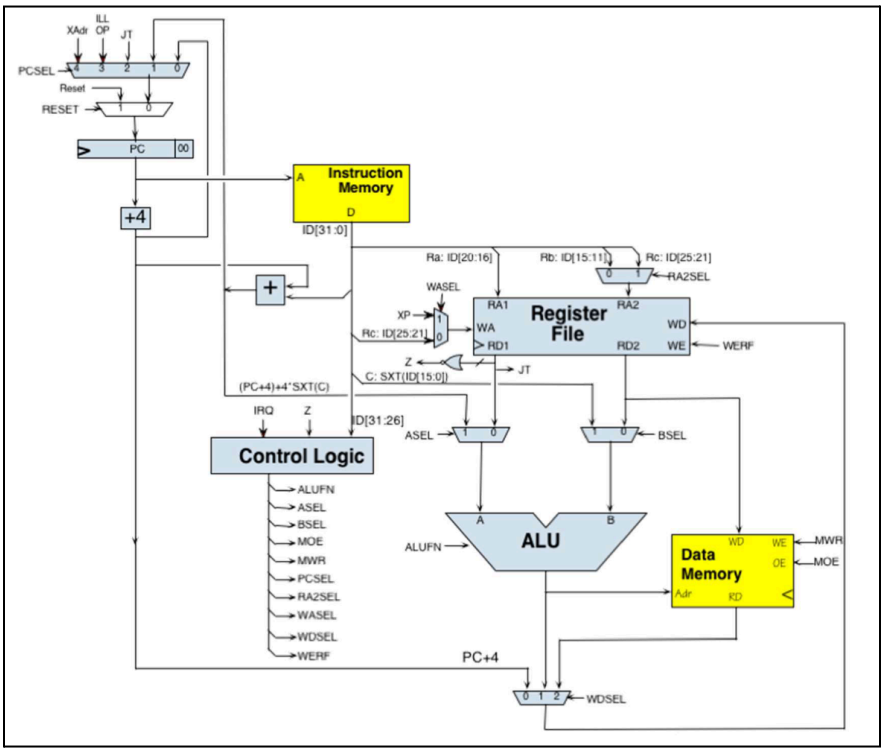
Building The Beta: A Reference Manual

This reference guide provides a simplified overview of the various components of the Beta architecture and how the control signals affect the operation of these components. This guide also provides a listing of the available Beta Assembly instructions. For a more detailed explanation of how each individual component works, refer to the "Building a Simple Processor: The Beta" lecture and other prior lectures as necessary.

Table of Contents

Building The Beta



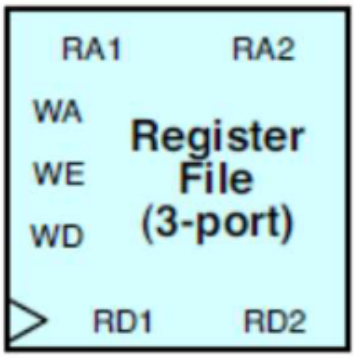
Source: MIT OpenCourseWare 6.004 Spring 2017

The Beta architecture is a 32-bit architecture that is a simplified version of modern CPU architectures, such as the x86 architecture or the more recent ARM and RISC-V architectures, that can handle a limited number of operations. It can handle various arithmetic and logical operations, such as addition, subtraction, binary shifting, value comparisons. In addition, the architecture supports various special instructions that are commonly found in other Assembly languages, including unconditional branching, conditional branching, as well as reading from and writing to memory. This functionality is provided by various components working together as one unit. Each core component is described in more detail below.

The Components

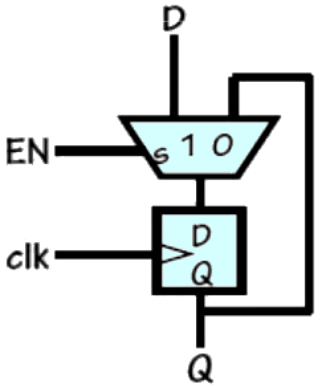
Register File

The register file contains 32, 32-bit registers, ranging from R0 to R31, and is able to store a limited amount of data that can be accessed quickly compared to the larger amount of data in memory that takes longer to access.



For reading values from the register file, the inputted values in Read Address 1 (RA1) and Read Address 2 (RA2) serve as the “select input” for a multiplexer to determine which register’s data should be accessed and outputted through Register Data 1 (RD1) and Register Data 2 (RD2) respectively.

When writing to the register file, the Write Address (WA) input will indicate the desired register to write to, where the Write Data (WD) input represents the data to be written to the specified register. The write is only executed if Write Enable (WE) is enabled (i.e. set to 1).



See the schematic to the right. If EN (aka Write Enable) is set to 1, the value in D (aka Write Data) is loaded into the register. Otherwise, the existing value in the register is kept.

A brief description of each input/output port of the register file is provided below:

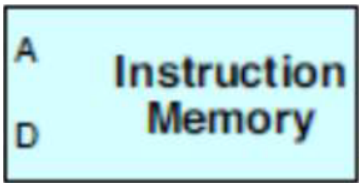
- Inputs
    - RA1 (Read Address 1): Specifies a register to read from
        - Typically some Ra register input from an instruction (bits 16-20)
    - RA2 (Read Address 2): Specifies a second register to read from.
        - Typically some Rb or Rc register input from an instruction (bits 11-15 or 21-25 respectively)
    - WA (Write Address): Specifies a register to write to.
    - WD (Write Data): Specifies the data to write to a register.
    - WE (Write Enable): Indicates whether or not to write to a register.
- Outputs
    - RD1 (Read Data 1): The data fetched from the register specified in RA1.
    - RD2 (Read Data 2): The data fetched from the register specified in RA2.

ALU

The ALU, also known as the arithmetic logic unit, is utilized in processing a large number of the available instructions in the Beta Assembly language. It can handle basic arithmetic, bitwise boolean, bitwise comparison, and bitwise shifting operations. The inputs and output are summarized below:

- Inputs
  - A: One of the 32-bit data inputs to be processed.
  - B: The other 32-bit data input to be processed.
  - ALUFN: A 6-bit control signal to select the ALU operation to perform.
- Outputs
  - (Unlabeled Output): The 32-bit output of the specified ALU operation.

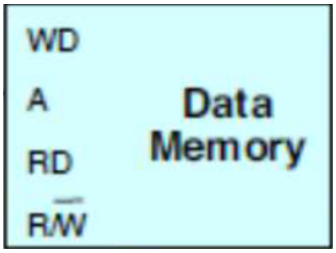
Instruction Memory



The Instruction Memory component serves as an access point into main memory to fetch instructions for the Beta system to perform. Its input and output are described below:

- Inputs
  - A (Address): The address of the next instruction in memory.
- Outputs
  - D (Data): The data of an instruction, such as opcodes, register addresses, and literals. Typically referred to as ID[31:0] to indicate 32 bits of data.

Data Memory



Similar to the Instruction Memory component, the Data Memory component serves as an access point into main memory, albeit not for instructions. Unlike the instruction memory component however, data can be written via the Data Memory component. As such, it is heavily used for Load and Store operations.

- Inputs
  - WD (Write Data): Data to be written to some location in memory.
  - A (Address): The address in memory to read from or write to
- Outputs
  - RD (Read Data): The data fetched from the specified address in memory.
  - R/W (Read/Write): Determines whether to read from or write to memory.

Control Signals

You may have noticed from the overall Beta implementation that there are some additional signals that have been added in addition to the standalone components to help the components communicate with each other. All of the Beta control signals, whether or not they were discussed in the previous section, are described below:

ALUFN: ALU Function

- Purpose: Determines which operation the ALU should perform
- Where It's Used: ALU Component
- Values: See Opcode Table in Practice Exams

ASEL: A Input Selection

- Purpose: Determines whether to use the RD1 register value or some result of the offset adder as the 'A' input to the ALU
- Where It's Used: Just before the A input to the ALU
- Values:
    - 0: Use RD1 as the A Input
    - 1: Use the result of the offset adder as the A Input
        - Result = (PC+4)+4*SXT(C)
        - In other words, take the current value of the PC, add 4, then add four times the sign-extended constant C (bits 0-15 in an instruction)

BSEL: B Input Selection

- Purpose: Determines whether to use the RD2 register value or some literal as the 'B' input to the ALU
- Where It's Used: Just before the B input to the ALU
- Values
    - 0: Use the value in RD2
    - 1: Use the sign-extended literal C (bits 0-15 in an instruction)

MOE: Memory Output Enable

- Purpose: Specifies whether or not to read from memory via Data Memory
- Where It's Used: Data Memory Component
- Values
    - 0: Data should not be read from memory
    - 1: Data is to be read from memory

MWR: Memory Write Enable

- Purpose: Specifies whether or not to write to memory via Data Memory.
- Where It's Used: Data Memory Component
- Values
    - 0: Data should not be written to memory
    - 1: Data is to be written to memory

PCSEL: PC Selection

- Purpose: Determines the next value of the PC register
- Where It's Used: Just before the PC register (and RESET multiplexer)
- Values:
    - 0: PC+4
        - The current value of PC, then add 4
    - 1: (PC+4)+4*SXT(C)
        - The result of the offset adder. See ASEL for more info
    - 2: JT
        - The value of RD1, typically used with JMP instructions
    - 3: ILLOP
        - A predefined value for illegal operations, usually 0x4 for Beta
    - 4: XAdr
        - A predefined value for interrupts, usually 0x8 for Beta

RA2SEL: RA2 Selection

- Purpose: Determines whether to use the Rb or Rc inputs from the current instruction as the RA2 input in the Register File.
- Where It's Used: Just before the Register File
- Values:
    - 0: Use some input Rb (bits 11-15 in an instruction)
    - 1: Use some input Rc (bits 21-25 in an instruction)

WASEL: Write Address Selection

- Purpose: Determines whether to use the special XP (aka exception pointer) register or the Rc register input as the WA input in the Register File.
- Where It's Used: Just before the Register File
- Values:
    - 0: Use some input Rc (bits 21-25 in an instruction)
    - 1: The value of the XP register (aka exception pointer)

WDSEL: Write Data Selection

- Purpose: Determines which data to use as the WD input in the Register File.
- Where It's Used: After the ALU and Data Memory, Before Looping Back to the Register File
- Values:
    - 0: PC+4
        - The current value of the PC register plus 4.
    - 1: The result of the ALU operation
    - 2: Value in RD
        - The value retrieved from Data Memory

WERF: Write Enable Register File

- Purpose: Determines whether or not to write to the Register File
- Where It's Used: Register File
- Values:
    - 0: Data should not be written to the Register File
    - 1: Data is to be written to the Register File

Control Logic Inputs

- Z: Tests if the bits from RD1 are all 0s. If so, Z is set to 1. If not, Z is set to 0.
- IRQ: Set to 1 if an interrupt request is received, and 0 if no interrupt is present.
- ID[31:26]: Instruction Data bits 26-31, specifies the opcode of the instruction to execute.

ISA Manual

With all of this hardware, it's time to utilize its potential with the Beta ISA, containing a variety of instructions that can be executed on a simple computer. Many of these instructions have some equivalent on modern architectures as well.

Arithmetic / Logical Operations Without Literal

ID[31:26]	ID[25:21]	ID[20:16]	ID[15:11]	ID[10:0]
Opcode	Rc	Ra	Rb	Unused

ADD

- Usage: ADD(Ra,Rb,Rc)
- Opcode: 100000
- Operation:
    - PC + 4 → PC
    - Reg[Ra] + Reg[Rb] → Reg[Rc]
- Description: The summation of the contents of registers Ra and Rb are written to register Rc. Carries and overflows are not handled, but can be computed after.

AND

- Usage: AND(Ra,Rb,Rc)
- Opcode: 101000
- Operation:
    - PC + 4 → PC
    - Reg[Ra] & Reg[Rb] → Reg[Rc]
- Description: The bitwise AND of the contents of registers Ra and Rb are written to register Rc.

CMPEQ

- Usage: CMPEQ(Ra,Rb,Rc)
- Opcode: 100100
- Operation:
    - PC + 4 → PC
    - If Reg[Ra] = Reg[Rb], then 1 → Reg[Rc]. Else, 0 → Reg[Rc]
- Description: If the contents of registers Ra and Rb are equal, the value 1 is written to register Rc. Otherwise, the value 0 is written to register Rc.

CMPLE

- Usage: CMPLE(Ra,Rb,Rc)
- Opcode: 100110
- Operation:
    - PC + 4 → PC
    - If Reg[Ra] ≤ Reg[Rb], then 1 → Reg[Rc]. Else, 0 → Reg[Rc]
- Description: If the contents of register Ra is less than or equal to the contents of register Rb, 1 is written to register Rc. Otherwise, 0 is written to register Rc.

CMPLT

- Usage: CMPLT(Ra,Rb,Rc)
- Opcode: 100101
- Operation:
    - PC + 4 → PC
    - If Reg[Ra] < Reg[Rb], then 1 → Reg[Rc]. Else, 0 → Reg[Rc]
- Description: If the contents of register Ra are less than the contents of register Rb, 1 is written to register Rc. Otherwise, 0 is written to register Rc.

DIV

- Usage: DIV(Ra,Rb,Rc)
- Opcode: 100011
- Operation:
    - PC + 4 → PC
    - Reg[Ra] / Reg[Rb] → Reg[Rc]
- Description: The quotient from the division of the contents of register Ra by the contents of register Rb are written to register Rc. Only using the low-order 32 bits.

MUL

- Usage: MUL(Ra,Rb,Rc)
- Opcode: 100010
- Operation:
    - PC + 4 → PC
    - Reg[Ra] * Reg[Rb] → Reg[Rc]
- Description: The product of the multiplication of the contents of registers Ra and Rb are written to register Rc.

OR

- Usage: OR(Ra,Rb,Rc)
- Opcode: 101001
- Operation:
    - PC + 4 → PC
    - Reg[Ra] | Reg[Rb] → Reg[Rc]
- Description: The bitwise OR of the contents of registers Ra and Rb are written to register Rc.

SHL

- Usage: SHL(Ra,Rb,Rc)
- Opcode: 101100
- Operation:
    - PC + 4 → PC
    - Reg[Ra] << $\text{Reg[Rb]}_{4:0}$ → Reg[Rc]
- Description: The result of left-shifting the contents of register Ra by the amount specified by the five low-order bits in register Rb is written to Rc. Zeros are added to the vacant bit positions.

SHR

- Usage: SHR(Ra,Rb,Rc)
- Opcode: 101101
- Operation:
    - PC + 4 → PC
    - Reg[Ra] >> $\text{Reg[Rb]}_{4:0}$ → Reg[Rc]
- Description: The result of right-shifting the contents of register Ra by the amount specified by the five low-order bits in register Rb is written to Rc. Zeros are added to the vacant bit positions.

SRA

- Usage: SRA(Ra,Rb,Rc)
- Opcode: 101110
- Operation:
    - PC + 4 → PC
    - Reg[Ra] >> $\text{Reg[Rb]}_{4:0}$ → Reg[Rc]
- Description: The result of an arithmetic right-shift of the contents of register Ra by the amount specified by the five low-order bits in register Rb is written to Rc. The sign bit (Reg[Ra]_{31}) is added to the vacant bit positions.

SUB

- Usage: SUB(Ra,Rb,Rc)
- Opcode: 100001
- Operation:
    - PC + 4 → PC
    - Reg[Ra] - Reg[Rb] → Reg[Rc]
- Description: The subtraction of the contents of register Rb from the contents of register Ra are written to register Rc. Borrows and overflows are not handled, but can be computed after.

XOR

- Usage: XOR(Ra,Rb,Rc)
- Opcode: 101010
- Operation:
    - PC + 4 → PC
    - Reg[Ra] ^ Reg[Rb] → Reg[Rc]
- Description: The bitwise XOR of the contents of registers Ra and Rb are written to register Rc.

XNOR

- Usage: XNOR(Ra,Rb,Rc)
- Opcode: 101011
- Operation:
    - PC + 4 → PC
    - ~(Reg[Ra] ^ Reg[Rb]) → Reg[Rc]
- Description: The bitwise XNOR of the contents of registers Ra and Rb are written to register Rc.

Arithmetic / Logical Operations With Literal

ID[31:26]	ID[25:21]	ID[20:16]	ID[15:0]
Opcode	Rc	Ra	Literal (Two's Complement)

ADDC

- Usage: ADDC(Ra,literal,Rc)
- Opcode: 110000
- Operation:
    - PC + 4 → PC
    - Reg[Ra] + SEXT(literal) → Reg[Rc]
- Description: The summation of the contents of register Ra and *literal* are written to register Rc. Carries and overflows are not handled, but can be computed after.

ANDC

- Usage: ANDC(Ra,literal,Rc)
- Opcode: 111000
- Operation:
    - PC + 4 → PC
    - Reg[Ra] & SEXT(literal) → Reg[Rc]
- Description: The bitwise AND of the contents of register Ra and *literal* are written to register Rc.

CMPEQC

- Usage: CMPEQ(Ra,literal,Rc)
- Opcode: 110100
- Operation:
    - PC + 4 → PC
    - If Reg[Ra] =SEXT(literal), then 1 → Reg[Rc]. Else, 0 → Reg[Rc]
- Description: If the contents of register Ra and *literal* are equal, 1 is written to register Rc. Otherwise, 0 is written to register Rc.

CMPLEC

- Usage: CMPLEC(Ra,literal,Rc)
- Opcode: 110110
- Operation:
    - PC + 4 → PC
    - If Reg[Ra] ≤ SEXT(literal), then 1 → Reg[Rc]. Else, 0 → Reg[Rc]
- Description: If the contents of register Ra is less than or equal to *literal*, 1 is written to register Rc. Otherwise, 0 is written to register Rc.

CMPLTC

- Usage: CMPLTC(Ra,literal,Rc)
- Opcode: 110101
- Operation:
    - PC + 4 → PC
    - If Reg[Ra] < SEXT(literal), then 1 → Reg[Rc]. Else, 0 → Reg[Rc]
- Description: If the contents of register Ra are less than *literal*, 1 is written to register Rc. Otherwise, 0 is written to register Rc.

DIVC

- Usage: DIVC(Ra,literal,Rc)
- Opcode: 110011
- Operation:
    - PC + 4 → PC
    - Reg[Ra] / SEXT(literal) → Reg[Rc]
- Description: The quotient from the division of the contents of register Ra by *literal* are written to register Rc. Only using the low-order 32 bits.

MULC

- Usage: MULC(Ra,literal,Rc)
- Opcode: 110010
- Operation:
    - PC + 4 → PC
    - Reg[Ra] * SEXT(literal) → Reg[Rc]
- Description: The product of the multiplication of the contents of register Ra and *literal* are written to register Rc.

ORC

- Usage: ORC(Ra,literal,Rc)
- Opcode: 111001
- Operation:
    - PC + 4 → PC
    - Reg[Ra] | SEXT(literal) → Reg[Rc]
- Description: The bitwise OR of the contents of register Ra and *literal* are written to register Rc.

SHLC

- Usage: SHLC(Ra,literal,Rc)
- Opcode: 111100
- Operation:
    - PC + 4 → PC
    - Reg[Ra] << $\text{literal}_{4:0}$ → Reg[Rc]
- Description: The result of left-shifting the contents of register Ra by the amount specified by the five low-order bits in *literal* is written to Rc. Zeros are added to the vacant bit positions.

SHRC

- Usage: SHRC(Ra,literal,Rc)
- Opcode: 111101
- Operation:
    - PC + 4 → PC
    - Reg[Ra] >> $\text{literal}_{4:0}$ → Reg[Rc]
- Description: The result of right-shifting the contents of register Ra by the amount specified by the five low-order bits in *literal* is written to Rc. Zeros are added to the vacant bit positions.

SRAC

- Usage: SRAC(Ra,literal,Rc)
- Opcode: 111110
- Operation:
    - PC + 4 → PC
    - Reg[Ra] >> $\text{literal}_{4:0}$ → Reg[Rc]
- Description: The result of an arithmetic right-shift of the contents of register Ra by the amount specified by the five low-order bits in *literal* is written to Rc. The sign bit (Reg[Ra]_{31}) is added to the vacant bit positions.

SUBC

- Usage: SUBC(Ra,literal,Rc)
- Opcode: 110001
- Operation:
    - PC + 4 → PC
    - Reg[Ra] - SEXT(literal) → Reg[Rc]
- Description: The subtraction of *literal* from the contents of register Ra are written to register Rc. Borrows and overflows are not handled, but can be computed after.

XORC

- Usage: XORC(Ra,literal,Rc)
- Opcode: 111010
- Operation:
    - PC + 4 → PC
    - Reg[Ra] ^ SEXT(literal) → Reg[Rc]
- Description: The bitwise XOR of the contents of register Ra and *literal* are written to register Rc.

XNORC

- Usage: XNORC(Ra,literal,Rc)
- Opcode: 111011
- Operation:
    - PC + 4 → PC
    - ~(Reg[Ra] ^ SEXT(literal)) → Reg[Rc]
- Description: The bitwise XNOR of the contents of register Ra and *literal* are written to register Rc.

Special Instructions

ID[31:26]	ID[25:21]	ID[20:16]	ID[15:0]
Opcode	Rc	Ra	Literal (Two's Complement)

BEQ / BF

- Usage: BEQ(Ra,label,Rc) or BF(Ra,label,Rc)
- Opcode: 011100
- Operation:
    - literal = ((OFFSET(label) - OFFSET(current instruction)) / 4) - 1
    - PC + 4 → PC
    - PC + 4 * SEXT(literal) → EA (effective address)
    - Reg[Ra] → TEMP
    - Reg[Rc] → PC
    - If TEMP = 0, then EA → PC
- Description: The address of the instruction after the BEQ or BF instruction is written to register Rc. If the contents of register Ra are zero, the PC is loaded with the target address. Otherwise, execution continues normally without branching. Note that the displacement *literal* is considered to be a signed word offset, meaning it is multiplied by 4 to convert it to a byte offset, sign extended to 32 bits, then added to the updated PC to form the target address.

BNE / BT

- Usage: BNE(Ra,label,Rc) or BT(Ra,label,Rc)
- Opcode: 011101
- Operation:
    - literal = ((OFFSET(label) - OFFSET(current instruction)) / 4) - 1
    - PC + 4 → PC
    - PC + 4 * SEXT(literal) → EA (effective address)
    - Reg[Ra] → TEMP
    - Reg[Rc] → PC
    - If TEMP ≠ 0, then EA → PC
- Description: The address of the instruction after the BNE or BT instruction is written to register Rc. If the contents of register Ra are non-zero, the PC is loaded with the target address. Otherwise, execution continues normally without branching. Note that the displacement *literal* is considered to be a signed word offset, meaning it is multiplied by 4 to convert it to a byte offset, sign extended to 32 bits, then added to the updated PC to form the target address.

JMP

- Usage: JMP(Ra,Rc)
- Opcode: 011011
- Operation:
    - PC + 4 → PC
    - Reg[Ra] & 0xFFFFFFFC → EA (effective address)
    - PC → Reg[Rc]
    - EA → PC
- Description: The address of the instruction following the JMP instruction is written to register Rc, then the PC is loaded with the contents of Ra. The low two bits of Ra are masked so the target address is aligned on a 4-byte boundary. Ra and Rc may specify the same register, as the calculation is done before the assignment of the new value. The unused literal field should be filled with zeros. Note that JMP can clear the supervisor bit (bit 31 of the PC) but not set it.

LD

- Usage: LD(Ra,literal,Rc)
- Opcode: 011000
- Operation:
    - PC + 4 → PC
    - Reg[Ra] + SEXT(literal) → EA (effective address)
    - Mem[EA] → Reg[Rc]
- Description: The effective address EA is computed with the summation of the contents of register Ra and the sign-extended 16-bit displacement literal. The location in memory specified by EA is read into register Rc.

LDR

- Usage: LDR(label,Rc)
- Opcode: 011111
- Operation:
    - literal = ((OFFSET(label) - OFFSET(current instruction)) / 4) - 1
    - PC + 4 → PC
    - PC + 4 * SEXT(literal) → EA (effective address)
    - Mem[EA] → Reg[Rc]
- Description: The effective address EA is computed by multiplying the sign-extended literal by 4 to convert it to a byte offset, and then adding it to the updated PC. The location in memory specified by EA is read into register Rc. Ra is ignored and should be 11111 (R31). The supervisor bit (bit 31 of the PC) is ignored (treated as zero) when computing the EA.

ST

- Usage: ST(Rc,literal,Ra)
- Opcode: 011001
- Operation:
    - PC + 4 → PC
    - Reg[Ra] + SEXT(literal) → EA (effective address)
    - Reg[Rc] → Mem[EA]
- Description: The effective address EA is computed with the summation of the contents of register Ra and the sign-extended 16-bit displacement literal. The contents of register Rc are written to the location in memory specified by EA.