Datapath Elements

The datapath elements are the functional blocks within a microprocessor that actually interact to perform *computational operations*. These tasks include reading/writing to memory, arithmetic, logic operations, and numerical shift operations. These elements form the building blocks of a complete datapath, discussed in the next chapter. Each of these datapath elements is built from the basic building blocks you have already seen in this course. All microprocessors contain these elements in some form or another, satisfying particular price/performance constraints. For example, an arithmetic unit that can perform floating point calculations is much more complicated (and expensive) than an arithmetic unit that only performs integer calculations. The elements we will talk about today are the register file, adder/subtractor, logical unit, and shift unit.

Register File

The **register file** is an addressable bank of registers. This bank of registers forms a block of memory that is local (i.e., on-board) the microprocessor. Each register is a 32-bit binary word, and our idealized register file consists of 32 registers. By *addressable*, we mean that an address, or 5-bit binary input, is used to specify which of the 32 registers that we wish to read from or write to. A sample schematic of simple register file for reading and writing is shown in Figure 1.

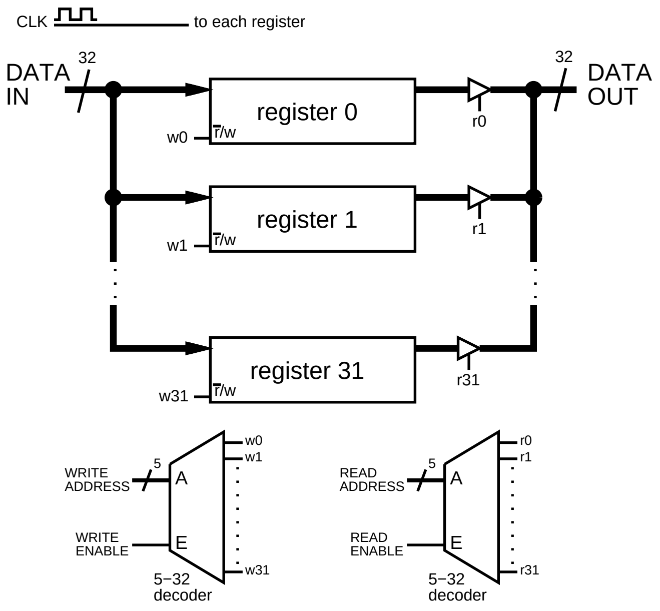

Figure 1: Generalized schematic of the inner workings of a register file

Each register in the register file is *clocked* like the registers and flip-flops you have already seen. In a typical CPU, different data is being read/written to the register file and the control lines are changing on every clock cycle, enabling a steady sequence of read and write operations.

In practice, a register file may have multiple read and write ports. In the register file we will be using in our datapath for this class, there are 2 read ports (X and Y) and one write port (Z). A functional block

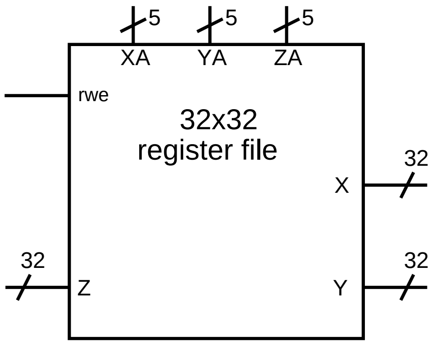

Figure 2: Control and I/O signals for a 32x32 register file with two read ports (X and Y) and 1 write port (Z).

for this register is shown in Figure 2. Each of the three ports has its own 5 bit address selection line. On every clock cycle, the registers specified by the XA and YA addresses are output (i.e., read) to the X and Y bus. The control line *rwe* is the *register write enable*. When *rwe* is set to 1, writing into the register file is enabled, and whatever data is present on the Z bus is written to the register specified by the ZA address bits. When *rwe* is set to 0, the data on the Z bus is ignored.

When we build our datapath in the next chapter to implement a simple microprocessor, the register file serves as a local memory for performing numerical, logical, and memory read/write operations. The registers are somewhat analogous to local (or temporary) variables when programming in a higher-level programming language.

Adder/Subtractor

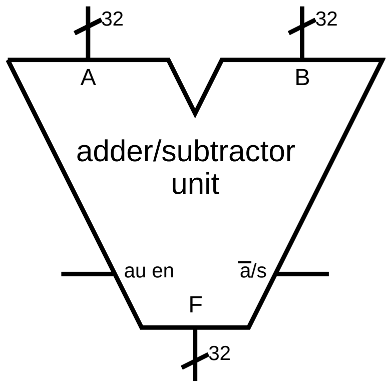

Figure 3: Adder/subtractor unit.

The adder/subtractor performs 32-bit addition subtraction. Its internal implementation is generally similar to the adder/subtractor we discussed earlier in the course. The inputs A and B are two 32-bit integers and the output F is A+B or A-B. The *au en* input enables the adder/subtractor when it is set to 1. The $\bar{a}/s$ input controls whether the operation to be performed is an addition or subtraction. The adder/subtractor has other outputs signals to indicate conditions such as numerical overflow; these signals will be discussed later in the course.

Logical Unit

The logical unit is a device for performing logical operations on two boolean variables. Recall that a truth-table for a 2 variable boolean function only has 4 possible rows, thus any possible truth-table for a two-variable function can be implemented by specifying the 4 bits that describe this truth-table, as shown in Fig. 4A. In practice, we can implement this by using a 4-1 multiplexor as a lookup table (Fig. 4B).

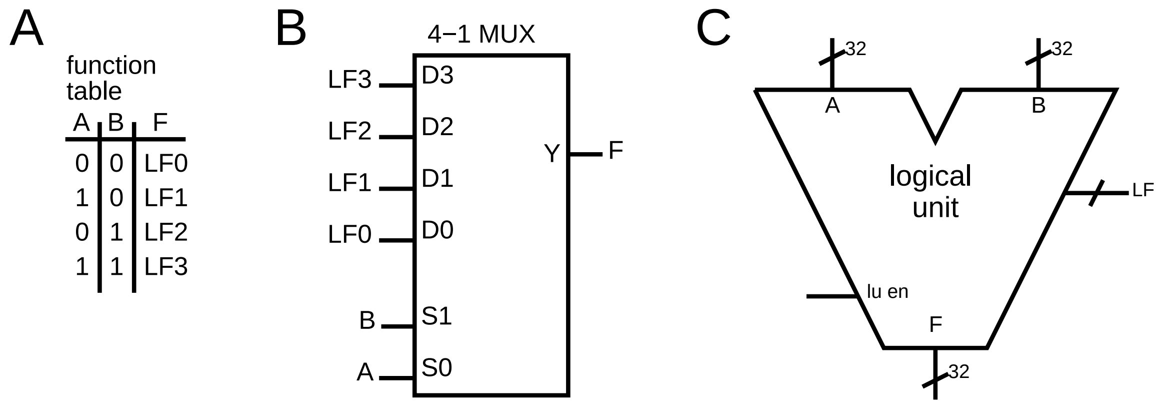

Figure 4: Logical unit implementation. A: Truth-table. B: Implemented as a lookup table using a 4-1 multiplexor. C: Symbol used for our 32-bit datapath.

Thus the boolean function to be implemented can be described by the 4 bit number LF, indicated by the 4 bits $LF_3LF_2LF_1LF_0$. For example, consider the operations listed in the table below and their corresponding values for the lookup table:

function	LF
AB	1000
$A+B$	1110
$A \bigoplus B$	0110
$\bar{A}+B$	1101

For our 32-bit datapath, we use the logical unit symbol shown in Fig. 4C. This logic unit has 2 32-bit inputs (A and B), a 32-bit output (F), the 4 LF bits that describe the logical function to be implemented, and *lu en*, which is an enable bit that enables the operation of the logical unit.

The examples above and MUX in Fig. 4B show the logical operations using two 1-bit values, A and B, as the input. How does the logical unit operate on 2 32-bit numbers, as shown in Fig. 4C? The logical operation specified by the LF bits is performed on all 32 bits in parallel and independently. Thus within the logical unit are 32 MUXes similar to Fig. 4B, and each MUX operates on a single *bit slice* of A and B. For example, if an AND operation is being performed, $F_0 = A_0B_0$, $F_1 = A_1B_1$, and so on.

It is common to work in terms of hexadecimal notation when dealing with 32-bit wide numbers. Below are examples of some logical operations performed using the logical unit.

A	B	LF	operation	F
0x4488CCFF	0x44444444	1110	OR	0x44CCCCFF
0x4488CCFF	0x0FF00FF0	1000	AND	0x04800CF0
0x12345678	0x0000FFFF	1000	AND	0x00005678
0x0FF0AAAA	0xFFFFAA55	0110	XOR	0xF00F00FF

Work out the above examples to understand that the results in the table are correct. Write out A and B in binary, one on top of each other, and then perform the specified boolean operation on each column of bits. Using the first row as an example:

A:	0100	0100	1000	1000	1100	1100	1111	1111
B:	0100	0100	0100	0100	0100	0100	0100	0100
A+B:	0100	0100	1100	1100	1100	1100	1111	1111

A common use for the bit-wise AND and OR functions is *masking*, which means to selectively set or zero out particular bits in a number. The AND function can be used to zero out particular bits, and the OR function can be used to set particular bits. The example above illstrates how the bits in B (the mask) are used to set particular bits in A.

Shift Unit

Another common operation performed by microprocessors are shift operations. The shift operation takes a binary number and shifts it left or right a specified number of bits. There are 3 different kinds of shift operations: *logical*, *arithmetic*, and *circular*. The difference among these operations are how the "end-conditions" are handled – where the bits go that are shifted off one end and what bits are introduced into the "empty bits" inserted at the other end of the shift. In the examples below, we consider the effects of each type of shift on the number A = 11100100.

- *Logical shift.* In the logical shift, the empty bits (the most-significant bits with a right shift, and the least-significant bits with a left shift) are filled with zeros; this is sometimes referred to as *padding*. A logical right-shift of 2 bits on A yields 00111001, while a logical left-shift of 2 bits on A yields 10010000.

- *Arithmetic shift.* In the arithmetic shift, the least-significant bits are padded with zero when left shifting. However, when right-shifting the most-significant bit is *copied* into the vacant bit positions. An arithmetic right-shift of 2 bits on A yields 11111001, while an arithmetic left-shift of 2 bits on A yields 10010000.

- *Circular shift.* In the circular shift, bits that are shifted off of one end of the number are inserted at the other end – the bits simply have their positions rotated. A circular right-shift of 2 bits on A yields 00111001 and an circular left-shift of 2 bits on A yields 10010011.

What are these shifts used for? We will see later in the semester that there are many uses for splitting a binary number into *bit fields* – groupings of bits within a number that have particular meaning, and all of these bit fields are packed together. These methods are often used in conjunction with the masking and bit-setting uses of the logical unit described earlier.

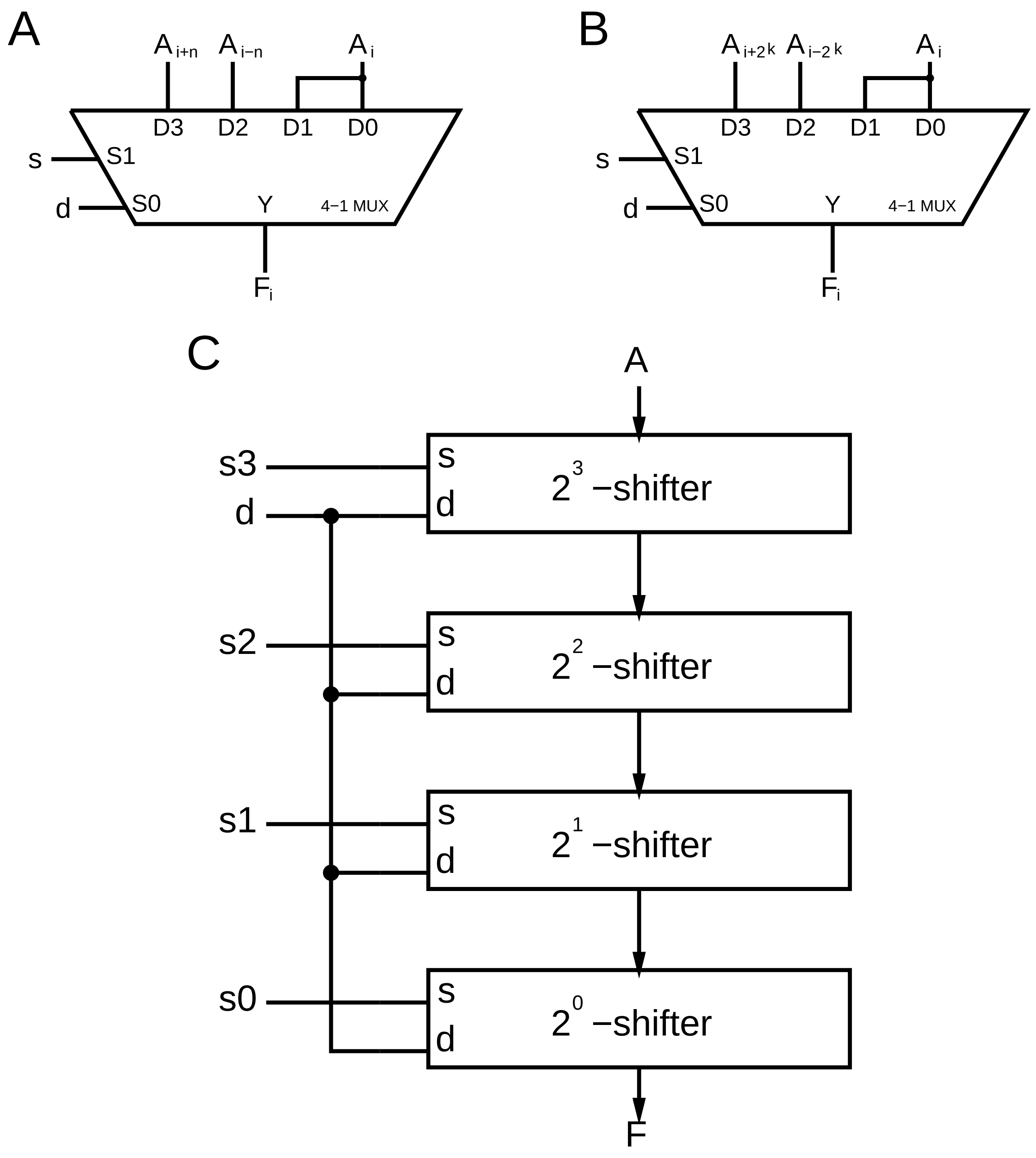

Figure 5: Shifter implementation. A: A single bit-slice of an n-shifter. B: A single bit-slice of a 2^k-shifter. C: Implementation of a barrel shifter by cascading 2^k-shifters.

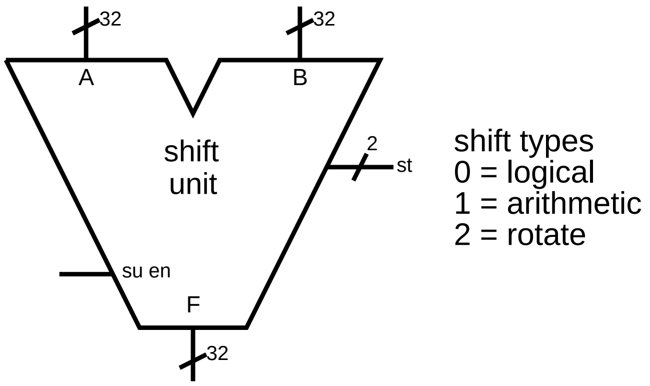

Figure 6: Shifter datapath element.

The arithmetic and logical shifts are also used to perform efficient integer multiplication and division by 2. A logical left-shift one bit is the same as multiplying an unsigned binary value by 2, and a logical right-shift is the same as dividing an unsigned binary value by 2. If the number is signed (in 2's-complement notation), the arithmetic shift performs the same operation (multiplication or division by 2) while preserving the sign of the number. An arithmetic left-shift can change the sign of a number – what does this mean? [1] Another use of the arithmetic right-shift is to convert a signed integer to a higher degree of precision, such as converting a 16-bit number to a 32-bit number.

How is a shift implemented in terms of logical devices? Just as in the logical unit described earlier, the output for each bit of a shift operation can be implemented using a 4-1 MUX, as shown in Fig. 5A. The two select inputs are s (1=shift, 0=no shift) and d (1=right, 0=left). The output F_i is either A_i (no shift), A_{i+n} (right shift n bits), nr A_{i-n} (left shift n bits). These inputs are slightly modified for the most and least significant bytes.

In practice, a shifter unit will shift a given input a specified number of bits left or right, while the MUX in Fig. 5A showing the shifter for a single bit-slice is "hard-wired" to shift left or right n bits. How do we design a shift unit that will shift left or right an arbitrary number of bits? There are many possible solutions you may think of, most of which involve rather complex combinatorial logic. The most compact solution is through the use of a cascade of 2^k-shifters, as shown in Fig. 5B. The 2^k-shifter is hardwired to shift an input left or right 2^k bits. For example, a 2^3-shifter will shift an input left or right 8 bits. By cascading several 2^k-shifters in a row, as in Fig. 5C, we can shift an arbitrary number of bits left or right. This cascade of shift units is called a *barrel shifter*.

For example, we wish to shift a number right 11 bits. Looking at Fig. 5C, d=1 (right shift), and $s_3s_2s_1s_0$ = 1011 (binary for 11). The upper unit shifts the number right 8 bits ($s_3 = 1$), the next unit does not shift at all ($s_2 = 0$), the next unit shifts the number right 2 bits ($s_1 = 1$), and the bottom unit shifts the number right 1 bit ($s_0 = 1$). The final output F has thus been shifted right 11 bits.

The shift unit to be used in our datapath is shown in Fig. 6. A is a 32-bit number containing the data to be shifted, and B is a 32-bit *signed* number indicating the number of bits to be shifted right (positive) or left (negative). The shifter also has inputs specifying the shift-type (*st*) and enabling the device (*su en*).

[1] Answer: An overflow condition occurred

Summary

In this chapter we have described the basic elements that comprise our single-cycle datapath that will be put together in the next chapter. By putting these elements together, will will have the basic building blocks for manipulating data (adder/subtractor, logical unit, shift unit) and reading/writing to local memory (register file).