CS-UY 2214 — E20 Manual

Jeff Epstein

Contents

1 Introduction

This manual covers the architecture of the E20 processor and its instruction set. The E20's architecture is more powerful than the E15, and its assembly language is more expressive. The encoding of its machine language is also more complicated, so we will introduce a program, called an *assembler*, to convert its assembly language into machine language, which can be directly interpreted by the processor.

2 Architecture

2.1 Registers

The E20 processor has eight numbered 16-bit registers. By convention, in E20 assembly language, we write each register name with a dollar sign, so the registers are `$0` through `$7`. However, register `$0` is special, because its value is always zero and cannot be changed. Attempting to change the value of register `$0` is valid, but will not produce any effect: the register is immutable. Therefore, we have only seven mutable registers that can actually be used as registers, `$1` through `$7`. The initial value of all registers is zero.

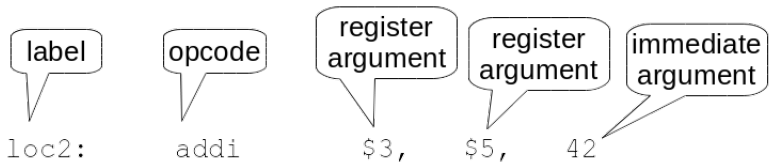
In addition, the E20 processor has a 16-bit program counter register, which cannot be accessed directly through the usual instructions. The program counter stores the memory address of the currently-executing instruction. The initial value of the program counter register is zero. After each non-jump instruction, the program counter is incremented. A jump instruction may adjust the program counter.

2.2 Instructions

Syntax Each instruction in E20 assembly language consists of an opcode, as well as zero, one, two, or three arguments. Opcodes are not case-sensitive. Numeric arguments are expressed in decimal. At least one space is required between the opcode and the first argument. Arguments are separated by commas. Optional spaces may occur around each comma. Optional spaces may occur before the opcode, as well as after the last argument, or immediately after the opcode if there are no arguments. A maximum of one instruction may appear on a single line.

Any opcode may optionally be prefixed by any number of label declarations, each consisting of a label name followed by a colon. At least one space is required between a label declaration and any subsequent label declaration or opcode on the same line. A label name is a non-empty sequence of letters, digits, and underscores; the first character may not be a digit. Labels may appear either on the same line as an instruction, or on a line by themselves. A label declaration may also appear on a line after the last instruction of a program. Label names are not case-sensitive.

Comments begin with a hash mark (`#`) and continue to the end of the line. Blank lines are ignored.



There are three kinds of arguments:

- register argument — an integer $0 \ldots 7$ prefixed with dollar sign. For example, `$4`
- immediate argument — may consist of a positive decimal integer, such as `34`; a negative decimal integer, such as `-4`; or a label name, such as `loc2`
- memory reference — a combination of an immediate argument and a register argument, with the latter contained in parentheses. For example, `0($2)` or `array($0)`

Examples The `add` instruction takes three register arguments: a destination register and two source registers, as follows:

```
add $1, $2, $3          # add the value of $2 and the value of $3, store sum into $1
```

The `addi` instruction takes a destination register, a source register, and an immediate value, as follows. Note that the third argument is an immediate value, not a register:

```
addi $1, $2, 3          # add the value of $2 and 3, store sum into $1
```

Due to the special property that register `$0` always has value zero, we can use the same opcode to store an immediate value into a register. Below, we add the immediate value to the value of register `$0`, so the sum will always equal the immediate value:

```
addi $1, $0, 3          # store 3 into $1
```

The `j` instruction takes an immediate argument, and jumps to the memory address indicated. The address may be expressed numerically or with a label:

```
j 8                     # jump to address 8
j somelabel             # jump to address given by label (defined elsewhere)
```

The `lw` instruction has one register argument and one memory reference argument. It will calculate a pointer given by the memory reference argument, and read the value stored at that memory address, storing it into the given register. Each memory reference argument has two parts: an immediate value (specified as a number or as a label) and a parenthesized register name. The memory address to be accessed is calculated by adding the immediate value and the value stored in the register:

```
lw $4, 0($3)            # the memory reference is 0($3). Here,
                        # the immediate part is zero, and the
                        # register is $3. Therefore, the address
                        # to be read is equal to the value of
                        # register $3, plus zero. The value at that
                        # memory location will be stored into
                        # register $4

lw $4, foo($0)          # the memory reference is foo($0). Here, the
                        # immediate part is foo, a label. The register
                        # part is $0, which we know has the value 0.
```

```
                        # Therefore, the address to be read is equal
                        # to the value of foo, plus zero. The value
                        # at that memory location will be stored into
                        # register $4

    lw $4, foo($3)      # the memory reference is foo($3). Here, the
                        # immediate part is foo, a label. The register
                        # part is $3. Therefore, the address to be read
                        # is equal to the value of foo, plus the value
                        # of register $3. The value at that memory location
                        # will be stored into register $4.
```

Further examples of instructions are given below, as well as in the provided example programs.

Common errors The format of the `addi` instruction specifies the following fields:

- opcode — 3 bits
- source register — 3 bits
- destination register — 3 bits
- immediate — 7 bits

This means that the immediate value must be expressed as a signed 7-bit number, in the range of $-64 \ldots 63$. Any value outside of that range is not expressible in the allocated number of bits, and therefore the assembler must reject it. A similar restriction exists for other instructions that take an immediate field, such as `jeq` and `slti`.

A consequence of the above stipulation is that it's impossible to directly set all bits of a register at once, with a single `addi` instruction. For that, we would need to use `lw` to read a stored value from memory.

2.3 Memory

The E20 processor has 8192 16-bit cells of memory. Initially, the program is loaded into memory, starting at address zero. The program counter is initially zero, so execution begins at the first instruction of the program. Memory cells that do not initially contain program instructions are initially set to zero.

The 8192 cells of memory are addressed using 13-bit addresses. When accessing a memory cell via a pointer (either through the program counter, or with the `lw` and `sw` opcodes), the 3 most significant bits of the pointer are ignored. Therefore, an instruction accessing memory via a pointer value 43222 (1010100011010110_2) refers to the same address as a pointer value 2262 (0000100011010110_2). Similarly, the instruction to be executed will be determined by the least significant 13 bits of the program counter.

E20 assembly language allows the use of *labels*. A label is a symbol that represents a known memory address. A label's name is a sequence of characters obeying the following rules: the first character may be a letter or an underscore; subsequent characters may be a letter, an underscore, or a digit; there must be at least one character. Label names are not case-sensitive.

A label may be declared in your assembly code by giving its name followed by a colon, preceding any instruction (including normal instructions, pseudo-instruction, and directives). Multiple labels may be declared at a single instruction, one after another. The value of the label will be the address of the instruction that it precedes. If the label precedes no instruction, the value of the label will be the address that a subsequent instruction would occupy. A label's name may be used as the `imm` field of any instruction, assuming that the value of the label can be expressed in the number of bits allocated to that field. A label's declaration need not precede its use.

Labels make it easier to write E20 assembly language programs. Instead of referring to addresses numerically, which will change as we add or remove instructions in the development of our program, we can

refer to addresses by name, which the assembly will convert to the appropriate numeric address. Labels can denote the address of an instruction (which can serve as the target of a jump instruction) or the address of a data location (which can serve as the target of load/store word instruction).

Here's an example program, with assembly language on the left, and the corresponding machine code on the right:

```
first_label:
    movi $1, 1
    j first_label
    j second_label
second_label:
    movi $2, 2
```

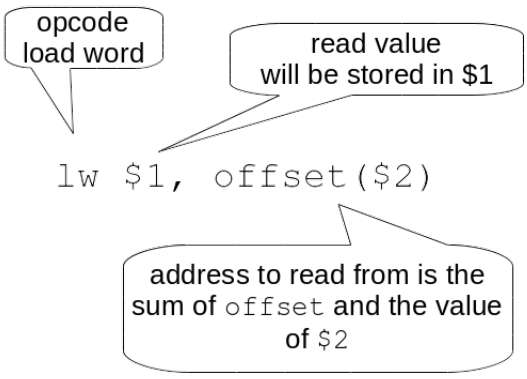
```
ram[0] = 16'b0010000010000001;
ram[1] = 16'b0100000000000000;
ram[2] = 16'b0100000000000011;

ram[3] = 16'b0010000100000010;
```

In the above program, the label `first_label` is declared before the instruction at address 0, therefore its value is zero. The `second_label` is declared before the instruction at address 3, so its value is 3. The instruction `j first_label` refers to `first_label`, so the immediate field of that instruction contains the value of that label. Similarly, the instruction `j second_label` has an immediate field containing the value of that label. In the above listing, the color of the labels matches to the color of the corresponding bits.

Memory can be read and written using the `lw` and `sw` instructions, respectively.

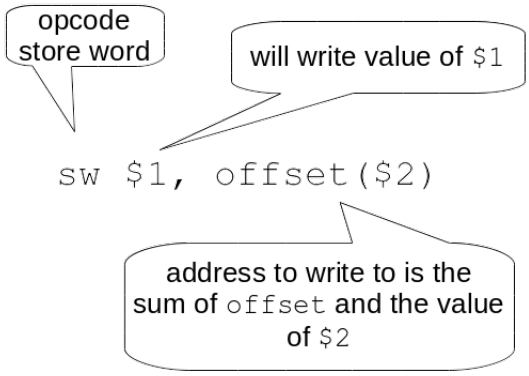
The `lw` (load word) instruction will read a memory cell, copying its value into a register:



- The first argument of `lw` is the register to copy the value into.
- The second argument of `lw` has two parts:
  - an immediate field, which can be a number or a label, and
  - a register field.

  The immediate will be summed with the value of the register, giving the memory address to read from.

The `sw` (store word) instruction will write to a memory cell, copying a value from a register:



- The first argument of `sw` is the register holding the value to be copied to be memory.
- The second argument of `sw` has two parts:
  - an immediate field, which can be a number or a label, and
  - a register field.

  The immediate will be summed with the value of the register, giving the memory address to write to.

Here's an example:

```
    lw $2, myvariable($0)
    movi $3, 1
    lw $4, myvariable($3)
myvariable:
    .fill 42
    .fill 97
```

```
ram[0] = 16'b1000000100000011;
ram[1] = 16'b0010000110000001;
ram[2] = 16'b1000111000000011;

ram[3] = 16'b0000000000101010;
ram[4] = 16'b0000000001100001;
```

In the above program, the instruction at address 0 calculates the memory address `myvariable + $0`, which is equal to `myvariable`. The value of the label `myvariable` is 3, because the label is declared at the instruction at memory address 3. Therefore, the first `lw` reads the value at that address, which is 42, and loads it into register `$2`. Then, in the instruction at address 2, we calculate the memory address `myvariable + $3`, which is `myvariable + 1`, which is `4`. The value at memory address 4 is is 97, which is loaded into register `$4`.

Variables We can use labels, `lw`, and `sw` to provide variables to our program. Consider the following:

```
beginning:
    lw $1, mycounter($0)
    addi $1, $1, 1
    sw $1, mycounter($0)
    j beginning
mycounter:
    .fill 0
```

```

ram[0] = 16'b1000000010000100;
ram[1] = 16'b0010010010000001;
ram[2] = 16'b1010000010000100;
ram[3] = 16'b0100000000000000;

ram[4] = 16'b0000000000000000;
```

In the above program, `mycounter` is a label with the value 4, referring to a memory address initialized to the value 0. In each iteration of the loop, the value from that address is read, incremented, and stored back into memory. The register `$1` is used only temporarily. This approach to storing data is convenient, because although we have relatively few registers, we have access to a large number of memory addresses.

Arrays The numeric relationship of memory addresses to each other allows us to loop through a sequence of adjacent memory addresses, effectively treating them as an array. For example:

```
    movi $1, 0
    movi $3, 0
loop:
    lw $2, myarray($1)
    add $3, $3, $2
    addi $1, $1, 1
    jeq $2, $0, done
    j loop
done:
    halt
myarray:
    .fill 5
    .fill 3
    .fill 20
    .fill 4
    .fill 5
    .fill 0
```

```
ram[0] = 16'b0010000010000000;
ram[1] = 16'b0010000110000000;

ram[2] = 16'b1000010100001000;
ram[3] = 16'b0000110100110000;
ram[4] = 16'b0010010010000001;
ram[5] = 16'b1100100000000001;
ram[6] = 16'b0100000000000010;

ram[7] = 16'b0100000000000111;

ram[8] = 16'b0000000000000101;
ram[9] = 16'b0000000000000011;
ram[10] = 16'b0000000000010100;
ram[11] = 16'b0000000000000100;
ram[12] = 16'b0000000000000101;
ram[13] = 16'b0000000000000000;
```

In the above program, `myarray` identifies the address of the beginning of an array of numbers, which is really just a sequence of memory locations. The program uses `$1` to store an index into the array, starting at zero, and incrementing it thereafter. A value from the array is read by the instruction at address 2. The loop will then sum each consecutive array element into a running total in `$3`, before continuing to the next array element. When it finds an array element whose value is zero, the loop ends. Thus, this program sums all element in the array, and the final value of `$3` will be 37.

Halting As in the E15, the end of the program is expressed by entering a tight loop. We use the `halt` pseudo-instruction for this purpose, which is implemented as a jump to itself. That is, the `halt` instruction is equivalent to the following:

```
    endofprogram: j endofprogram
```

Common errors Although a label may be declared before or after its use, it is an error to refer to a label that has not been declared at all in the program. For example, the following program cannot be assembled:

```
    lw $1, mycounter($0)
    addi $1, $1, 1
    sw $1, mycounter($0)
    j beginning             # undeclared label!
mycounter:
    .fill 0
```

It is an error to declare the same label more than once in a program. On the other hand, it is not an error to declare a label that is never referred to.

Jumps E20 has several jump opcodes:

- `j` — jump, with a 13-bit destination address
- `jal` — jump and link, with a 13-bit destination address
- `jr` — jump to register, with a register containing a 16-bit destination address

- `jeq` — jump if equal, with a 7-bit relative destination address

In the case of `jr`, the register argument provides a 16-bit address, which is loaded into the program counter.

In the case of `j` and `jal`, the instruction format provides only a 13-bit destination address. The provided 13 bits will be copied into the least-significant 13 bits of the program counter, and the remaining 3 bits of the program counter will be set to zero.

In the case of `jeq`, the instruction format provides only a 7-bit relative destination address, which is interpreted as a signed 2's complement number, sign-extended[1], and added to the 16-bit address of the subsequent instruction; this sum is stored into the program counter. This means that using only this instruction, it is impossible to jump to a location more than $2^{7-1} - 1 = 0111111_2 = 63_{10}$ cells ahead of the current program counter, or more than $-2^{7-1} = 1000000_2 = -64_{10}$ behind it.

In E20 assembly language, it is an error to provide a immediate value, either numerically or by label, that exceeds the allowed number of bits for the immediate field of the given instruction. For example, `j 9000` cannot be assembled, since 9000 cannot be expressed as a 13-bit unsigned integer. Similarly, `jeq $0, $0, 90` cannot be assembled if it is located at address 0, since $90 - 0 - 1 = 89$ cannot be expressed as a 7-bit 2's complement integer.

2.4 Comparison

E20 supports equality comparison and less-than comparison.

Equality Equality comparison is achieved with the `jeq` instruction. Its arguments are two registers, and an immediate value that indicates a memory location. If the values of its two register arguments are equal, then the program will jump to the given address. Otherwise, the program will proceed to the subsequent instruction. For example, consider the following program:

```
    jeq $1, $0, they_are_equal      # if $1 is 0, go to they_are_equal
    addi $1, $1, 1                  # we execute this only when $1 is not 0
    j done                          # avoid fall-through to next instruction
they_are_equal:
    addi $1, $1, 2                  # we execute this only when $1 is 0
done:
                                    # end of program
```

The above E20 assembly program might be expressed roughly using the following Python code:

```
if Reg1 == 0:
    Reg1 += 2
else:
    Reg1 += 1
```

Note that the `j` instruction is necessary. In contrast, consider the following program, identical except for the omission of the `j` instruction:

```
    jeq $1, $0, they_are_equal      # if $1 is 0, go to they_are_equal
    addi $1, $1, 1                  # we execute this only when $1 is not 0
they_are_equal:
    addi $1, $1, 2                  # we execute this regardless
done:
                                    # end of program
```

[1] To *sign-extend* is to increase the number of bits of a 2's complement number, in such a way that does not change the value it represents. The most-significant bit of the number, which represents the sign (negative or non-negative) of the value, must be reproduced into new bits positions. For example, consider a 4-bit 2's complement number, $1011_2 = -5_{10}$. If we sign-extend this to an 8-bit number, it will become 11111011_2.

In this case, we need to sign-extend the 7-bit address to 16 bits, because the result must be added to a signed 16-bit number.

This version would be expressed with the following Python code:

```
if Reg1 != 0:
    Reg1 += 1
Reg1 += 2
```

Less-than Less-than comparison is achieved with the `slt` or `slti` instructions, in conjunction with `jeq`. The `slt` or `slti` instructions will set a given register to 1 if its second argument is less than its third argument, and to 0 otherwise. Then, we can use `jeq` to check the value of the given register. For example:

```
    slt $1, $2, $3                  # $1 will be set to 1 if $2 < $3
    jeq $1, $0, not_less            # if $1 is now 0, we know $2 >= $3
    addi $2, $2, 1                  # we execute this only when $2 < $3
    j done                          # avoid fall-through to next instruction
not_less:
    addi $2, $2, 2                  # we execute this only when $2 >= $3
done:
                                    # end of program
```

The above E20 assembly program might be expressed roughly using the following Python code:

```
if Reg2 < Reg3:
    Reg2 += 1
else:
    Reg2 += 2
```

Note that `slt` and `slti` perform unsigned comparison. That is, both comparands are assumed to be non-negative binary numbers.

Less-than-or-equals We can combine the above approaches to provide less-than-or-equals comparison. First we compare for equality, then for less-than.

```
    jeq $2, $3, less_or_equal       # jump if $2 and $3 are equal, otherwise fallthrough
    slt $1, $2, $3                  # $1 will be set to 1 if $2 < $3
    jeq $1, $0, not_less_or_equal   # if $1 is now 0, we know $2 > $3
less_or_equal:
    addi $2, $2, 1                  # we execute this only when $2 <= $3
    j done                          # avoid fall-through to next instruction
not_less_or_equal:
    addi $2, $2, 2                  # we execute this only when $2 > $3
done:
                                    # end of program
```

2.5 Subroutines

The `jal` and `jr` instructions can be used together to achieve *subroutines*, a flow control mechanism that allows execution to return to an early point of code, and can be invoked from multiple places within a program. A subroutine is therefore like a function, but does not necessarily encompass the parameters and return values that functions usually entail.

Before we discuss how subroutines work in E20, consider the following C++ code:

```
1 int x = 0;
2 int y = 0;
3
```

```
 4  void sub1() {
 5      x++;
 6      y--;
 7  }
 8
 9  void main() {
10      sub1();
11      sub1();
12  }
```

In the above code, execution starts at line 10. The subroutine `sub1` is invoked, which does stuff in lines 5 and 6, and execution then returns to line 11. At line 11, the subroutine is invoked again, lines 5 and 6 are executed again, and execution returns to line 12, whereupon the program ends.

The subtlety of such code is the mechanism by which these invocations and returns are implemented. How does `sub1` know where to return to when it finishes, particularly when it returns to a different place each time?

On most architectures, this feat is accomplished by storing the address of the subsequent instruction at the time of a subroutine's invocation. Then, when the subroutine has finished, it copies that stored address into the program counter. This approach is used by the E20 processor.

- The `jal` (jump and link) instruction is similar to the ordinary jump instruction (`j`), with the key difference that in addition to storing the immediate value into the program counter, `jal` will store the address of the subsequent instruction into `$7`.
- The `jr` (jump to register) instruction copies the address in the given register into the program counter. Therefore the instruction `jr $7` will jump to the address in the register `$7`, where it was previously stored by `jal`.

Below is a direct E20 translation of the above C++ program. Note that we use `jal sub1` to invoke the subroutine, and `jr $7` to return from it.

```
main:
    jal sub1              # invoke subroutine
    jal sub1              # invoke subroutine again
    halt                  # end program

sub1:
    lw $1, x($0)          # x++
    addi $1, $1, 1
    sw $1, x($0)

    lw $1, y($0)          # y--
    addi $1, $1, -1
    sw $1, y($0)

    jr $7                 # return from subroutine

x:
    .fill 0
y:
    .fill 0
```

Common errors A limitation of subroutines on E20 is that the value of `$7` must be preserved for the duration of the subroutine. If the subroutine modifies `$7`, then the `jr` instruction may return to an unintended location.

A corollary of the above limitation is that a subroutine cannot call another subroutine. Since the `jal` instruction modifies `$7`, this makes recursion awkward on E20.

Both of these limitation have workarounds. For example, a subroutine could save the value of `$7` into another register or into memory. It would then be free to manipulate `$7`, including by invoking other subroutines, as long as it restores the original value of `$7` before it returns.

2.6 E15 vs E20

The E20 processor can be considered a more powerful version of the E15. Although they have many features in common and their instruction sets are similar, the E20 expands some capabilities.

	E15	E20
registers	four 4-bit registers	seven 16-bit registers
memory	16 12-bit cells of read-only instruction memory	8192 16-bit cells of instruction/-data memory
instruction format	all instructions have 4-bit opcode field, two 2-bit register fields, and 4-bit immediate field	three formats: • 3-bit opcode field, three 3-bit register fields, 4-bit immediate field • 3-bit opcode field, two 3-bit register fields, 7-bit immediate field • 3-bit opcode field, 13-bit immediate field

3 Instruction set

Here we discuss the instructions that form the E20 instruction set. We categorize the instructions by their format, which is based on the number of register arguments. The format determines how many bits are allocated to each argument.

In addition to instructions proper, we discuss *pseudo-instructions*, which are assembly mnemonics that are translated to another instruction. Finally, we cover *assembler directives*, which change the behavior of the assembler but do not correspond to any specific instruction.

In the following subsections, we give the following information about each instruction:

- the instruction name and syntax — The header of each subsection gives the assembly language syntax.
- the instruction format — Within each 16-bit instruction, we show which bits must have which values.
- the informal behavior — We give a brief prose description of the behavior of the instruction when executed.
- the formal behavior — We use a symbolic notation to express the behavior of the instruction. Here, we use the following symbols:
  - `<-`: indicates that the location on the left will be updated with the value on the right
  - `$reg` (and variants, such as `$regA`, `$regSrc`, etc): the instruction's register argument

- `imm`: the instruction's immediate argument
- `R`: the processor's register file, indexed by the name of a register, $0 \ldots 7$
- `Mem`: the processor's memory, indexed by a memory address
- `pc`: the program counter

Unless otherwise specified, immediate values can be positive, negative, or zero and are represented in 2's complement. Unless otherwise specified, all instructions increment the program counter.

3.1 Instructions with three register arguments

3.1.1 add $regDst, $regSrcA, $regSrcB

15 14 13	12 11 10	9 8 7	6 5 4	3 2 1 0
3 bits	3 bits	3 bits	3 bits	4 bits
000	regSrcA	regSrcB	regDst	0000

Example: `add $1, $0, $5`
Adds the value of registers `$regSrcA` and `$regSrcB`, storing the sum in `$regDst`.
Symbolically: `R[regDst] <- R[regSrcA] + R[regSrcB]`

3.1.2 sub $regDst, $regSrcA, $regSrcB

15 14 13	12 11 10	9 8 7	6 5 4	3 2 1 0
3 bits	3 bits	3 bits	3 bits	4 bits
000	regSrcA	regSrcB	regDst	0001

Mnemonic: *Subtract*
Example: `sub $1, $0, $5`
Subtracts the value of register `$regSrcB` from `$regSrcA`, storing the difference in `$regDst`.
Symbolically: `R[regDst] <- R[regSrcA] - R[regSrcB]`

3.1.3 or $regDst, $regSrcA, $regSrcB

15 14 13	12 11 10	9 8 7	6 5 4	3 2 1 0
3 bits	3 bits	3 bits	3 bits	4 bits
000	regSrcA	regSrcB	regDst	0010

Example: `or $1, $0, $5`
Calculates the bitwise OR of the value of registers `$regSrcA` and `$regSrcB`, storing the result in `$regDst`.
Symbolically: `R[regDst] <- R[regSrcA] | R[regSrcB]`

3.1.4 and $regDst, $regSrcA, $regSrcB

15 14 13	12 11 10	9 8 7	6 5 4	3 2 1 0
3 bits	3 bits	3 bits	3 bits	4 bits
000	regSrcA	regSrcB	regDst	0011

Example: `and $1, $2, $5`
Calculates the bitwise AND of the value of registers `$regSrcA` and `$regSrcB`, storing the result in `$regDst`.
Symbolically: `R[regDst] <- R[regSrcA] & R[regSrcB]`

3.1.5 slt $regDst, $regSrcA, $regSrcB

15 14 13	12 11 10	9 8 7	6 5 4	3 2 1 0
3 bits	3 bits	3 bits	3 bits	4 bits
000	regSrcA	regSrcB	regDst	0100

Mnemonic: *Set if less than*
Example: `slt $1, $2, $5`
Compares the value of `$regSrcA` with `$regSrcB`, setting `$regDst` to 1 if `$regSrcA` is less than `$regSrcB`, and to 0 otherwise.

The comparison performed is unsigned, meaning that the two operands are treated as unsigned 16-bit integers, not 2's complement integers. Therefore, `0x0000 < 0xFFFF`.
Symbolically: `R[regDst] <- (R[regSrcA] < R[regSrcB]) ? 1 : 0`

3.1.6 jr $reg

15 14 13	12 11 10	9 8 7	6 5 4	3 2 1 0
3 bits	3 bits	3 bits	3 bits	4 bits
000	reg	000	000	1000

Mnemonic: *Jump to register*
Example: `jr $1`
Jumps unconditionally to the memory address in `$reg`.

The jump destination is expressed as an absolute address. All 16 bits of the value of `$reg` are stored into the program counter.
Symbolically: `pc <- R[reg]`

3.2 Instructions with two register arguments

3.2.1 slti $regDst, $regSrc, imm

15 14 13	12 11 10	9 8 7	6 5 4 3 2 1 0
3 bits	3 bits	3 bits	7 bits
111	regSrc	regDst	imm

Mnemonic: *Set if less than, immediate*
Example: `slti $1, $2, some_label`
Example: `slti $1, $2, 30`
Compares the value of `$regSrc` with sign-extended `imm`, setting `$regDst` to 1 if `$regSrc` is less than `imm`, and to 0 otherwise.

The comparison performed is unsigned, meaning that the two operands are treated as unsigned 16-bit integers, not 2's complement integers. Therefore, `0x0000 < 0xFFFF`. This is true even though the argument is expressed as a 7-bit signed number.
Symbolically: `R[regDst] <- (R[regSrc] < imm) ? 1 : 0`

3.2.2 lw $regDst, imm($regAddr)

15 14 13	12 11 10	9 8 7	6 5 4 3 2 1 0
3 bits	3 bits	3 bits	7 bits
100	regAddr	regDst	imm

Mnemonic: *Load word*
Example: `lw $1, some_label($2)`

Example: `lw $1, 5($2)`
Example: `lw $1, -1($2)`
Calculates a memory pointer by summing the signed number `imm` and the value `$regAddr`, and loads the value from that address, storing it in `$regDst`.
The memory address is interpreted as an absolute address. The least significant 13 bits of the value of `$regAddr + imm` are used to index into memory.
Symbolically: `R[regDst] <- Mem[R[regAddr] + imm]`

3.2.3 sw $regSrc, imm($regAddr)

15 14 13	12 11 10	9 8 7	6 5 4 3 2 1 0
3 bits	3 bits	3 bits	7 bits
101	regAddr	regSrc	imm

Mnemonic: *Store word*
Example: `sw $1, some_label($2)`
Example: `sw $1, 5($2)`
Example: `sw $1, -1($2)`
Calculates a memory pointer by summing the signed number `imm` and the value `$regAddr`, and stores the value in `$regSrc` to that memory address.
The memory address is interpreted as an absolute address. The least significant 13 bits of the value of `$regAddr + imm` are used to index into memory.
Symbolically: `Mem[R[regAddr] + imm] <- R[regSrc]`

3.2.4 jeq $regA, $regB, imm

15 14 13	12 11 10	9 8 7	6 5 4 3 2 1 0
3 bits	3 bits	3 bits	7 bits
110	regA	regB	rel_imm

where rel_imm = imm - pc - 1

Mnemonic: *Jump if equal*
Example: `jeq $1, $0, some_label`
Example: `jeq $2, $3, 23` (jumps to address 23)
Compares the value of `$regA` with `$regB`. If the values are equal, jumps to the memory address identified by the address `imm`, which is encoded as the signed number `rel_imm`.
The jump destination, `imm`, is encoded as a relative address, `rel_imm`. That is, when a jump is performed, the value `rel_imm` is sign-extended and added to the successor value of the program counter. Therefore, the actual address that will be jumped to is equal to the current program counter plus one plus the immediate value. This means that when encoding a `jeq` instruction to machine code, the field in the least significant seven bits must be the difference between the desired destination and one plus the program counter.
Symbolically: `pc <- (R[regA] == R[regB]) ? pc+1+rel_imm : pc+1`

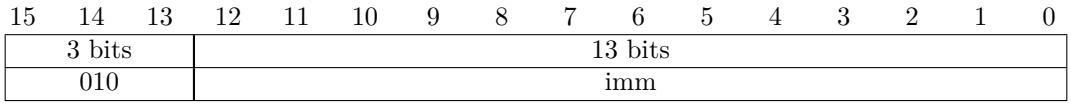
3.2.5 addi $regDst, $regSrc, imm

15 14 13	12 11 10	9 8 7	6 5 4 3 2 1 0
3 bits	3 bits	3 bits	7 bits
001	regSrc	regDst	imm

Mnemonic: *Add immediate*
Example: `addi $1, $0, some_label`
Example: `addi $2, $2, -5`
Adds the value of register `$regSrc` and the signed number `imm`, storing the sum in `$regDst`.
Symbolically: `R[regDst] <- R[regSrc] + imm`

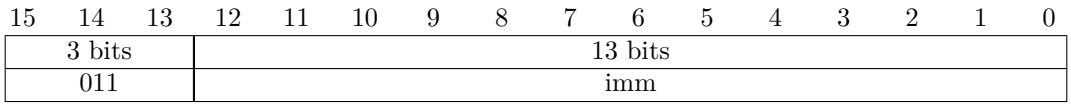
3.3 Instructions with no register arguments

3.3.1 j imm



Mnemonic: *Jump*
Example: `j some_label`
Example: `j 42`
Jumps unconditionally to the memory address `imm`.
The jump destination is expressed as a non-negative absolute address. All 13 bits of `imm` are stored into the program counter, while the most significant 3 bits of the program counter will be set to zero.
Symbolically: `pc <- imm`

3.3.2 jal imm



Mnemonic: *Jump and link*
Example: `jal some_label`
Example: `jal 42`
Stores the memory address of the next instruction in sequence in register `$7`, then jumps unconditionally to the memory address `imm`.
The jump destination is expressed as a non-negative absolute address. All 13 bits of `imm` are stored into the program counter, while the most significant 3 bits of the program counter will be set to zero.
Symbolically: `R[7] <- pc+1; pc <- imm`

3.4 Pseudo-instructions

Pseudo-instructions, unlike proper instructions, do not have their own unique encoding in machine language. Instead, pseudo-instructions are a shorthand form of expressing more complicated assembly instructions. The assembler will translate a pseudo-instruction into an actual instruction, which will then be assembled normally.

3.4.1 movi $reg, imm

Mnemonic: *Move immediate*
Example: `movi $2, 55`
Example: `movi $7, some_label`
Copies the value `imm` to the register `$reg`.
The `movi $reg, imm` instruction is translated by the assembler as `addi $reg, $0, imm`.

3.4.2 nop

Mnemonic: *No operation*
Performs no operation, other than incrementing the program counter.
The `nop` instruction is translated by the assembler as `add $0, $0, $0`.

3.4.3 halt

Performs no operation at all. The program counter is not incremented, resulting in an infinite loop. By convention, represents the end of the program.

The `halt` instruction is translated by the assembler as an unconditional jump (`j`) to the current memory location.

3.5 Assembler directives

Directives are not processor instructions, but rather commands to the assembler itself.

3.5.1 .fill imm

Example: `.fill some_label`

Example: `.fill 42`

Inserts a 16-bit immediate value directly into memory at the current location. This directive instructs the assembler to put a number into the place where an instruction would normally be. The immediate value may be specified numerically (positive, negative, or zero) or with a label.

3.6 Undefined bit patterns

Any bit pattern not covered by any of the previous sections is not valid machine code and its interpretation is undefined.

4 Examples

Here are some examples of E20 assembly language programs. Further examples, including the machine code translation, are available on the class website.

4.1 Math

```
# add, addi, and sub work as you expect, except
# they take three arguments. The first argument
# is the destination, the other two are the
# sources.
# There is no subi. Instead you have to use addi
# with a negative immediate.
addi $1, $0, 5      # $1 := 5
addi $2, $1, -2     # $2 := $1 + (-2)
add $3, $1, $2      # $3 := $1 + $2

# Notice that $0 is special, because it is always
# zero. So using $0 as a source lets us effectively
# do a movi, as in the first instruction below.
addi $4, $0, 55     # $4 := 55
sub $5, $4, $1      # $5 := $4 - $1

# We also have bitwise operators AND and OR.
# (but not ori and andi). As above, the first
# operand is the destination register.
or $6, $2, $5
```

```
and $7, $2, $5

halt       # end the program with an infinite loop
```

4.2 Loops

```
# A simple loop, counting down from 10
# Here, we introduce a pseudo-instruction as a synonym
# for addi. Specifically:
#      movi $1, 10   <===>   addi $1, $0, 10
# In other words, movi is really adding zero to the
# immediate and storing the result in $1.
# We also introduce jeq, which compares its first two
# arguments and conditionally jumps to the label in
# the third.

    movi $1, 10         # Initialize counter to 10
beginning:
    jeq $1, $0, done    # if $1 == $0, go to done
    addi $1, $1, -1     # Decrement $1
    j beginning         # go to top of loop
done:
    halt                # we've finished
```

4.3 Subroutines

```
# Example of simple subroutine. At the jal
# instruction, we call subroutine proc.
# It store a value in $3, then returns.
# Upon return, we go back to the address
# after the subroutine invocation. Thus,
# the addi instructions will be executed
# in the following order:
#   X1, X2, X3

    addi $1, $0, 1     # X1: assign 1 to $1
    jal proc
    addi $2, $0, 2     # X3: assign 2 to $2
    halt

proc:
    addi $3, $0, 3     # X2: assign 3 to $3
    jr $7
```

4.4 Variables

```
# Examples of variables in memory.
# var1 is a label identifying a memory address
# containing 30, and var2 is a label identifying
```

```
# a memory address containing 5. We load the value
# at var1 into $1, and the value at var2 into $2.
# Then we AND them into $3, and OR them into $4.
# Then we store $3 into var3.

    lw $1, var1($0)    # read from address var1 + 0
    lw $2, var2($0)    # read from address var2 + 0
    and $3, $1, $2     # AND the values together
    or $4, $1, $2      # then OR them together
    sw $3, var3($0)    # write the AND result into memory

    halt               # program ends

var1:                  # declare a label
    .fill 30           # insert the value 30 into memory here

var2:
    .fill 5

var3:
    .fill 0
```

5 Hardware implementation

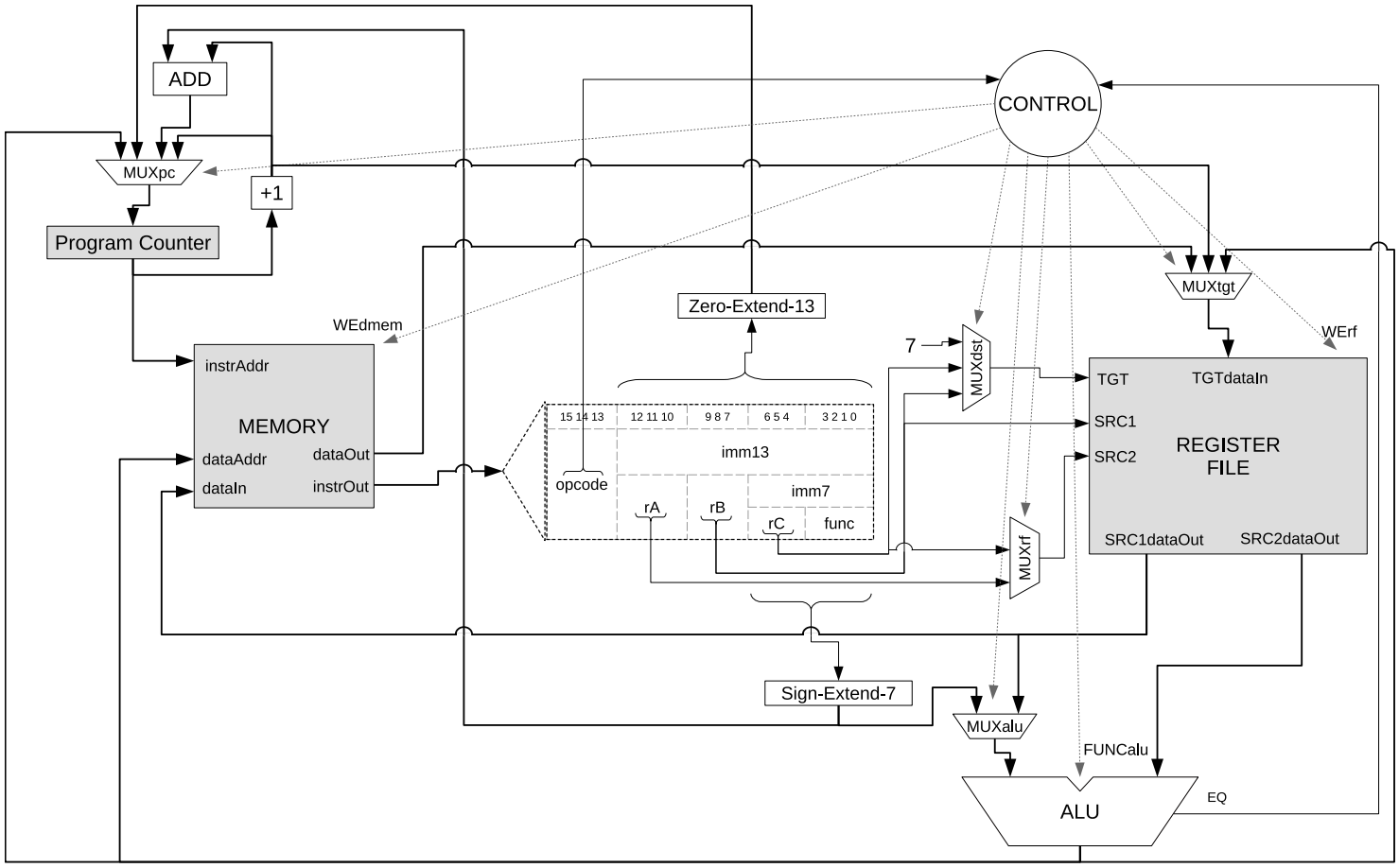
There are three version of the E20 processor: a single-cycle version, a multicycle version, and a pipelined version. Each version has different performance characteristics, but implements the same architecture: that is, all three versions must interpret the same machine code and produce identical results, although the way they achieve those results will differ.

5.1 Single-cycle version

Here we present details of the hardware implementation of the single-cycle version of the E20 processor. In this version, each instruction executes in its entirety in one clock cycle.

5.1.1 Circuit diagram

In the following diagram, thick lines represent 16-bit wires. Dotted lines represent control signals.



Every cycle, the program counter selects an instruction to be read from memory, which is passed to the control module. The control module acts as a decoder, which, on the basis of the current instruction, sets all of the control signals appropriately, and all of the control and data lines are set up. Then the new values are stored in memory, the register file, and the program counter register.

5.1.2 Control signals

The control signals are set based on the basis of only the opcode and the EQ wire from the ALU. The EQ wire will be 1 when the result of the ALU operation is zero, and 0 otherwise.

The control signals are as follows:

FUNC_{alu}	This wire directs the ALU which operation to perform.
MUX_{alu}	This wire directs a mux to select one of two inputs to the ALU: either from a register, or a sign-extended value from the immediate field.
MUX_{pc}	This 2-bit wire directs a mux to select one of four signals as the next value of the program counter: the ALU output, the increment of the current program counter, the zero-extended 13-bit immediate field, or the sum of the immediate field and the increment of the current program counter.
MUX_{rf}	This wire directs a mux to select one of two register selectors as input to the read input of the register file.
MUX_{tgt}	This 2-bit wire directs a mux to select one of three signals as input to the write selector of the register file: the incremented program counter, the ALU output, or the data memory output.
MUX_{dst}	This 2-bit wire directs a mux to select one of three possible register selectors indicating where to write the result of the instruction: the second register field, the third register field, or the literal 7.
WE_{rf}	This wire enables or disables the write port of the register file. If the signal is 1, the register file will write the value of TGTdata to register TGT. Otherwise, writing is blocked.
WE_{dmem}	This wire enables or disables the write port of memory. If the signal is 1, memory will write the value of dataValIn at address dataAddr. Otherwise, writing is blocked.

The control wires can carry the following concrete values with the indicated meaning:

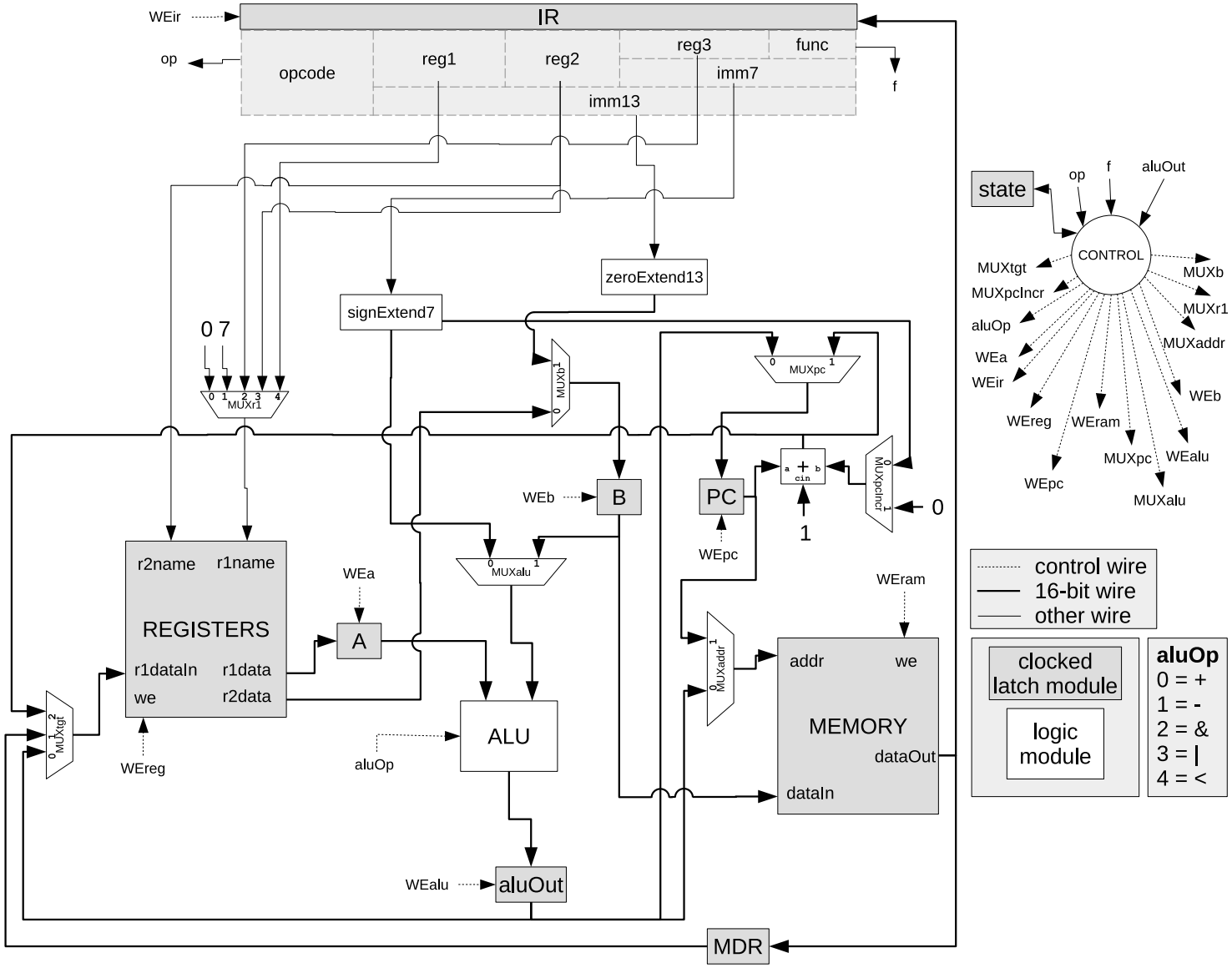
Control signal	Values
FUNC_{alu}	0 = add 1 = subtract 2 = and 3 = or 4 = slt
MUX_{alu}	0 = register 1 = immediate
MUX_{pc}	0 = ALU 1 = pc+1 2 = pc+1+imm 3 = 13-bit immediate
MUX_{rf}	0 = rA 1 = rC
MUX_{tgt}	0 = ALU 1 = pc+1 2 = dmem
MUX_{dst}	0 = rB 1 = rC 2 = literal 7
WE_{rf}	0 = disable 1 = enable
WE_{dmem}	0 = disable 1 = enable

5.2 Multicycle version

Here we present details of the hardware implementation of the multicycle version of the E20 processor. In this version, each instruction executes in more than one clock cycle.

5.2.1 Circuit diagram

In the following diagram, thick lines represent 16-bit wires. Thin lines represent wires of less than 16 bits. Dotted lines represent control signals. The control unit, shown to the right, has implicit connection to many components, including all the multiplexors, but for simplicity the actual wires are not shown.



In addition to the architectural registers `$0`...`$7`, this version of the processor has several microarchitectural registers that are not accessible to the programmer:

- `IR` — the instruction register store the current instruction.
- `A` and `B` — these registers temporarily hold the input to the ALU before the EXEC stage.
- `aluOut` — this register stores the output of the ALU after the EXEC stage.
- `MDR` — the memory data register stores output from the memory unit between the MEM and WB stage.

- `state` — stores the current state of execution. Used exclusively by the control unit.

5.2.2 High-level state diagram

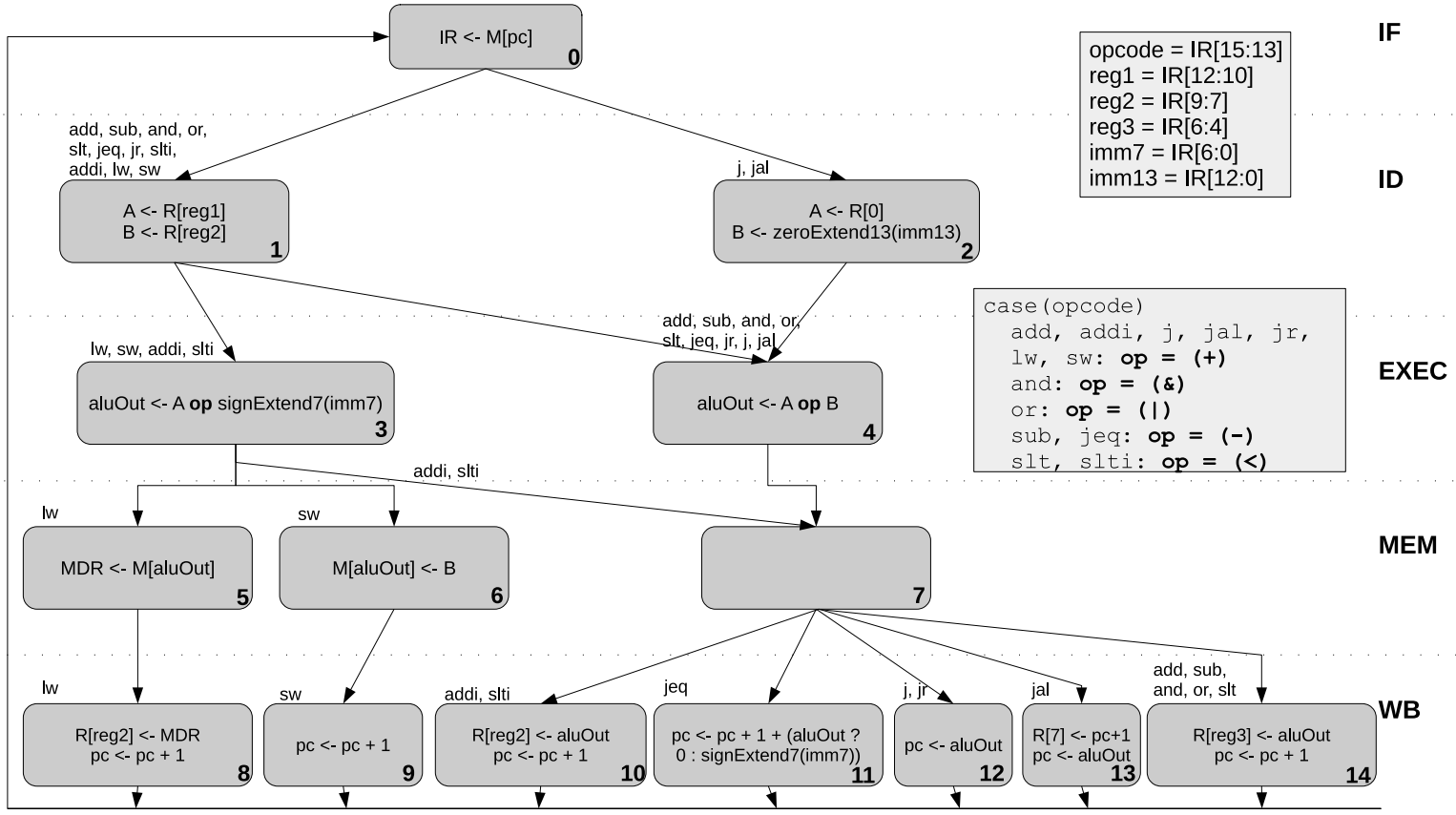
In the following diagram, we show the relationship between the states of the multicycle processor.

Execution of all instructions is divided into five *stages*: instruction fetch (IF), instruction decode (ID), execute (EXEC), memory (MEM), and writeback (WB). Each stage executes in one clock cycle. All instructions execute each of the five stages in sequence. In each stage, each instruction will be in one of several possible *states*, each identified by a number between 0 and 14. The determination of which state to use is made by the control module, based on the instruction's opcode.

Execution of all instructions begins at state 0. Transitions to subsequent states, shown by arrows, are predicated by the opcode. After executing the last stage of each instruction, execution resumes with the next instruction at state 0.

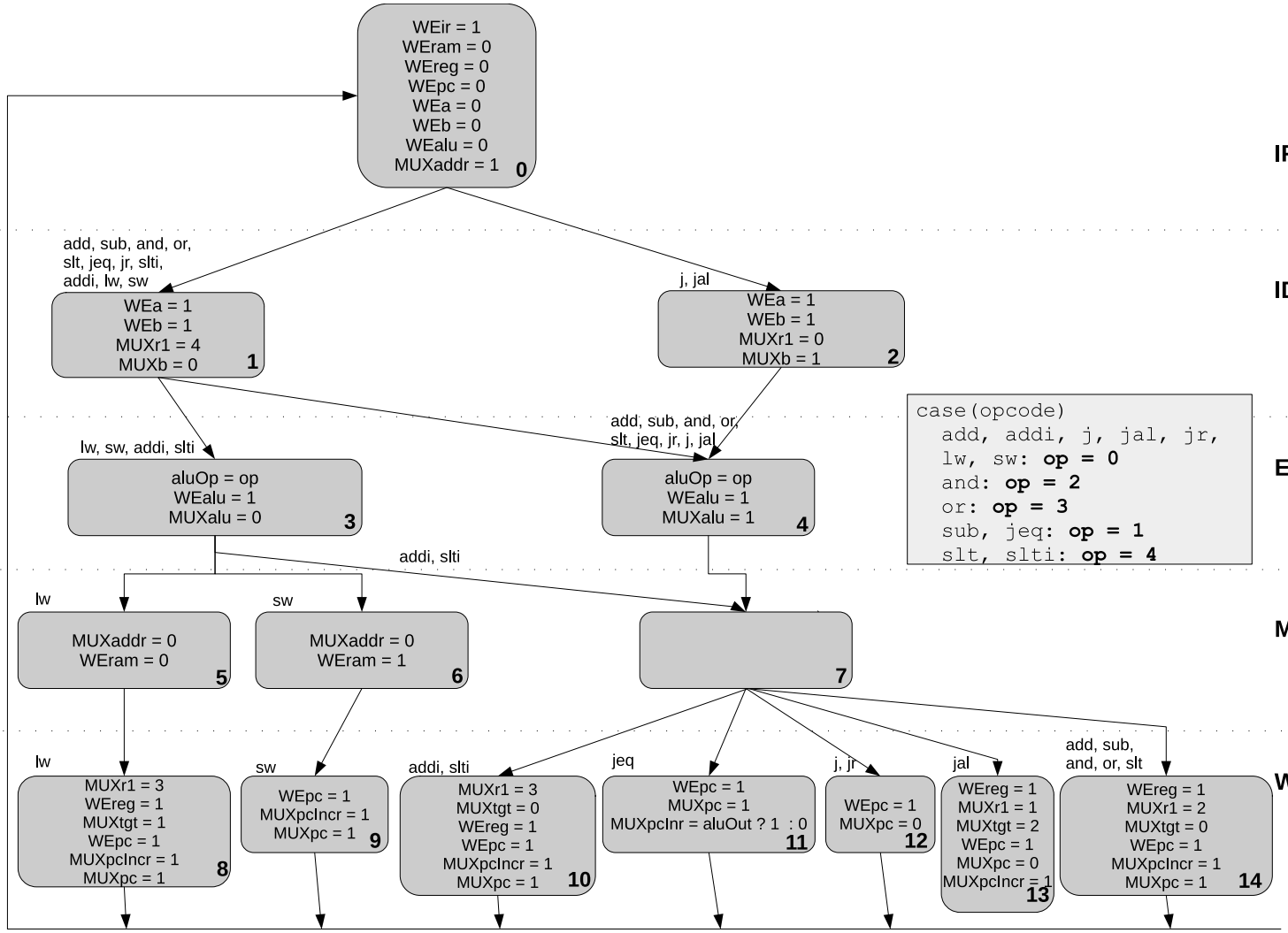
Within the box for each state are shown the symbolic micro-instructions necessary to complete that stage. The advancement of the state for the subsequent clock cycle is implicit. Similar to the notation used in section 3, the micro-instructions indicate storage operations using the left arrow (`<-`), memory access using `M[...]` and architectural register access using `R[...]`.

The terms to the right side show which execution stage each state belongs to.



5.2.3 Low-level state diagram

The low-level state diagram is similar to the high-level diagram shown above, however instead of showing symbolic micro-instructions, each state shows the exact control signals necessary to complete that stage.



5.3 Pipelined version

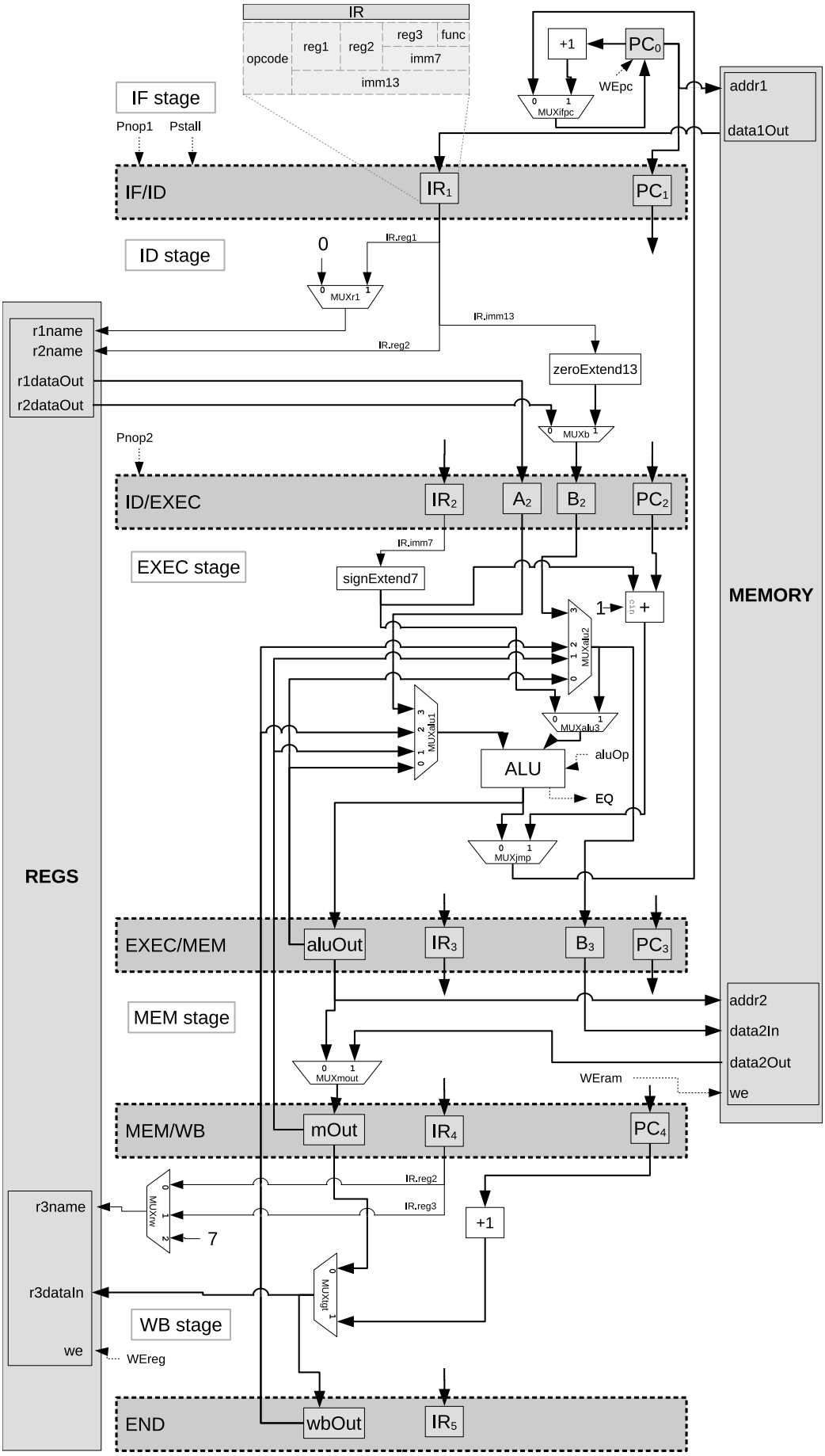
Here we present details of the hardware implementation of the pipelined version of the E20 processor. As in the multicycle version, each instruction executes in more than one clock cycle. In addition, the processor supports executing parts of different instructions at the same time, in different parts of the processor.

5.3.1 Circuit diagram

In the following diagram, thick lines represent 16-bit wires. Thin lines represent wires of less than 16 bits. Dotted lines represent control signals. Control modules are shown separately.

Clocked latch modules are shown in gray, and logic modules are represented by white boxes.

To avoid clutter, lines representing wires between successive pipeline registers are omitted. For example, the ID_1 pipeline register passes its value to the ID_2 pipeline register, even though the middle section of the line between them is hidden.


IR
opcode
reg1
reg2
reg3
func
imm7
imm13
+1
PC0
WEpc
MUXifpc
addr1
data1Out
IF stage
Pnop1
Pstall
IF/ID
IR1
PC1
ID stage
0
IR.reg1
MUXr1
IR.imm13
IR.reg2
r1name
r2name
r1dataOut
r2dataOut
zeroExtend13
MUXb
Pnop2
ID/EXEC
IR2
A2
B2
PC2
IR.imm7
signExtend7
EXEC stage
1
+
MUXalu2
MUXalu1
MUXalu3
ALU
aluOp
EQ
MUXjmp
MEMORY
REGS
EXEC/MEM
aluOut
IR3
B3
PC3
MEM stage
addr2
data2In
data2Out
we
WEram
MUXmout
MEM/WB
mOut
IR4
PC4
IR.reg2
IR.reg3
MUXrw
7
+1
MUXtgt
r3name
r3dataIn
we
WB stage
WEreg
END
wbOut
IR5

5.3.2 Stages

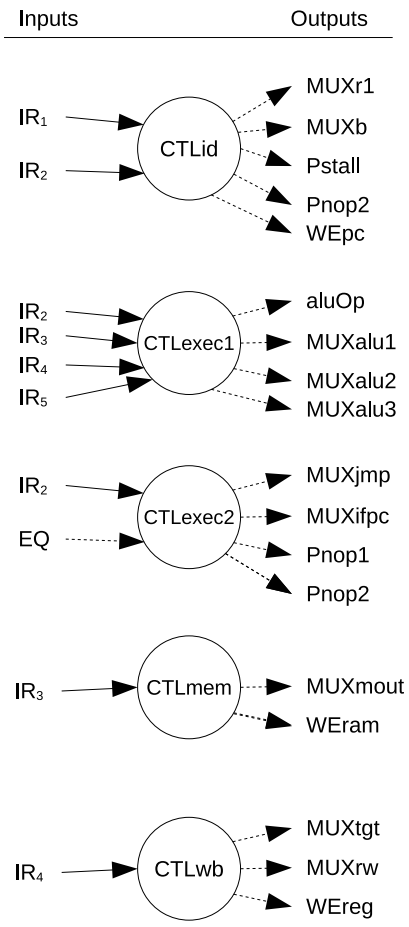
In the circuit diagram, you can see that the processor is divided into areas matching each of the five stages of execution:

1. Each instruction starts in the first stage, instruction fetch (IF), at the top of the diagram. This stage is responsible for reading the instruction from memory. Memory is represented by the tall gray box on the right.
2. Then, each instruction enters the instruction decode (ID) stage, where its register values are read from the register file. The register file is represented by the tall gray box on the left.
3. The third stage is the execute (EXEC) stage, where most instructions have their operands processed by the ALU. Much of the complexity of this stage is due to *forwarding*.
4. Instructions that access memory will do so in the memory (MEM) stage. Other instructions still use this stage, but perform no useful work in it.
5. Finally, in the writeback (WB) stage, the instruction's final result is stored back into the register file. After this stage, each instruction is retired.

Between each stage, intermediary values are stored into *pipeline registers*, indicated by the wide gray boxes with dotted border.

5.3.3 Control modules and wires

The processor has several control modules. Their inputs and outputs are described here.



CTLid implements decoding and stalling. The module examines the instructions in IR_1 and IR_2; if the latter instruction is an `lw` and the former instruction reads a register written by that `lw`, then we must stall. To perform the stall, the module asserts Pstall, which holds the current instruction in the IF/ID pipeline register, preventing an update; disables WEpc, preventing an update to the main program counter; and also asserts Pnop2, which replaces the instruction in the ID/EXEC pipeline register with a `nop`, thus introducing a bubble. If there is no need to stall, then CTLid will set MUXb and MUXr1 for the registers needed by IR_1 and to store the instruction's operands in A_2 and B_2.

CTLexec1 controls inputs of the execution stage. In a simple case, it examines IR_2, sets the appropriate aluOp for that opcode, and passes operands in A_2, B_2, and IR_2.imm7 to the ALU. The module also implements forwarding: if either or both of the register operands read by the instruction in IR_2 is also written by the instruction in IR_3, IR_4, or IR_5, then we must pass that result from aluOut, mOut, or wbOut, respectively, directly to the execution unit, by setting the control of MUXalu1, MUXalu2, and MUXalu3.

CTLexec2 handles jumps and mispredictions. If IR_2 holds an unconditional jump instruction (`j`, `jal`, or `jr`), or a conditional jump instruction (`jeq`) when the two register operands are equal, then the processor must flush the pipeline and prepare to fetch the instruction at the destination address. To flush the instructions presently in the fetch and decode stages, the module asserts Pnop1 and Pnop2, which replace the instructions in the IF/ID and ID/EXEC pipeline registers with a `nop`. To fetch the next instruction, the module sets MUXifpc, which overrides the usual fetch logic of simply incrementing the previous program counter.

CTLmem reads and writes data from the memory unit. If IR_3 contains `sw`, it asserts WEram to write the value in B_3 to the address in aluOut. If IR_3 contains `lw`, it sets MUXmout so that the value of the address in aluOut is written to mOut. In other cases, the value of aluOut is passed through to mOut.

CTLwb writes final values to the register file. For most instructions, the register to be written will be in IR_4.reg2 or IR_4.reg3, and the value to be stored will be in mOut. For `jal`, the value of the program counter plus one will be stored in register `$7`. For `sw` and other jump instructions, no value is written.

The control wires can carry the following concrete values with the indicated meaning:

Control signal	Values	Description
Pnop1	0=no flush 1=flush	Replace IR_1 with `nop`
Pstall	0=no stall 1=stall	Disable write to IR_1 and PC_1
MUXifpc	0=MUXjmp 1=PC_0+1	Select new value of PC_1
WEpc	0=disable 1=enable	Write enable for main program counter
MUXr1	0=literal 0 1=IR_1.reg1	Select register to read
MUXb	0=r2dataOut 1=IR_1.imm13	Select new value of B_2
Pnop2	0=no flush 1=flush	Replace IR_2 with `nop`
MUXalu1	0=aluOut 1=mOut 2=wbOut 3=A_2	Select ALU input
MUXalu2	0=aluOut 1=mOut 2=wbOut 3=B_2	Select ALU input and B_3
MUXalu3	0=IR_2.imm7 1=MUXalu2	Select register value or immediate to ALU
MUXjmp	0=ALU 1=PC_2+IR_2.imm7+1	Select target in case of jump
EQ	0=not equal 1=equal	Result of equality of ALU operands
aluOp	0=add 1=subtract 2=and 3=or 4=slt	Select ALU operation
MUXmout	0=aluOut 1=data2Out	Select new value of mOut
WEram	0=disable 1=enable	Write enable for memory
MUXrw	0=IR_4.reg2 1=IR_4.reg3 2=literal 7	Select register to write
MUXtgt	0=mOut 1=PC_4+1	Select value to write to register
WEreg	0=disable 1=enable	Write enable for registers

5.3.4 Architectural differences

A design goal of all three versions (single-cycle, multicycle, and pipelined) of the E20 processor is to support the same architecture: that is, they should run the same machine code and get the same results. However, the pipelined micro-architecture compels us to compromise in this goal. For example, consider the following program:

```
    movi $1, 1
    sw $0, target($0)   # replace the instruction at target with nop
target:
    add $1, $1, $1      # this instruction should not be run
    halt
```

The above program modifies itself: the `sw` instruction stores the value 0 (corresponding to the `nop` instruction) into the subsequent memory cell, which is then executed. This is a valid E20 program, which, when run on the single-cycle or multicycle versions of the processor, will halt with the value 1 in register `$1`, because the `add` is never run.

However, if we run the same program on the pipelined processor, we get a different result: in that case, the `add` is fetched into the pipeline before the `sw`'s MEM stage overwrites it in memory. The `add` is executed, and the final value of `$1` is 2, not 1. Therefore the behavior of the pipelined E20 differs from earlier versions.

Strictly speaking, the pipelined E20 violates the architectural design that we established with the earlier versions. However, this is a worthwhile sacrifice, thanks to the increased performance of the pipelined version. Furthermore, in practice, most programs do not modify themselves, so this design characteristic is unlikely to "break" many programs.

The alternative would be to have the pipelined processor automatically flush the pipeline in response to self-modification. However, detecting a self-modifying program would add significant complexity to the design.