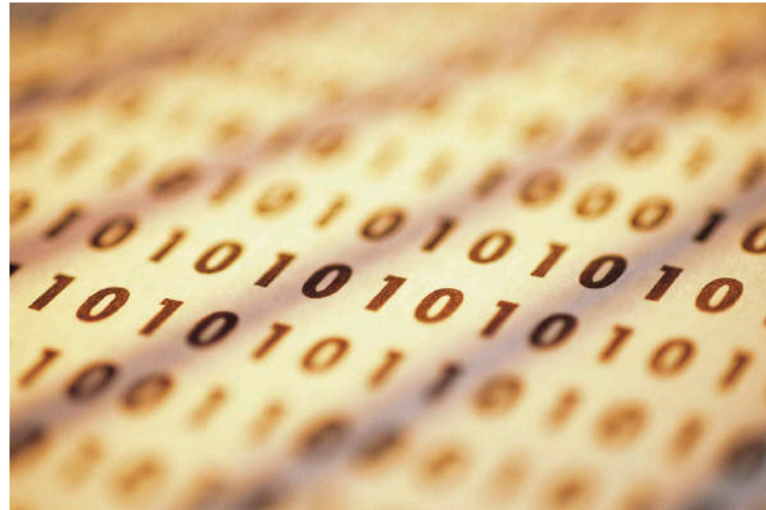
Instruction encoding

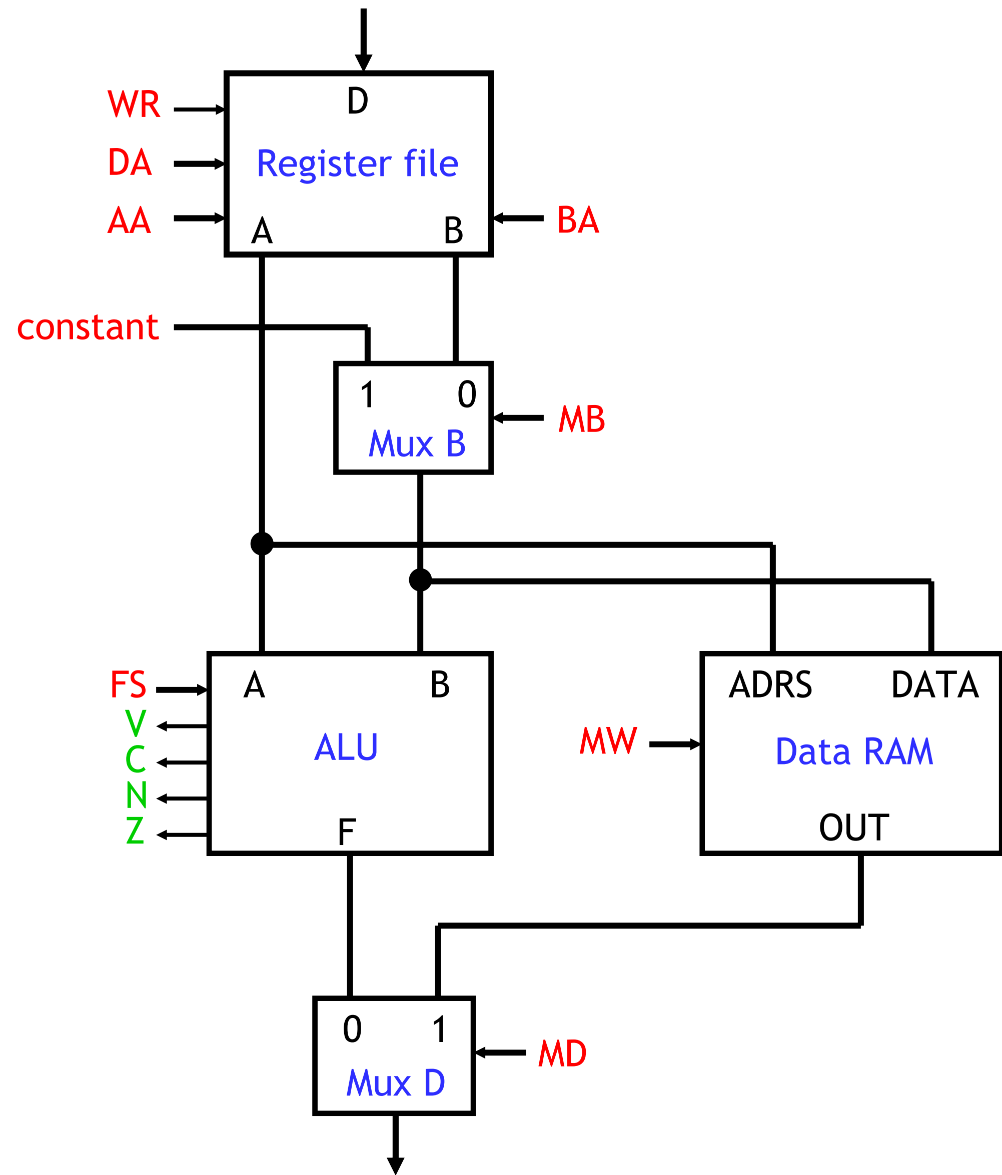


- We’ve already seen some important aspects of processor design.
  - A datapath contains an ALU, registers and memory.
  - Programmers and compilers use instruction sets to issue commands.
- This week we’ll complete our processor with a control unit that converts assembly language instructions into the correct datapath signals.
  - Today we’ll see how assembly instructions can be stored in a binary format, suitable for hardware manipulation.
  - Tomorrow we’ll show all of the implementation details for our sample datapath and assembly language.

©2000-2003 Howard Huang

Datapath review

- The datapath contains all of the circuitry and memory to do a variety of computations.
- The actual computations are determined by the various datapath control inputs in red.
  - AA, BA and MB select the sources for operations.
  - FS picks an ALU function.
  - MW = 1 to write to RAM.
  - MD, WR and DA allow data to be written back to the register file.
- The status bits V, C, N and Z provide further information about the ALU output.



Instruction set review

- We'll use the three-address, register-to-register instruction set from last week, because it matches our datapath closely.
- Data manipulation instructions have one destination register and up to two sources, which can be two registers or a register and a constant.

ADD	R1, R2, R3	R1 ← R2 + R3
SUB	R1, R2, #2	R1 ← R2 - 2

- Data transfer instructions use register-indirect addressing mode to copy data between registers and memory.

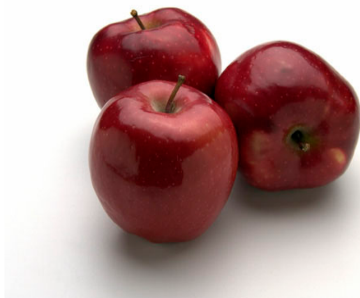
LD	R1, (R2)	R1 ← M[R2]
ST	(R3), R1	M[R3] ← R1

- Jumps and branches on different conditions can also be executed.

JMP	LABEL1	PC ← LABEL1
BZ	R2, LABEL2	if R2 = 0 then PC ← LABEL2

From assembly to machine language

- We already saw how many types of data are represented with 0s and 1s.
  - Unsigned numbers are stored as their binary equivalents.
  - Signed numbers are represented using two's-complement.
  - Text is stored as sequences of ASCII values.
- Today we'll show how assembly language instructions can be represented in a binary machine language format.
- The idea of representing programs in binary and storing them in memory, due to Alan Turing, is what permits computers to perform different tasks easily, just by loading different programs.
- Go ahead, ask about the apples.



Instruction fields

```
SUB   R1, R2, #2
JMP   LABEL1
BZ    R2, LABEL2
```

- Each assembly language instruction contains several components.
  - An operation, such as SUB and BZ.
  - Some instructions include a destination register, like R1.
  - There may be one or two source operands, such as R2 and #2.
  - Branches and jumps include an address, like LABEL1 and LABEL2.
- We can encode an assembly instruction in binary by encoding each of the components, or fields, separately.

Instruction formats

- There are many instructions in our assembly language, but they can be split into just three categories, by the number and types of operands.
- Register format instructions have one destination and two sources, all of which are registers.

```
ADD   R1, R2, R3
```

- Immediate format instructions also have one destination register, but the sources include one register and one constant value.

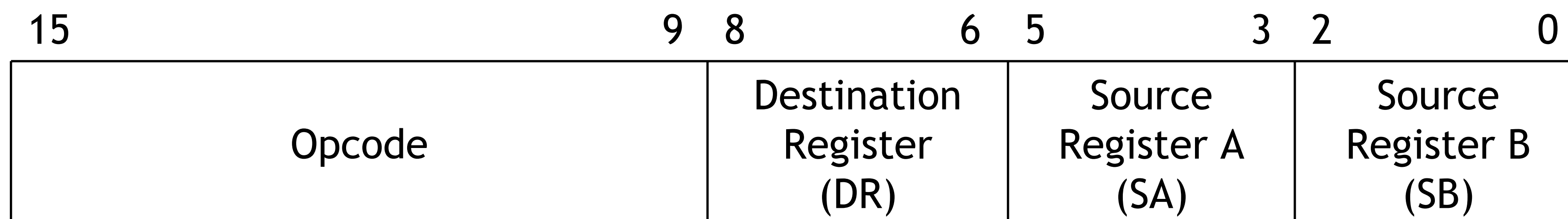
```
SUB   R1, R2, #2
```

- Jump and branch format instructions always need a target address, and there may be a source register as well.

```
JMP   LABEL1
BZ    R2, LABEL2
```

- We will encode each assembly instruction as a 16-bit binary value.

Register format

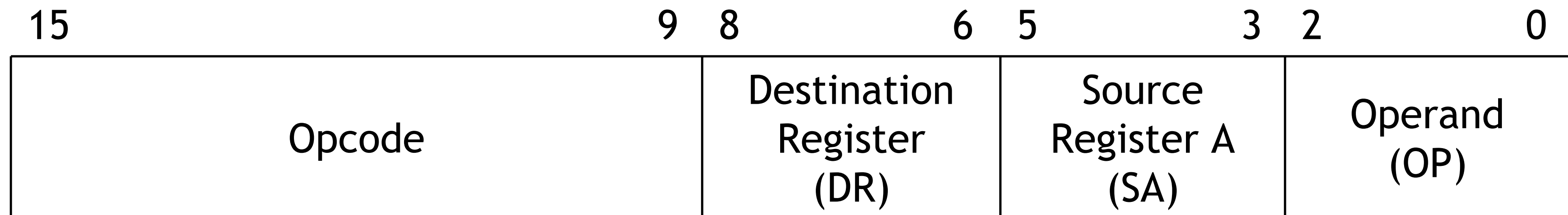


- An example register-format instruction follows.

ADD R1, R2, R3

- Our binary representation for these instructions will include three fields.
  - A 7-bit opcode field that specifies the operation (e.g., ADD).
  - A 3-bit destination register, DR (e.g., R1).
  - Two 3-bit source registers, SA and SB (e.g., R2 and R3).

Immediate format



- An example immediate-format instruction is shown below.

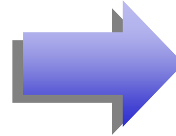
SUB R1, R2, #2

- Immediate-format instructions will consist of four fields.
  - A 7-bit instruction opcode.
  - A 3-bit destination register, DR.
  - A 3-bit source register, SA.
  - A 3-bit constant operand, OP (e.g., #2).

PC-relative jumps and branches

- We will use PC-relative addressing for jumps and branches, where the operand specifies the number of addresses to jump or branch from the current instruction.
- We can assume each instruction occupies one 16-bit word of memory.

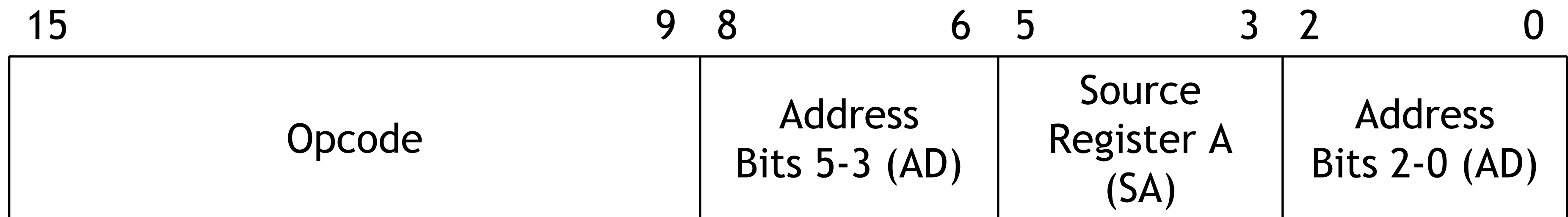
```
     LD   R1, #10
     LD   R2, #3
     JMP  L
K    LD   R1, #20
     LD   R2, #4
L    ADD  R3, R3, R2
     ST   (R1), R3
```



```
1000  LD   R1, #10
1001  LD   R2, #3
1002  JMP  +3
1003  LD   R1, #20
1004  LD   R2, #4
1005  ADD  R3, R3, R2
1006  ST   (R1), R3
```

- The operand is a *signed* value, so you can go either forward or backward.

Jump and branch format

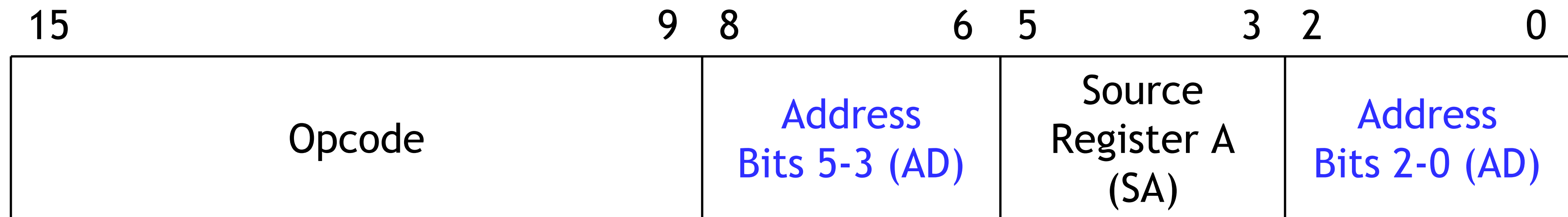


- Here are two example jump and branch instructions.

```
BZ    R2, +19
JMP   -5
```

- Jump and branch format instructions include two or three fields.
  - A 7-bit instruction opcode (e.g., BZ or JMP).
  - A 3-bit source register SA for branch conditions (e.g., R2).
  - A 6-bit address field, AD, for jump or branch offsets (e.g., +19 or -5).
- Our branch instructions include only one source register, but other types of branches can also be done, as discussed in the current homework.

The address field AD



15 — 9	8 — 6	5 — 3	2 — 0
Opcode	Address Bits 5-3 (AD)	Source Register A (SA)	Address Bits 2-0 (AD)

- AD is treated as a six-bit two's complement number, so you can branch up to 31 addresses forward (2^5-1) or 32 addresses backward (-2^5).
- The address field is split into two parts just so the SA field can occupy the same position (bits 5-3) in all three instruction formats.

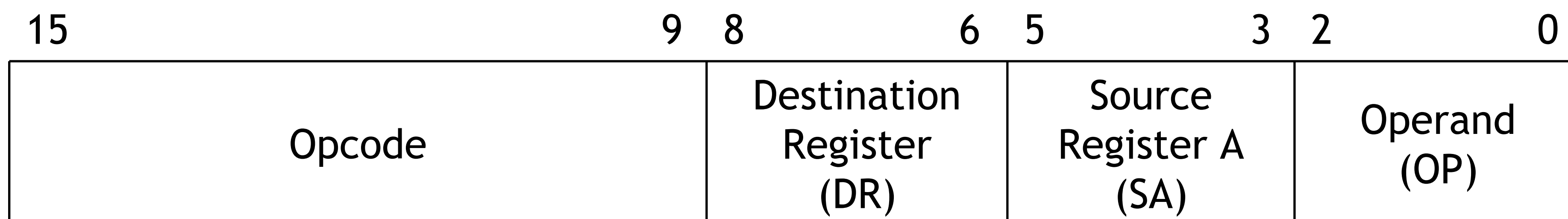
Instruction format uniformity

- Notice how we tried to keep the different instruction formats as similar as possible.
  - The Opcode field always appears in the same position (bits 15-9).
  - DR is in the same place for register and immediate instructions.
  - The SA field also appears in the same position, even though this forced us to split AD into two parts for jumps and branches.
- On Wednesday we'll see how this leads to a simpler control unit.

15 — 9	8 — 6	5 — 3	2 — 0
Opcode	Destination Register (DR)	Source Register A (SA)	Source Register B (SB)
Opcode	Destination Register (DR)	Source Register A (SA)	Operand (OP)
Opcode	Address Bits 5-3 (AD)	Source Register A (SA)	Address Bits 2-0 (AD)

Instruction formats and the datapath

- The instruction formats are closely related to the datapath design.
  - Because our register addresses DR, SA and SB are each three bits long, this instruction set can only support eight registers.
  - The constant operand OP is also three bits long. Its value will have to be sign-extended if the ALU supports wider inputs and outputs.
- Conversely, if the datapath supported more registers or larger constants, we would have to increase the length of our instructions.



Filling in the operand fields

- It’s easy to fill in the operand fields in our binary instructions, given the original assembly instruction.
- Registers are represented by their numbers. For example, we can denote the operand R1 by 001.

Instruction	Opcode	DR	SA	SB
ADD R1, R2, R3	-------	001	010	011

- Constants and address offsets are just stored in two’s-complement form. For example, the constant #2 is 010, while the address +19 is encoded as 010011.

Instruction	Opcode	DR	SA	OP
SUB R1, R2, #2	-------	001	010	010

Instruction	Opcode	AD	SA	AD
BZ R2, +19	-------	010	010	011

Filling in the opcode fields

- The hard part is mapping operations like ADD and BZ to binary opcodes!
- One solution is to just select a random code for each possible operation.
  - This is (more or less) what we'll do for the last assignment, MP4.
  - However, the selection of opcodes can have a significant impact on the complexity of the processor, much like the assignment of states in sequential circuit design.
- Today we'll see a more methodical approach to assigning opcodes.
  - Similar operations will have similar opcodes.
  - This will make our hardware much simpler in Wednesday's lecture.
- Unfortunately you'll have to understand Wednesday's material to fully appreciate some of the opcode selection choices we make today.

Organizing our operations

- To start, we'll divide our operations into four categories. The instructions in each category will need similar datapath control signals, as we'll show on Wednesday.

Instruction category	Opcode bits 6-5
Register-format ALU operations	00
Data transfer operations	01
Immediate ALU operations	10
Branches and jumps	11

- We'll use the two most significant bits in every opcode to indicate which category the instruction belongs to. The rest of the opcode will indicate a particular operation *within* that category.

Instruction	Opcode	
ADD R1, R2, R3	00	-----
LD R1, (R2)	01	-----
SUB R1, R2, #2	10	-----
BZ R2, +19	11	-----

Immediate and register-based ALU operations

- For both register and immediate ALU operations, a simple approach is to fill in the rest of the opcode bits with the matching ALU function selection code.
- For example, a register-based XOR would have the opcode 0001100.
  - The first two bits 00 denote a register-based ALU instruction.
  - 01100 denotes the ALU's add function.
- Here are two other examples.

Instruction	Opcode	
ADD R1, R2, R3	00	00010
SUB R1, R2, #2	10	00101

FS	Operation
00000	F = A
00001	F = A + 1
00010	F = A + B
00011	F = A + B + 1
00100	F = A + B'
00101	F = A + B' + 1
00110	F = A − 1
00111	F = A
01000	F = A ∧ B (AND)
01010	F = A ∨ B (OR)
01100	F = A ⊕ B
01110	F = A'
10000	F = B
10100	F = sr B (shift right)
11000	F = sl B (shift left)

Register-indirect data transfer operations

- Our second general category of instructions are data transfer operations, but we have just two of them: loads and stores.
- We can use the third bit of the opcode to distinguish between them.
  - The first three bits of the opcode for ST will be 010.
  - The first three bits of the opcode for LD will be 011.

Instruction		Opcode		
ST	(R1), R2	01	0	XXXX
LD	R1, (R2)	01	1	XXXX

- The rest of the opcode bits aren't needed here.
  - Keeping the same length for all of our opcodes, even when they're not all used, makes Wednesday's control unit easier to design.
  - This also leaves room for expansion in the instruction set; if we later decide to add other types of loads and stores, we'll have four extra bits to work with.

Immediate-mode data transfer operations

- What about immediate loads and stores?

```
LD  R1, #19
ST  (R2), #–5
```

- These will actually get implemented as immediate *arithmetic* operations which use the ALU's transfer capability.
- We'll see more about this on Wednesday, but for now you should know that these immediate transfers are in a different category from register-indirect transfers.

FS	Operation
00000	F = A
00001	F = A + 1
00010	F = A + B
00011	F = A + B + 1
00100	F = A + B'
00101	F = A + B' + 1
00110	F = A – 1
00111	F = A
01000	F = A $\wedge$ B (AND)
01010	F = A $\vee$ B (OR)
01100	F = A $\oplus$ B
01110	F = A'
10000	F = B
10100	F = sr B (shift right)
11000	F = sl B (shift left)

Jump instructions

- Our last category included jump and branch instructions.
- Just as for data transfer instructions, we'll use one additional opcode bit to distinguish between jumps and branches.
  - Branch opcodes will all start with the three bits 110.
  - Jump opcodes will start with the bits 111.

Instruction	Opcode		
BZ R2, +19	11	0	----
JMP –5	11	1	XXXX

- Our instruction set has only one PC-relative jump instruction so the rest of the opcode bits are unused, much like for loads and stores.

Branch instructions

- We can include all the branch conditions discussed last week, which are repeated on the right.
- We'll just pick a three-bit code to denote each possible branch condition, and insert it into the branch opcode.
- This means that one of the four remaining opcode bits will be left unused; the book leaves the *middle* bit unused.
- For example, BZ has the opcode 110X011.
  - The bits 110 indicate a branch.
  - 011 specifies branch if zero.

Branch	Condition	Code
BC	Carry set	000
BN	Negative	001
BV	Overflow	010
BZ	Zero	011
BNC	Carry clear	100
BNN	Positive	101
BNV	No overflow	110
BNZ	Non-zero	111

Instruction	Opcode			
BZ R2, +19	11	0	X	011
BNN R3, −5	11	0	X	101

Sample instruction encodings

- We're finally done! Now we know the binary fields that each machine language instruction requires, and we know how to encode all of them.
- Here are the complete machine language translations of some example assembly language instructions.
  - The meaning of bits 8-0 depends on the instruction format.
  - The colors are not supposed to blind you, but to help you distinguish between destination, source, constant and address fields.

Instruction	Format	Bits 15-9 (Opcode)	Bits 8-6	Bits 5-3	Bits 2-0
ADD R1, R2, R3	Register	0000010	001	010	011
LD R1, (R0)	Immediate	011xxxx	001	000	xxx
SUB R5, R5, #2	Immediate	1000100	101	101	010
BZ R1, +19	Jump/branch	110x011	010	001	011
JMP –5	Jump/branch	111xxxx	111	xxx	011

Summary

- Today we defined a binary machine language for the instruction set from last Wednesday.
  - Each assembly language instruction contains several component fields.
  - Different instructions are encoded using different binary formats, but keeping those formats uniform will help simplify our hardware.
  - We also tried to assign similar opcodes to "similar" instructions.
  - Register, constant and address offset operands are represented as just unsigned or signed binary numbers.
- This instruction encoding is closely related to our example datapath. For example, our opcodes include ALU function selection codes, and the number of usable registers is limited by the length of each instruction.
- This is just *one* example of how to define a machine language. You will be using a different instruction encoding for MP4, for instance.
- Tomorrow we'll show how to build a control unit that corresponds to our datapath and instruction set. This will complete our processor!