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-213_{10}转换为 7 进制的 7s complement,首先计算$213_{10} = 423_7$，再取$1000_7 - 423_7 = 244_7$
Largest value:
Fixed point: `010.11` $= 0\times2^2 + 1\times2^1 + 0\times2^0 + 1\times2^{-1} + 1\times2^{-2} = 2.75$
Floating point representation: Use scientific notation to convert binary numbers to below (without sign bit)

$$1.\text{fraction} \times 2^{\text{exponent}}$$

Sign bit (1): 1 for negative, 0 for positive
Exponent bit (E): the exponent value of the radix, for example 1.fraction $\times 2^4$, the exponent is 4, where the exponent bit is 0100_2
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 n 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:

`1010` `00001111000011110000111100` `00`

Most significant 4bits of PC+4 | 26bits Target address specified in instruction | Default 2bit "00" for word address

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

R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt	immediate		
J	opcode	target address				

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

	RegDst	ALUSrc	MemTo Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop op1	ALUop op0
R-type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

i. `0x8df80000 # lw $24, 0($15) #Inst.Addr = 0x100`

IF / ID	
No Control Signal	------
PC+4	0x104
OpCode	0x23
Rs	$15
Rt	$24
Rd	X
Funct	X
Imm (16)	0

ID / EX	
MToR	1
RegWr	1
MemRd	1
MemWr	0
Branch	0
RegDst	0
ALUsrc	1
ALUop	00
PC+4	0x104
ALUOpr1	116
ALUOpr2	X
Rt	$24
Rd	X
Imm (32)	0

EX / MEM	
MToR	1
RegWr	1
MemRd	1
MemWr	0
Branch	0
BrcTgt	X
isZero?	X
ALURes	116
ALUOpr2	X
DstRNum	$24

MEM / WB	
MToR	1
RegWr	1
MemRes	Mem(116)
ALURes	X
DstRNum	$24

ii. `0x1023000C # beq $1, $3, 12 #Inst.Addr = 0x100`

IF / ID	
No Control Signal	------
PC+4	0x104
OpCode	4
Rs	$1
Rt	$3
Rd	X
Funct	X
Imm (16)	12

ID / EX	
MToR	X
RegWr	0
MemRd	0
MemWr	0
Branch	1
RegDst	X
ALUsrc	0
ALUop	01
PC+4	0x104
ALUOpr1	102
ALUOpr2	104
Rt	X
Rd	X
Imm (32)	12

EX / MEM	
MToR	X
RegWr	0
MemRd	0
MemWr	0
Branch	1
BrcTgt	0x134
isZero?	0
ALURes	-2
ALUOpr2	X
DstRNum	X

MEM / WB	
MToR	X
RegWr	0
MemRes	X
ALURes	X
DstRNum	X

iii. `0x0285c822 # sub $25, $20, $5 #Inst.Addr = 0x100`

IF / ID	
No Control Signal	------
PC+4	0x104
OpCode	0
Rs	$20
Rt	$5
Rd	$25
Funct	22
Imm (16)	X

ID / EX	
MToR	0
RegWr	1
MemRd	0
MemWr	0
Branch	0
RegDst	1
ALUsrc	0
ALUop	10
PC+4	0x104
ALUOpr1	121
ALUOpr2	106
Rt	X
Rd	$25
Imm (32)	X

EX / MEM	
MToR	0
RegWr	1
MemRd	0
MemWr	0
Branch	0
BrcTgt	X
isZero?	X
ALURes	15
ALUOpr2	X
DstRNum	$25

MEM / WB	
MToR	0
RegWr	1
MemRes	X
ALURes	15
DstRNum	$25

> 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

	add $3, $1, $2	lw $3, 20($1)	beq $1, $2, label
Fetch	Read inst. at [PC]	Read inst. at [PC]	Read inst. at [PC]
Decode & Operand Fetch	○ Read [**$1**] as *opr1* ○ Read [**$2**] as *opr2*	○ Read [**$1**] as *opr1* ○ Use ***20*** as *opr2*	○ Read [**$1**] as *opr1* ○ Read [**$2**] as *opr2*
ALU	*Result* = *opr1* + *opr2*	*MemAddr* = *opr1* + *opr2*	*Taken* = (*opr1* == *opr2*)? *Target* = (**PC**+4) + **ofst×4**
Memory Access		Use *MemAddr* to read from memory	
Result Write	*Result* stored in **$3**	*Memory* data stored in **$3**	if (*Taken*) **PC** = *Target*

Opcode	ALUop	Instruction Operation	Funct field	ALU action	ALU control
lw	00	load word	xxxxxx	add	0010
sw	00	store word	xxxxxx	add	0010
beq	01	branch equal	xxxxxx	subtract	0110
R-type	10	add	10 0000	add	0010
R-type	10	subtract	10 0010	subtract	0110
R-type	10	AND	10 0100	AND	0000
R-type	10	OR	10 0101	OR	0001
R-type	10	set on less than	10 1010	set on less than	0111

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: (T_k=Time of operation in stage k, N=Number of stages, I=number of instructions)
> All non-pipelined must take the slowest stage time as T_k
> Single cycle non-pipelined: count all stages, whether the stage is used or not: Cycle Time $= \sum_{k=1}^{N} T_k$, $T = I \times$ Cycle Time
> Multi-cycle non-pipelined: count stage based on different instruction, or use average CPI: $CT_{multi} = max(T_k)$, $T =$ Cycles × Cycle Time = $I \times$ Average CPI × CT_{multi}
> Pipelined: take the slowest stage/operation time as T_k, T_d for pipeline register delay

$$CT = max(T_k) + T_d,\ T = (I + N - 1) \times (max(T_k) + T_d)$$

Speedup:

$$\text{Speedup} = \frac{T_{seq}}{T_{pipeline}} \approx N$$

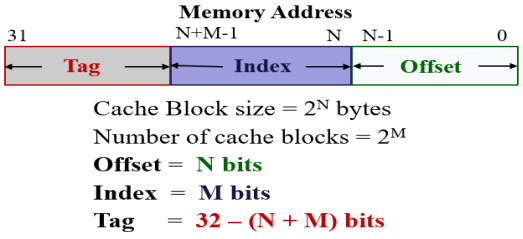
Cache

Cache hit and miss

Average Access Time = Hit rate × Hit time + (1 − Hit rate) × Miss Penalty
> Hit time = time to access the cache
> Miss Penalty = time to access the cache + memory

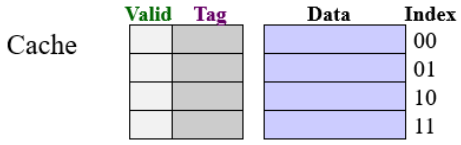
Direct Mapped Cache

> Memory Address Structure



Cache Block size = 2^N bytes
Number of cache blocks = 2^M
Offset = N bits
Index = M bits
Tag = 32 – (N + M) bits

> 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:

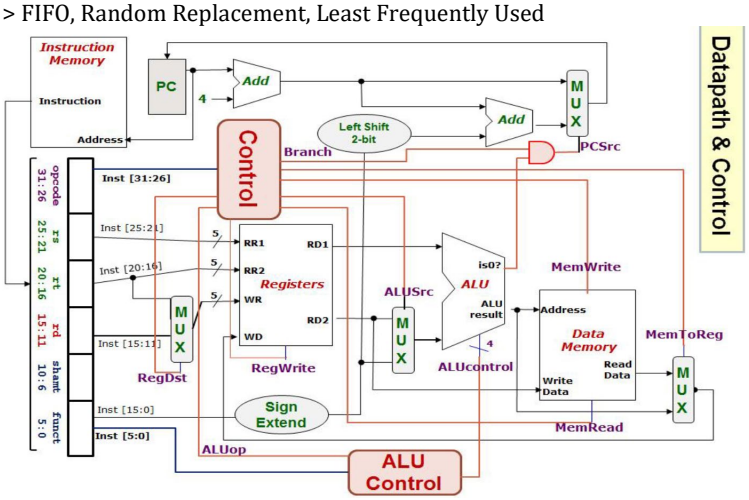
> Write allocate: 从主内存将数据块加载到缓存中，然后更改相应的数据。是否立即写回主存取决于 write policy
> Write around: 更改的数据直接写入主存，缓存不参与操作。这意味着后续读取会导致缓存缺失，这个策略通常用于不太会用到的数据，或缓存很快会被填满的情况

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



MIPS assembly language

Category	Instruction	Example	Meaning
Arithmetic	add	add $s1, $s2, $s3	$s1 = $s2 + $s3
	subtract	sub $s1, $s2, $s3	$s1 = $s2 - $s3
	add immediate	addi $s1, $s2, 100	$s1 = $s2 + 100
Data transfer	load w ord	lw $s1, 100($s2)	$s1 = Memory[$s2 + 100]
	store w ord	sw $s1, 100($s2)	Memory[$s2 + 100] = $s1
	load byte	lb $s1, 100($s2)	$s1 = Memory[$s2 + 100]
	store byte	sb $s1, 100($s2)	Memory[$s2 + 100] = $s1
	load upper immediate	lui $s1, 100	$s1 = 100 * 2^{16}
Conditional branch	branch on equal	beq $s1, $s2, 25	if ($s1 == $s2) go to PC + 4 + 100
	branch on not equal	bne $s1, $s2, 25	if ($s1 != $s2) go to PC + 4 + 100
	set on less than	slt $s1, $s2, $s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0
	set less than immediate	slti $s1, $s2, 100	if ($s2 < 100) $s1 = 1; else $s1 = 0
Uncondi-tional jump	jump	j 2500	go to 10000
	jump register	jr $ra	go to $ra
	jump and link	jal 2500	$ra = PC + 4; go to 10000