CMPT 250 : Week 7 (Oct 15 to OCT 22)

1. SEQUENTIAL CIRCUIT MULTIPLICATION (cont'd)

1.1. SAVING REGISTERS

An examination of the multiplication process in the context of the logic diagram reveals that both the X and P registers are shifted right at the same time. Furthermore, initially P(LO) is not required and is set to 0.

By storing X in P(LO) at the beginning of multipication, and then examining P(0) rather than X(0), we can save the cost an X register. This revised algorithm is defined in the following ASM diagram:

1.2. SIGNED MULTIPLICATION

There are two possible approaches for signed multiplication:

1. Convert the operands to positive values, remembering the original signs. Then perform unsigned multiplication. If the signs are different, negate the result.
2. Develop an algorithm that includes the sign determination as part of the process.

Booth's algorithm is an example of an algorithm that determines the sign a a by-product of the multiplication.

Booth's algorithm is based on the fact that any number can be expressed as the sum and difference of different powers of 2. Obviously, every number can be expressed a sum, as this is reflected by its binary representation where a "1" in the *i*th postion to the right of the least significant bit means that 2^i should be one of the summands.

What is not so clear is that there are other representations if subtraction of terms is permitted and Booth's discovery was a way of determining these terms from the pattern of bits in the binary representation.

Booth's Algorithm

Begin by adding a bit (called b_{-1}) to the right of the least significant bit of the binary number. We now proceed from right to left, comparing pairs of adjacent bits b_i, b_{i-1}, i = 0,1,..,n-1, to detect the following patterns:

- **00** or **11**: Increment *i* by 1.

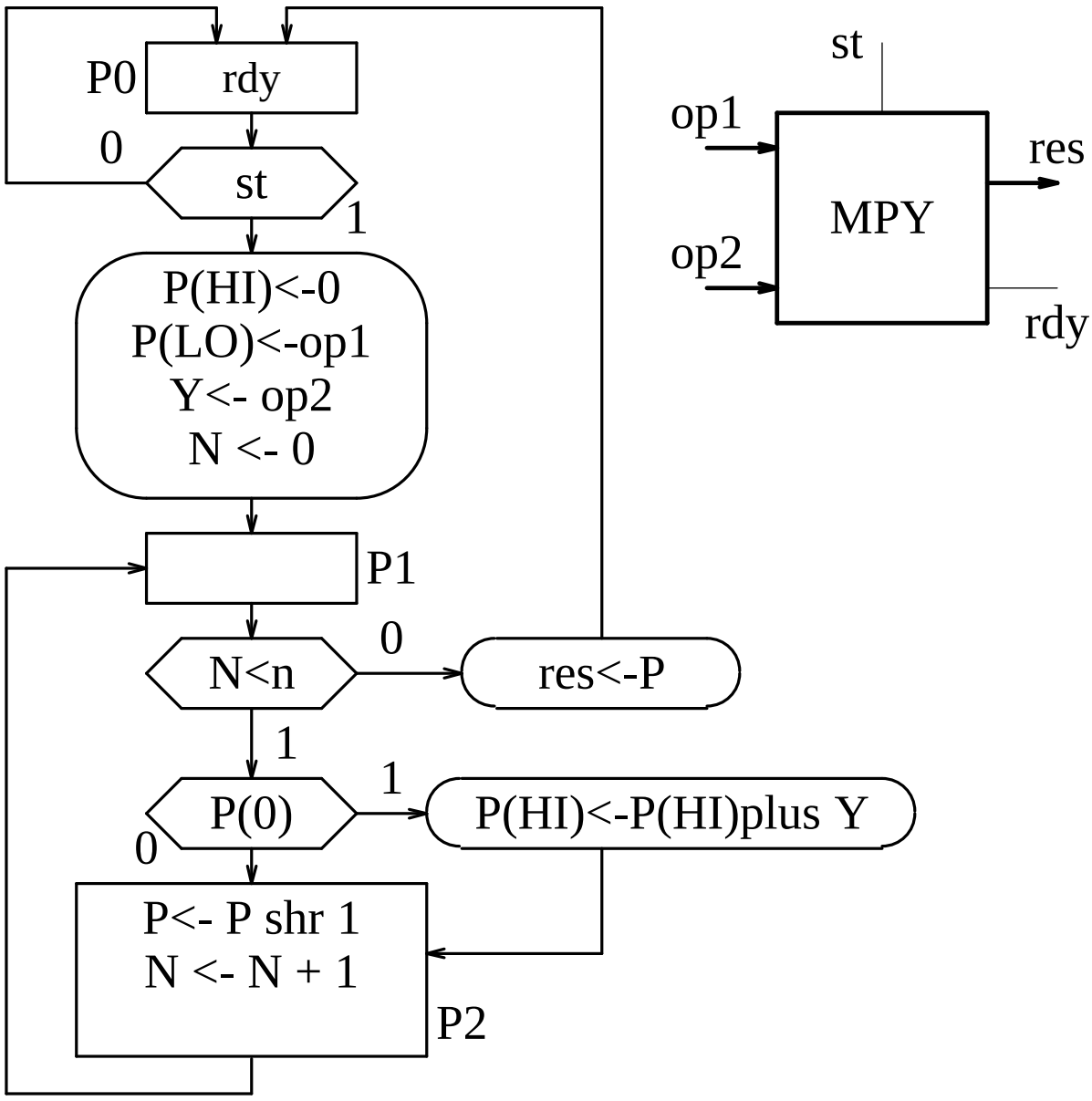

Figure 1-1: The multiplier algorithm (v2)

- **01**: Add 2^i to the sum. Increment *i* by 1.
- **10**: Subtract 2^i from the sum. Increment *i* by 1.

For example the binary number 011010 Can be expressed as follows. (b_{-1} is in parentheses. "^" denotes bits examined):

```
011010(0) => do not add or subtract 2^0.
     ^ ^

011010(0) => subtract 2^1 from the sum.
    ^^

011010(0) => add 2^2 to the sum.
   ^^

011010(0) => subtract 2^3 from the sum.
  ^^

011010(0) => do not add or subtract 2^4.
 ^^

011010(0) => add 2^5 to the sum; and stop.
^^

011010 = 26 = 2^5 - 2^3 + 2^2 - 2^1
```

This test can be used in the multiplication algorithm. The bits of the multiplier are tested according to the rules above, The multiplicand is either added to the product or subtracted from the product if adjacent bits differ. After every case the product is shifted right. Note that the an arithmetic shift rather than a logical shift is performed to preserve the most significant bit of the product which maintains the sign.

The previous mulitplication ASM can be modified to incorporate Booth's test. The product register P is extended by 1 bit so that P(0) holds b_{-1}. Thus `P(HI) = P(n..n/2+1)`, and `P(LO) = P(n/2..1)`. The value placed on the output bus `res` is P(HI,LO) and does not include P(0). This is illustrated on the next page. The significance of Booth's algorithm is that an addition or subtraction only occurs when adjacent bits are different. If this occurs infrequently, then few of these operations are required, and performance is improved.

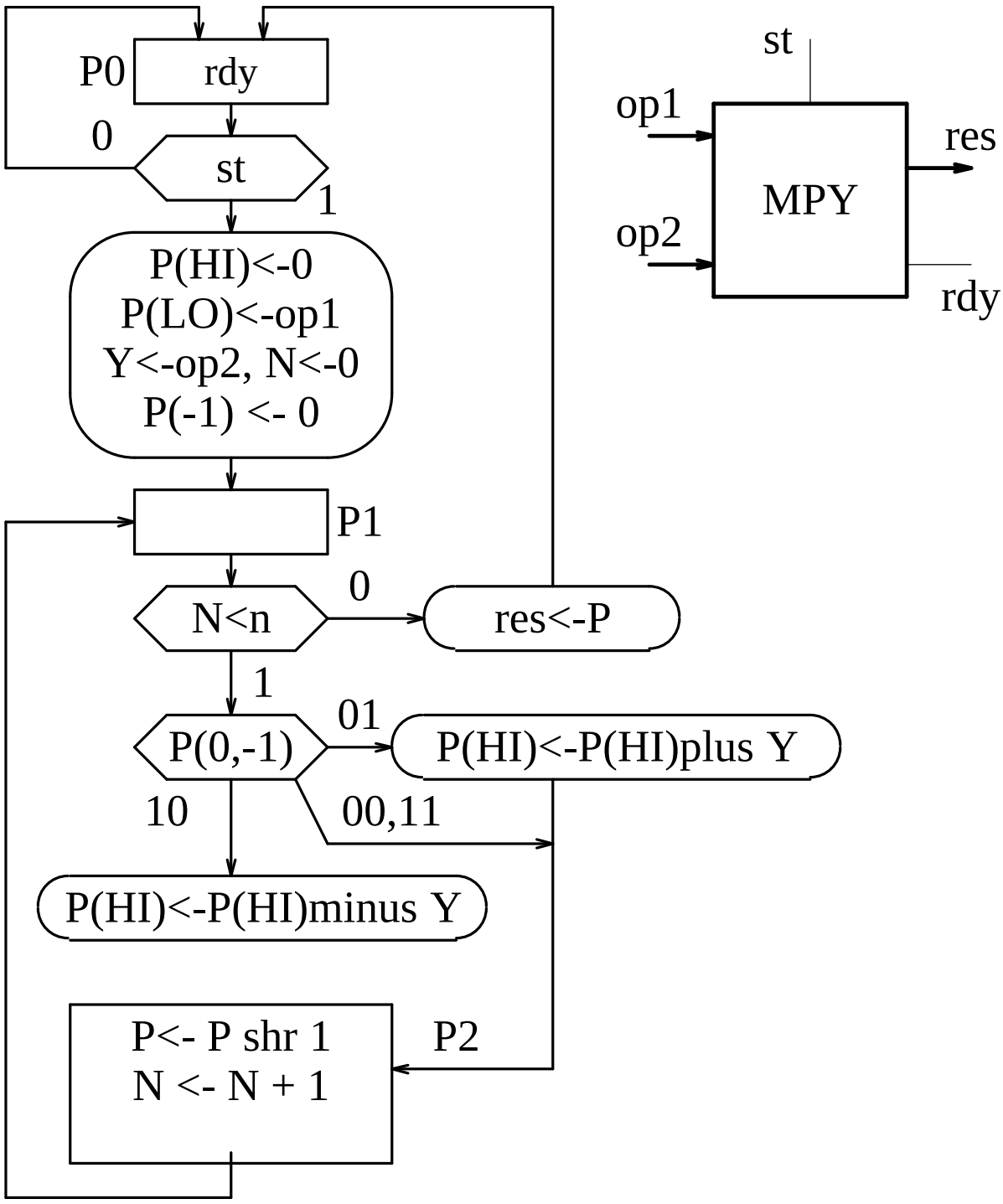

Figure 1-2: Booth's multiplier algorithm

2. INSTRUCTION SET ARCHITECTURES & CPU DESIGN

Designing the central processing unit (CPU) of a simple computer provides a good opportunity to illustrate the application of the tools and techniques covered in the course, as well as to identify the fundamental considerations that affect the eventual architecture.

2.1. PROBLEM SPECIFICATION AND ANALYSIS

A general purpose processor will be one which is capable of performing the tasks specified by any algorithm.

To be able to define an algorithm, ignoring hardware considerations, requires a set of basic operations, called the instruction set, with which the algorithm can be defined. With a sufficiently robust instruction set, any algorithm can be defined from the same set of instructions. Step 1 of the specification is the formulation of a suitable instruction set.

Computer processor design is determined primarily by the instruction set that is to be interpreted by it. To be able to define any algorithm a *complete* instruction set is required; that is, one that provides the following types of operations:

- Data storage and retrieval, including input and output functions.
- Arithmetic, logic, and shifting functions.
- Conditional branching.

Each instruction must eventually be described formally in terms of the primitive operations that are provided by the processor on which the instruction is to be executed. In general the following considerations influence the choice of instructions to be included in an instruction set:

1. **completenenss**: Data transfer, arithmetic, logic, test/branching, and input/output can be performed by the instruction set.
2. **efficiency**: The most frequently used instructions can be performed raptidly.
3. **regularity**: Instructions should be implemented:
    - so as to adopt a reasonably uniform instruction format,
    - so that similar features in two instructions are implemented in similar ways,
    - so that both operations of "complementary" pairs, when they exist, are provided (eg. add/subtract, shift left/shift right).

As a general rule, uniformity of instruction definition leads to simpler processor

designs. Flexibility of use while preserving uniformity can be achieved in instruction set architectures in three significant ways:

1. By classifying instructions into the major groups described above and using a common instruction format for eqch:
    a. Data Transfer: LOAD, STORE, MOVE, PUSH, SERVE, ...

       Design factors: Choice of register types and data flow paths.

    b. Arithmetic/Logic: ADD, SUB, AND, MASK, ...

       Design requirements: Combinational Logic to provide the operations and data flow paths to supply the operands.

    c. Shifting: SHRL, SHRA, CIR, ...

       Design consideration: Whether to provide shifting as part of a register function, or as a combinational circuit.

    d. Testing/Branching: BEQ, BLE, <interrupt>, ...

       Design requirements: Components for generating and collecting testable conditions, and modifying or saving the instruction pointer.

2. By fixing the number of operands that are provided by arithmetic instructions. Machine design strategies can be classified by the number of operands that must be explicitly provided by computational instructions:
    a. **0-operand machines** These machines do not require any explicit identification as to where the operand values will be found. Therefore they are always in the same place. This design is frequently modelled by *stack machines*; that is, those that use a push-down-store to hold the operands and result.

    b. **1-operand machines** The destination and one operand are implied, while one operand is provided explicitly by the instruction. Historically, the implied operand and destination were provided by a single general purpose register called an accumulator; hence this model is often referred to as an accumulator machine.

    c. **multi-operand machines** The operands and result are specified explicitly in the instruction. Depending on where the operands will be found or result will be stored two versions are possible:
        i. **register-register machines** All the operands and the result are held within the processor using a set of

general purpose registers, often implemented with a register file (see below).

ii. **register-memory machines** include at least one reference to memory in addition to referencing specific internal registers.

Most contemporary CPUs today have adopted a multi-operand architecture becuase of the performance benefits obtained in executing such instructions.

The following example program segments illustrate how the instruction formats as well as program organization are determined by the number of operands that are required. each program segment is a machine language translation of the high-level assignment statement, `F = (A + B) * (C - D)`:

- **one-operand architecture**

```
LOAD A          AC <- M[A]
ADD  B          AC <- AC + M[A]
STO  TMP    M[TMP] <- AC
LOAD C          AC <- M[C]
SUB  D          AC <- AC - M[D]
MULT TMP     MP,AC <- AC * M[TMP]
STM  F+4    M[F+4] <- MP
STO  F        M[F] <- AC
```

From this small sample, the cpu datapath will need to include one general work register, `AC`, as well as an "overflow" register `MP` for holding the results of multiplications. In addition, combinational logic will be required to provide the arithmetic operations.

- **register-memory architecture**

```
LDDA A          ACCA <- M[A]
ADDA B          ACCA <- ACCA + M[B]
LDDB C          ACCB <- M[C]
SUBB D          ACCB <- ACCB - M[D]
MUL        ACCA,ACCB <- ACCA * ACCB
STD  F
```

Additional work registers reduce the need to store temporary results in memory as was the case with the one-operand machine. Additional switching logic will be needed if the internal data lines are shared by the work registers. The same general requirements exist for providing arithmetic functions as occurred in the one-operand architecture.