$$T = N \times CPI \times t_c$$
$$CPI_{mem} = CPI_{hit} + m \times penalty_{miss}$$

$$P\ (watts) = Energy\ (joules^1) \div Time\ (seconds)$$

Instruction set	SIMD length	SP_Flop/cycle (32 bits)	DP_Flop/cycle (64 bits)
SSE2	128-bits	4	2
AVX	256-bits	8	4
AVX2 (includes FMA)	256-bits	8 / 16 (FMA)	4 / 8 (FMA)
AVX-512 (includes FMA)	512-bits	16 / 32 (FMA)	8 / 16 (FMA)

Multiprogramming:
Serial: aplicacions una darrera l'altra
Single processor, multiprogrammed: Es van intercanviant
Multiple processors, multiprogramed: S'executen amb el mateix número de cores fins al final
Multiplex: intercalar sense compartir cores

Speed–up $S_p = T_1 \div T_p$
Efficiency $E_p = S_p \div P$

$$S_p = \frac{T_1}{T_p} = \frac{T_1}{(1-\varphi) \times T_1 + (\varphi \times T_1/P)}$$
$$S_p = \frac{1}{((1-\varphi) + \varphi/P)}$$

$$S_{p\to\infty} = \frac{1}{(1-\varphi)}$$

Scaling:
Increase the number of processors P with constant problem size (strong scaling → reduce the execution time)
Increase the number of processors P scaling (linearly, quadratically, ...) problem size with P (weak scaling → solve larger problem)

Arithmetic and logic: `add`, `sub`, `and`, `or`, `xor`, `sra`, `srl`, `sll`

$op\ rd,\ rs1,\ rs2 \rightarrow rd = rs1\ op\ rs2$ with
$op = \{+,\ -,\ and,\ or,\ xor,\ shift\ arith/logical\ right/left\}$

Memory access: `lw`, `sw` (but also `lb`, `sb`, `lh` and `sh`)

$lw\ rd,\ disp(rs1) \rightarrow rd = mem[rs1 + disp]$
$sw\ rs2,\ disp(rs1) \rightarrow mem[rs1 + disp] = rs2$

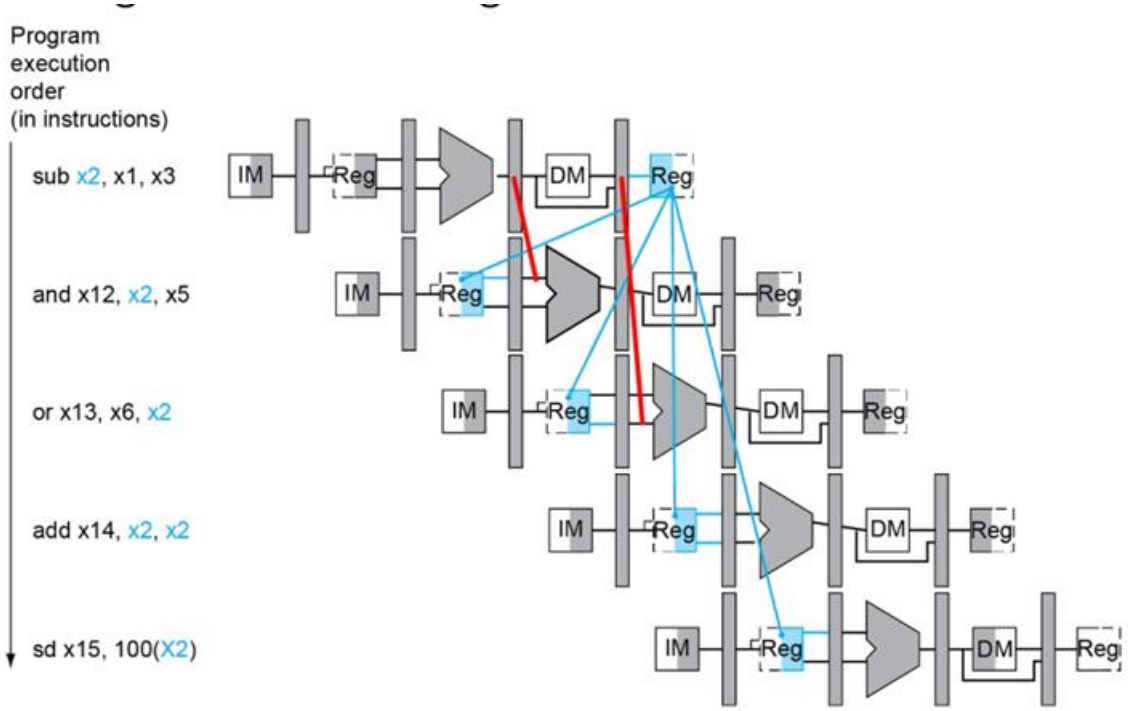
Control flow change: `beq` (but also `bne`, `blt`, `bge`)

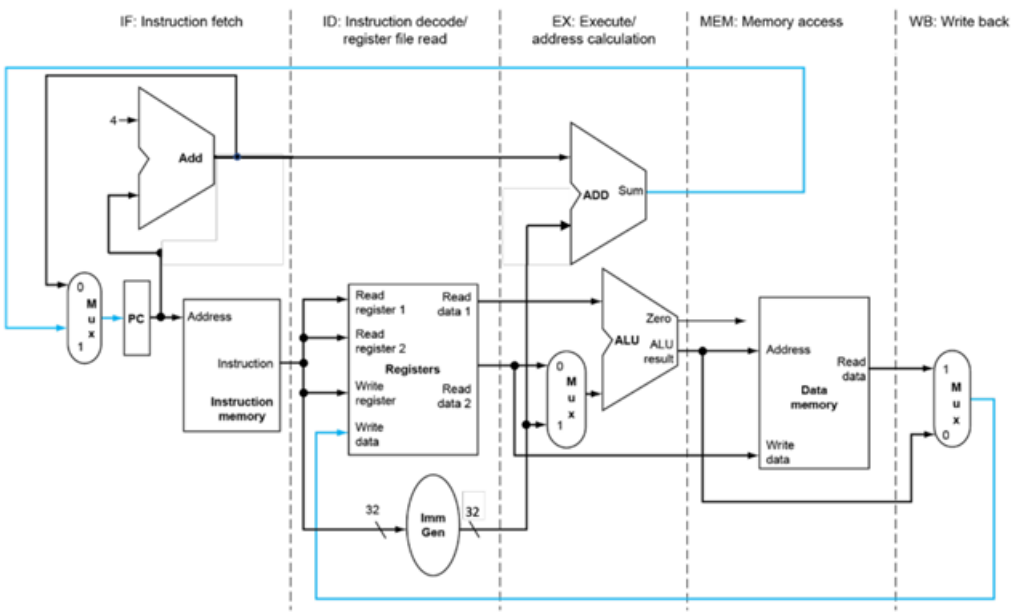
$beq\ rs2,\ rs1,\ disp \rightarrow if\ (rs2 = rs1)\ then\ PC = PC + disp$

The address of the instruction to execute: `PC` register, which needs to be incremented by 4 to point to the next instruction in the program (by default)
A memory with the instructions of the program
All instructions in the ISA subset read 1 or 2 operands from registers (located in the register file: x0–x31)
All instructions in the ISA subset perform one ALU operation (arithmetic: add/sub; logic: and/or; or address calculation +: lw/sw) or 2 (beq: comparison and address calculation)
lw/sw instructions read/write data from/to memory
All instructions in the ISA subset, except lw and beq, write 1 result in a register (located in the register file)

- $t_c = 200\ ps$
- $CPI = 1$ ideally, except when:
  - `lw` in which the compiler can not reorder instructions $\rightarrow$ 2 (one cycle penalty due to stalled cycle)
  - beq when branch is misspredicted $\rightarrow$ $1 + x$ (being x the penalty cycles due to instructions nullified in the wrong path, 3 in our design)

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, and, or)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch	200 ps	100 ps	200 ps			500 ps



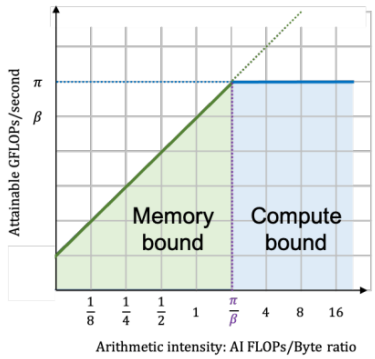


```
void dgemm(int n, double *A, double *B, double *C) {
    double sum;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) {
            sum = 0.0;
            for (int k = 0; k < n; k++)
                sum = sum + A[i*n+k] * B[k*n+j];
            C[i*n+j] += sum;
        }
}
```

$$AI = \frac{Flops}{Bytes} = \frac{2 \times n^3}{[3\ loads + 1\ store] \times n^2 \times 8\ bytes} = \frac{2 \times n^3}{32 \times n^2} = \frac{n}{16}$$

Attainable performance (GFLOPs/second) = $\min(\pi, \beta \times AI)$

- π GFLOPs/second = peak FP performance
- β GByte/second = peak memory bandwidth
- AI = Arithmetic intensity, i.e. FLOPs per Byte ratio



Example, bandwidth for UPI clocked at 5.2 GHz is 41.6 GB/s:

$$5.2\ GHz \times 2\ edges/cycle \times 2\ directions \times 16\ bits \div 8\ bits/byte$$

Write update

Action	P0 $	hit/miss	P1 $	hit/miss4	P2 $	hit/miss3	LLC[X]	hit/miss2	mem[X]	mem[Y]
									0	3
P0 load X	0	miss					0	miss	0	3
P1 load X			0	miss			0	hit	0	3
P0 store X	1	hit	1	update			1	update	0	3
P2 load X	1		1		1	miss	1	hit	0	3
P2 store X	2	update	2	update	2	hit	2	update	0	3
P1 load X	2		2	hit	2		2		0	3
P0 load Y	3	miss	2		2		2		0	3

Write invalidate

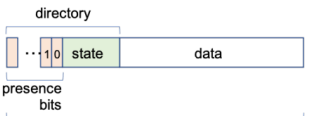
Action	P0 $	hit/miss	P1 $	hit/miss4	P2 $	hit/miss3	LLC[X]	hit/miss2	mem[X]	mem[Y]
									0	3
P0 load X	0	miss					0	miss	0	3
P1 load X			0	miss			0	hit	0	3
P0 store X	1	hit	--	inval			--	inval	0	3
P2 load X	1		--		1	miss	1	update	0	3
P2 store X	--	inval	--		2	hit	--	inval	0	3
P1 load X	--		2	miss	2		2	update	0	3
P0 load Y	3	miss	2		2		2		0	3

(assume this load causes eviction of X in P0 $)

Coherence protocols:

- Write-update: writing processor broadcasts **the whole line** with the new value and forces all others to update their copies
- Write-invalidate: writing processor forces all others to invalidate their copies; **the line** with the new value is provided to others when requested or when flushed from cache

- **Each** line in LLC can be in three different states (**2 bits**):
  - Modified (M): dirty copy of the line in one local cache
  - Shared (S): clean copy of the line in one or more local caches
  - Uncached (U): no copies of line in any cache
- **Each** line in LLC includes a presence bit list to indicate which local caches have copies



Memory channel: I Data is transferred on both the rising and falling edges of the clock signal (at a given frequency), with a transfer rate of frequency × 2 (dual rate) I Data is transferred 64 bits at a time I Bandwidth is transfer rate × 64 (number of bits transferred) ÷ 8 (number of bits/byte) I Multiple channels to increase total bandwidth: number of channels × bandwidth per channel I Multiple DIMM per channel to increase capacity

MM1: Memory Module 1 (Home)
MC0: Memory Cache 0 (Processor Cache 0)
LLC0: Last Level Cache 0
CHA: Cache Home Agent
MHA1: Memory Home Agent 1
r0: Read from Processor 0
w0 (1): Write from Processor 0 with sequence number 1
r2: Read from Processor 2
w1 (2): Write from Processor 1 with sequence number 2NoC: Network on chip
NIC: Network Interface Card
NUCA: Non uniform cache access
UPI: Ultra Path Interconnect
NUMA: Non uniform memory access
DIMM: Dual Inline Memory Module

Memory Access	hit/miss	Cache line value			Directory in LLC				Value in LLC	Value in MM	Comments
		cache2	cache1	cache0	State	Presence 2	Presence 1	Presence 0			
		-	-	-	-	-	-	-	-	0	Variable in MM
r0	miss			0	S	0	0	1	0	0	From MM to P0 and LLC
w0 (1)	hit			1	M	0	0	1	-	0	MM and LLC not updated
r1	miss		1	1	S	0	1	1	1	0	Line from P0, LLC updated
w2 (2)	miss	2	-	-	M	1	0	0	-	0	Line from LLC, P0/P1 invalidated
r1	miss	2	2	-	S	1	1	0	2	0	Line from P2, LLC updated
w0 (5)	miss	-	-	5	M	0	0	1	-	0	Line from LLC, P1/P2 invalidated
w1 (3)	miss	-	3	-	M	0	1	0	(5) -	0	Line from P0, P0 inval., LLC upd.
r2	miss	3	3	-	S	1	1	0	3	0	Line from P1, LLC updated
r1	hit	3	3	-	S	1	1	0	3	0	
r0	miss	3	3	3	S	1	1	1	3	0	Line from LLC

Each line in main memory has one entry in the **directory**:

- **State**: they track the state of cache lines in its memory, as before MSU (**2 bits**)
- **Presence bits**: tracks the list of sockets having a copy of a line (**as many bits as sockets in the node**)

Memory Access	hit/miss	Value in		Directory in LLC0			Value in LLC0	Comments	Value in		Directory in LLC1			Value in LLC1	Directory in MM1			Value in MM1
		MC1	MC0	State	Presence 1	Presence 0			MC3	MC2	State	Presence 3	Presence 2		State	Presence 1	Presence 0	
		-	-	-	-	-		Initial state and value in MM1 (home)	-	-	-	-	-	-	U	0	0	0
r0	miss		0	S	0	1	0	From MM1 to MC0 and LLC0							S	0	1	0
w0 (1)	hit		1	M	0	1	-	Notify MHA1							M	0	1	0
r2	miss		1	S	0	1	1	Line from MC0, LLC0/MM1 updated		1	S	0	1	1	S	1	1	1
w1 (2)	miss	(1) 2	-	M	1	0	-	Line from LLC0, MC0 invalidated, notify MHA1, MC2/LLC1 invalidated		-	-	-	-	-	M	0	1	1
r2	miss	2	-	S	1	0	2	Line from MC1, LLC0 updated, MM1 updated		2	S	0	1	2	S	1	1	2
w0 (5)	miss	-	(2) 5	M	0	1	-	Line from LLC0, MC1 invalidated, notify MHA1, MC2/LLC1 invalidated		-	-	-	-	-	M	0	1	2
w2 (3)	miss	-	-	-	-	-	-	Line from MC0, MC0/LLC0 invalidated		(5) 3	M	0	1	-	M	1	0	5
r1	miss	3	-	S	1	0	3	Line from MC2, LLC1/MM1 updated		3	S	0	1	3	S	1	1	3
r2	hit	3	-	S	1	0	3	No changes		3	S	0	1	3	S	1	1	3
r0	miss	3	3	S	1	1	3	Line from LLC1, no changes in directory		3	S	0	1	3	S	1	1	3

Property: the number of links in a leaf switch that go down to the nodes equals the number of links that go to its parents (spine)

With two levels of 36-port switches the maximum would be 18 spine switches and 36 leaf switches for a total of $36 \times 18 = 648$ nodes

Efficiency $= (Flops\ per\ node \times \#nodes) \div ((power\ per\ node+) \times \#nodes + (switch\_power \times \#switches))$

KB: 2^10 Bytes
MB: 2^20 Bytes
GB: 2^30 Bytes

```
#pragma omp parallel for private(val) reduction(+:sum)
for (row = 0; row < Rows; row++)
{
   bla bla code
   sum += val;
}
```

OMP
Critical – Bloqueja tota la secció, overhead
Atomic – Només bloqueja l'element concret (pot ser una posició en un vector)
Barrier

MPI_Reduce

- Applies a reduction operation on all tasks in the communicator `comm` and places the result in one task (`root`). `count` elements starting at `sendbuf` from each process are individually aggregated, applying operation `op`, into `recvbuf` in `root`.
- `MPI_Reduce (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)`
- Some predefined reduction operations
  - `MPI_SUM, MPI_PROD`
  - `MPI_MAX, MPI_MIN, MPI_MAXLOC, MPI_MINLOC`

Private(i)/First private{} – Variable per a cada thread no inicialitzada/inicialitzada;
En un parallel for, qualsevol variable que no sigui la del bucle, s'ha de declarar privada "private(j)". Sinó per defecte serà Shared()
Reduction – Opcions: `+, -, *, &, |, ^, && i ||; (operador: variable1, ..., variableN)`

Schedule: static – automàtic, per quan les iteracions tarden el mateix; dynamic – per quan les operacions tarden diferent; en les dues opcions: schedule(_, chunk) per indicar la quantitat manualment

MPI:
- MPI_Comm_size(MPI_COMM_WORLD, &variable)
- MPI_Comm_rank
- MPI_Barrier(MPI_Comm comm);
- MPI_Send(void* message, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm, communicator);
- MPI_Recv(void* data, int count, MPI_Datatype datatype, int from, int tag, MPI_Comm comm, MPI_Status* status);

MPI_Bcast

- Broadcasts (sends) a message from the process with rank `root` to all other processes in communicator `comm`.
- `MPI_Bcast (void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`

MPI_Scatter

- Distributes distinct messages from a single source rank (`root`) to each other rank in communicator `comm`.
- `MPI_Scatter (void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm)`

MPI_Gather

- Gathers distinct messages from each rank in communicator `comm` to a single destination rank (`root`).
- `MPI_Gather (void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`

MPI_Allreduce

- All ranks get the result of the reduction operation that would be produced by `MPI_Reduce`.
- `MPI_Allreduce (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`