Topic 10: Instruction Representation

CSE 30: Computer Organization and Systems Programming
Summer Session II

Dr. Ali Irturk
Dept. of Computer Science and Engineering
University of California, San Diego

Stored-Program Concept

- Computers built on 2 key principles:
  1) Instructions are represented as numbers.
  2) Therefore, entire programs can be stored in memory to be read or written just like numbers (data).
- Simplifies SW/HW of computer systems:
  - Memory technology for data also used for programs

Consequence #1: Everything Addressed

- Since all instructions and data are stored in memory as numbers, everything has a memory address: instructions, data words
  - both branches and jumps use these
- C pointers are just memory addresses: they can point to anything in memory
  - Unconstrained use of addresses can lead to nasty bugs; up to you in C; limits in Java
- One register keeps address of instruction being executed: "Program Counter" (PC)
  - Basically a pointer to memory: Intel calls it Instruction Address Pointer, a better name

Consequence #2: Binary Compatibility

- Programs are distributed in binary form
  - Programs bound to specific instruction set
  - Different version for Macintosh and IBM PC
- New machines want to run old programs ("binaries") as well as programs compiled to new instructions
- Leads to instruction set evolving over time
- Selection of Intel 8086 in 1981 for 1st IBM PC is major reason latest PCs still use 80x86 instruction set (Pentium 4); could still run program from 1981 PC today

Instructions as Numbers

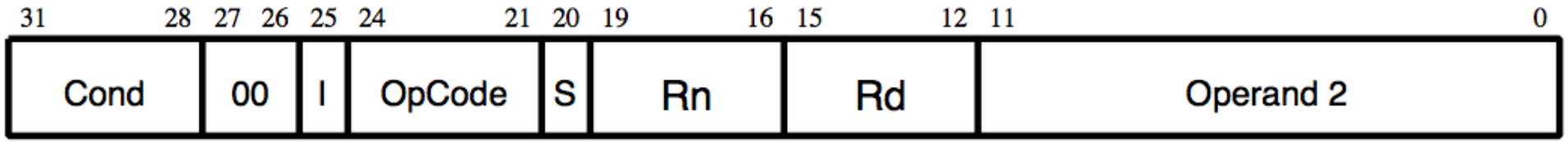
- Most of the data that we work with is in words (32-bit blocks):
  - Each register is a word.
  - `LDR` and `STR` both access memory one word at a time.
- So how do we represent instructions?
  - Remember: Computer only understands 1s and 0s, so "`ADD r0,r0,#0`" is meaningless.
  - "Regular" ARM instructions are 32 bits.
  - Thumb ARM instructions are 16 bits.

Instructions as Numbers

- Divide instruction word into "fields"
- Each field tells computer something about instruction
- There is an attempt to reuse fields across instructions as much as possible. Simple = less complex hardware which generally leads to better performance

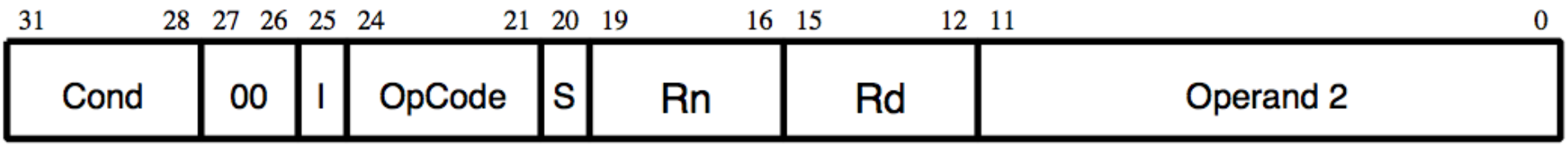
Data Processing Instructions

- Define "fields":



- Important: Each field is viewed as an independent unsigned integer, not as part of a 32-bit integer.
  - Consequence: 5-bit fields can represent any number 0-31, while 6-bit fields can represent any number 0-63.

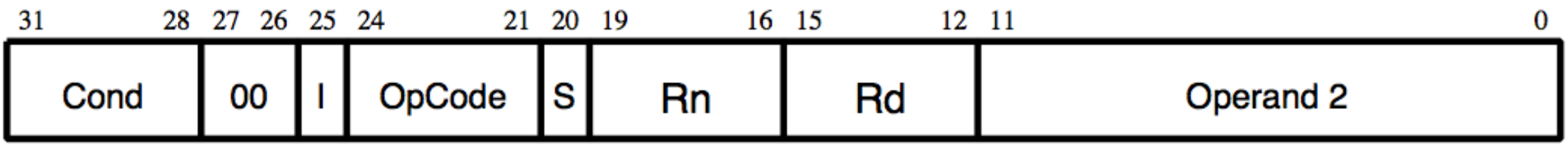
Data Processing Instructions



- What do these field integer values tell us?
  - OpCode: specifies the specific data processing instruction

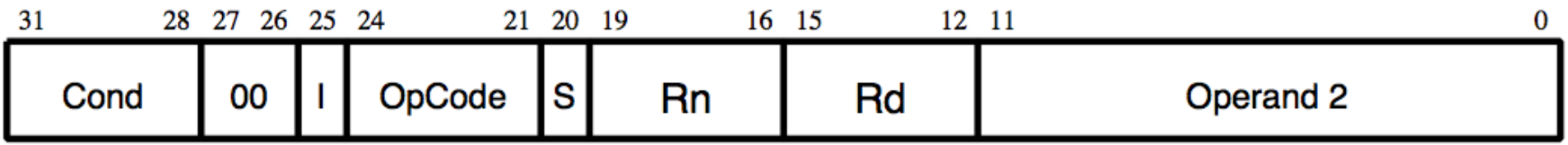
0000 = AND - Rd:= Op1 AND Op2
0001 = EOR - Rd:= Op1 EOR Op2
0010 = SUB - Rd:= Op1 - Op2
0011 = RSB - Rd:= Op2 - Op1
0100 = ADD - Rd:= Op1 + Op2
0101 = ADC - Rd:= Op1 + Op2 + C
0110 = SBC - Rd:= Op1 - Op2 + C - 1
0111 = RSC - Rd:= Op2 - Op1 + C - 1
1000 = TST - set condition codes on Op1 AND Op2
1001 = TEQ - set condition codes on Op1 EOR Op2
1010 = CMP - set condition codes on Op1 - Op2
1011 = CMN - set condition codes on Op1 + Op2
1100 = ORR - Rd:= Op1 OR Op2
1101 = MOV - Rd:= Op2
1110 = BIC - Rd:= Op1 AND NOT Op2
1111 = MVN - Rd:= NOT Op2

Data Processing Instructions

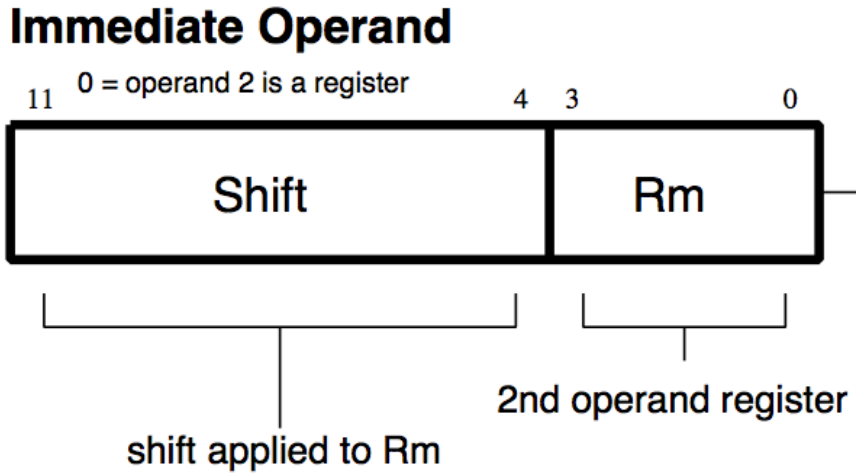


- What do these field integer values tell us?
  - Rd: Destination Register
  - Rn: 1st Operand Register
- Each register field is 4 bits, which means that it can specify any unsigned integer in the range 0-15. Each of these fields specifies one of the 16 registers by number.

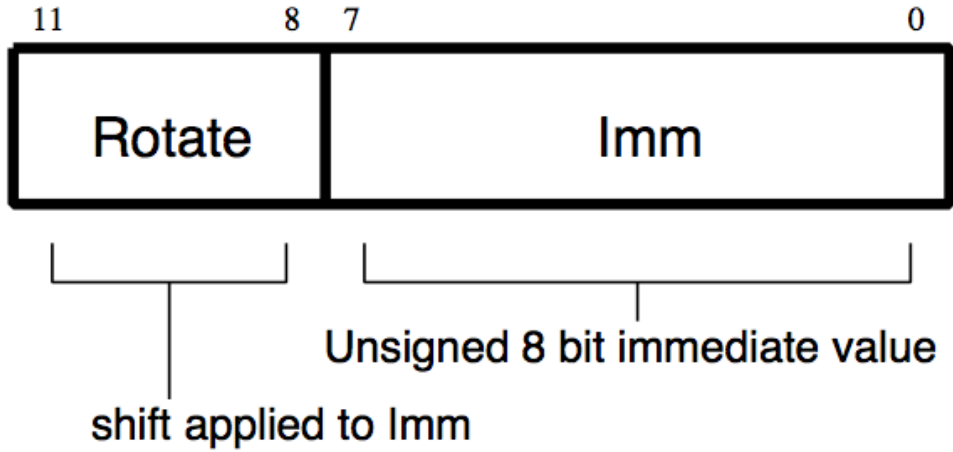
Data Processing Instructions



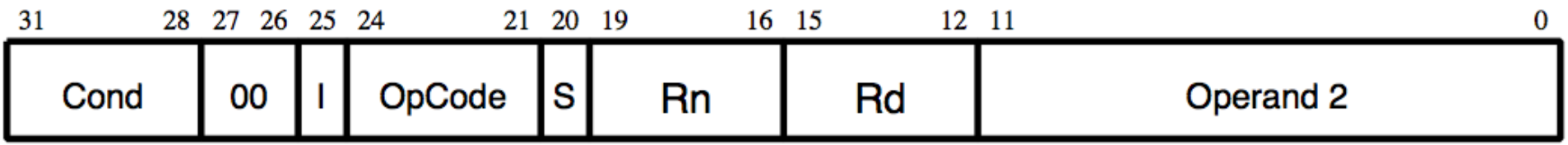
- More fields:
  - I (Immediate)
    - 0: Operand 2 is a Register



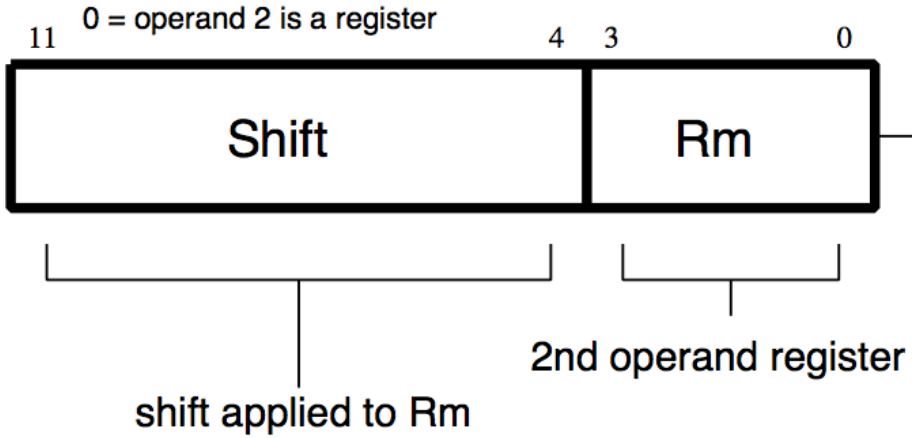
    - 1: Operand 2 is an Immediate



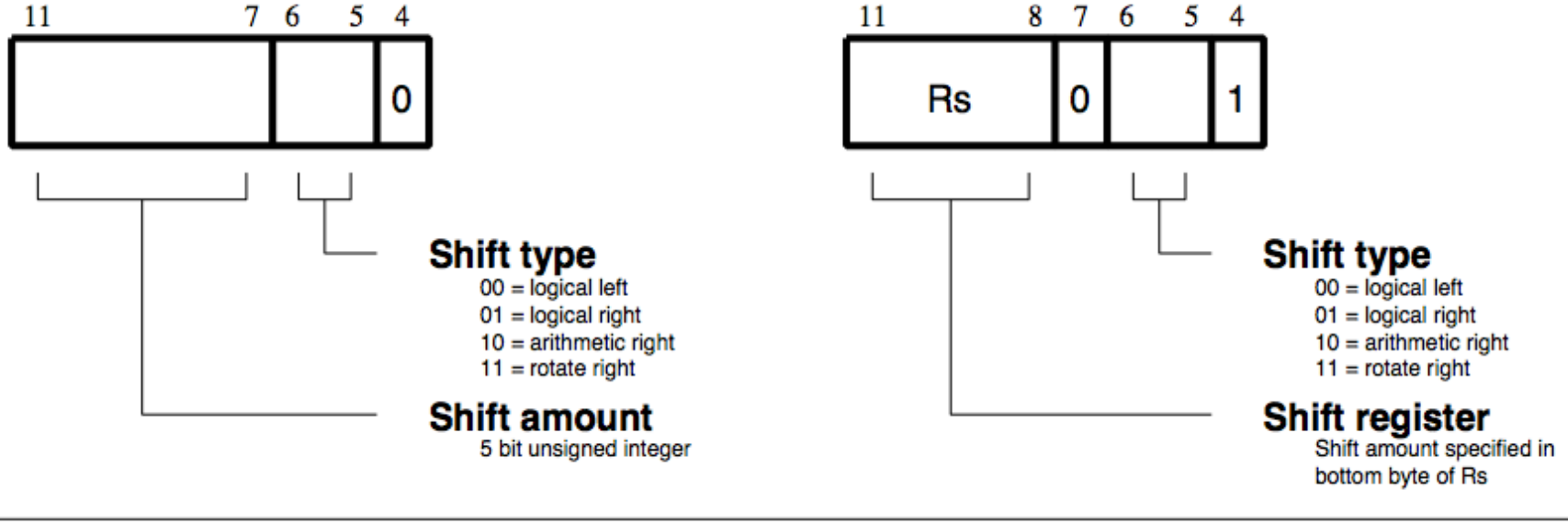
Data Processing Instructions



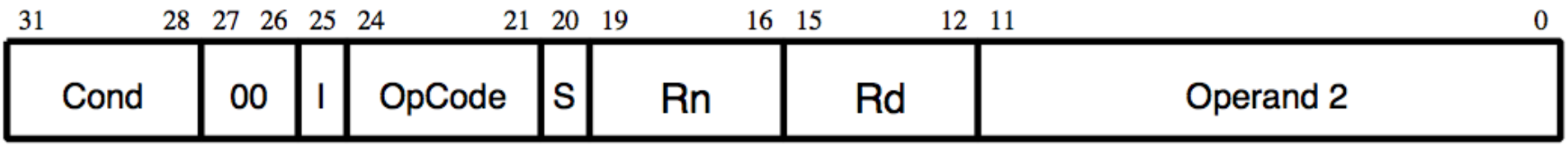
- Operand 2:



- Shift field:



Data Processing Instructions



- More fields:
  - S (Set Condition Fields)
    - 0: Do not alter condition fields
    - 1: Set condition fields
    - These set the C, N, Z, V flags in the CPSR
  - Cond (Condition Field)
    - Determines whether the instruction resulted is committed
    - Used with ADDEQ, ADDNE, ADDGT, etc.

Condition Field

Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

Data Processing Example

- ARM Instruction: `ADD r0,r1,r2`

`cond = 1110` (always – unconditional)

`S = 0` (not ADDS)

`I = 0` (second operand not a constant)

`opcode = 0100` (look up)

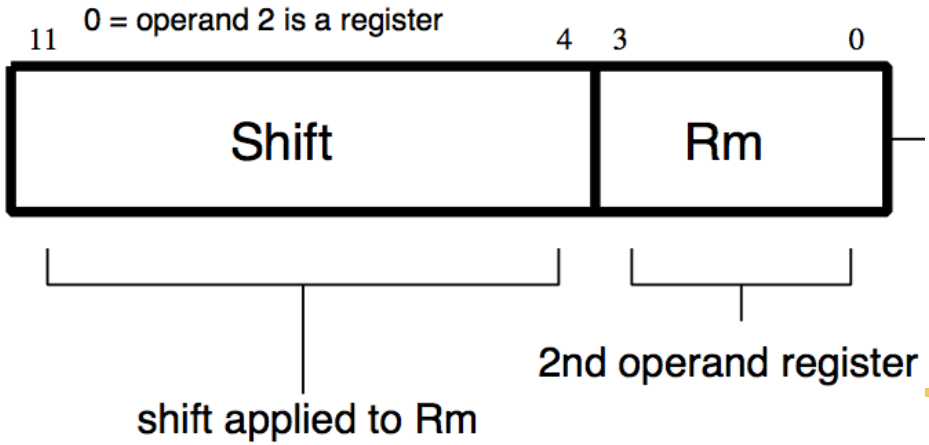
`rd` = 0 (destination *operand*)

`rn` = 1 (first *operand*)

`operand 2`

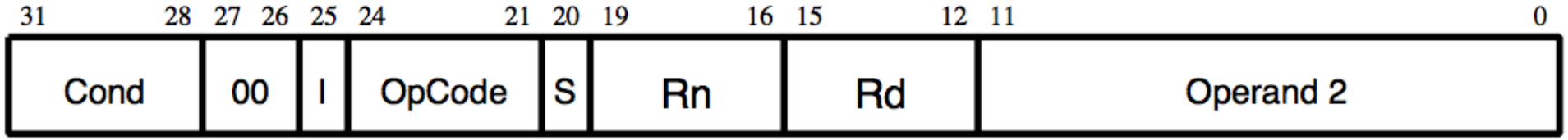
`shift = 0`

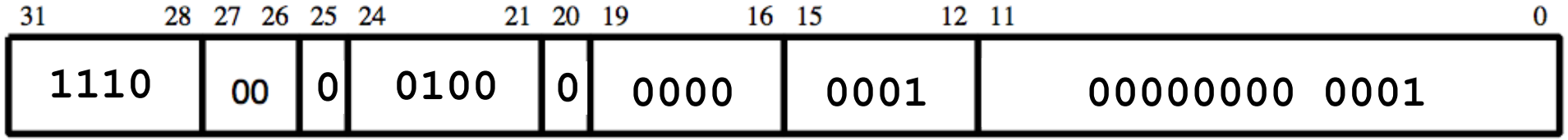
`Rm = 2`



Data Processing Example

- ARM Instruction: `ADD r0,r1,r2`





Binary number per field representation:

hex representation: 0xE0810002

- Called a Machine Language Instruction

Data Processing Example

- ARM Instruction: `SUBEQ a1,v7,r2,LSL #3`

`cond = 0000` (equal)

`S = 0` (not SUBEQS)

`I = 0` (second operand not a constant)

`opcode = 0010` (look up)

`rd = 0000` (destination *operand:* a1 = r0)

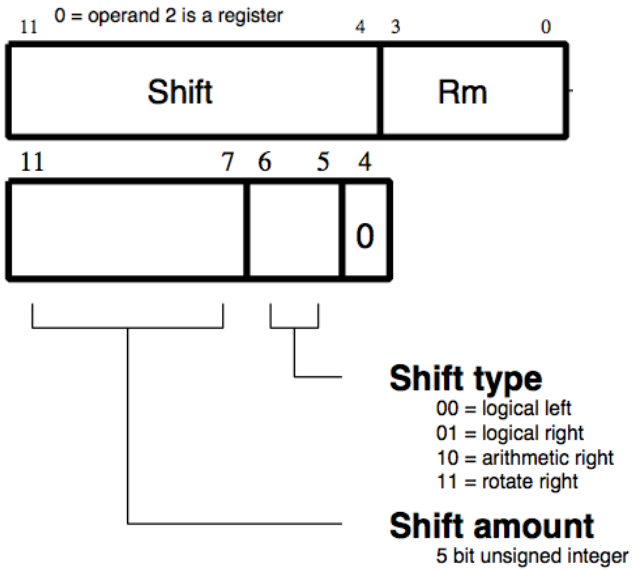
`rn = 1010` (1st *operand*: *v7 = r10*)

`operand 2`

`rm = 0010` (2nd *operand*: *r2*)

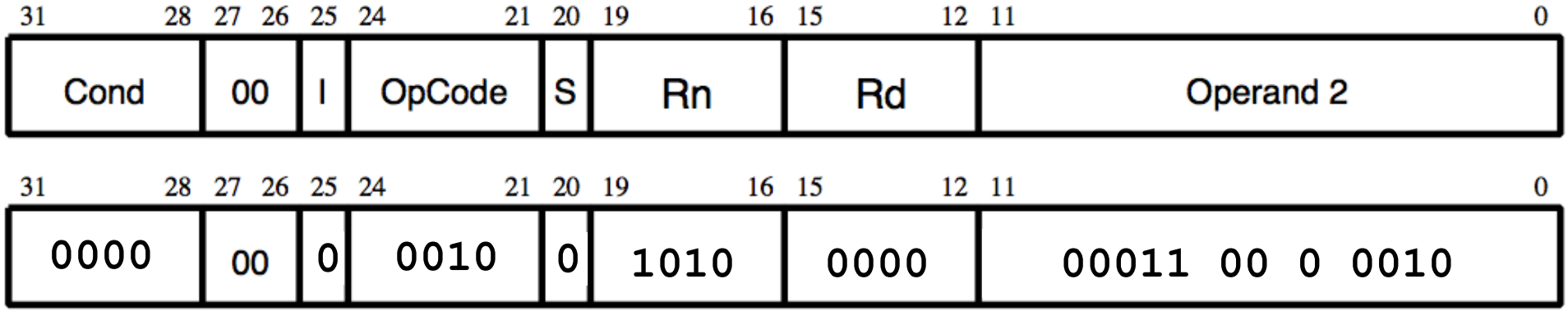
`shift amount = 00011`

`Shift Type = 00`

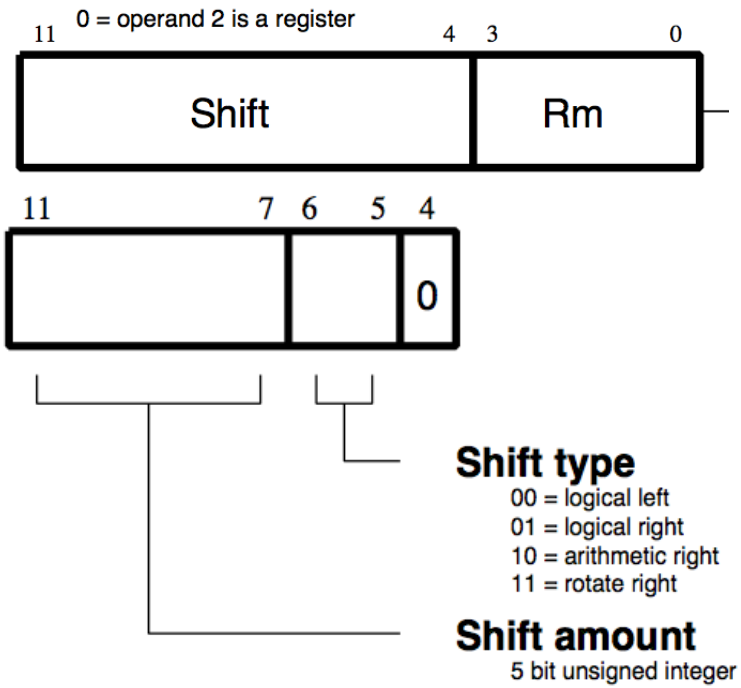


Data Processing Example

- ARM Instruction: `SUBEQ a1,v7,r2,LSL`



Hex representation: 0x004A0182



Data Processing Example

- ARM Instruction: `BIC v6,a1,a2,ROR v2`

  `cond = 1110` (always)

  `S = 0` (not BICS)

  `I = 0` (second operand not a constant)

  `opcode = 1110` (look up)

  `rd = 1001` (destination *operand:* v6 = r9)

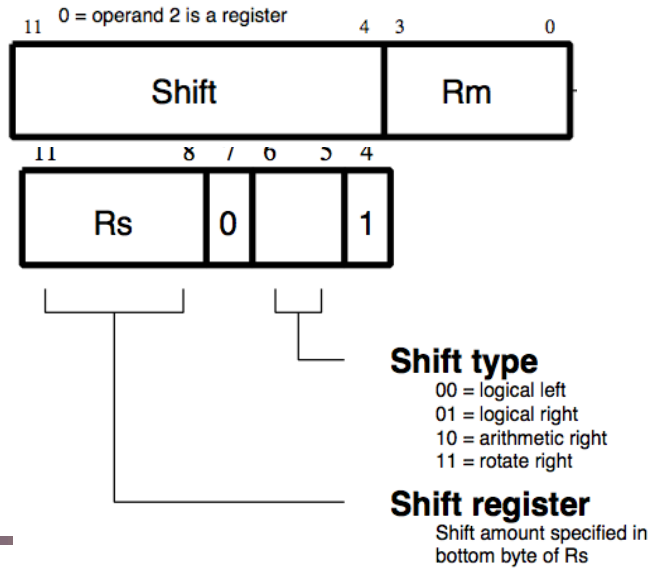
  `rn = 0000` (1st *operand*: *a1* = *v0*)

  `operand 2`

  `rm = 0001` (2nd *operand*: *a2* = *v1*)

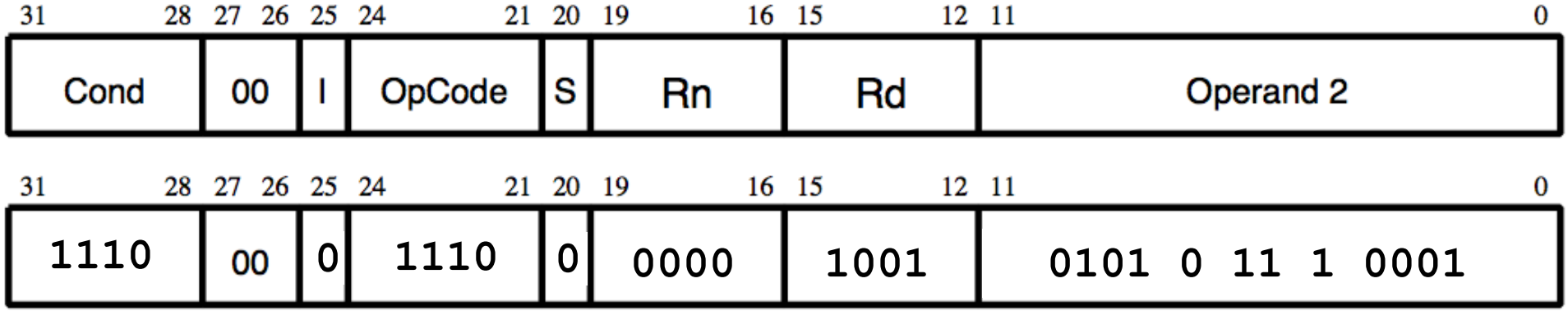
  `rs = 1010` (shift operand: v2 =r5)

  `Shift Type = 11 (ROR)`

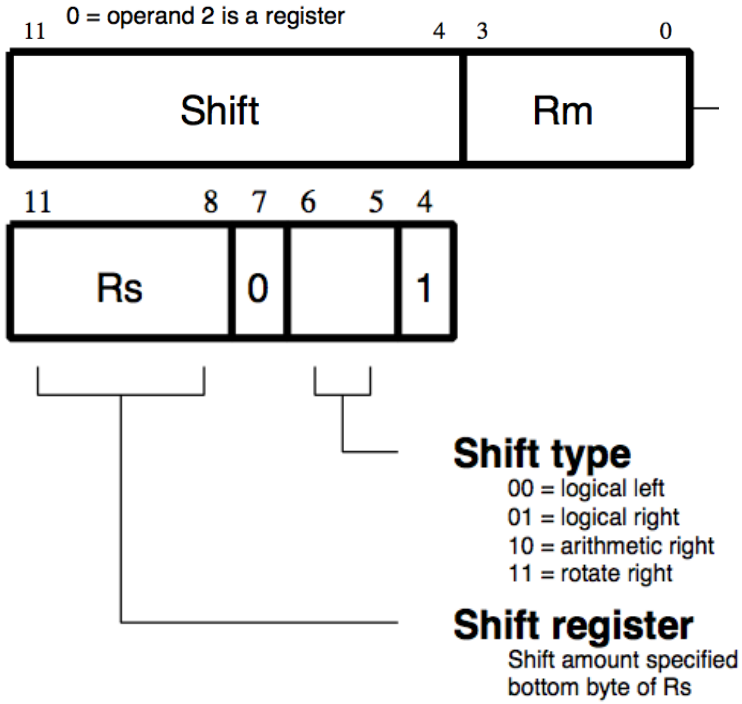


Data Processing Example

- ARM Instruction: `BIC v6,a1,a2,ROR v2`



Hex representation: 0xE1C09571



Data Processing Example

- ARM Instruction: `EORGTS r6,r2,#10`

  `cond = 1100` (greater than)

  `S = 1`

  `I = 1` (second operand is a constant)

  `opcode = 0001` (look up)

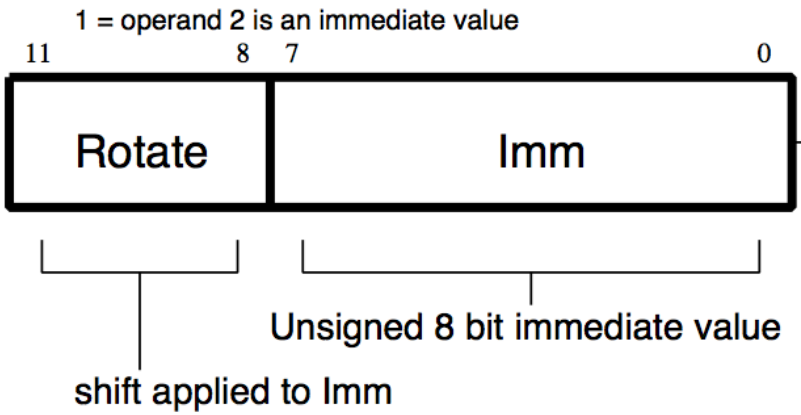
  `rd = 0110` (destination *operand:* r6)

  `rn = 0010` (1st *operand*: *r2*)

  `operand 2`

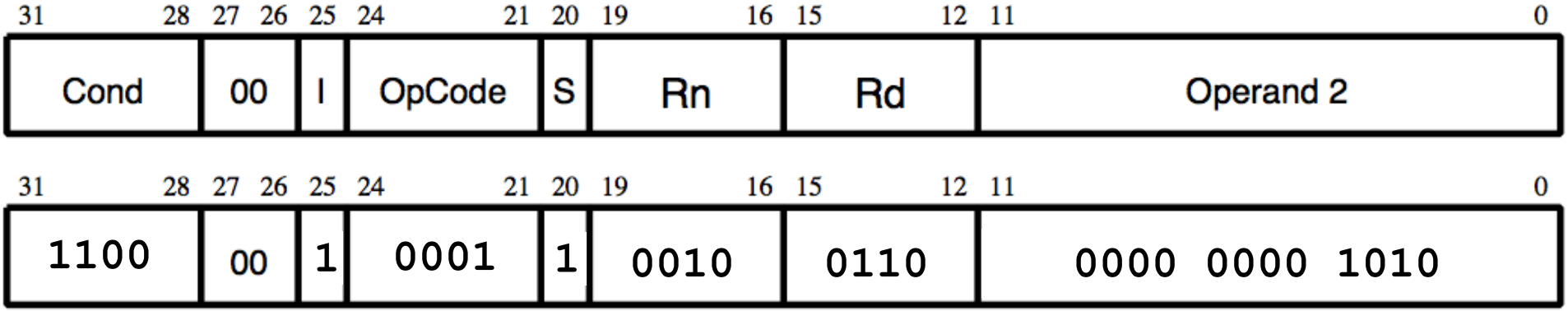
  `rotate = 0000`

  `imm = 0x0A`

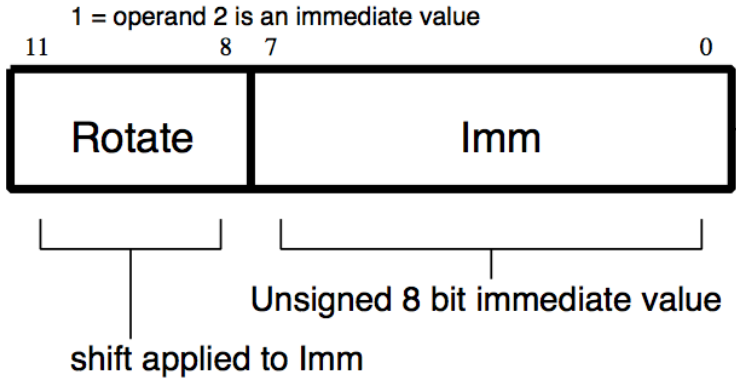


Data Processing Example

- ARM Instruction: `EORGTS r6,r2,#10`

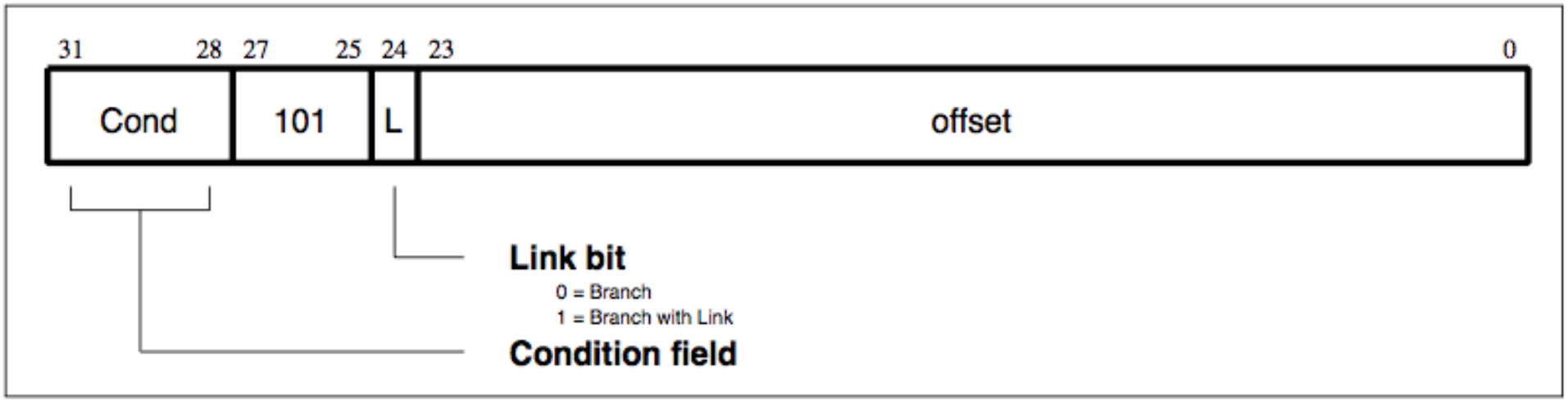


Hex representation: 0xC232600A

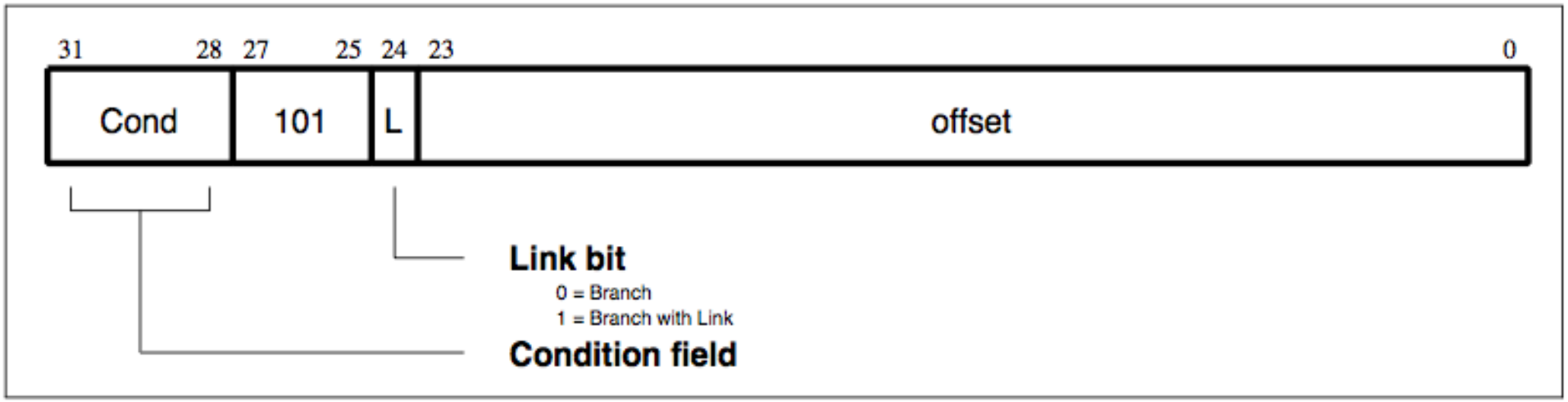


Branch Instructions

- What fields do we need for Branch Instructions?
  - Opcode
  - Label
  - Condition

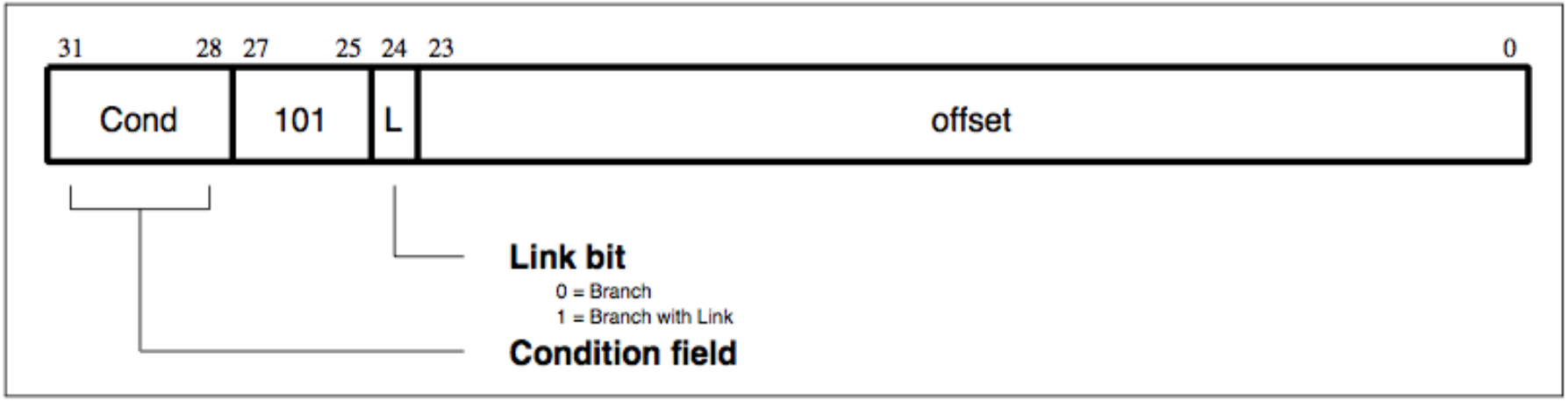


Branch Instructions



- Opcode
  - 101 for Branch
  - Link bit is set if BL
- Condition – same as before (`EQ, LT, GT,` etc.)

Branch Instructions



- Label
  - Uses offset, i.e., how far to branch. PC = PC + offset.
  - As opposed to absolute address of the instruction.
  - 24 bits allows us to branch +/- 8 Mbytes.
  - Instructions 32 bits = 4 bytes, so add implicit 00 at end. Now can branch +/- 32 Mbytes.
  - Equivalent to +/- 8 Minstructions.
  - Start counting 2 instructions later (due to pipelining).

Branches: PC-Relative Addressing

- Solution to branches in a 32-bit instruction: PC-Relative Addressing
- Let the 24-bit `offset` field be a signed two's complement integer to be *added* to the PC if we take the branch.
- Now we can branch +/- 2^{23} bytes from the PC, which should be enough to cover almost any loop.
- Any ideas to further optimize this?

Branches: PC-Relative Addressing

- Note: Instructions are words, so they're word aligned (byte address is always a multiple of 4, which means it ends with 00 in binary).
  - So the number of bytes to add to the PC will always be a multiple of 4.
  - So specify the `offset` in words.
- Now, we can branch +/- 2^{23} words from the PC (or +/- 2^{25} bytes), so we can handle loops 4 times as large.

Branches: PC-Relative Addressing

- Branch Calculation:
  - If we don't take the branch:

    PC = PC + 4

  - If we do take the branch:

    PC = (PC + 8) + (`offset` * 4)

    Why two? Pipelining starts next two following instructions before current one is finished.

Branches: PC-Relative Addressing

- Branch Calculation:
  - Observations
    - `offset` field specifies the number of words to jump, which is simply the number of instructions to jump.
    - `offset` field can be positive or negative.
    - Due to hardware, add `immediate` to (PC+8), not to PC

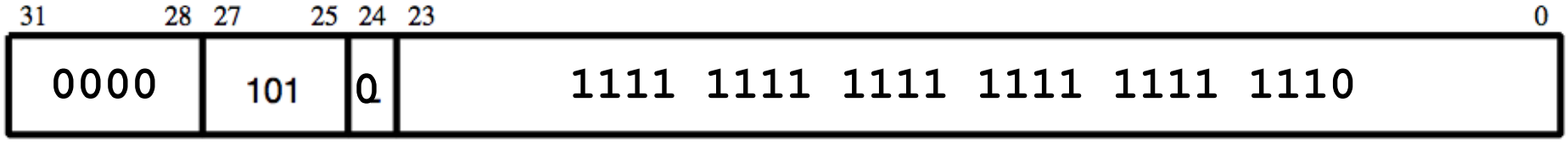
Branch Example

- ARM Instruction: `here BEQ here`

  `cond = 0000` (equal)

  `L = 0` (not a BL instruction)

  `Offset = -2 = 0xFFFFFE` Number of instructions to add to (or subtract from) the PC, starting at two instructions *following* the branch (PC is two instructions ahead due to pipelining)

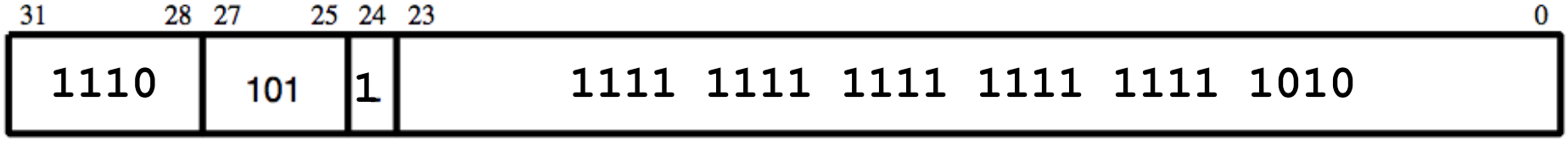


Hex representation: 0x0AFFFFFE

Branch Example

- ARM Instruction:

```
there       ADD r0, r1, r2
            SUBEQ a1, v7, r2, LSL #3
            BIC v6, a1, a2, ROR v2
            EORGTS  r6,r2,#1
here        BL there
```



cond = 1110 (always)

L = 1 (BL instruction)

Offset = -6 110 = 0xFFFFFA

Hex representation: 0xEBFFFFFA

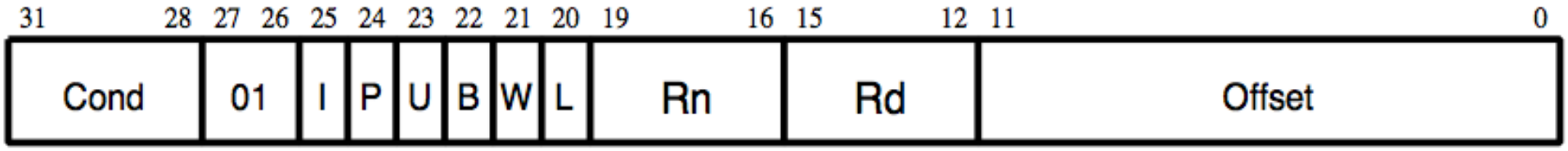
Large Offsets

- Chances are that offset is small enough to fit in the immediate field.
- How do we usually use branches?
  - Answer: `if-else`, `while`, `for`
  - Loops are generally small: typically up to 50 instructions
- Conclusion: may want to branch to anywhere in memory, but a branch often changes PC by a small amount
- What if too big?
  - Load branch target into a register (IP) and BX

Questions on PC-addressing

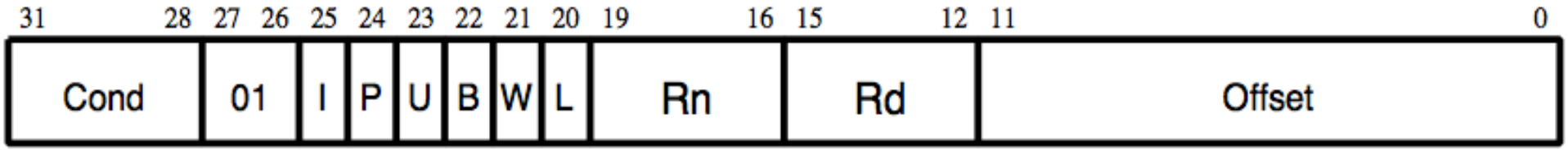
- Does the value in branch field change if we move the code?
- What do we do if destination is > 2^23 instructions away from branch?
- Since its limited to +- 2^23 instructions, doesn't this generate lots of extra ARM instructions?

Data Transfer Instructions



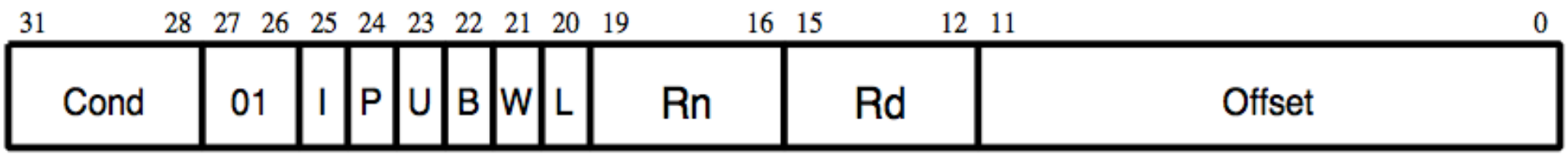
- Cond: condition field. Execute instruction when condition is met.
- I: immediate field. Determines type of offset.
  - 0: if `offset` is an immediate
  - 1: if `offset` is a register
- P: pre/post indexing bit.
  - 0: post - add `offset` after transfer
  - 1: pre - add `offset` before transfer

Data Transfer Instructions



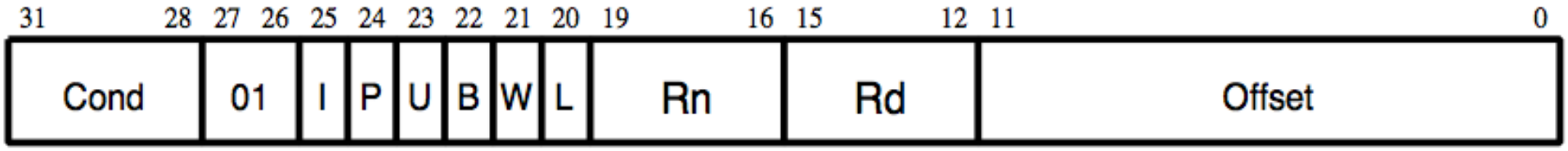
- U: up/down bit.
  - 0: down - subtract `offset` from base
  - 1: up - add `offset` to base
- B: byte/word bit.
  - 0: transfer word quantity
  - 1: transfer byte quantity
- W: write-back bit.
  - 0: no write-back
  - 1: write address into base

Data Transfer Instructions

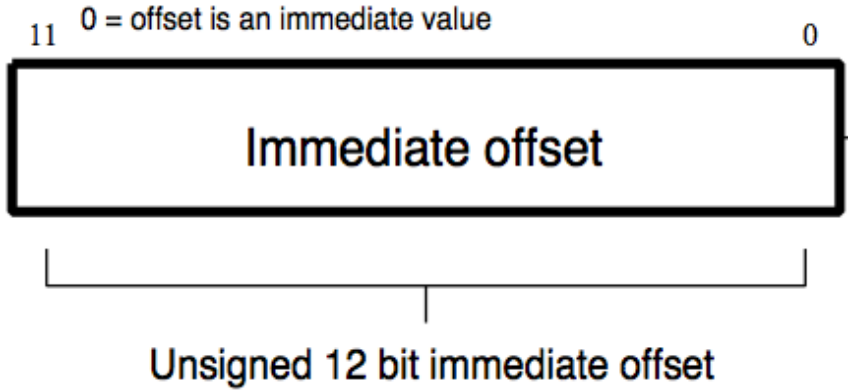


- L: load/store bit.
  - 0: store into memory
  - 1: load from memory
- Rn: base register
- Rd: source/destination register

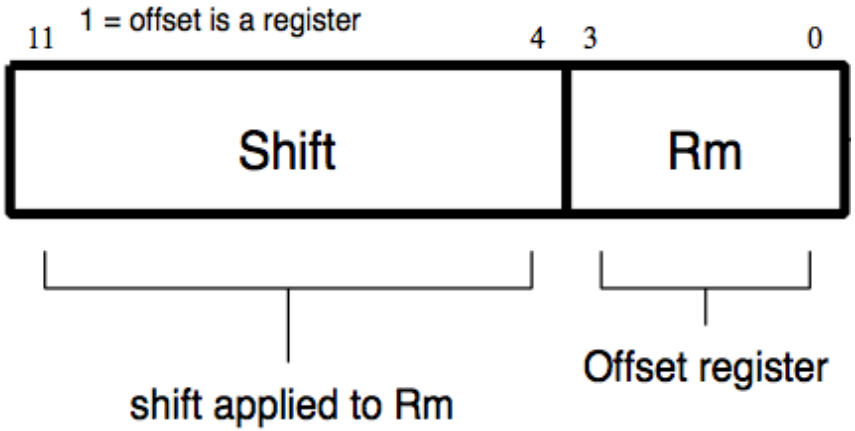
Data Transfer Instructions



- Offset: source/destination register
  - I bit = 0

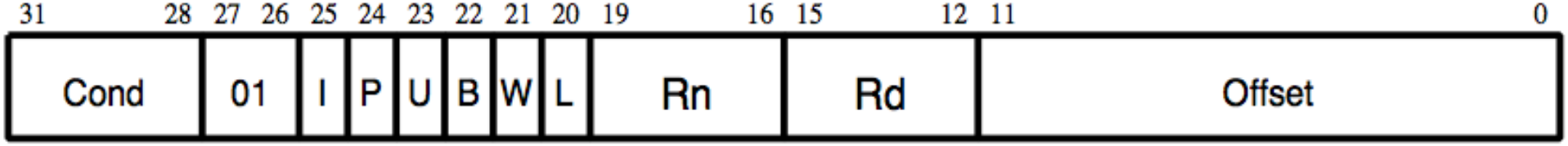


  - I bit = 1



Data Transfer Example

- ARM Instruction: `LDR r8,[sp]`



`cond = 1110` (always – unconditional)

`I = 0` (immediate: there is no offset, or offset = 0)

`P = 1` (pre-increment: there is no offset)

`U = 1` (add offset to base; there is no offset)

`B = 0` (transfer word)

`W = 0` (no writeback)

`L = 1` (load)

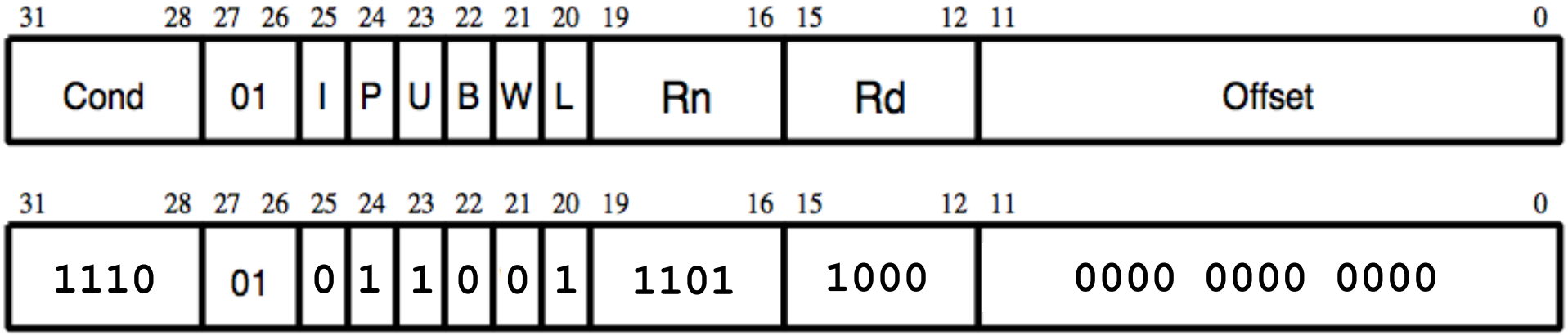
`rd = 8 = 1000` (destination *operand*)

`rn = 13 = 1101` (base *register*)

`Offset = 0` (there is no offset)

Data Transfer Example

- ARM Instruction: `LDR r8,[sp]`

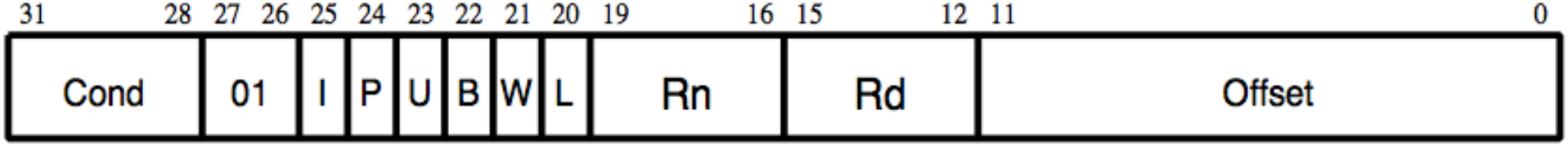


Binary number per field representation:

hex representation: 0xE59D8000

Data Transfer Example

- ARM Instruction: `STRB r10,[sp,#-4]!`



`cond = 1110` (always – unconditional)

`I = 0` (offset is an immediate value)

`P = 1` (pre-index – add immediate before transfer)

`U = 0` (subtract offset from base)

`B = 1` (transfer byte)

`W = 1` (write address into base)

`L = 0` (store)

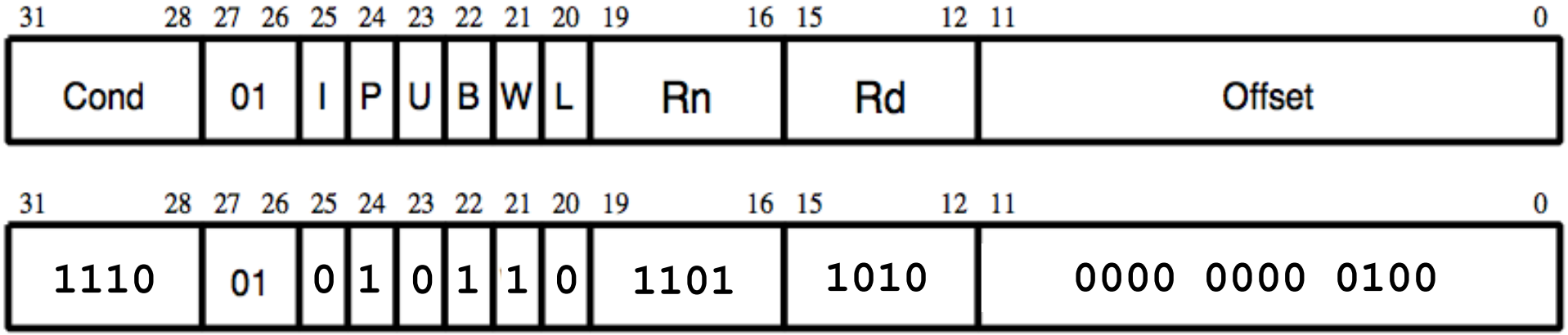
`rd = 10 = 1010` (destination *operand*)

`rn = sp = r13 = 1101` (base *register*)

`Offset = 0x004`

Data Transfer Example

- ARM Instruction: `LDR r8,[sp]`

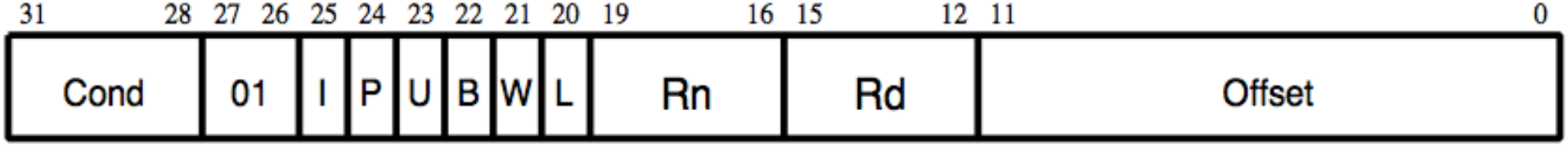


Binary number per field representation:

hex representation: 0xE56DA004

Data Transfer Example

- ARM Instruction: `LDRB r2,[r1],-r3, LSR #4`



`cond = 1110` (always – unconditional)

`I = 1` (offset is a register)

`P = 0` (post-index – add immediate after transfer)

`U = 0` (subtract offset from base)

`B = 1` (transfer byte)

`W = 0` (write address into base)

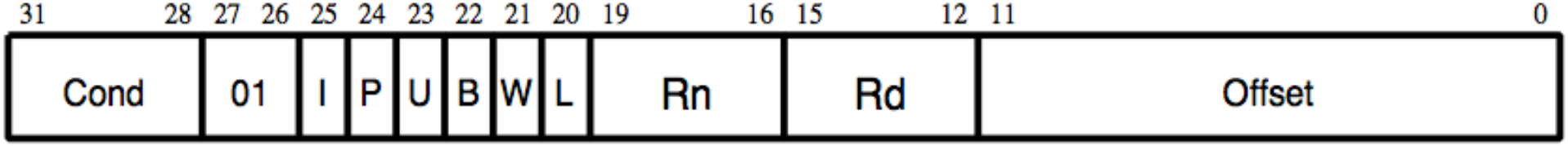
`L = 1` (load)

`rd = 2 = 0010` (destination *operand*)

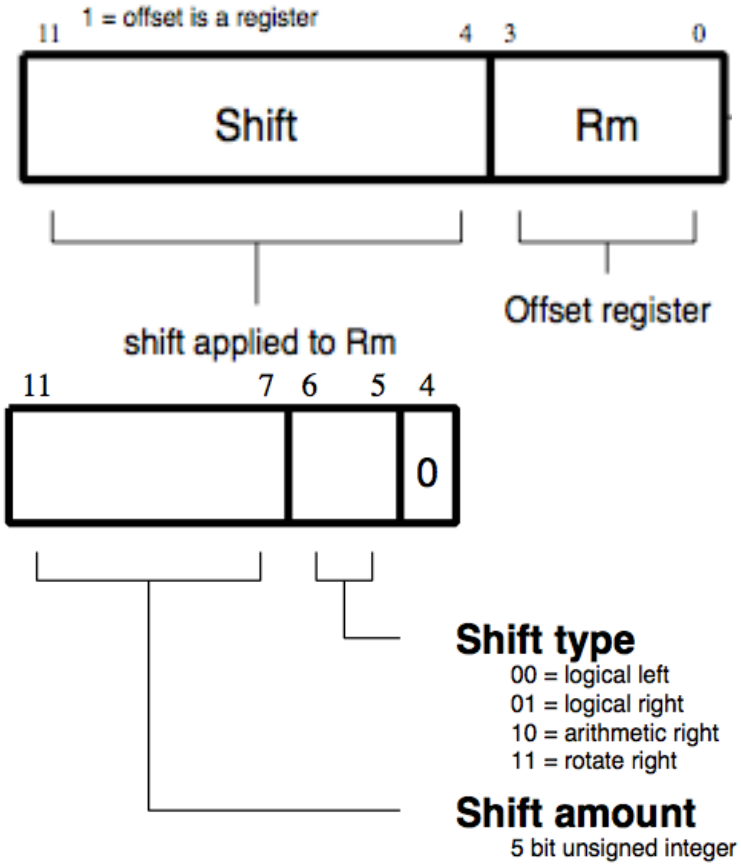
`rn = 1 = 0001` (base *register*)

Data Transfer Example

- ARM Instruction: `LDRB r2,[r1],-r3, LSR #4`

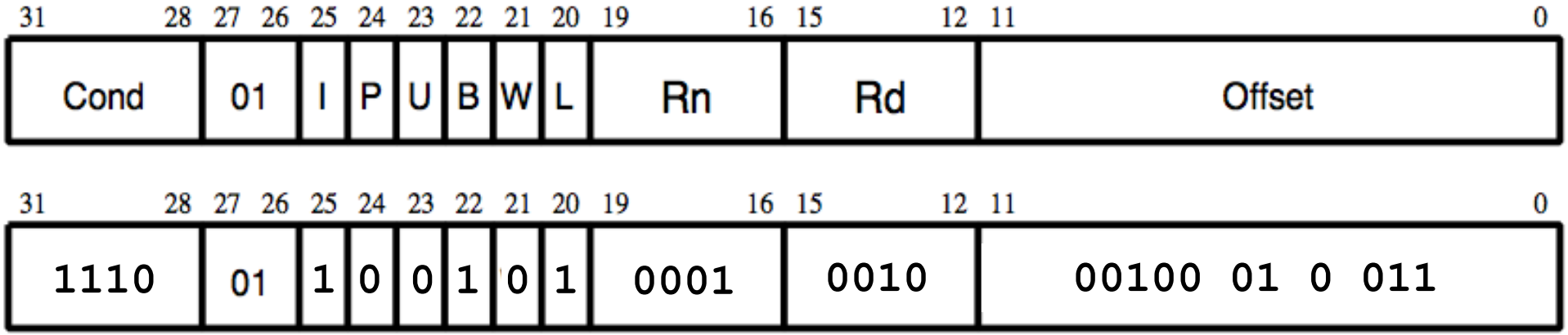



```
Offset:
 shift:
   shift amount = 4 = 00100
   shift type = 01
 rm = 3 = 0011
```



Data Transfer Example

- ARM Instruction: `LDR r8,[sp]`



Binary number per field representation:

hex representation: 0xE651223

Conclusion

- Computer actually stores programs as a series of these 32-bit numbers.
- Each instruction has different number of fields that define it (opcode, registers, immediates, etc.)
- ARM Machine Language Instruction:
  32 bits representing a single instruction