CS420 – Lecture 2

Raghavendra Kanakagiri
Slides: Marc Snir

Spring 2023

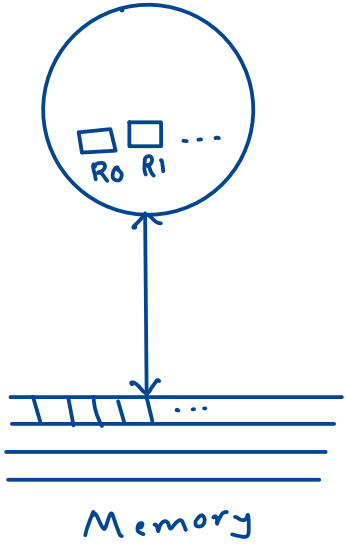
Recap

It's also a branch structure

```
sum = 0;
for (int i=1; i<=N; i++)
{
    sum += A[i];
}
```

```
R0 : sum
R1 = i
R2 = N
R3 = addr(A)
```

```
.loop
ld   R4,[R3]
add  R0,R0,R4
add  R1,R1,#1
add  R3,R3,#4
cmp  R1,R2
bne  .loop
```



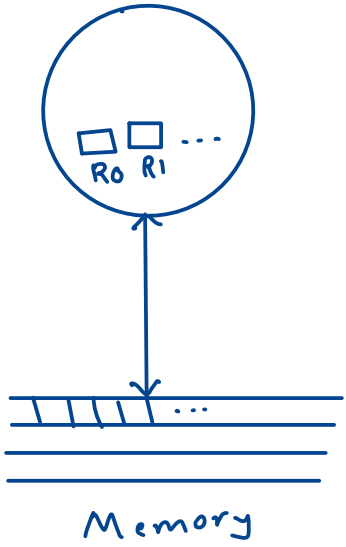
Recap

C/C++

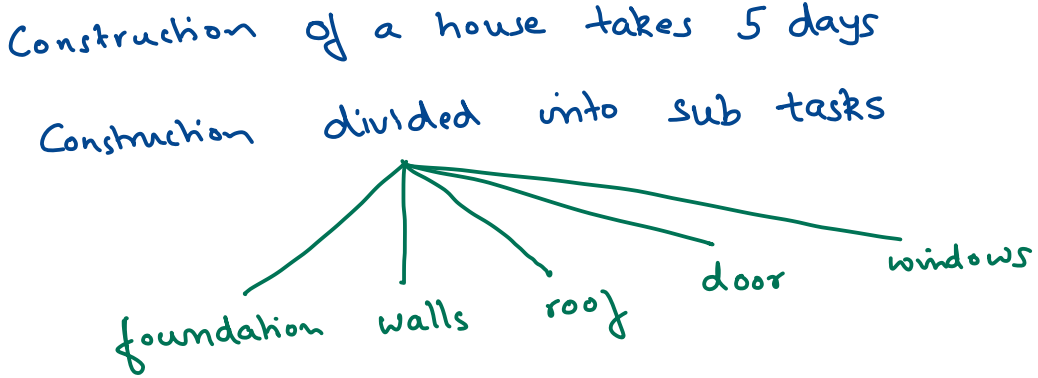
Assembly

Machine Code

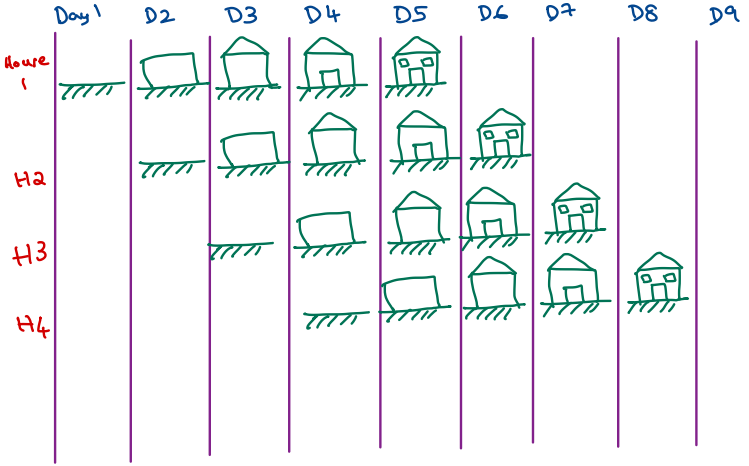
Next: Instruction Level Parallelism (ILP)



Pipelining


Construction of a house takes 5 days
Construction divided into sub tasks
foundation
walls
roof
door
windows

Pipelining



→ After 4 days, it appears as if a new house is constructed/completed every single day

→ Pipelined

$$\text{Speedup} = \frac{\text{old time}}{\text{new time}}$$

To construct n houses

without pipelining: $n \cdot 5 = 5n$ days

with pipelining: $(5-1) + n \cdot 1 = (5-1) + n$ days

$(5-1)$: no house (startup cost)

$n \cdot 1$: one house completed every single day

Speedup : $\frac{5n}{(5-1)+n}$

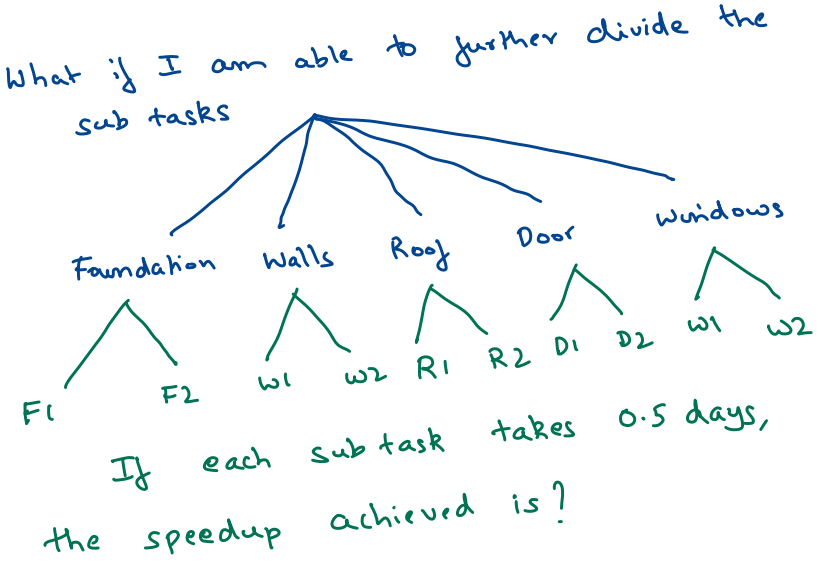
If n is say 1 million

Speedup = $\frac{5M}{1M+3} \approx 5$

For sufficiently large n, $(5-1)$ is negligible

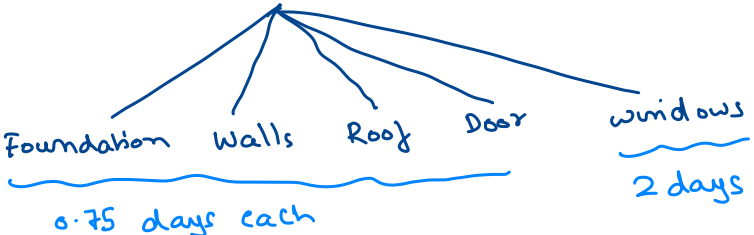
Speedup = $\frac{5n}{n} = 5$

Pipelining


What if I am able to further divide the sub tasks
Foundation
Walls
Roof
Door
windows
F1
F2
W1
W2
R1
R2
D1
D2
W1
W2
If each sub task takes 0.5 days,
the speedup achieved is?

Pipelining

What happens if the division is as below



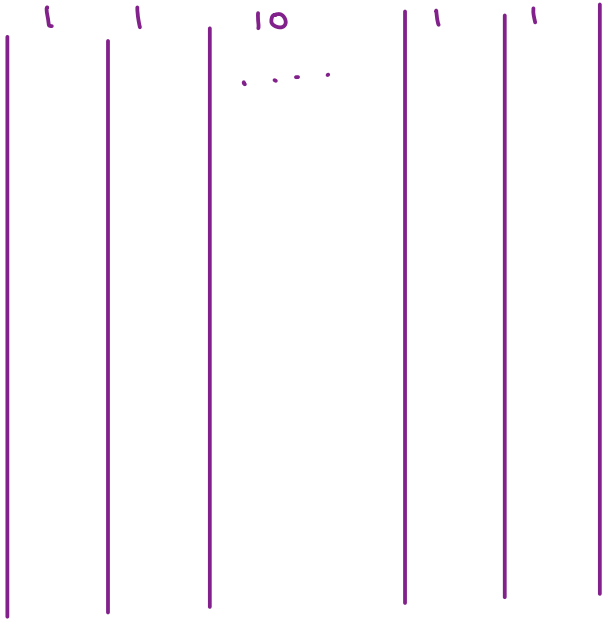
The first 4 tasks take only 0.75 days each.

Placing the windows takes 2 days

A single house still takes 5 days to complete

Speedup = ?

Pipelining



Speedup = ?

New house is finished every 10 days.

Why not 12 days?

Instruction Level Parallelism

IF Instruction Fetch

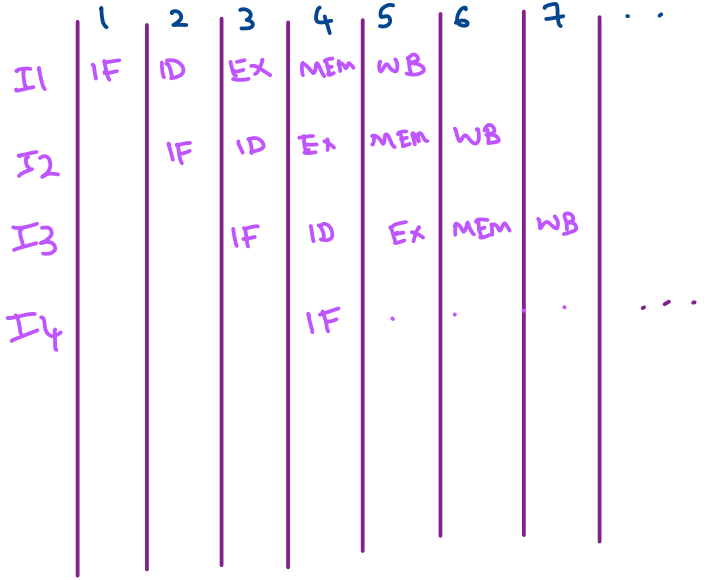
ID/RF Instruction Decode

EX Execute

MEM Memory

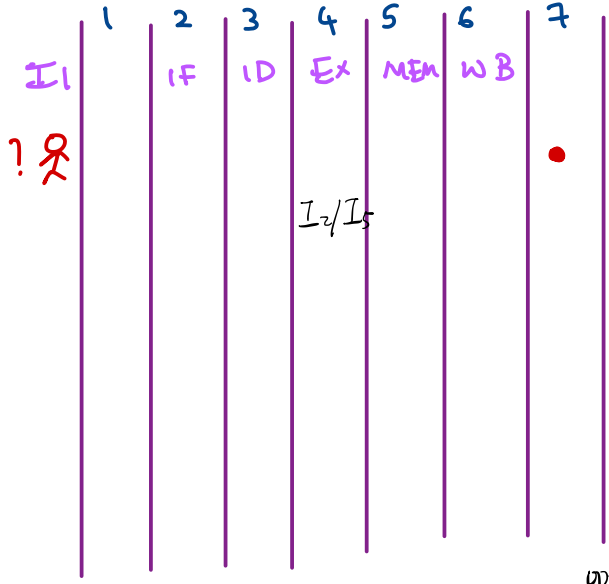
WB Write back

Branch Hazard



I1: a = b + c
I2: e = a + d
I3: f = e * 3
I4: g = f + h + a

Branch Hazard



I1: if(x > 0) {

I2: a = b + c;

I3: b = 8 + e;

}

I4: else {

I5: a = 8 + c;

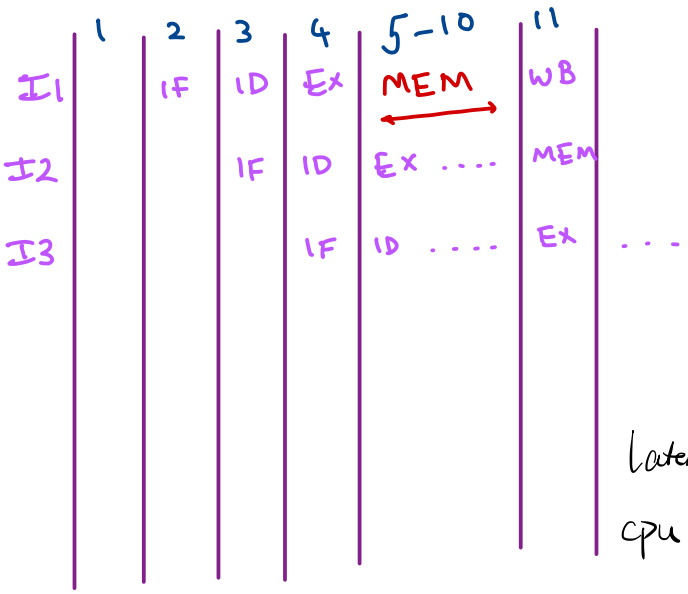
I6: b = e + c;

}

branch prediction

What if prediction is wrong and operations modify the values?

Memory Operation



• loop

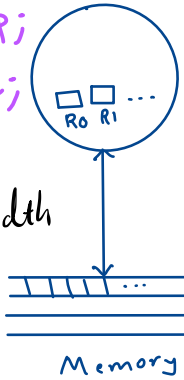
I1: x = a[i];

I2: y = x + 8;

I3: i = i + 1;

I4: z = 3 + p;

I5: j = z + x;



latency v.s. bandwidth

cpu v.s. gpu

Recap

Pipeline

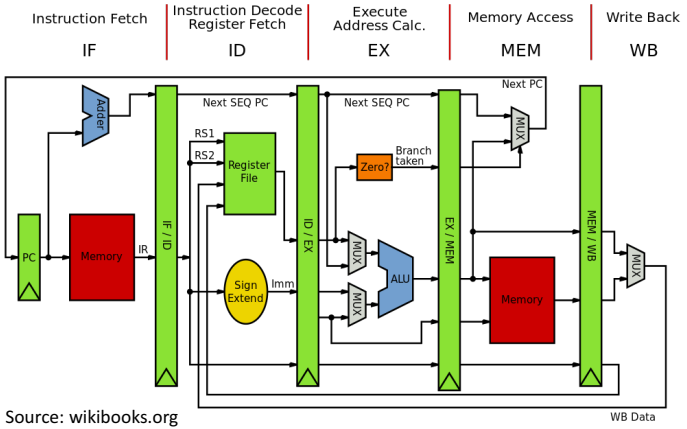
Branches

Memory

Pipelining

- Pipeline increases *throughput* – number of houses constructed per day
- Pipeline does not decrease *latency* (slightly increases it) – time for coordination
- Pipeline enables more specialization (assembly line)

Reducing Gate Delays
Pipelined Processor



Source: wikibooks.org

Dependencies cause pipeline bubbles

- *pipeline bubble:* empty stage in pipeline
- Compute `a[i]=a[i-1]*s;` assume only constrained resource is floating point pipeline.

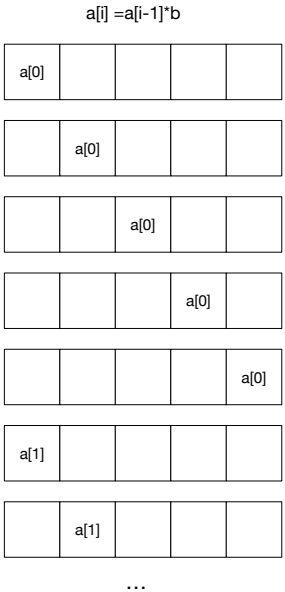
can't compute in parallel. because a[i] depends on a[i-1].

Dependencies cause pipeline bubbles

- *pipeline bubble:* empty stage in pipeline
- Compute `a[i]=a[i-1]*s`; assume only constrained resource is floating point pipeline.
- Dependency distance one implies only one stage of the pipeline is utilized (utilization $1/m$, if pipeline has depth m)

Dependencies cause pipeline bubbles

- *pipeline bubble:* empty stage in pipeline
- Compute `a[i]=a[i-1]*s`; assume only constrained resource is floating point pipeline.
- Dependency distance one implies only one stage of the pipeline is utilized (utilization $1/m$, if pipeline has depth m)

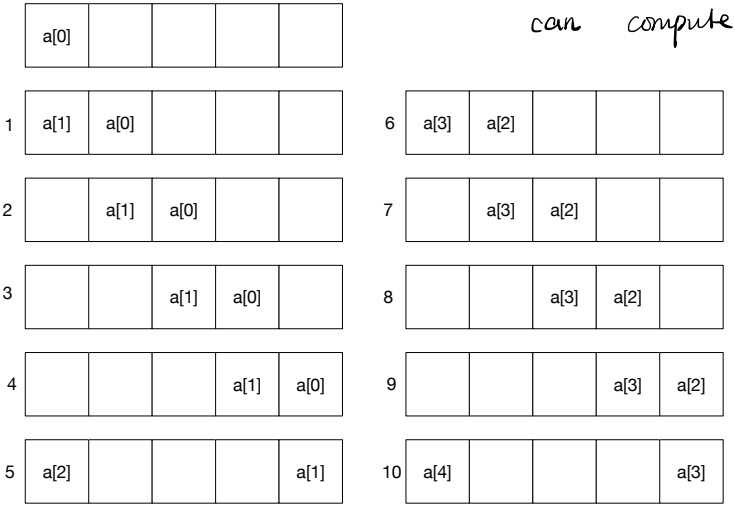


Dependency distance 2, utilization $2/m$

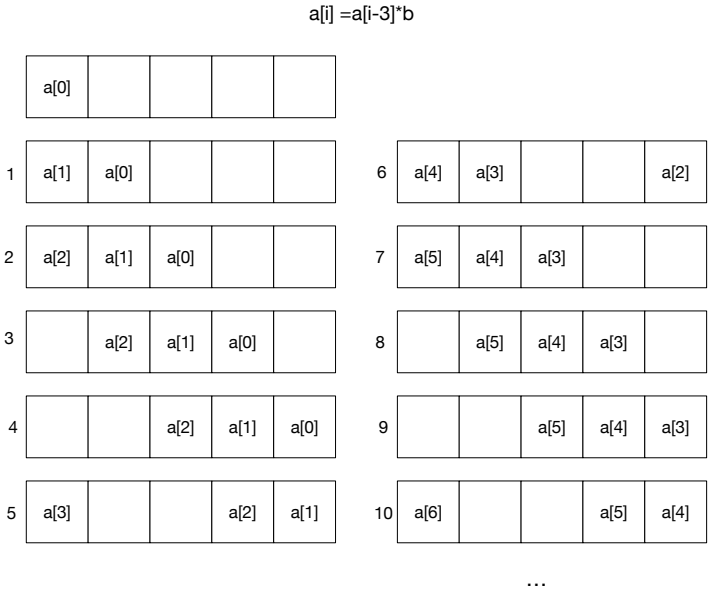
a[i] =a[i-2]*b

dependency lag ≥ 2

can compute in parallel way.



Dependency distance 3, utilization $3/m$; ...



True Dependences

```
for(i=2;i<N;i++)
a[i]=s*a[i-2];
```

- *Loop dependence*: iteration i depends on iteration $i-2$ (dependence distance 2).
- $\Rightarrow$ Can compute at same time iteration i and $i-1$, but no more.
- Need to optimize code, in order to compute simultaneously two iterations

unroll to reduce branches

```
for(i=2;i<N-1;i+=2) {
a[i]=s*a[i-2];
a[i+1]=s*a[i-1]
}
if (N%2) a[N-1]=s*a[N-3]
```

more instructions in a branch. And instructions are parallel.

- More efficient code: Fewer branches, can pipeline (but need extra code to handle odd N); good if N is large
- Compiler will do this *loop unrolling* on its own, if possible (with higher optimization levels)
- *True Dependence:* Read after write (RAW) - producer/consumer dependence – Later iteration uses result of previous iteration.

not necessary in one step.

False dependences 1

```
for(i=0;i<N;i++)
a[i]=s*a[i+1];
```

- *Anti dependence:* Write after read (WAR) – Later iteration updates variable used by earlier iteration. Seems to prevent pipelining
- Not "true dependence" – can be avoided by using temporary variables:

```
for(i=0;i<N;i+=2) {
t1=a[i+1];
t2 =a[i+2];
a[i] = s*t1;
a[i+1]=s*t2;
}
```

Then modifying a[i+1] won't affect a[i].

There are cases compiler can't automatically do it.

- Compiler will do this, using registers as temporary variables
- Code can be unrolled 3,4,... times; but, if unroll too much, may not have enough registers (register pressure)

False dependences 2

```
for(i=0;i<N;i++)
s=a[i];
```

- *Output dependence:* Write after write (WAW) –Updates need to occur in the right order
- Can be (usually) avoided by not performing first write:

```
s=a[N-1];
```

Register renaming

need to learn assembly language.

```
1.ldr r1,[#1024]
2.add r1, r1, #2
3.str r1, [#148]
4.ldr r1, [#2090]
5.add r1, r1, #4
6.str r1 [#3000]
```

have false (WAR) dependence on register r1
Compiler can solve problem by using another register

```
1.ldr r1,[#1024]
2.add r1, r1, #2
3.str r1, [#148]
4.ldr r2, [#2090]
5.add r2, r2, #4
6.str r2 [[#3000]
```

4-5-6 can execute concurrently with 1-2-3

900 processer

Problem: Number of registers small (e.g., 16) – compiler runs out of registers for more complicated code.
Solution: Hardware uses "hidden" registers during execution

```
1.ldr r1,[#1024]
2.add r1, r1, #2
3.str r1, [#148]
\\ hardware allocates a new copy
\\ of r1; new copy used subsequently
4.ldr r1, [#2090]
5.add r1, r1, #4
6.str r1 [[#3000]
```

Control dependence

can't execute $i=5$ until $i=4$ because it may break.

But if it breaks then roll back.

```
for(i=0;i<N;i++)
if(a[i][0] ==key) {
value = a[i][1];
break
}
```

- Whether iteration $i+1$ executes depends on the result of the test in iteration i.
- Can start, speculatively, to execute iteration $i+1$ – e.g., load `a[i+1][*]` – as long as speculation can be undone, if wrong.
- Processors do *branch prediction* and speculate on the most likely branch

Loop fusion

```
for(i=0;i<N;i++)
    a[i]=a[i]+b[i];
for(i=0;i<N;i++)
    b[i]=b[i]*s;
```

We *fuse* the two loops

```
for(i=0;i<N;i++) {
    a[i]=a[i]+b[i];
    b[i]=b[i]*s;
}
```

Loop fusion

- reduces number of branches;
- usually reduce number of loads (`b[i]` will be loaded only once);
- enable further optimizations

Optimize?

```
#include <stdlib.h>
#define N 25
#define s 27
int main() {
  int i;
  int a[N];
  a[0]=1;
  for(i=1; i<N; i++) {
    a[i]=s+a[i-1];
  }
  return 0;
}
```

loads ↓

true dependency.

Avoid memory accesses

temp — register (faster)

a[i] — memory

source code

```
temp = a[0];
for (i=1; i<N; i++) {
  temp = s+temp;
  a[i]=temp
}
```

→ why exact 4 times?

Running time improved by ×4. Do not need to wait for store to complete before starting next operation.

Optimizations

- *Loop unrolling* can be done manually; a good optimizing compiler will do it automatically
- *Register renaming* is done by hardware – no need to do it
- Loop fusion can be done manually; good optimizing compiler will do it automatically
- *Branch prediction* is done by hardware
- *Executing loads earlier* – compiler optimization
- Hardware can also provide *out-of-order* execution: If instruction will surely execute (no branch) and is ready to execute (no dependence) that is can be executed out of order
- Hardware can provide *speculative execution* (e.g., with branch prediction). If speculation turns out wrong, it is undone.