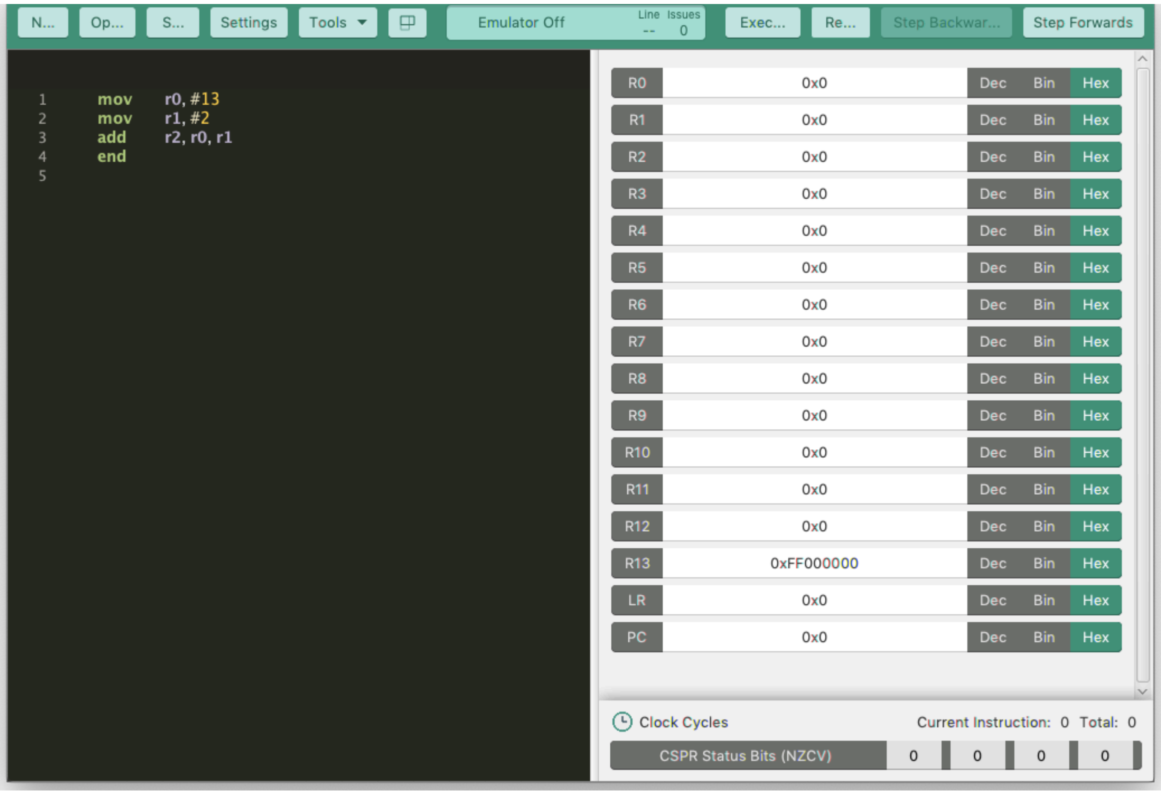
THE ARM INSTRUCTION SET

- VisUAL emulates a subset of the ARM instruction set

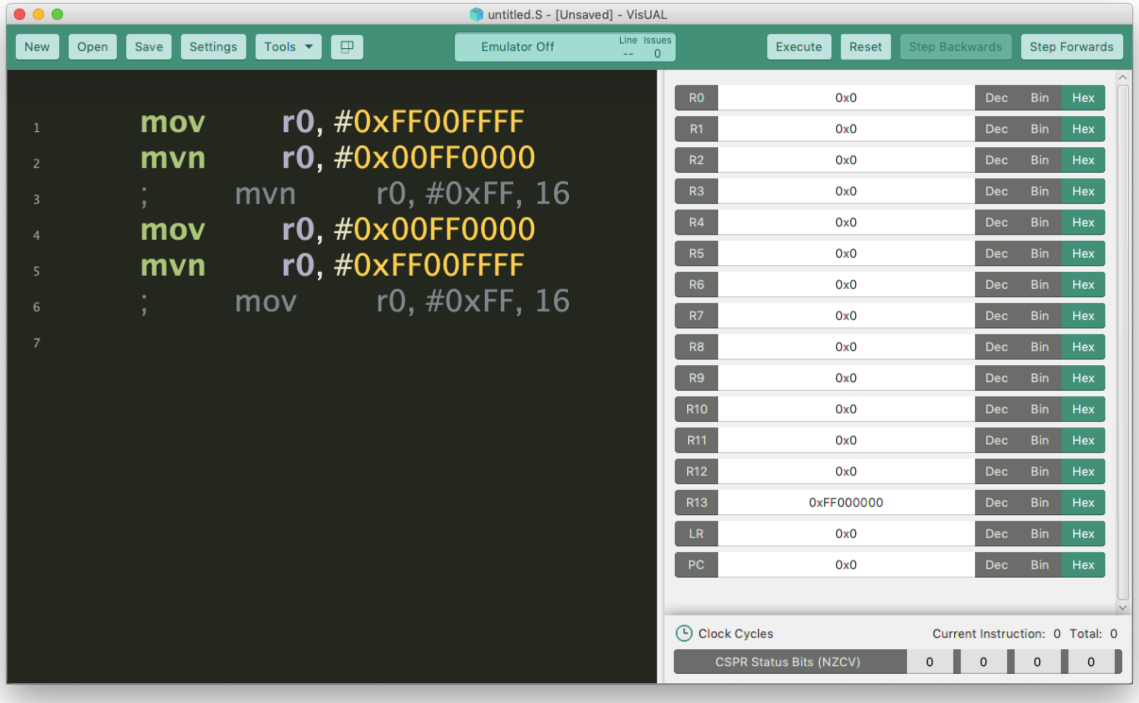


```
        mov     r0, #13
        mov     r1, #2
        add     r2, r0, r1
        end
```

R0	0x0	Dec	Bin	Hex
R1	0x0	Dec	Bin	Hex
R2	0x0	Dec	Bin	Hex
R3	0x0	Dec	Bin	Hex
R4	0x0	Dec	Bin	Hex
R5	0x0	Dec	Bin	Hex
R6	0x0	Dec	Bin	Hex
R7	0x0	Dec	Bin	Hex
R8	0x0	Dec	Bin	Hex
R9	0x0	Dec	Bin	Hex
R10	0x0	Dec	Bin	Hex
R11	0x0	Dec	Bin	Hex
R12	0x0	Dec	Bin	Hex
R13	0xFF000000	Dec	Bin	Hex
LR	0x0	Dec	Bin	Hex
PC	0x0	Dec	Bin	Hex

THE ARM INSTRUCTION SET

- The two commented-out instructions are not part of the emulated subset



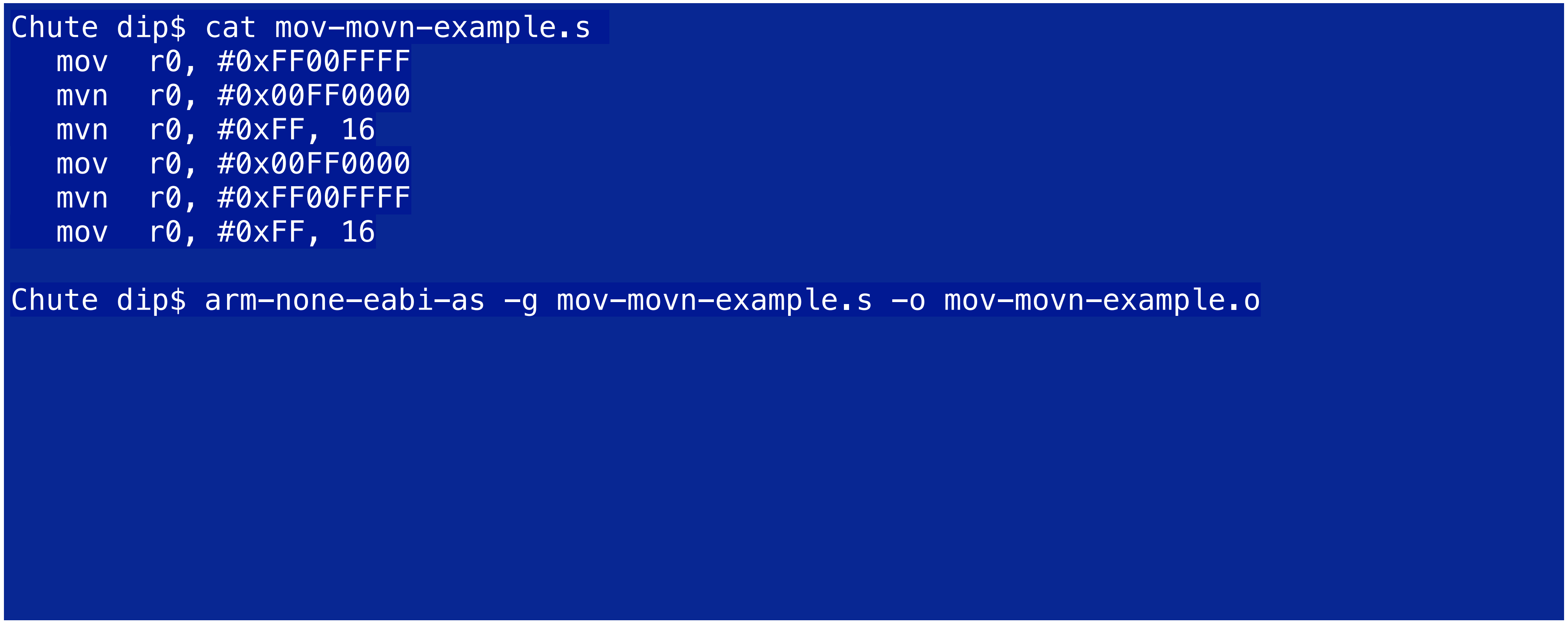
```
        mov     r0, #0xFF00FFFF
        mvn     r0, #0x00FF0000
        ;       mvn     r0, #0xFF, 16
        mov     r0, #0x00FF0000
        mvn     r0, #0xFF00FFFF
        ;       mov     r0, #0xFF, 16
```

R0	0x0
R1	0x0
R2	0x0
R3	0x0
R4	0x0
R5	0x0
R6	0x0
R7	0x0
R8	0x0
R9	0x0
R10	0x0
R11	0x0
R12	0x0
R13	0xFF000000
LR	0x0
PC	0x0

CSPR Status Bits (NZCV): 0 0 0 0

DATA PROCESSING INSTRUCTION ENCODING

- To analyze the encoding of these instructions the source assembly code (mov-movn-example.s) is translated to machine language (ML) using a real assembler (GNU arm-none-eabi-as)

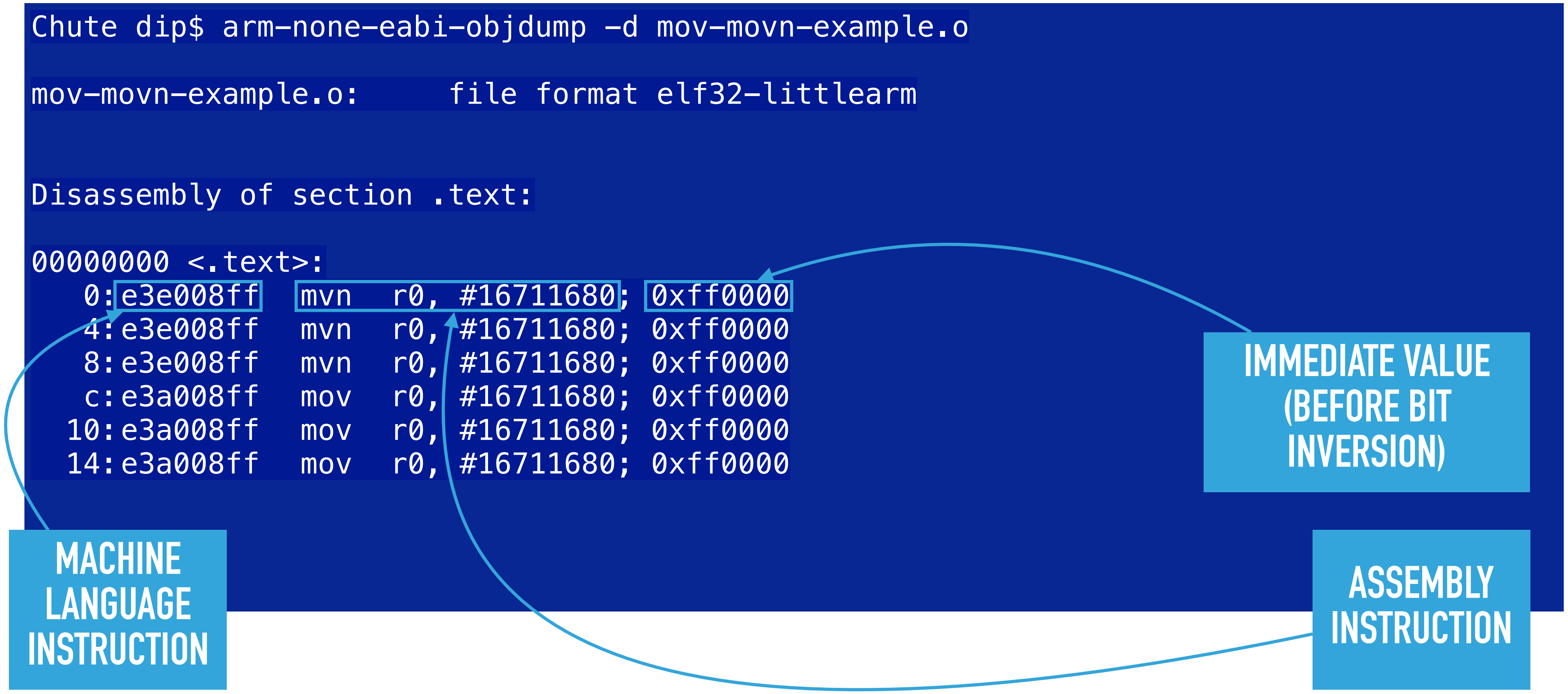


```
Chute dip$ cat mov-movn-example.s
   mov   r0, #0xFF00FFFF
   mvn   r0, #0x00FF0000
   mvn   r0, #0xFF, 16
   mov   r0, #0x00FF0000
   mvn   r0, #0xFF00FFFF
   mov   r0, #0xFF, 16

Chute dip$ arm-none-eabi-as -g mov-movn-example.s -o mov-movn-example.o
```

DATA PROCESSING INSTRUCTION ENCODING

- The resulting object code (mov-movn-example.o) is then disassembled (GNU arm-none-eabi-objdump) revealing how the six instructions were encoded



```
Chute dip$ arm-none-eabi-objdump -d mov-movn-example.o

mov-movn-example.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <.text>:
   0:e3e008ff   mvn   r0, #16711680; 0xff0000
   4:e3e008ff   mvn   r0, #16711680; 0xff0000
   8:e3e008ff   mvn   r0, #16711680; 0xff0000
   c:e3a008ff   mov   r0, #16711680; 0xff0000
  10:e3a008ff   mov   r0, #16711680; 0xff0000
  14:e3a008ff   mov   r0, #16711680; 0xff0000
```

IMMEDIATE VALUE (BEFORE BIT INVERSION)
MACHINE LANGUAGE INSTRUCTION
ASSEMBLY INSTRUCTION

DATA PROCESSING INSTRUCTION ENCODING

- The ML instructions to load an immediate into a register are `MOV Rd, #Imm, Rotate` and `MVN Rd, #Imm, Rotate`
  - The pseudo-instructions `MOV Rd, #LegalValue32` and `MVN Rd, #LegalValue32` are accepted by the Assembler
    - The Assembler takes care, if possible, of converting these into a `MOV Rd, #Imm, Rotate` and `MVN Rd, #Imm, Rotate`
    - The three assembly instructions below share the same Machine Language encoding (`mvn r0, #0xFF, 16`)

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
1	1	1	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1	1	1	1	1	1	1
E				3				E				0				0				8				F				F			
al						i	mvn									r0				Rotate=8x2=16				Imm=0xFF							
Cond				0	0	I	OpCode				S	Rn				Rd				Operand2											

Assembly instruction	ML encoding	Encoded instruction in assembly
`mov r0, #0xFF00FFFF`		
`mvn r0, #0x00FF0000`	E3E008FF	`mvnal r0, #0xFF, 16`
`mvn r0, #0xFF, 16`		

DATA PROCESSING INSTRUCTION ENCODING

- If the desired 32-bit immediate is `0x00FF0000`
  - Only one bit (bit 22) changes with respect to the encoding of `mov r0, #0xFF00FFFF`
    - The encoded instruction is `mov r0, #0xFF, 16`

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
1	1	1	0	0	0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1	1	1	1	1	1	1
E				3				A				0				0				8				F				F			
al						i	mov									r0				Rotate=8x2=16				Imm=0xFF							
Cond				0	0	I	OpCode				S	Rn				Rd				Operand2											

Assembly instruction	ML encoding	Encoded instruction in assembly
`mov r0, #0x00FF0000`	E3A008FF	`moval r0, #0xFF, 16`
`mvn r0, #0xFF00FFFF`		
`mov r0, #0xFF, 16`		

DATA PROCESSING INSTRUCTION ENCODING

- Shifted register second operand with instruction specified shift amount
  - bits 11-7 contain the shift/rotate amount
  - bits 6-5 indicate one of the shift operations: **`LSL (0b00)`**, **`LSR (0b01)`**, **`ASR (0b10)`**, and **`ROR (0b11)`**
  - bit 4 is **`0`**, bits 3-0 indicate the register containing the value to be shifted or rotated

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
1	1	1	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	1	1	1	0	0	0	0	0
E				1				A				0				1				0				E				0			
al						r	mov									r1				1					ROR			r0			
Cond				0	0	I	OpCode				S	Rn				Rd				Operand2											

Assembly instruction	ML encoding	Encoded instruction in assembly
`ror r1, r0, #1`	`E1A010E0`	`moval r1, r0, ROR #1`
`mov r1, r0, ROR #1`		

DATA PROCESSING INSTRUCTION ENCODING

- Shifted register second operand with instruction specified shift amount
  - bits 11-7 contain the shift/rotate amount
  - bits 6-5 indicate one of the shift operations: **`LSL (0b00)`**, **`LSR (0b01)`**, **`ASR (0b10)`**, and **`ROR (0b11)`**
  - bit 4 is **`0,`** bits 3-0 indicate the register containing the value to be shifted or rotated

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
1	1	1	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	0
E				1				A				0				1				1				8				0			
al						r	mov									r1				3					LSL			r0			
Cond				0	0	I	OpCode				S	Rn				Rd				Operand2											

Assembly instruction	ML encoding	Encoded instruction in assembly
`lsl r1, r0, #3`	`E1A01180`	`moval r1, r0, LSL #3`
`mov r1, r0, LSL #3`		

DATA PROCESSING INSTRUCTION ENCODING

- Rotate Right Extended (RRX) second operand (ROR special case)
  - CPSR C flag as a 33rd bit
  - Rotates right by 1 bit. **Encoded as ROR #0,** the assembler translates **`mov r1, r0, ROR #0`** into **`mov r1, r0`**

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
1	1	1	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0
E				1				A				0				1				0				6				0			
al						r	mov									r1				0					RRX			r0			
Cond				0	0	I	OpCode				S	Rn				Rd				Operand2											

Assembly instruction	ML encoding	Encoded instruction in assembly
`rrx r1, r0`	E1A01060	`moval r1, r0, RRX`
`mov r1, r0, RRX`		
`mov r1, r0, ROR #0`	E1A01000	`moval r1, r0`

DATA PROCESSING INSTRUCTION ENCODING

- Shifted register second operand with register specified shift amount
  - bits 11-8 contain the shift/rotate amount, bit **7** must be set to **`0`**
  - bits 6-5 indicate one of the shift operations: **`LSL (0b00)`**, **`LSR (0b01)`**, **`ASR (0b10)`**, and **`ROR (0b11)`**
  - bit 4 is **`1,`** bits 3-0 indicate the register containing the value to be shifted or rotated

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
1	1	1	0	0	0	0	0	0	1	0	0	0	1	0	1	0	1	0	0	0	0	1	0	0	0	1	1	0	1	1	1
E				0				4				5				4				2				3				7			
al						r	sub					r5				r4				2					LSR			r7			
Cond				0	0	I	OpCode				S	Rn				Rd				Operand2											

Assembly instruction	ML encoding	Encoded instruction in assembly
`sub r4,r5,r7,LSR r2`	`E0454237`	`subal r4, r5, r7, LSR r2`

BRANCH INSTRUCTION ENCODING

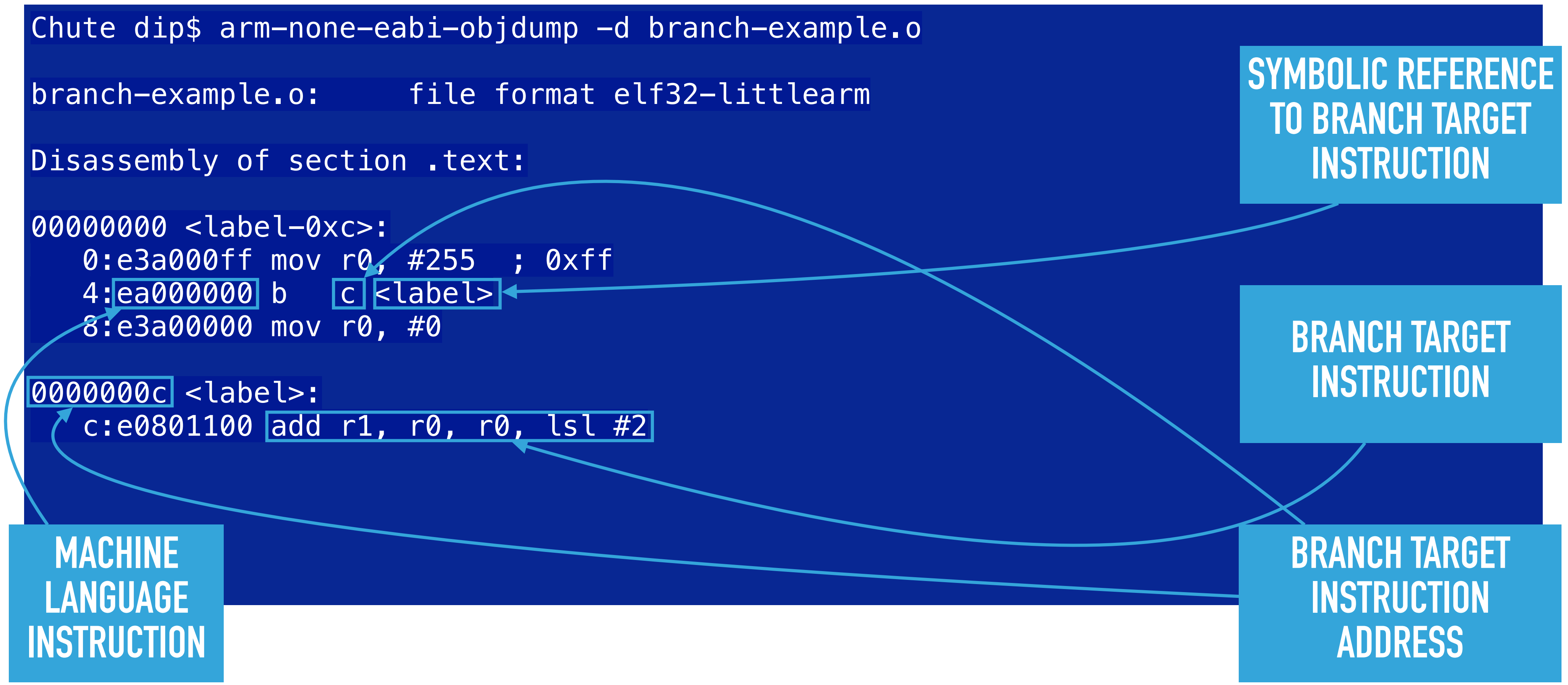
- To analyze the encoding of these instructions the source assembly code (branch-example.s) is translated to machine language (ML) using the assembler (GNU arm-none-eabi-as)

```
Chute dip$ cat branch-example.s
   mov r0, #0xff
   b label
   mov r0, #0x00
label:
   add r1, r0, r0, lsl #2

Chute dip$ arm-none-eabi-as branch-example.s -o branch-example.o
```

BRANCH INSTRUCTION ENCODING

- The resulting object code (branch-example.o) is then disassembled (GNU arm-none-eabi-objdump)



```
Chute dip$ arm-none-eabi-objdump -d branch-example.o

branch-example.o:     file format elf32-littlearm

Disassembly of section .text:

00000000 <label-0xc>:
   0:e3a000ff mov r0, #255  ; 0xff
   4:ea000000 b   c <label>
   8:e3a00000 mov r0, #0

0000000c <label>:
   c:e0801100 add r1, r0, r0, lsl #2
```

SYMBOLIC REFERENCE TO BRANCH TARGET INSTRUCTION
BRANCH TARGET INSTRUCTION
MACHINE LANGUAGE INSTRUCTION
BRANCH TARGET INSTRUCTION ADDRESS

BRANCH INSTRUCTION ENCODING

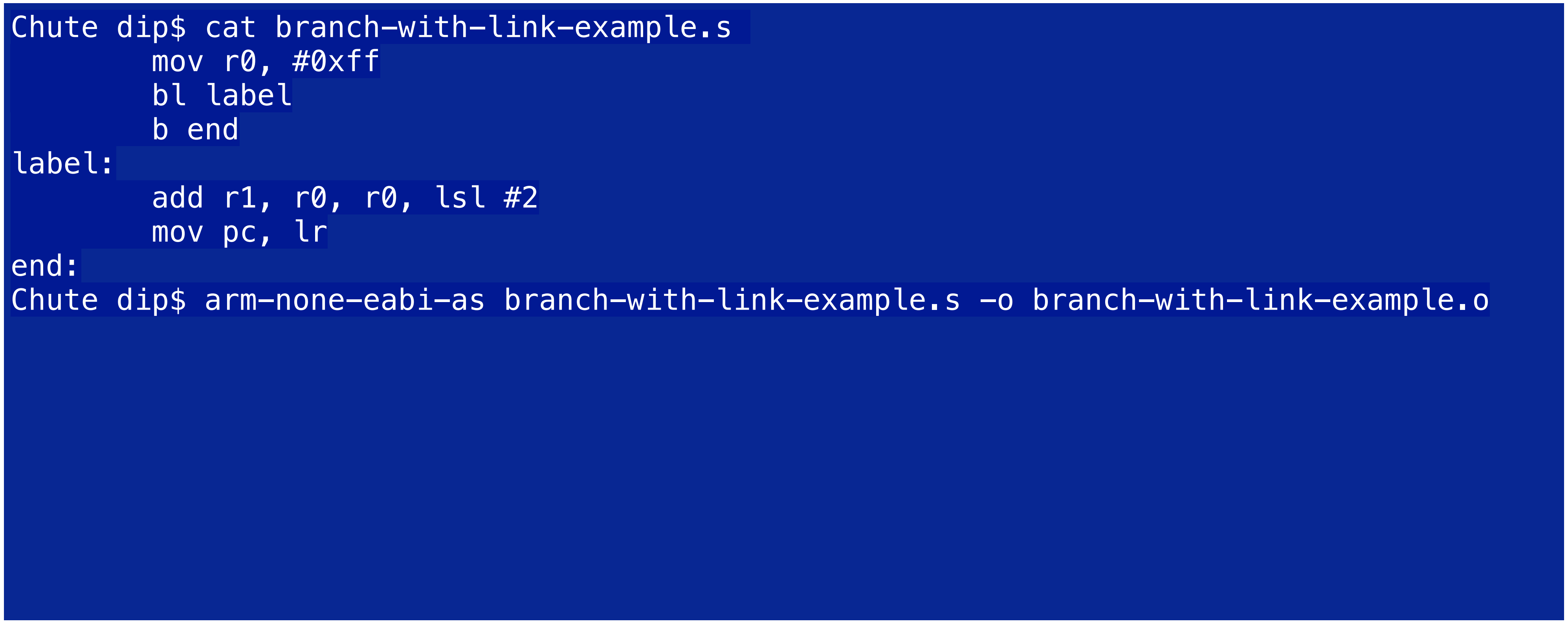
- Branch and Branch with Link instructions
  - bits 23-0 contain the offset value (two's complement); the offset is relative to the address of the branch instruction +2 words
    - the offset is shifted by two positions to the left and added to PC
  - bit 24 indicates whether the value PC-8 must be copied into LR (Link) before branching or not

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
1	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
E				A				0				0				0				0				0				0			
al							B	0000000																							
Cond							L	Offset																							

Assembly instruction	ML encoding	Encoded instruction in assembly
`b label`	EA000000	`bal c`

BRANCH INSTRUCTION ENCODING

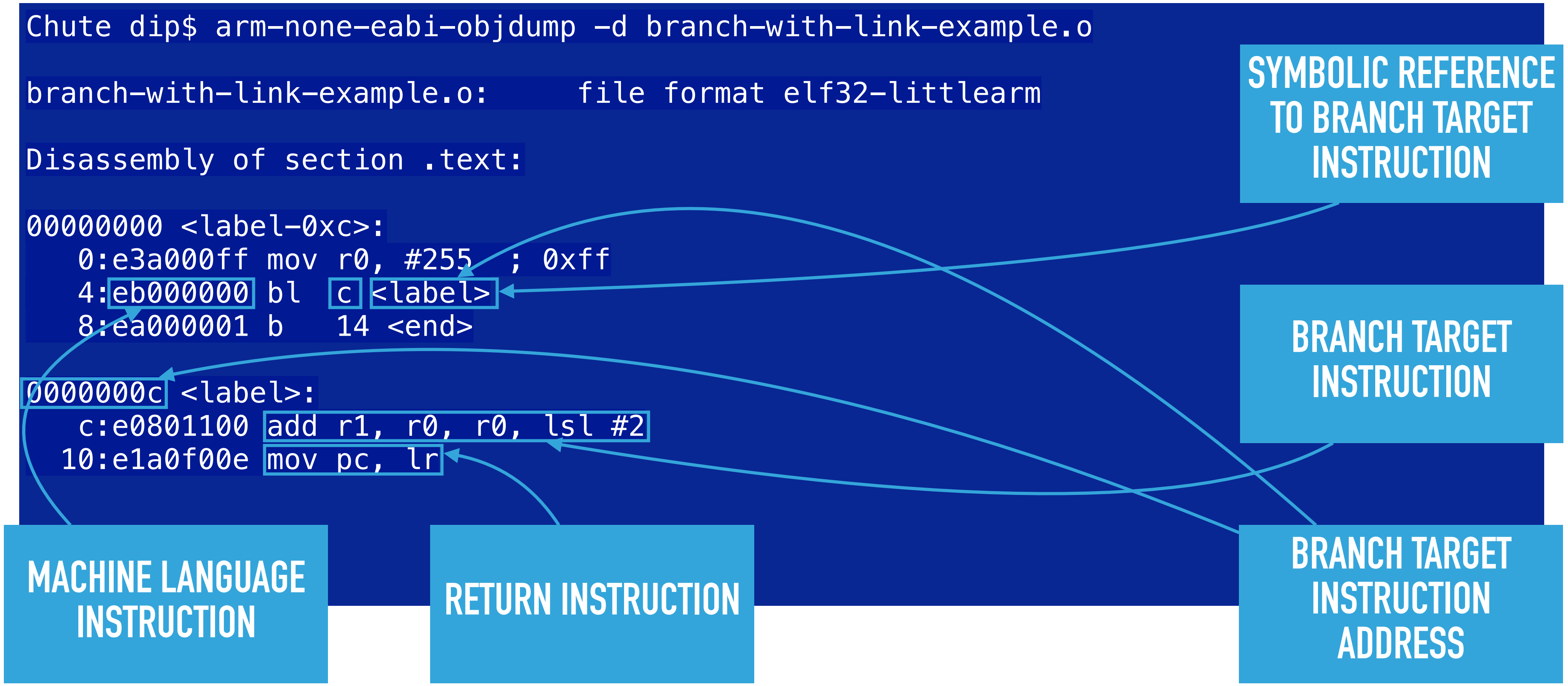
- Branch with link example (branch-with-link-example.s)



```
Chute dip$ cat branch-with-link-example.s
        mov r0, #0xff
        bl label
        b end
label:
        add r1, r0, r0, lsl #2
        mov pc, lr
end:
Chute dip$ arm-none-eabi-as branch-with-link-example.s -o branch-with-link-example.o
```

BRANCH INSTRUCTION ENCODING

- Disassembly of the Branch with Link example object code (branch-with-link-example.o)



```
Chute dip$ arm-none-eabi-objdump -d branch-with-link-example.o

branch-with-link-example.o:     file format elf32-littlearm

Disassembly of section .text:

00000000 <label-0xc>:
   0:e3a000ff mov r0, #255  ; 0xff
   4:eb000000 bl  c <label>
   8:ea000001 b   14 <end>

0000000c <label>:
   c:e0801100 add r1, r0, r0, lsl #2
  10:e1a0f00e mov pc, lr
```

SYMBOLIC REFERENCE TO BRANCH TARGET INSTRUCTION
BRANCH TARGET INSTRUCTION
MACHINE LANGUAGE INSTRUCTION
RETURN INSTRUCTION
BRANCH TARGET INSTRUCTION ADDRESS

BRANCH INSTRUCTION ENCODING

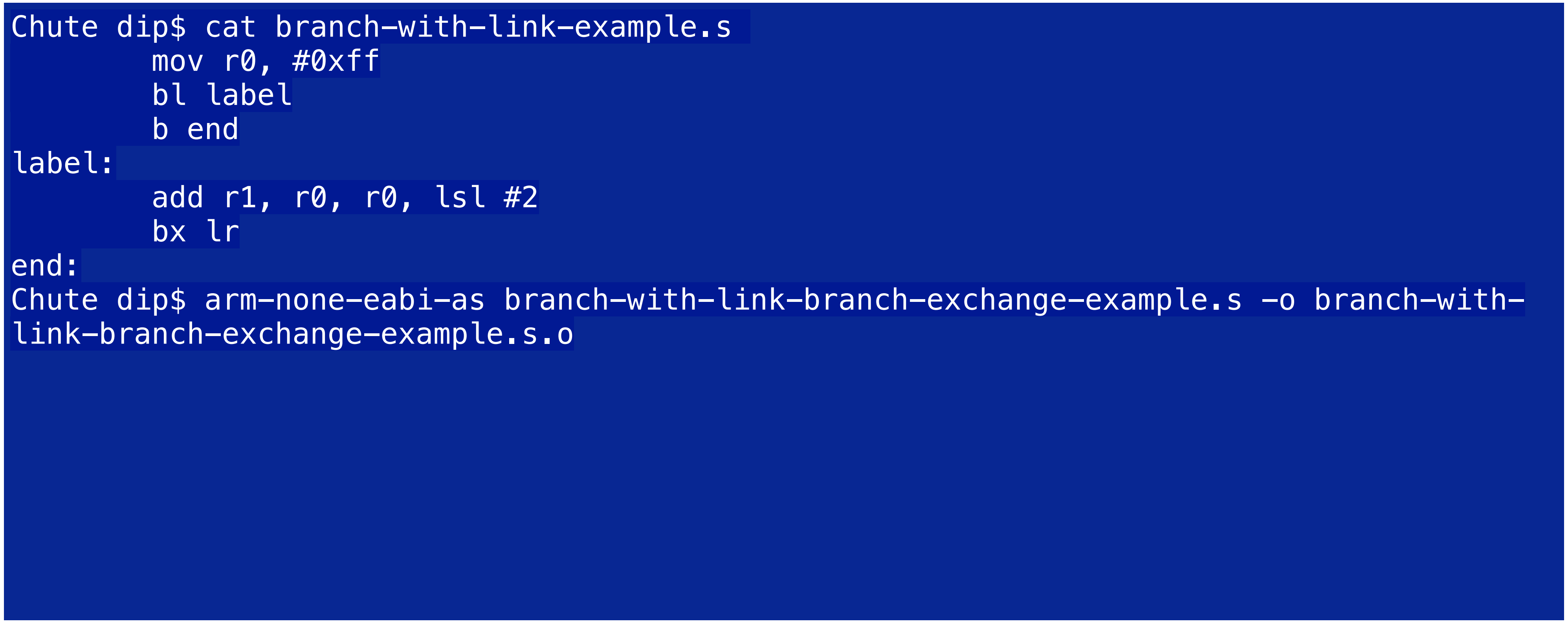
- Branch and Branch with Link instructions
  - bits 23-0 contain the offset value (two's complement); the offset is relative to the address of the branch instruction +2 words
    - the offset is shifted by two positions to the left and added to PC
  - bit 24 indicates whether the value PC-4 is copied into LR (Link) before branching or not

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
1	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
E				B				0				0				0				0				0				0			
al							L	0000000																							
Cond							L	Offset																							

Assembly instruction	ML encoding	Encoded instruction in assembly
`bl label`	`EB000000`	`blal c`

BRANCH INSTRUCTION ENCODING

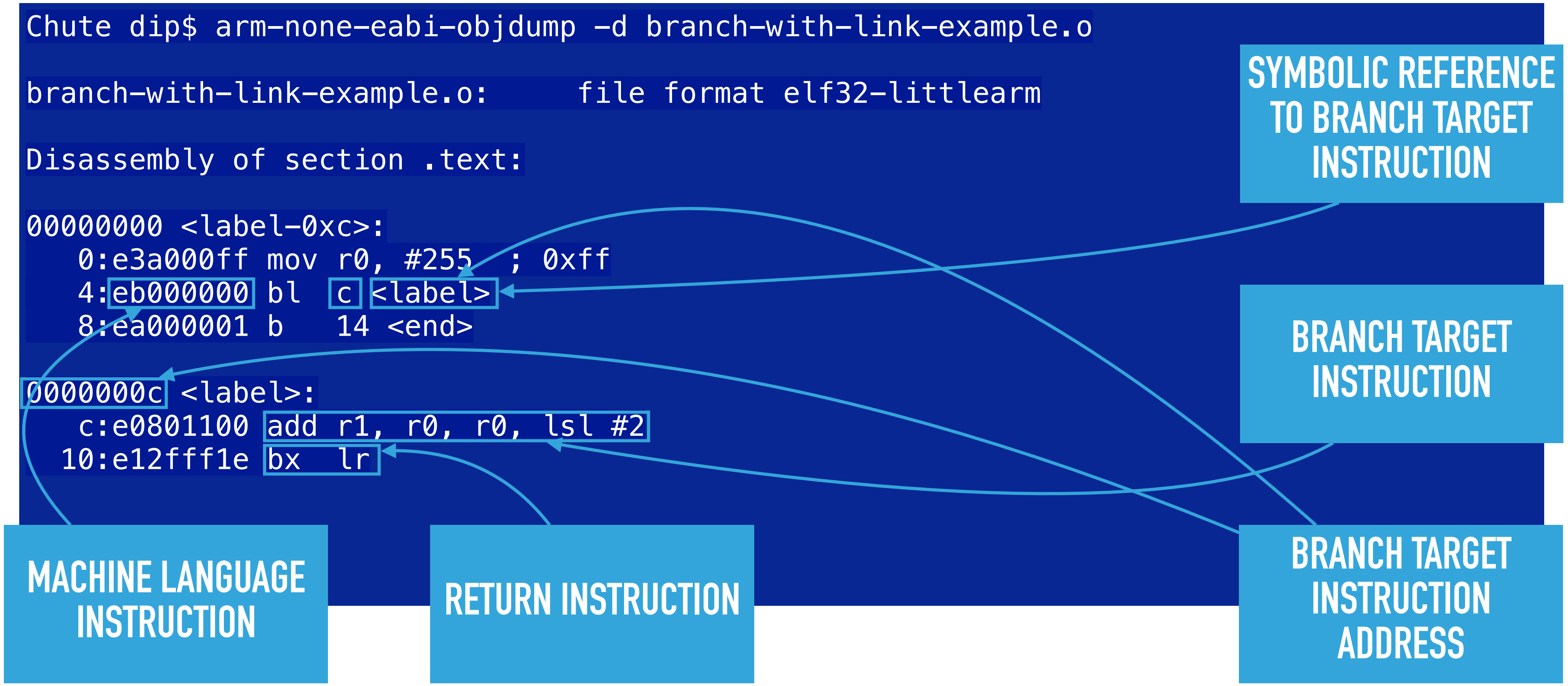
- Branch with Link and Branch eXchange example (branch-with-link-branch-exchange-example.s



```
Chute dip$ cat branch-with-link-example.s
        mov r0, #0xff
        bl label
        b end
label:
        add r1, r0, r0, lsl #2
        bx lr
end:
Chute dip$ arm-none-eabi-as branch-with-link-branch-exchange-example.s -o branch-with-
link-branch-exchange-example.s.o
```

BRANCH INSTRUCTION ENCODING

- Disassembly of the Branch with Link and Branch eXchange example object code (branch-with-link-branch-exchange-example.o)



```
Chute dip$ arm-none-eabi-objdump -d branch-with-link-example.o

branch-with-link-example.o:     file format elf32-littlearm

Disassembly of section .text:

00000000 <label-0xc>:
   0:e3a000ff mov r0, #255  ; 0xff
   4:eb000000 bl  c <label>
   8:ea000001 b   14 <end>

0000000c <label>:
   c:e0801100 add r1, r0, r0, lsl #2
  10:e12fff1e bx  lr
```

SYMBOLIC REFERENCE TO BRANCH TARGET INSTRUCTION
BRANCH TARGET INSTRUCTION
MACHINE LANGUAGE INSTRUCTION
RETURN INSTRUCTION
BRANCH TARGET INSTRUCTION ADDRESS

BRANCH INSTRUCTION ENCODING

- Branch eXchange instruction
  - the target address is contained in the register specified in bits 3-0
  - execution continues in ARM state or Thumb state depending on the value of bit 0 of the target address
    - if the bit is set execution continues in Thumb mode, otherwise it continues in ARM-32 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
1	1	1	0	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	0
E				1				2				F				F				F				1				E			
al																												lr			
Cond																												Rn			

Assembly instruction	ML encoding	Encoded instruction in assembly
`bx lr`	`E12FFF1E`	`bxal lr`

ARM ASSEMBLY PROGRAMMING

- Exercise 1 (ARM Assembly, Simulation)
    - Write an assembly program that loads 2 in register `r0`, 3 in `r1` and puts `r1+r2` into `r2`
        - This deals with basic ARM assembly syntax

ARM ASSEMBLY PROGRAMMING

- Exercise 2 (ARM Assembly, Simulation)
  - Write an assembly program that loads 2 in register `r0` and then executes `r0:=r0*33`
    - This deals with basic ARM assembly syntax and the flexible second operand

ARM ASSEMBLY PROGRAMMING

- Exercise 3 (ARM Assembly, Simulation)
  - Write an assembly program that loads 2 in register `r0` and then executes `r0:=r0*15`
    - This deals with basic ARM assembly syntax and the flexible second operand.