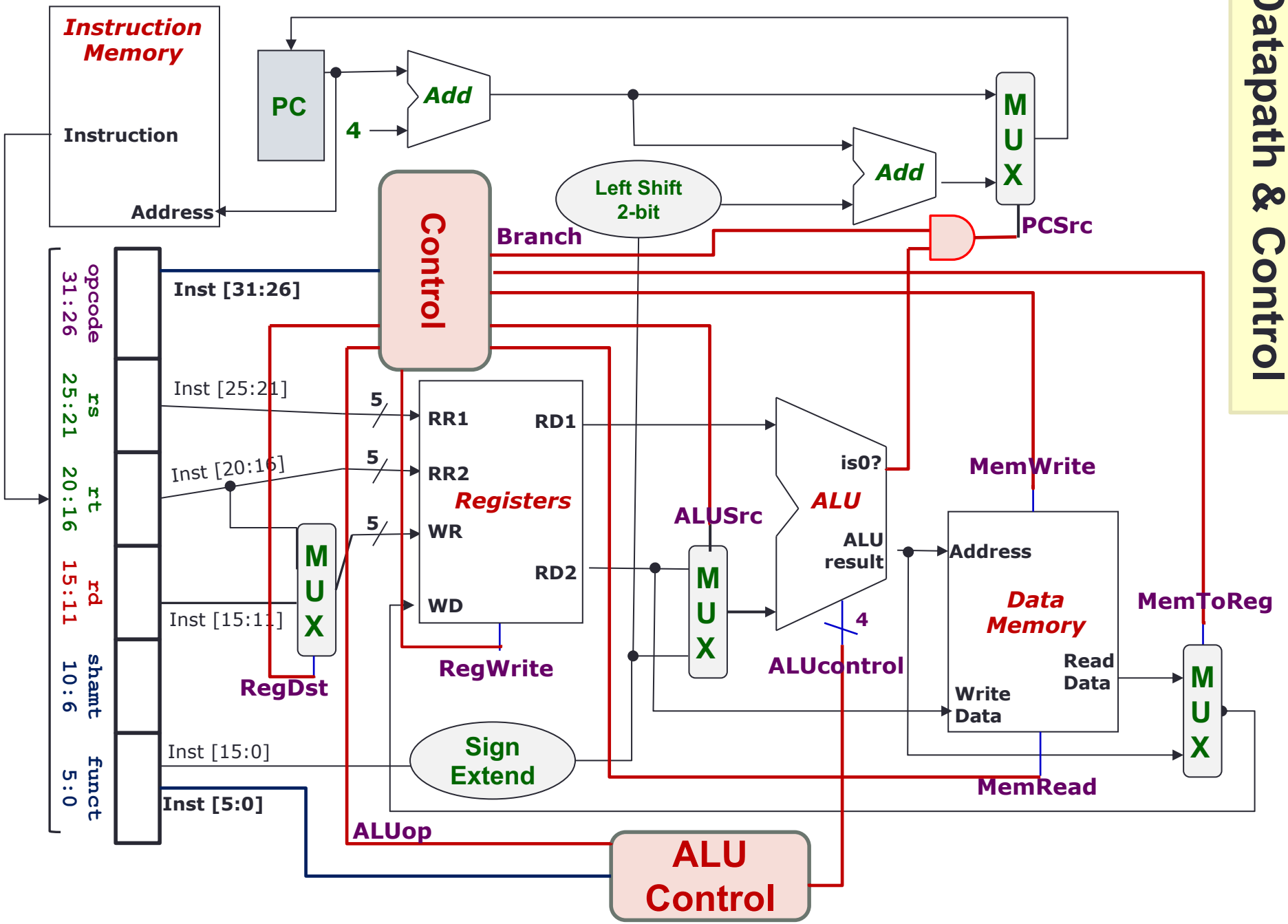
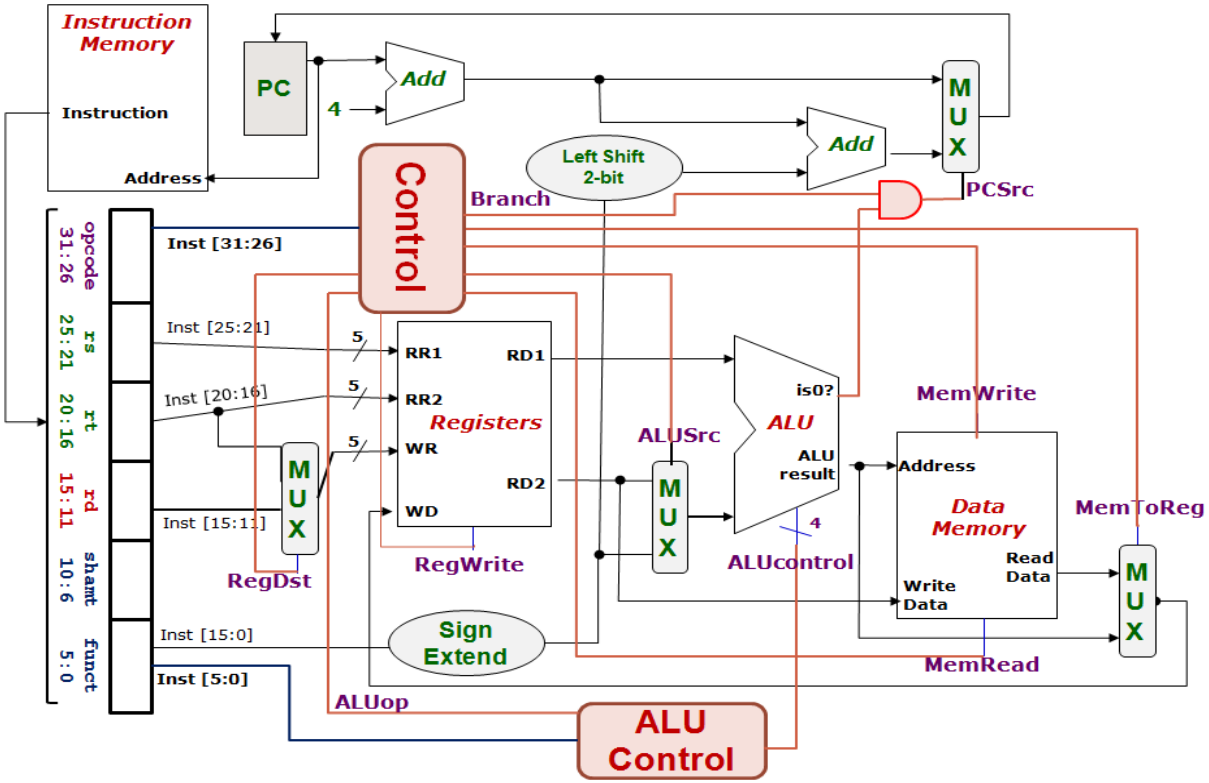
Datapath & Control



5. Control Design: **Outputs**

	RegDst	ALUSrc	MemTo Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop	
								op1	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



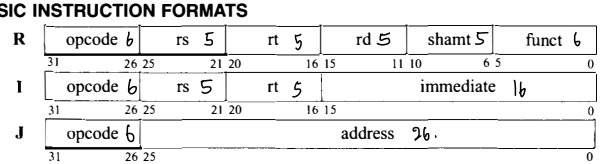
MIPS Reference Data

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	0 / 20_{hex}
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		0 / 21_{hex}
And	and	R	R[rd] = R[rs] & R[rt]		0 / 24_{hex}
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	j	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	jr	R	PC=R[rs]		0 / 08_{hex}
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs]+SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs]+SignExtImm](15:0)}	(2)	25_{hex}
Load Linked	ll	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	nor	R	R[rd] = ~ (R[rs] | R[rt])		0 / 27_{hex}
Or	or	R	R[rd] = R[rs] | R[rt]		0 / 25_{hex}
Or Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}
Set Less Than	slt	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		0 / $2a_{hex}$
Set Less Than Imm.	slti	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0	(2)	a_{hex}
Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b_{hex}
Set Less Than Unsig.	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	0 / $2b_{hex}$
Shift Left Logical	sll	R	R[rd] = R[rt] << shamt		0 / 00_{hex}
Shift Right Logical	srl	R	R[rd] = R[rt] >> shamt		0 / 02_{hex}
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28_{hex}
Store Conditional	sc	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (*atomic*) ? 1 : 0	(2,7)	38_{hex}
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29_{hex}
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	0 / 22_{hex}
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]		0 / 23_{hex}

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)
(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

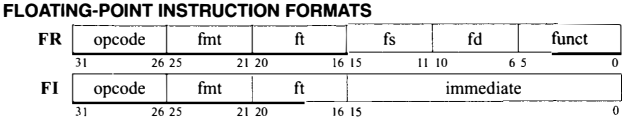
BASIC INSTRUCTION FORMATS



ARITHMETIC CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION		OPCODE / FMT /FT / FUNCT (Hex)
Branch On FP True	bc1t	FI	if(FPcond)PC=PC+4+BranchAddr	(4)	11/8/1/--
Branch On FP False	bc1f	FI	if(!FPcond)PC=PC+4+BranchAddr	(4)	11/8/0/--
Divide	div	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]		0/--/--/1a
Divide Unsigned	divu	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	(6)	0/--/--/1b
FP Add Single	add.s	FR	F[fd]= F[fs] + F[ft]		11/10/--/0
FP Add Double	add.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}		11/11/--/0
FP Compare Single	c.x.s*	FR	FPcond = (F[fs] *op* F[ft]) ? 1 : 0		11/10/--/*y*
FP Compare Double	c.x.d*	FR	FPcond = ({F[fs],F[fs+1]} *op* {F[ft],F[ft+1]}) ? 1 : 0		11/11/--/*y*
* (*x* is eq, lt, or le) (*op* is ==, <, or <=) (*y* is 32, 3c, or 3e)					
FP Divide Single	div.s	FR	F[fd] = F[fs] / F[ft]		11/10/--/3
FP Divide Double	div.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}		11/11/--/3
FP Multiply Single	mul.s	FR	F[fd] = F[fs] * F[ft]		11/10/--/2
FP Multiply Double	mul.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}		11/11/--/2
FP Subtract Single	sub.s	FR	F[fd]=F[fs] - F[ft]		11/10/--/1
FP Subtract Double	sub.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}		11/11/--/1
Load FP Single	lwc1	I	F[rt]=M[R[rs]+SignExtImm]	(2)	31/--/--/--
Load FP Double	ldc1	I	F[rt]=M[R[rs]+SignExtImm]; F[rt+1]=M[R[rs]+SignExtImm+4]	(2)	35/--/--/--
Move From Hi	mfhi	R	R[rd] = Hi		0 /--/--/10
Move From Lo	mflo	R	R[rd] = Lo		0 /--/--/12
Move From Control	mfc0	R	R[rd] = CR[rs]		10 /0/--/0
Multiply	mult	R	{Hi,Lo} = R[rs] * R[rt]		0/--/--/18
Multiply Unsigned	multu	R	{Hi,Lo} = R[rs] * R[rt]	(6)	0/--/--/19
Shift Right Arith.	sra	R	R[rd] = R[rt] >>> shamt		0/--/--/3
Store FP Single	swc1	I	M[R[rs]+SignExtImm] = F[rt]	(2)	39/--/--/--
Store FP Double	sdc1	I	M[R[rs]+SignExtImm] = F[rt]; M[R[rs]+SignExtImm+4] = F[rt+1]	(2)	3d/--/--/--

FLOATING-POINT INSTRUCTION FORMATS

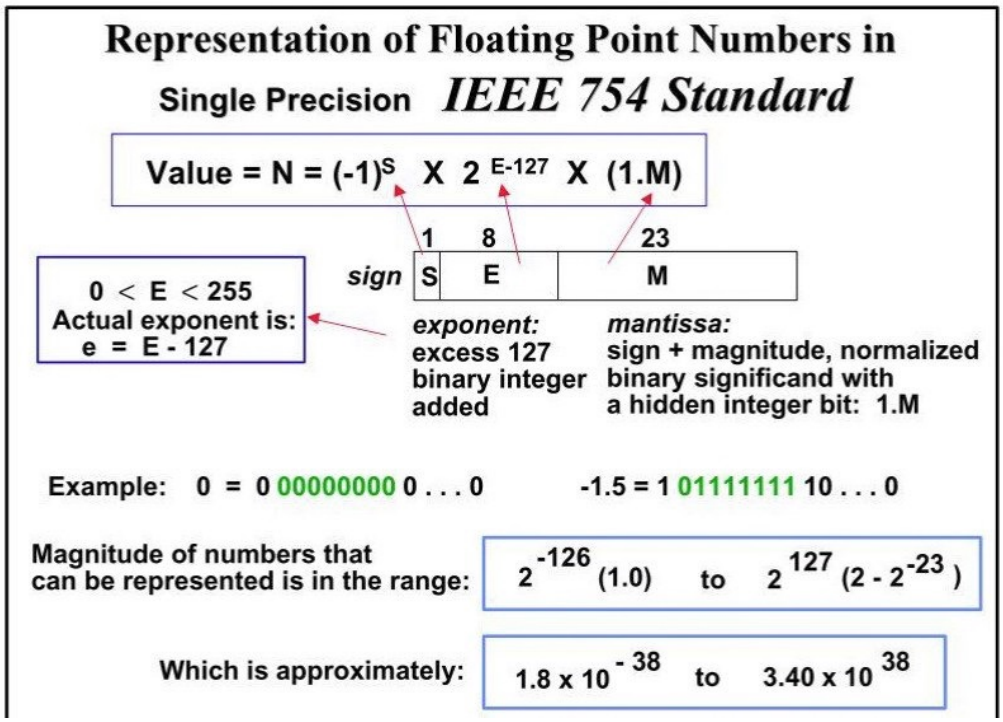


PSEUDOINSTRUCTION SET

NAME	MNEMONIC	OPERATION
Branch Less Than	blt	if(R[rs]<R[rt]) PC = Label
Branch Greater Than	bgt	if(R[rs]>R[rt]) PC = Label
Branch Less Than or Equal	ble	if(R[rs]<=R[rt]) PC = Label
Branch Greater Than or Equal	bge	if(R[rs]>=R[rt]) PC = Label
Load Immediate	li	R[rd] = immediate
Move	move	R[rd] = R[rs]

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$vl	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes



$$F = (-1)^S \; 2^{(E-127)}(1+M/2^{23})$$ where the F - decimal

Check our example:
$F = (-1)^0 \cdot 2^{(134-127)} \cdot (1+ 1810432 / 2^{23}) = 2^7 \cdot (1+0{,}2158203125) = 128 \cdot 1{,}2158203125 = 155{,}625$

Radix and Diminished Radix complement

← Prev Next →

The mostly used complements are 1's, 2's, 9's, and 10's complement. Apart from these complements, there are many more complements from which mostly peoples are not familiar. For finding the subtraction of the number base system, the complements are used. If **r** is the base of the number system, then there are two types of complements that are possible, i.e., r's and (r-1)'s. We can find the r's complement, and (r-1)'s complement of the number, here r is the radix. The r's complement is also known as **Radix complement** (r-1)'s complement, is known as **Diminished Radix complement**.

If the base of the number is 2, then we can find 1's and 2's complement of the number. Similarly, if the number is the octal number, then we can find 7's and 8's complement of the number.

There is the following formula for finding the r's and (r-1)'s complement:

r' s= complement=$(r^n)_{10}$-N

(r-1)' s complement={$(r^n)_{10}$-1}-N

In the above formulas,

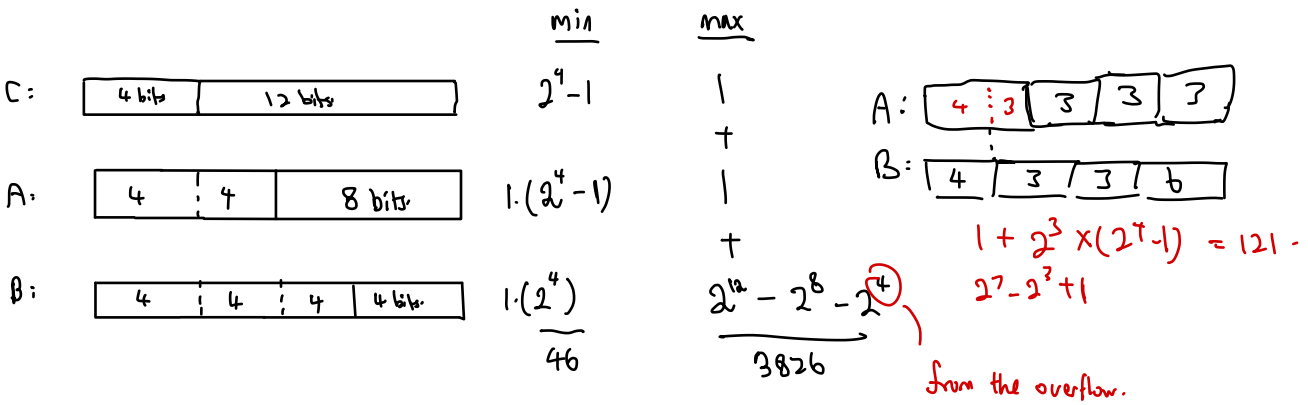
- The n is the number of digits in the number.
- The N is the given number. (in base 10)
- The r is the radix or base of the number.

We generalize $(r-1)$'s-complement (also called radix diminished complement) to include fraction as follows:

$(r-1)$'s complement of $N = r^n - r^{-m} - N$

where n is the number of integer digits and m the number of fractional digits. (If there are no fractional digits, then $m = 0$ and the formula becomes $r^n - 1 - N$ as given in class.)

For example, the 1's complement of 011.01 is $(2^3 - 2^{-2}) - 011.01 = (1000 - 0.01) - 011.01 = 111.11 - 011.01 = 100.10$.



```
typedef struct {
  int numer[1];
  int *denom;
} rational;
```

pointer. *

```
*(x.numer)   = x_num * y_num;
*(x.denom)   = x_den * y_den;
*(y->numer) = x_num * y_num;
*(y->denom) = x_den * y_den;
```

1 x 2 = 2 → copied number (always one copied)
2 x 5 = 10 → pointer.
1 x 2 = 2 → pointer
2 x 5 = 10 → pointer.

For C code, use box and pointer diagrams.

```
typedef struct {
    int first, second;
} pair_t;
void g(int *arr, pair_t pair) {
    *arr = 55;
    pair.first = 66;
    pair.second = 77;
}
```

not change ⇒ just a copy.

4. Pick all of the statements below that are TRUE about the MIPS datapath.

   a. A single cycle implementation is better than multicycle implementation since every MIPS instruction takes exactly the same amount of time to go through the datapath.

   b. A multicycle implementation is better than single-cycle since not all instructions need to pass through every stage of the datapath.

   c. In an N stage datapath where each stage may take a different amount of time to complete, a multicycle implementation potentially allows up to N times speedup over a single cycle implementation if there are instructions that must pass through only one stage.

   d. In a multicycle implementation where every stage takes the same amount of time to complete, the instruction that passes through the most stages will have the same execution time as in a single-cycle implementation.

   e. In a multicycle implementation where every stage may take a different amount of time to complete, the instruction that takes the longest will NOT have the same execution time as in a single cycle implementation.

(interp. of stages)

15. The most negative number that can be represented in this number system is _____.

Largest exponent = $2^5 - 1 = 31$ (6 bit exponent in 2's complement)

∴ Most negative: $-1.11111111 \times 2^{31}$ (9 bit normalised mantissa w/o hidden bit)

$= -4286578688$.

10. **[CHALLENGING]** What is the ***maximum*** possible number of MIPS instructions that can be inserted into the regions marked with "**`Code omitted`**"? Note that the code can be inserted into any one of the two regions. We are only interested in the total. For simplicity, write your answer in terms of $2^x \pm y$ [2 marks]

256MB = 2^{26} instr. max range

∴ $2^{26} - 9$ — 9 instr. included.

- | (bitwise OR) → returns 1 when either corresponding bit in a or b is 1.
- & (bitwise AND) → returns 1 when both bits are 1, good for masking.
- ^ (bitwise XOR) → returns 1 in each bit position when either but not both are 1.
- ~ (one's complement) → NOT operator for bits
- << (left shift) } shift bits to left / right.
- >> (right shift) }

```
void swap(int *a, int *b) {
    *a = *a ^ *b;
    *b = *b ^ *a;
    *a = *a ^ *b;
}
```

Critical Paths:

(a) SUB instruction (R-type):

Critical Path:
I-Mem →Reg.File → MUX(ALUSrc) → ALU → MUX(MemToReg) → Reg.File

Note: I-MEM → Control is a parallel path, the earliest signal needed is the ALUSrc. So, as long as the Control latency is lesser than Reg.File access latency, then it will not be in the critical path. Once the signal is generated, the Control latency will no longer affect the overall delays.

Similarly, there is another path to calculate the next PC (I-MEM → Control → AND → MUX(PCSrc) which is again not critical to the overall latency.

Latency = 400 + 200 + 30 + 120 + 30 + 200 = 980ps

(b) LW instruction:

Critical Path:
I-Mem → Reg.File → ALU → DataMem → MUX(MemToReg) → Reg.File

Latency = 400 + 200 + 120 + 350 + 30 + 200 = 1300ps

Note: The path I-Mem → Immediate → MUX(ALUSrc) occurs simultaneously with the above.

(c) BEQ instruction:

Critical Path:
I-Mem → Reg.File → MUX(ALUSrc) → ALU → AND → MUX(PCSrc)

Latency = 400 + 200 + 30 + 120 + 20 + 30 = 800ps

Since LW has the longest latency. The overall cycle time of the whole machine is determined by LW, i.e. at least 1300ps.