Harvard Extension School — CS 93: Computer Architecture

Femto-MC Principles of Operation

Name:	McCoy R. Becker
HUID:	11459560
Collaborators:	No collaborators.

1 Block diagram (50 points)

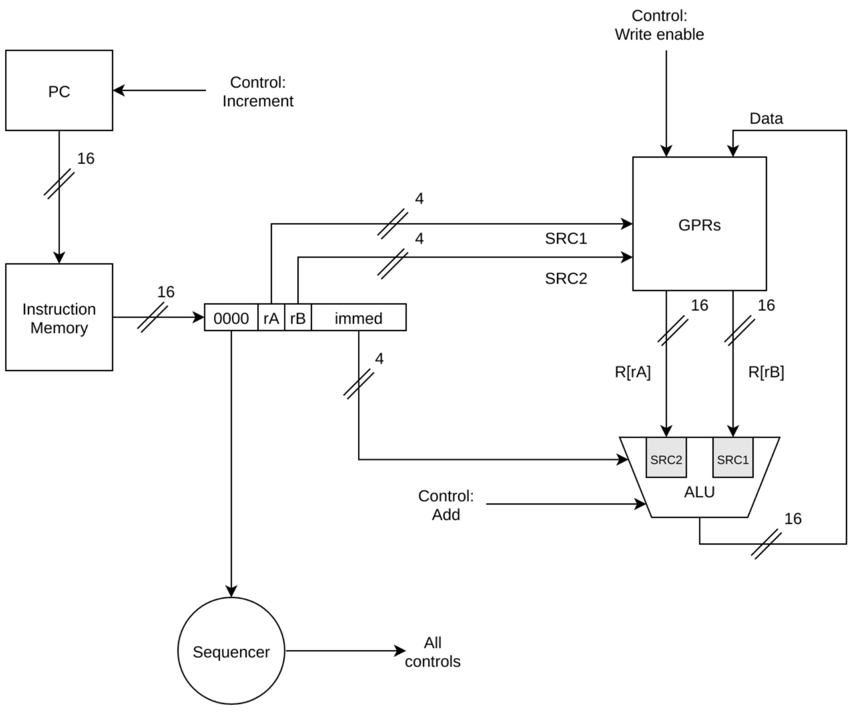
General characteristics The following list summarizes basic characteristics of the architecture.

- The basic word size of the design is 16 bits.
- There are 16 general purpose registers available to the assembly programmer, plus one special register (R0) which (by convention) should always read 0.
- The PC cannot be accessed by the user as a general purpose register.
- There is no special allocation for the stack. The register array is accessed through regular instructions.
- There is no processor status word.
- The instruction set contains shift instructions, which will be implemented at the hardware level by a barrel shifter. This allows multiple bit shifts in either direction.

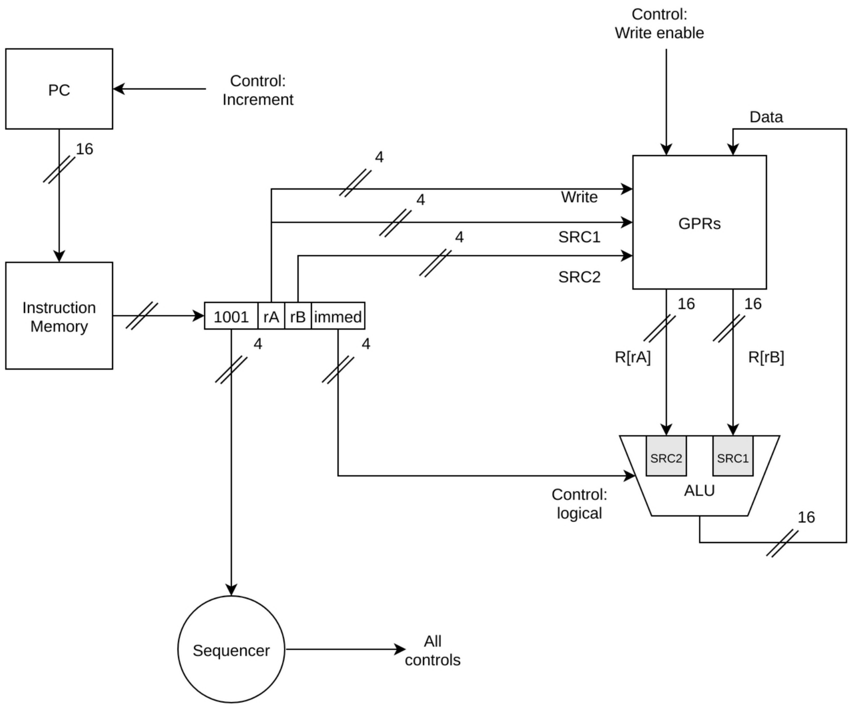
Flow of control The figures in the following section provide flow of control diagrams for each instruction in a sequential implementation of the architecture. There are a set of common elements which deserve a description which appear in each data and control flow diagram.

- The current instruction is loaded from instruction memory into an instruction register.
- Multiplexers (MUX) components are represented by trapezoids. The full data flow framework uses MUX components which select among multiple sets of various bit length inputs.
- Control elements are fed from a sequencer component. The sequencer acts as a decoder from OP code to correct control bits (including ALU function control bits).
- There are 16 general purpose registers (GPRs). These are addressed by rA and rB bit-field components of the current instruction. To accommodate this addressing, each of those components is 4-bits wide.
- The $immed$ bit-field width is either 4 or 8 bits. This is annotated in the diagrams and is interpreted on a per instruction basis.

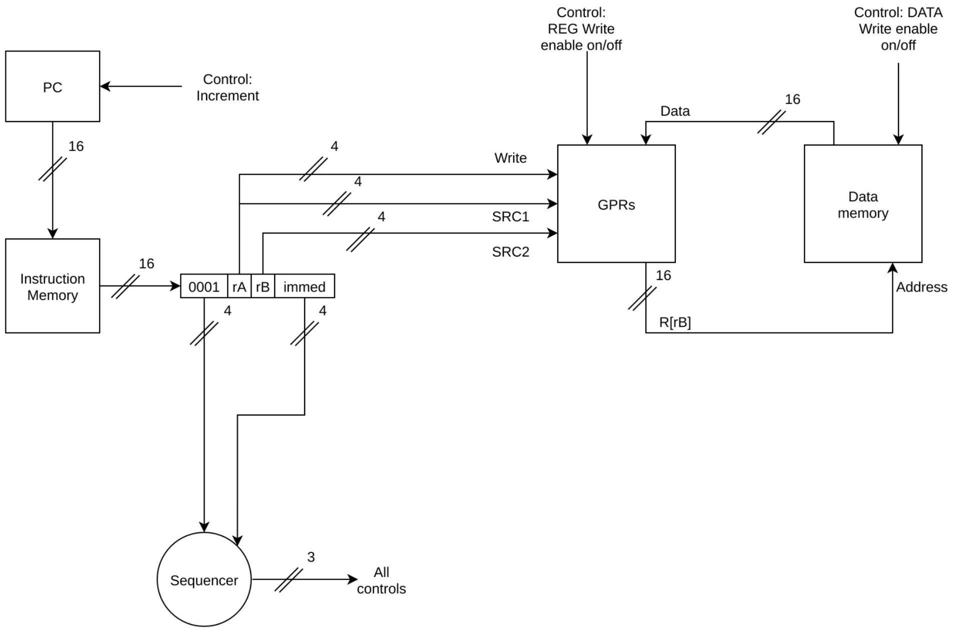
(a) **ADD** (Add values and store)



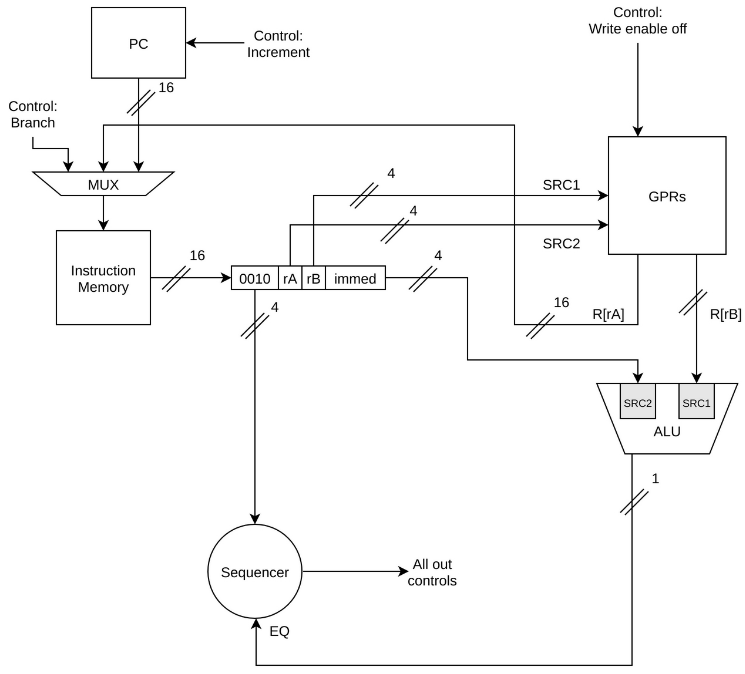
(b) **LOG** (LOG values and store)



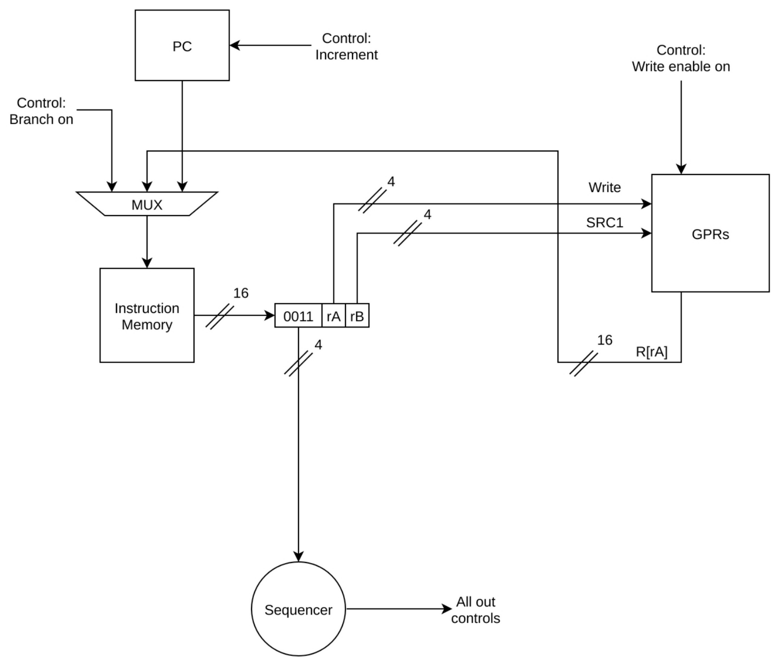
(c) **MEM** (MEM values and store)



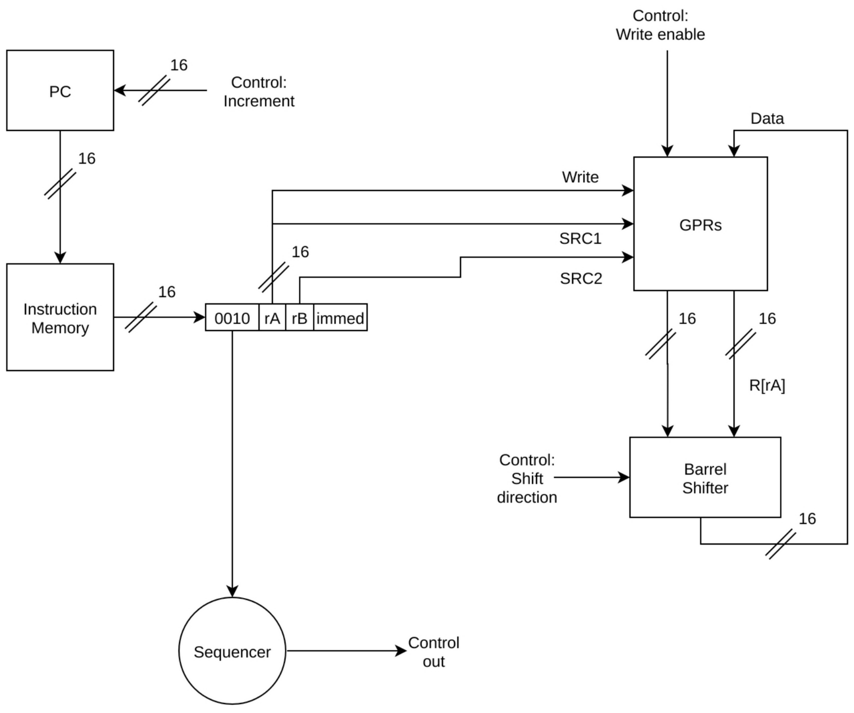
(d) **BEQ** (Branch if equal)



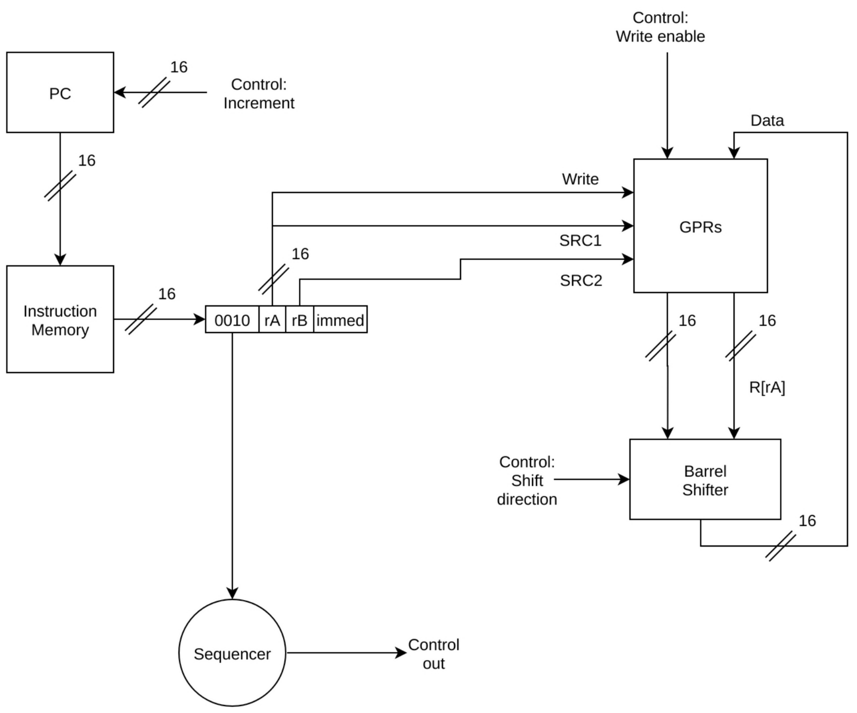
(e) **JUMP** (Unconditional jump)



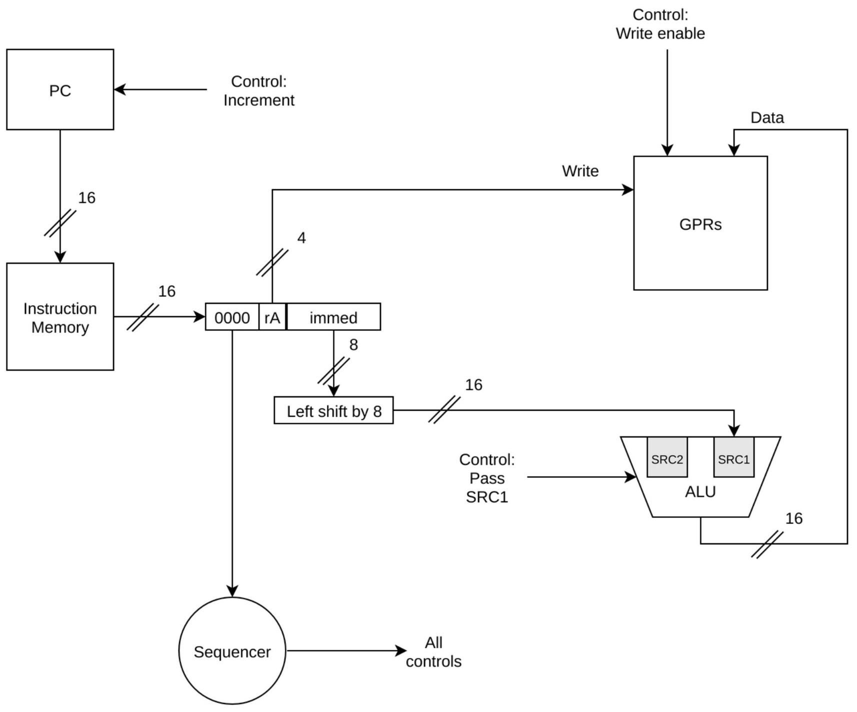
(f) **SHIFTL** (Shift left)



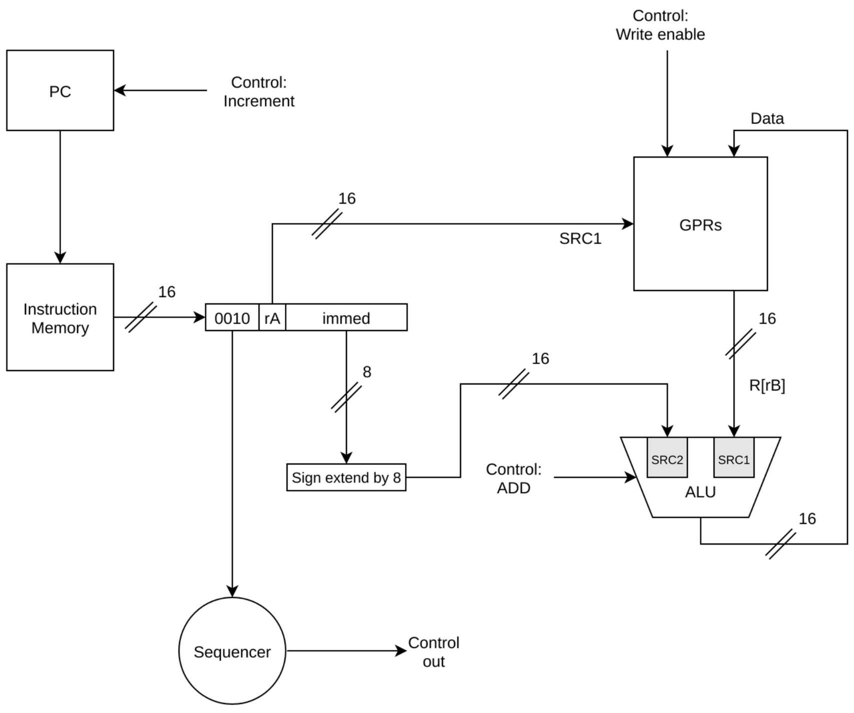
(g) **SHIFTR** (Shift right)



(h) **LUI** (Load upper immediate)



(i) **ADDI** (Add immediate and store)



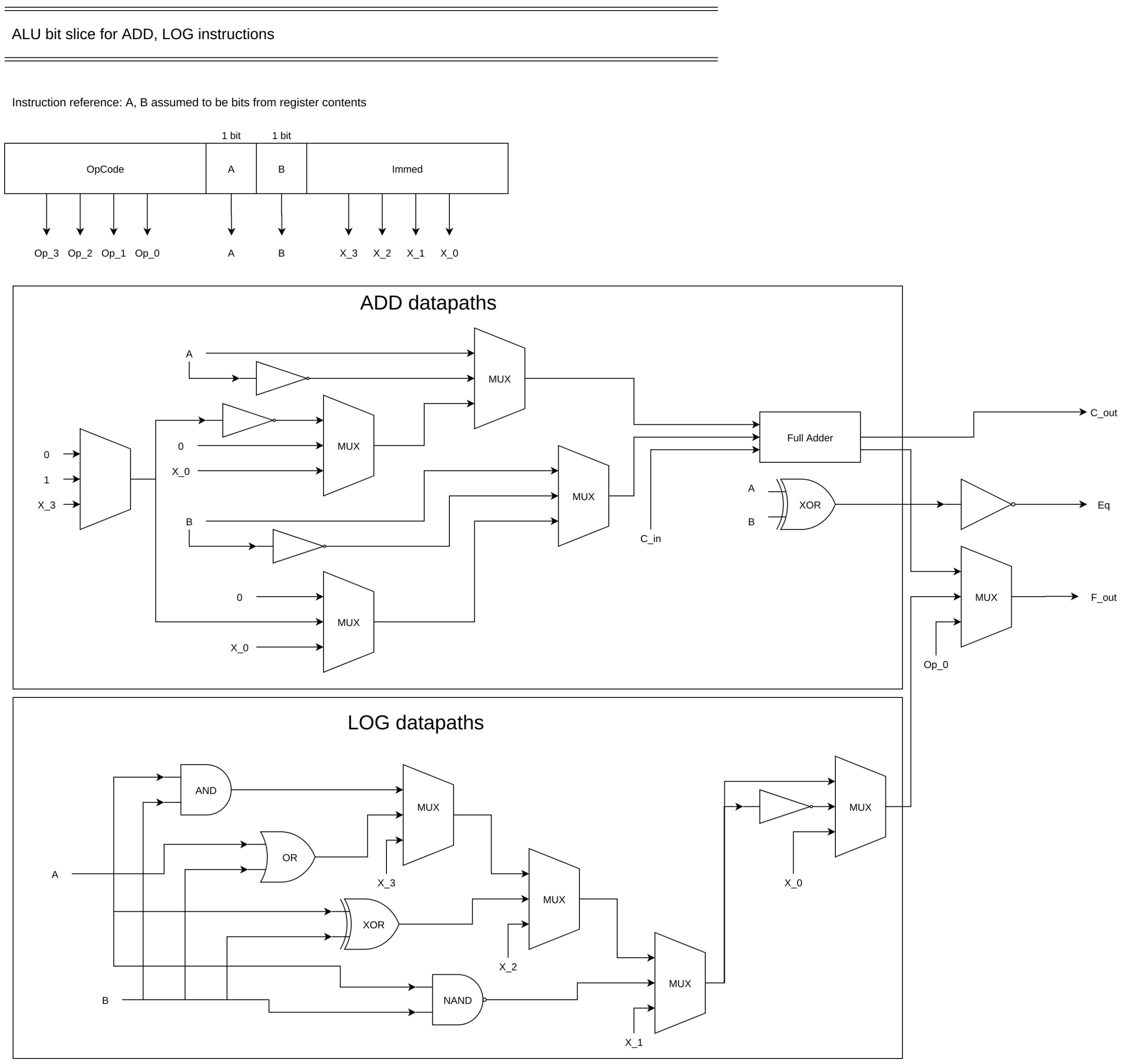


Figure 1: ADD and LOG datapaths in an ALU bitslice implementation for Femto-MC. These datapaths also include an auxiliary bit which allows the ALU to automatically compute if the two inputs are equal, to be used in BEQ instructions.

The full dataflow diagram is shown above. This dataflow diagram is a concatenation of the above diagrams, with the inclusion of control MUX components which select which data is asserted for particular instructions.

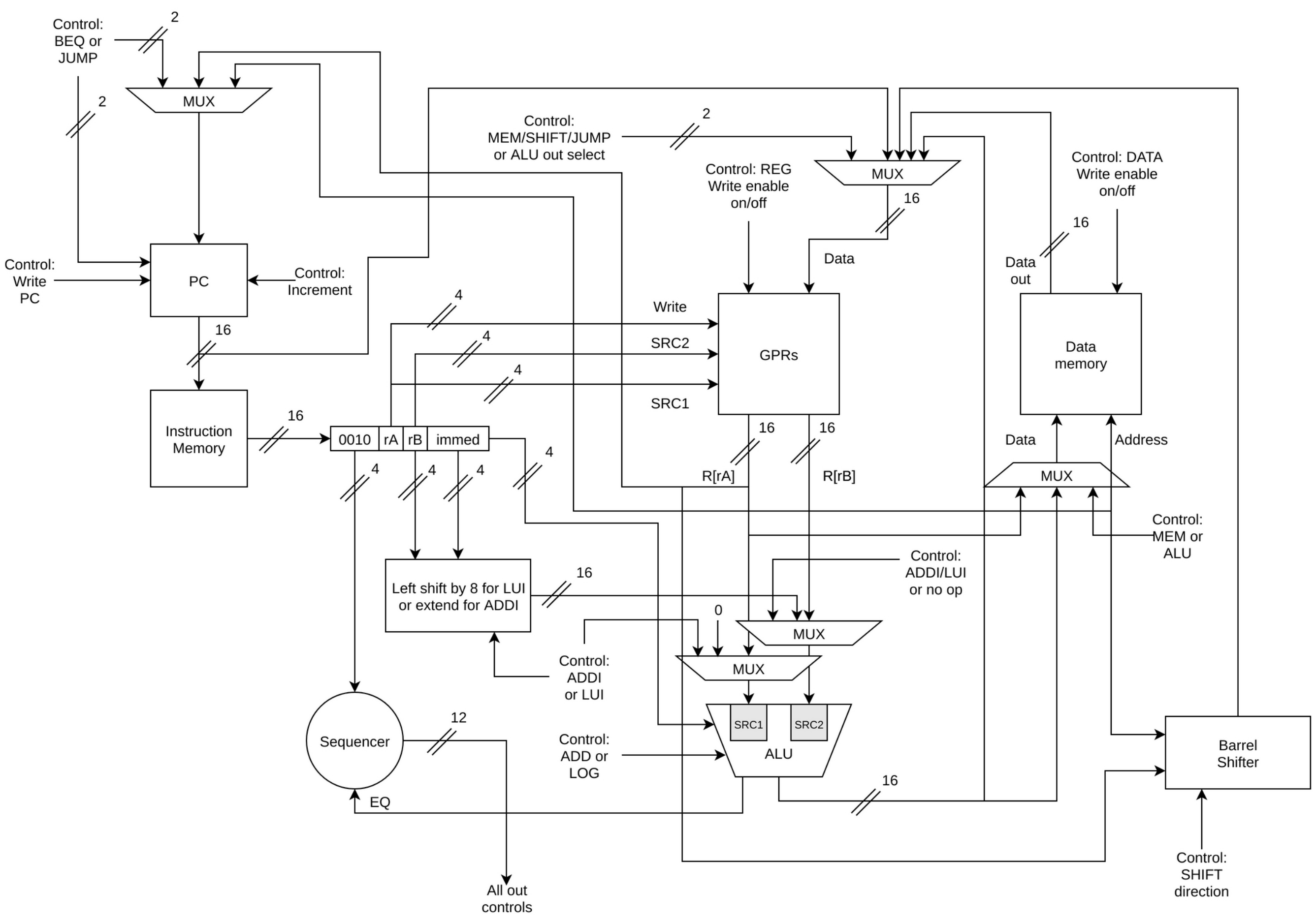


Figure 2: The full flow of control and data in a sequential implementation of Femto-MC. The addition of control to MUX components allows each of the data paths shown above to be encapsulated in a single framework.

2 Instruction set (100 points)

In the following, I describe my instruction set (called Femto-MC). Femto-MC consists of three atomic instruction types: 2 register RR instructions, 2 register RRI instructions, and single register RI instructions. These are shown below.

Figure 3: 2 register RR-type instructions ($X_3X_2X_1X_0$ is specified by a constant $immed$)

instruction	4 bits	4 bits	4 bits	4 bits
ADD	1000	reg A	reg B	$X_3X_2X_1X_0$
LOG	1001	reg A	reg B	$X_3X_2X_1X_0$
MEM	0001	reg A	reg B	$X_3X_2X_1X_0$

Figure 4: 2 register RRI-type instructions

instruction	4 bits	4 bits	4 bits	4 bits
BEQ	0010	reg A	reg B	immed
JUMP	0011	reg A	reg B	unused
SHIFTL	0100	reg A	reg B	immed
SHIFTR	0101	reg A	reg B	immed

(a) ADD regA, regB, immed

ADD is a configurable arithmetic instruction which takes in two register operands and 4 function bits in $immed$. Depending on the function bits, the operation computes different arithmetic functions on the register operands.

- 0000: normal signed integer addition.
- 0001: negate the first operand, then perform integer addition.
- 0011: negative and increment the first operand, then perform integer addition.
- 1001: negate the second operand, then perform integer addition.
- 1011: negate and increment the second operand, then perform integer addition.

This operation allows the expression of the following functions for register contents A and B

- $A + B$
- $(\bar{A}) + B$
- $(\bar{A} + 1) + B$
- $A + (\bar{B})$
- $A + (\bar{B} + 1)$

This allows the user to expression addition, negation, subtraction using the same instruction.

(b) LOG regA, regB, immed

LOG is a configurable logical instruction which takes in two register operands and 4 function bits in $immed$. Depending on the function bits, the operation computes different logical functions on the register operands.

- 000Z: AND the two operands.
- 100Z: OR the two operands.
- X10Z: XOR the two operands.
- XX1Z: NAND the two operands.

The least significant bit (here, Z), when asserted, negates the result.

(c) MEM regA, regB, immed

MEM is a configurable memory load and save instruction which allows word and byte addressing for loads and saves. The semantics of MEM are as follows: $regB$ holds the location in memory which $regA$ will either load or save to. The $immed$ bits determine the functionality of the instruction:

- 0000: load word from memory at address stored in $regB$ into $regA$.
- 0001: save word from $regA$ into memory at address stored in $regB$.

This instruction allows addressing of up to 64 kB of word addressable memory.

Note on endianess: the memory system is assumed to be little endian.

(d) BEQ regA, regB, immed

BEQ is a conditional branch instruction with the following semantics: if the contents of $regB$ are equal to $immed$, the program counter register branches directly to the location stored in $regA$.

This instruction can be used to implement conditional logic. For example, if $immed$ is set to 0, the BEQ instruction will branch only if none of the bits in $regB$ are asserted. This can be used to emulate an unconditional JUMP instruction, with the caveat that the current program counter register contents PC are not incremented and then stored in $regA$.

(e) JUMP regA, regB, immed

JUMP is an unconditional jump instruction. This instruction ignores the $immed$ bits. The program counter contents PC are incremented $PC + 1$ and stored in $regA$, then the location stored as the contents of $regB$ are loaded into the program counter register.

(f) SHIFTL regA, regB

SHIFTL is a left shift instruction. It shifts the contents of $regA$ to the left by the contents of $regB$, then stores the result of that shift back in $regA$. This is a non-destructive operation (bits are rotated around the word).

(g) SHIFTR regA, regB

SHIFTR is a right shift instruction. It shifts the contents of $regA$ to the right by the contents of $regB$, then stores the result of that shift back in $regA$. This is a non-destructive operation (bits are rotated around the word).

(h) LUI regA, immed

LUI is a load upper immediate instruction. It takes the contents of $immed$, shifts the contents to the left by 8 bits and then stores the result in $regA$. This has the effect of storing any 8 bits specified by $immed$ in the most significant byte of $regA$.

(i) ADDI regA, immed

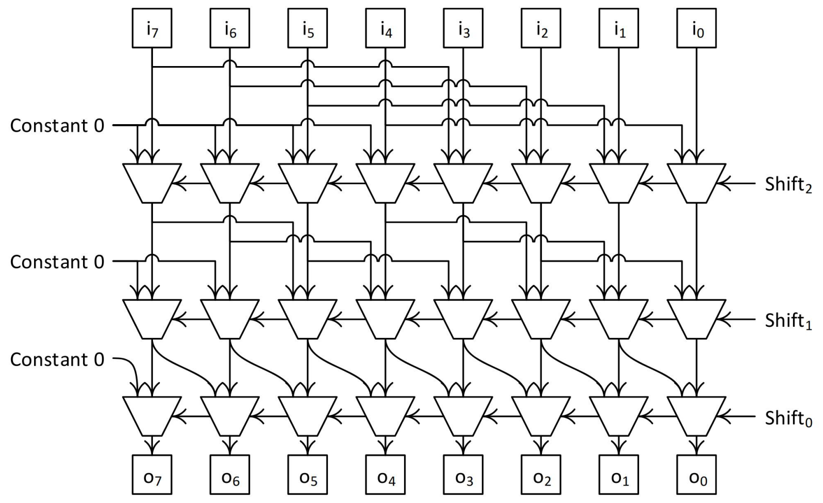
ADDI is an add immediate instruction. It takes the contents of $immed$ and adds them to the contents of $regA$, storing the result back in $regA$. ADDI can be used as a simple incrementer, or an adder for numbers up to 256 (using the 8 bits of $immed$).

ADDI is most useful when following an LUI instruction. The combination of LUI and ADDI allows storing any number up to 2^16 in any register. This allows memory accesses for the entire memory address space of 64 kB.

Summary table The summary table for the assembly language for this instruction set is given below. The assembly language programmer can utilize addressing to 16 general purpose registers. In the following summary table, I use the de-reference operator $*$ to refer to the word stored in any particular register.

Figure 6: The Femto-MC 16-bit assembly language.

Assembly code syntax	Opcode	Description
add regA, regB, immed	1000	Add $*regA$ and $*regB$ and store the result in $regA$. The optional function bit modifiers specified in $immed = X_3X_2X_1X_0$ provide the following modifications to the instruction. X_3 selects which register output to modify (0 goes A, 1 goes B). X_0 negates the modified register output. X_1 increments the modified register output. Negation is performed before incrementing.
log regA, regB, immed	1001	Apply a logical operation to operands $*regA$ and $*regB$ and store the result in $regA$. The function bit modifiers are used to select which logical operation to perform.
mem regA, regB, immed	0001	Load/save the word/byte at memory address $Memory[*regB]$ into $regA$. The functionality is determined by the function bits $X_4X_3X_2X_1$. 0000 is load word, 0001 is save word.
beq regA, regB, immed	0010	Branch to $PC+1+*regA$ if $*regB$ is equal to $immed$. This can be used to assert conditions (i.e. $immed$ might most often be 1 or 0 to take advantage of the functionality of the ALU to return whether or not two operands are equal).
jump regA, regB	0011	Jump immediately to $PC + 1 + *regB$ and store $PC + 1$ in $regA$.
shiftl regA, regB, immed	0100	Store the result of a constant shift to the left by $*regB$ of $*regA$ in $regA$.
shiftr regA, regB, immed	0101	Store the result of a constant shift to the right by $*regB$ of $*regA$ in $regA$.
lui regA, immed	0000	Load the constant specified by $immed$ into the least significant 8 bits of $regA$ and then left shift $*regA$ by 8.
addi regA, immed	1010	Add $immed$ to $*regA$ and store the result in $regA$.



Special feature My implemented special feature is barrel shifter which implements arbitrary circular shifts in either direction. The implementation of this feature followings the logarithmic-time shifting scheme from the course documents (where a single direction shifter is implemented). I've included the figure from the course documents here for reference. The barrel shifter consists of two of this unidirectional shifters with a selector MUX to determine which direction. Shifting is non-destructive - bits are not destroyed if shifted past a word boundary.