## Number System

**1s complement:** Flip all bits | **2s complement:** Flip all bits then add 1

**Ns complement:** For Ns complement negative number, 首先计算其正数在该进制下的数字，再使用比其大一位的10…0减去正数以得到负数。例如转换为7进制的7s complement,首先计算，再取

Largest value:

**Fixed point:** 010.11 =

**Floating point representation:** Use scientific notation to convert binary numbers to below (without sign bit)

**Sign bit (1)**: 1 for negative, 0 for positive

**Exponent bit (E):** the exponent value of the radix, for example , the exponent is 4, where the exponent bit is

**Mantissa bit (M):** the fractional part, since IEEE 754 has one hidden bit, so the integer part is ignored. Truncate if the number of fractional parts is greater than the specified value. If it is less than the specified value, write it and then continue from the beginning

**Excess:** The excess number is the offset, for example, in excess-4096 16 bit number system, the most negative number is 0000 0000 0000 0000 which is 0-4096=-4096. The most positive number is 65536-1-4096=61439

**ASCII:** digits( 0 - 9 ) < capital letter ( A - Z ) < lowercase letter ( a - z )

**Overflow:** positive+ positive= negative, or negative + negative = positive

## Data size

1 byte = 8 bits

1 character in ASCII system is 1 byte

1 MIPS instruction is 4 bytes,

opcode = 6 bits, rs = 5 bits, rt = 5 bits, rd = 5 bits, shamt = 5 bits, immediate= 16 bits, address = 26 bits

## Datapath:

**R-format: (add, sub, sll)**

**add $8, $9, $10:** rd = 8 (destination register)

rs = 9 (first operand) rt = 10 (second operad)

**sll $8, $9, 4:**  rd = 8, rt = 9, shamt = 4

**I-format: (addi, subi, lw, sw)**

**J-format:** jumps to:

文本

描述已自动生成

**addi $8, $8, 10:** rs = rt = $8, immediate = 10



**Execution cycle:** Fetch(IF) -> Decode(ID) -> Operand Fetch -> Execute (ALU) -> Result Write

**Fetch Stage:** Fetch instruction from memory (Program Counter PC)

**Decode Stage:** Use register file to read all necessary data

> Register file input:

> > Read register 1 (RR1): read rs (operand 1) 源寄存器

> > Read register 2 (RR2): read rt (operand 2) 目标寄存器

> > Write Register (WR): read rd (destination register) 目的地寄存器 for R-format, rt for I-format

> > Write Data (WD): the ALU output, for R format sub $25, $20, $5, WD should be [$20]-[$5]

> Register file output:

> > Read data 1: content of rs (operand 1)

> > Read data 2: content of rt for R-format, immediate for I-format

> Control Signal RegWrite to control the writing of register

> Control Signal RegDst for WR side MUX to indicate I-J or R format, 0 for R format

> Control Signal ALUSrc for RD2 side MUX to indicate I or R-J format, 0 for R-J format

**ALU stage:** implement arithmetic and logical operations

> ALU input:

> > Opr1: content of rs ([rs])

> > Opr2: according to control signal ALUSrc, content of rt for R, immediate for I

> ALU output:

> > isZero: For branch instruction to handle equal/not equal check

> > ALU result: result of operation

> ALU control signal ALUcontrol: 4-bit signal, to control the operation function (AND, OR, add, sub…)

**Memory Stage:** store and load ALU stage result to memory (for read/load operation only like lw, sw. Other instruction like add, or keep idle)

> Data Memory Input:

> > Address: the output of ALU result (ALU computes address)

> > Write Data: if the action is write, import from rt (RD2)

> Data Memory Output:

> > Read Data: if the action is read, output the memory address

> Data Memory Control Signal: MemWrite and MemRead

> For Non-memory instruction (add or sub), the output of ALU stage and RD2’s output rt is inputted to a MUX, controlled by MemReg, indicate whether result come from memory or ALU.

**Register Write Stage:** write result of computation into register for instructions add, or, sub

> The output of this stage is the input of WD in register file (Decode Stage)

**Other control signal:**

> PCSrc: Select the next PC value

> > For regular instruction, the next PC value should be PC+4

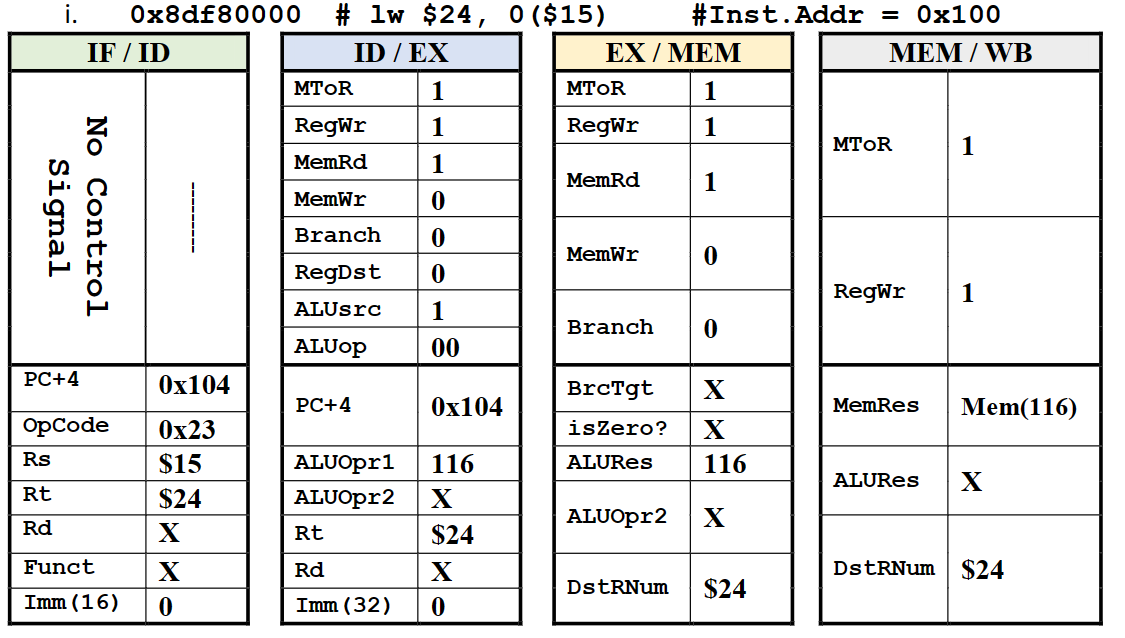
> > For branch instruction but not enter the branch, the next PC value also PC+4

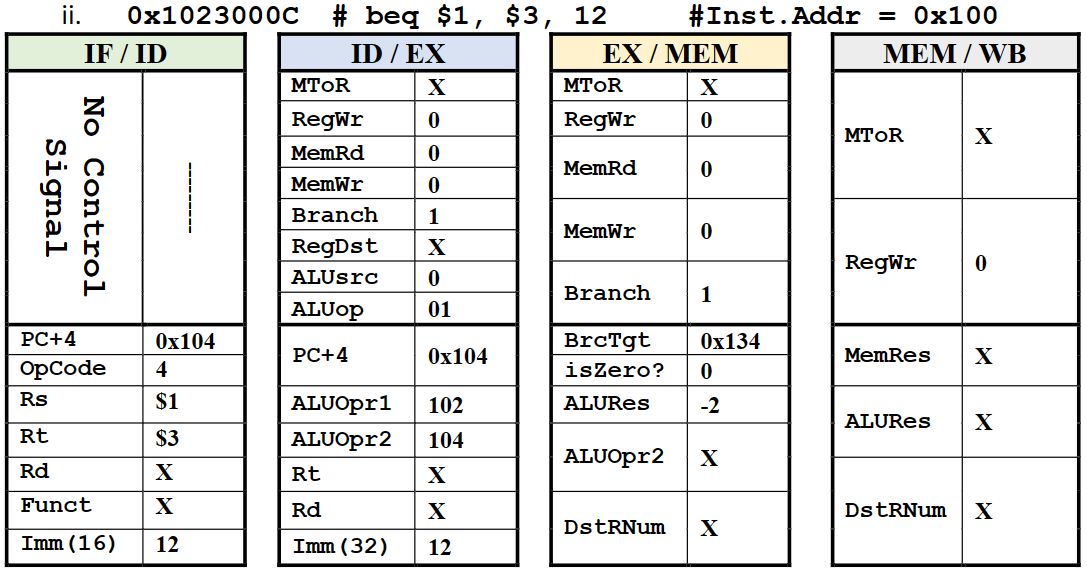
> > For branch instruction that enter the branch, the next PC value should be PC+4+(immediate\*4)

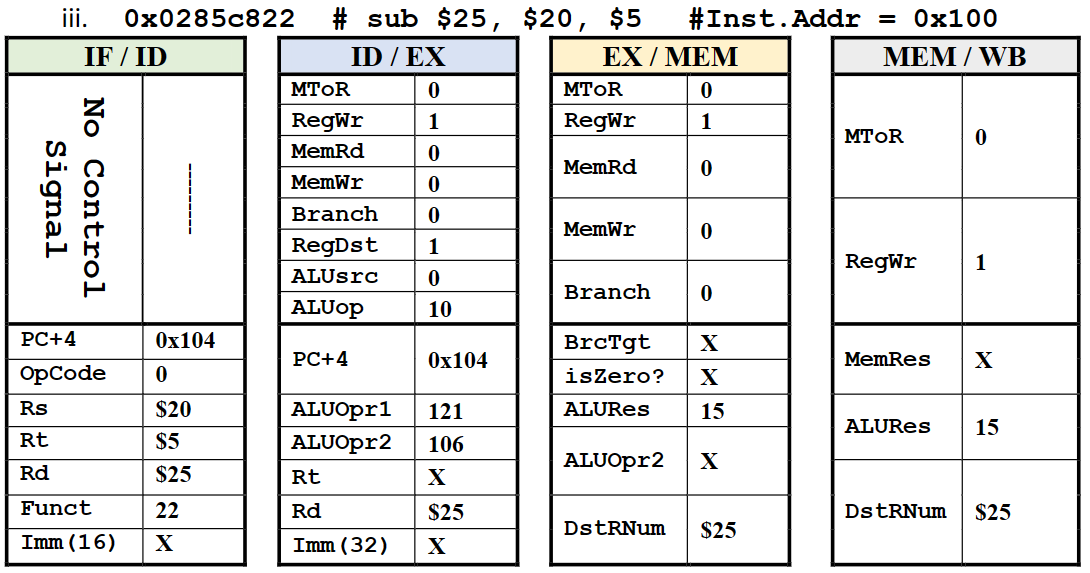
**I, J, R Format instruction detailed Datapath value**

表格

描述已自动生成







> MtoR(Memory to Register): 结果是否来自内存(1 for lw, 0 for sub)

> RegWr(Register Write):

是否需要写入寄存器(1 for lw sub sll, 0 for beq, sw)

> MemRd(Memory Read):

是否需要从内存读取(1 for lw, 0 for sub beq sw sll)

> MemWr(Memory Write):

是否需要写入内存(1 for sw, 0 for lw sub beq sll)

> Branch:

是否是分支指令 (beq)

> RegDst:

是否选择rt作为目标寄存器 (1 for sub sll, 0 for sw lw)

> ALUSrc:

ALU的第二个操作数是否来自immediate (1 for lw sw, 0 for beq sub)

> ALUop:

表示ALU的操作类型

表格

描述已自动生成

## 表格 中度可信度描述已自动生成

## Pipeline

> Different instructions pass different stages in pipeline

| Instruction | IF | ID | ALU | MEM | WB |
| --- | --- | --- | --- | --- | --- |
| Arithmetic | X | X | X |  | X |
| Branch | X | X | X |  |  |
| Load | X | X | X | X | X |
| Store | X | X | X | X |  |

**Running time:** (=Time of operation in stage , =Number of stages, =number of instructions)

> All non-pipelined must take the slowest stage time as

> Single cycle non-pipelined: count all stages, whether the stage is used or not:

> Multi-cycle non-pipelined: count stage based on different instruction, or use average CPI:

> Pipelined: take the slowest stage/operation time as , for pipeline register delay

**Speedup:**

## Cache

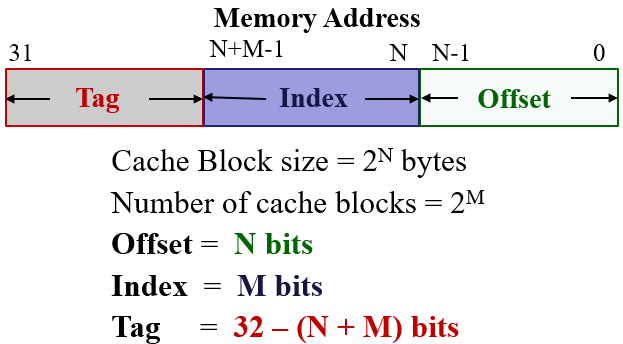
### Cache hit and miss

> Hit time = time to access the cache

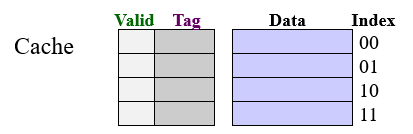
> Miss Penalty = time to access the cache + memory

**Direct Mapped Cache**

> Memory Address Structure



> Cache Structure



### Write policy

> Write through: 当缓存中的数据被修改时，同步写入主内存。优点是一致性高，缺点是额外的时间开销。使用write buffer当作缓冲。

> Write back: 引入dirty bit，只有在缓存块被替换时，脏位为1的缓存块才会写回memory

### Write miss policy:

> Write allocate: 从主内存将数据块加载到缓存中，然后更改相应的数据。是否立即写回主存取决于write policy

> Write around: 更改的数据直接写入主存，缓存不参与操作。这意味着后续读取会导致缓存缺失，这个策略通常用于不太会用到的数据，或缓存很快会被填满的情况

### Set Associative Cache

> 缓存被分为多个set，每个set包含几个block。例如2-way set associative cache就是每个set有两个block

>原先的index field变为set index field，即决定写入哪个set

**Block replacement policy:**

> LRU (Least recently used): hit的block被排到最前面，如果需要替换block，则替换最后（最不常用）的block

> FIFO, Random Replacement, Least Frequently Used

图示

描述已自动生成

表格

描述已自动生成