CSE340

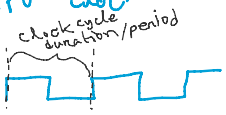
Tuesday, February 6, 2024 10:20 PM

Chapter 1:-

- ✓ Moore's prediction
- ✓ Computers are pervasive
- ✓ IC cost:-
  - Dies per wafer = $\frac{\text{Wafer area}}{\text{Die area}}$
  - Cost per die = $\frac{\text{Cost per wafer}}{\text{Die per wafer} \times \text{Yield}}$ → Proportion of working dies
  - Yield = $\frac{1}{\left\{1 + \left(\frac{\text{Defects per area} \times \text{Die area}}{2}\right)\right\}^2}$
- ✓ Judge performance on the basis of Throughput (efficiency)
- ✓ Performance = $\frac{1}{\text{Execution time}}$

$$n = \frac{\text{Performance}_A}{\text{Performance}_B} = \frac{\text{Execution time}_B}{\text{Execution time}_A}$$

- ✓ CPU time = Burst time
- ✓ Elapsed time = CPU time + Others (idle, context switch, ...)
- ✓ CPU clock



- ✓ clock speed/rate
- ✓ Clock cycle time = $\frac{1}{\text{clock rate}}$
- ✓ CPU time = No of cycles required × Clock cycle time = $\frac{\text{No of cycles required}}{\text{clock rate}}$
- ✓ CPI (Cycles per instruction → R, I, J)
- ✓ Clock cycles required = Instruction Count × CPI
- ✓ Avg CPI from an instruction mix
- ✓ Power = Capacitive load × Voltage^2 × Frequency
- ✓ SPEC ratio
  - → for every program, SPEC ratio = $\frac{\text{Reference time}}{\text{Execution time}}$

Description	Name	Instruction Count x 10^9	CPI	Clock cycle time (seconds x 10^-9)	Execution Time (seconds)	Reference Time (seconds)	SPECratio
Perl interpreter	perlbench	2684	0.42	0.556	627	1774	2.83
GNU C compiler	gcc	2322	0.67	0.556	863	3976	4.61
Route planning	mcf	1786	1.22	0.556	1215	4721	3.89
Discrete Event simulation - computer network	omnetpp	1107	0.82	0.556	507	1630	3.21
XML to HTML conversion via XSLT	xalancbmk	1314	0.75	0.556	549	1417	2.58
Video compression	x264	4488	0.32	0.556	813	1763	2.17
Artificial Intelligence:							

Computer A: 2GHz clock, 10s CPU time

Designing Computer B

- Aim for 6s CPU time
- Can do faster clock, but causes 1.2 × clock cycles

How fast must Computer B clock be?

For A, CPU time = $\frac{\text{clocks needed}}{\text{frequency}}$

$\Rightarrow 10s = \frac{C_{need}}{2GHz}$

$\therefore C_{need} = 20\ GHz\ s$

For B, $6s = \frac{1.2 \times C_{need}}{freq}$

$\Rightarrow freq = \frac{1.2 \times 20GHz\,s}{6s} = 4GHz$

✓ SPEC ratio

→ for every program, spec ratio = $\frac{\text{Reference time}}{\text{Execution time}}$

→ Then take the geometric mean

Description	Name	Instruction Count x 10^9	CPI	Clock cycle time (seconds x 10^-9)	Execution Time (seconds)	Reference Time (seconds)	SPECratio
Perl interpreter	perlbench	2684	0.42	0.556	627	1774	2.83
GNU C compiler	gcc	2322	0.67	0.556	863	3976	4.61
Route planning	mcf	1786	1.22	0.556	1215	4721	3.89
Discrete Event simulation - computer network	omnetpp	1107	0.82	0.556	507	1630	3.21
XML to HTML conversion via XSLT	xalancbmk	1314	0.75	0.556	549	1417	2.58
Video compression	x264	4488	0.32	0.556	813	1763	2.17
Artificial Intelligence: alpha-beta tree search (Chess)	deepsjeng	2216	0.57	0.556	698	1432	2.05
Artificial Intelligence: Monte Carlo tree search (Go)	leela	2236	0.79	0.556	987	1703	1.73
Artificial Intelligence: recursive solution generator (Sudoku)	exchange2	6683	0.46	0.556	1718	2939	1.71
General data compression	xz	8533	1.32	0.556	6290	6182	0.98
Geometric mean							2.36

✓ Amdahl's Law:- Parallel Processing

There will always be some series part along with parallel part.

$$\therefore T_{new} = \frac{T_{parallel}}{\text{no of cores}} + T_{series}$$

$T_{improved}$ = T_{new}; $T_{parallel}$ → $T_{affected}$; T_{series} → $T_{unaffected}$; no of cores → Improvement factor

✓ $MIPS = \frac{\text{Instruction Count}}{\text{Execution time} \times 10^6} = \frac{\text{Clock rate}}{CPI \times 10^6}$

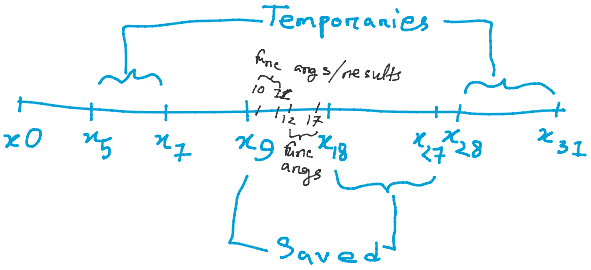
✓ Chapter 2:-

- Addition, subtraction

  add rd, rs1, rs2

  Ex: add a, b, c // a = b + c

- Word = 32 bits
- 32 registers (64bit each)
  - x0: the constant value 0
  - x1: return address
  - x2: stack pointer
  - x3: global pointer
  - x4: thread pointer
  - x5 – x7, x28 – x31: temporaries
  - x8: frame pointer
  - x9, x18 – x27: saved registers
  - x10 – x11: function arguments/results
  - x12 – x17: function arguments



f = (g + h) - (i + j);

- f, …, j in x19, x20, …, x23

```
add  x5 , x20, x21
add  x6 , x22, x23
sub  x19, x5 , x6
```

- f, …, j in x19, x20, …, x23

```
add  x6 , x22, x23
sub  x19, x5 , x6
```

- Loading data from RAM :- ld rd , offset(Base)
- Storing data to RAM :- sd rs , offset(Base)
- A[12] = h + A[8];
  - h in x21, base address of A in x22

```
ld   x5 , 64(x22)
add  x5 , x21, x5
sd   x5 , 96(x22)
```

- Adding constants :- addi rd, rs, const (immediate)
- R-format instruction :- (Arithmatic instruction)

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- I-format instruction :- (constant operation, load)

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

- S-format instruction :- (store)

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- Shift operation :-

funct6	immed	rs1	funct3	rd	opcode
6 bits	6 bits	5 bits	3 bits	5 bits	7 bits

- Logical operations :-
  - Instructions for bitwise manipulation

Operation	C	Java	RISC-V
Shift left	<<	<<	slli
Shift right	>>	>>>	srli
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit XOR	^	^	xor, xori
Bit-by-bit NOT	~	~	

R-format

NOT (all 1). 1 ⊕ 0 = 1, 1 ⊕ 1 = 0

- **Conditional / Branching :-**

```
if a == b :        (a -> x5, b -> x6)
    a += 1
else :
    b += 1
```

```
    bne  x5, x6, Else
    addi x5, x5, 1
    beq  x0, x0, End
Else:
    addi x6, x6, 1
End: ....
```

- **Looping :-**

```
while (save[i] == k) i += 1;
```

- i in x22, k in x24, address of save in x25

```
Loop: slli x5, x22, 3        // i x 8 , 2^3 = 8
      add  x5, x5, x25       // save + i x 8
      ld   x9, 0(x5)
      bne  x9, x24, Exit
      addi x22, x22, 1
      beq  x0, x0, Loop
Exit: ....
```

- **More conditions :-**
  - blt (<)
  - bge (>=)

- **Procedure call :** `jal x1, procedureLabel`
  - " **return :** `jalr x0, 0(x1)`

Leaf Procedure Example

- C code:

```
long long int leaf_example (
   long long int g, long long int h,
   long long int i, long long int j) {
  long long int f;
  f = (g + h) - (i + j);
  return f;
}
```

- RISC-V code:

```
leaf_example:
  addi sp,sp,-24
  sd   x5,16(sp)
  sd   x6,8(sp)
  sd   x20,0(sp
  add  x5,x10,x11
  add  x6,x12,x1
  sub  x20.x5.x6
```

Save x5, x6, x20 on stack

x5 = g + h
x6 = i + j
f = x5 – x6

```
  long long int i, long long int j) {
  long long int f;
  f = (g + h) - (i + j);
  return f;
}
```

- Arguments g, …, j in x10, …, x13
- f in x20
- temporaries x5, x6
- Need to save x5, x6, x20 on stack

```
sd   x20,0(sp
add  x5,x10,x11
add  x6,x12,x1
sub  x20,x5,x6
addi x10,x20,0
ld   x20,0(sp)
ld   x6,8(sp)
ld   x5,16(sp)
addi sp,sp,24
jalr x0,0(x1)
```

x5 = g + h
x6 = i + j
f = x5 – x6
copy f to return register
Resore x5, x6, x20 from stack
Return to caller

Non-Leaf Procedure Example

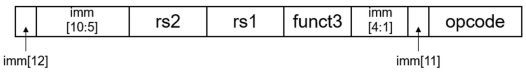
- C code:

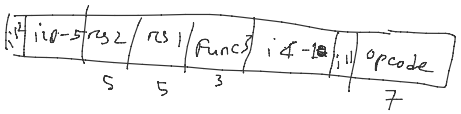
```
long long int fact (long long int n)
{
  if (n < 1) return 1
  else return n * fact(n - 1);
}
```

```
fact:
    addi sp,sp,-16          Save return address and n on stack
    sd   x1,8(sp)
    sd   x10,0(sp)
    addi x5,x10,-1          x5 = n - 1
    bge  x5,x0,L1           if n >= 1, go to L1
    addi x10,x0,1           Else, set return value to 1
    addi sp,sp,16           Pop stack, don't bother restoring values
    jalr x0,0(x1)           Return
L1: addi x10,x10,-1         n = n - 1
    jal  x1,fact            call fact(n-1)
    addi x6,x10,0           move result of fact(n - 1) to x6
    ld   x10,0(sp)          Restore caller's n
    ld   x1,8(sp)           Restore caller's return address
    addi sp,sp,16           Pop stack
    mul  x10,x10,x6         return n * fact(n-1)
    jalr x0,0(x1)           return
```

```
fact:
    addi sp, sp, -16
    sd   x1, 8(sp)      // return addr
    sd   x10, 0(sp)     // n
    addi x5, x0, 1      // x5 = 1
    bge  x10, x5, Else  // n ≥ 1
    addi x10, x0, 1     // return x10 = 1
    addi sp, sp, 16
    jalr x0, 0(x1)
Else:
    addi x10, x10, -1   // n = n-1
    jal  x1, fact
    addi x6, x10, 0     // x6 = Fact(n-1)
    ld   x10, 0(sp)     // x10 = caller's n
    ld   x1, 8(sp)      // caller's return address
    addi sp, sp, 16
    mul  x10, x10, x6   // n x fact(n-1)
    jalr x0, 0(x1)
```

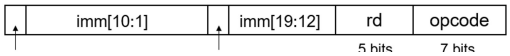
- SB format:

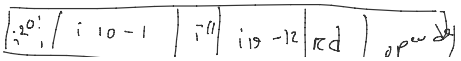




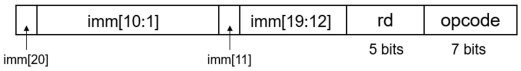
- PC-relative addressing
  - Target address = PC + immediate × 2

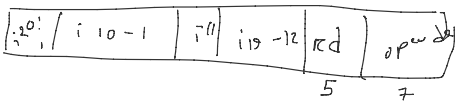
- UJ format: JAL





- UJ format:

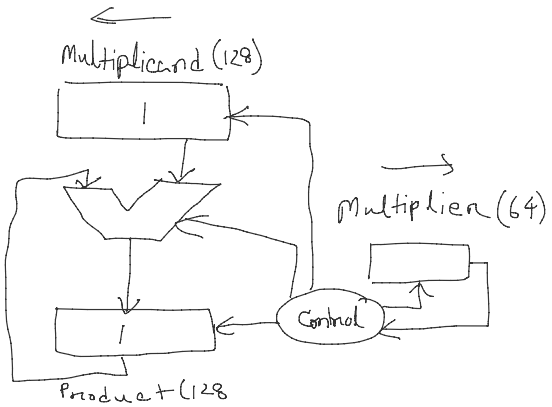




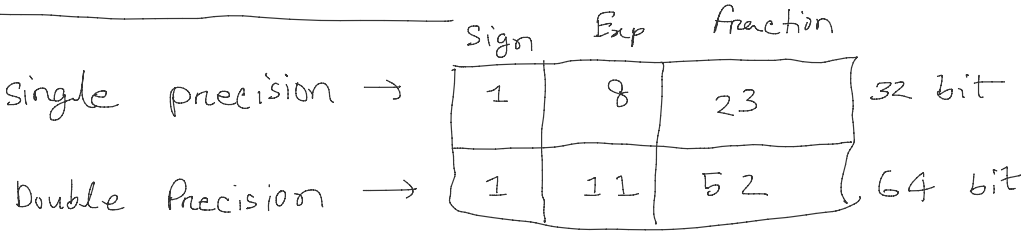
Name (Field Size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

Chapter 3 :-

✓ Multiplication Hardware :-



IEEE-754 format :-



Chapter 4 :-

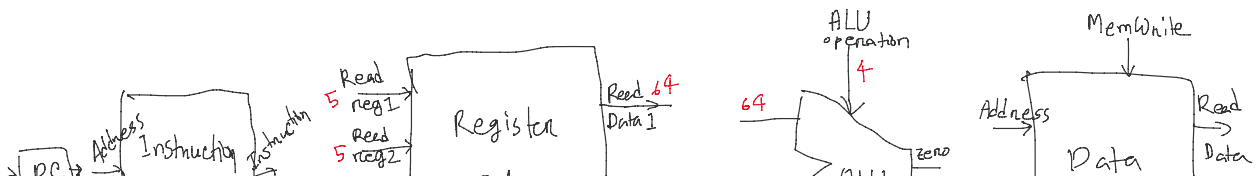
Fetch → Decode → Execute → memory → Write Back

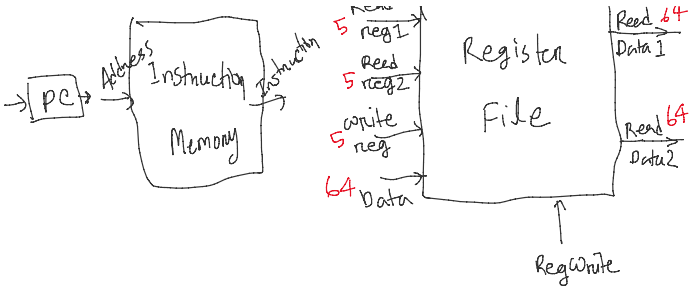
All types of operations

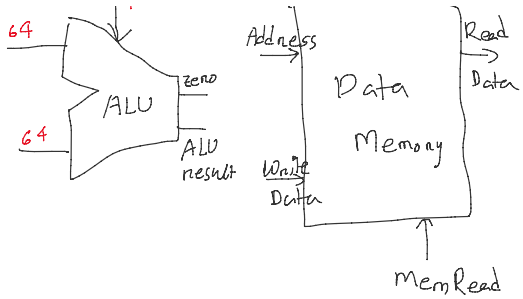
↓

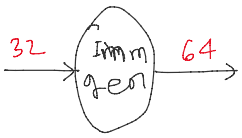
Then decide what to do seeing the type of operation

- Arithmetic
- Load/Store
- Branch

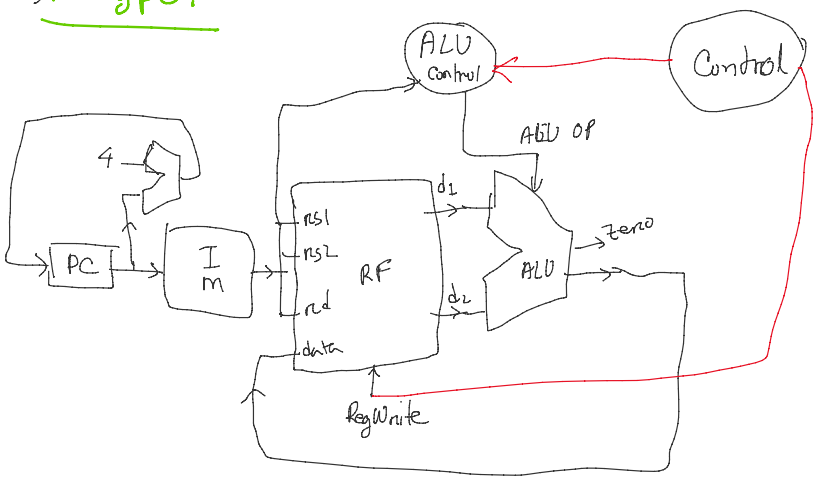




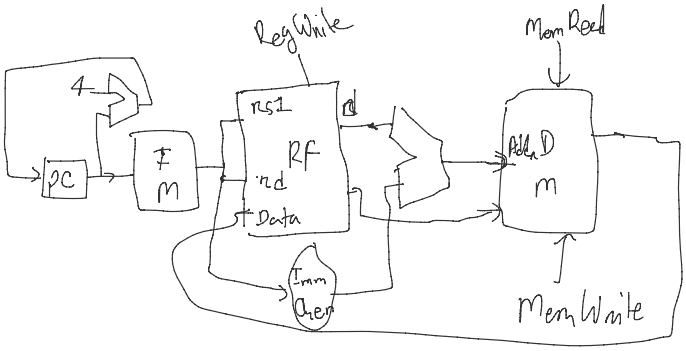




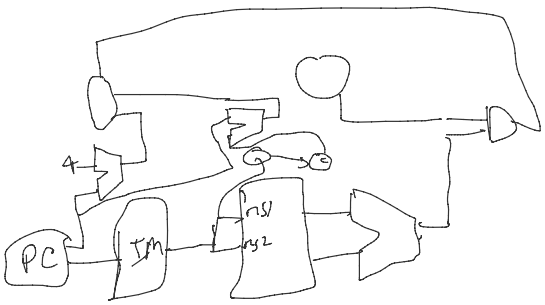
R-type :- Arithmetic



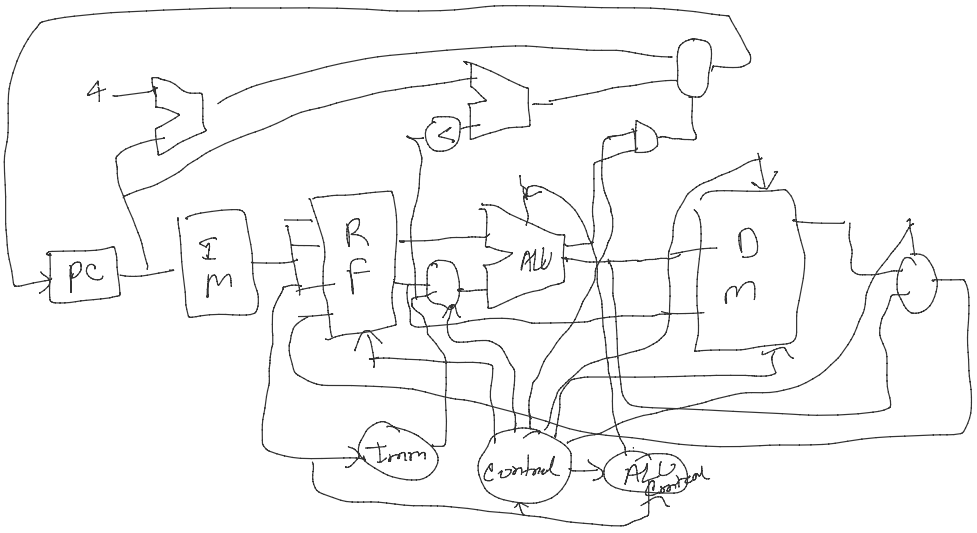
I-type :- load/store



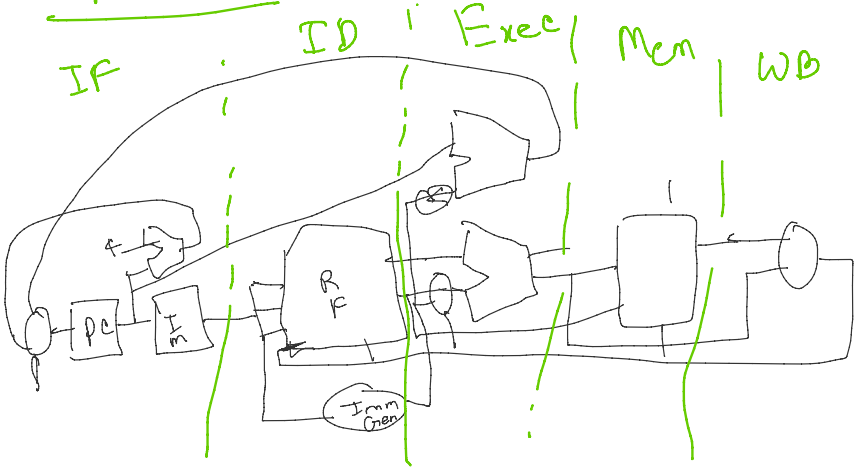
Branch :-



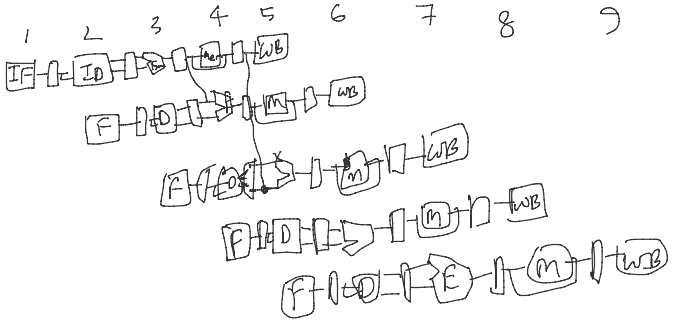
Full Datapath :-



Pipeline :-



```
1   sub  x2, x1,x3
2   and  x12,x2,x5
3   or   x13,x6,x2
4   add  x14,x2,x2
5   sd   x15,100(x2)
```



IC = 5

$CPI = \frac{9}{5} = 1.8$