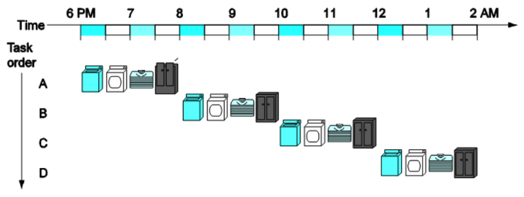
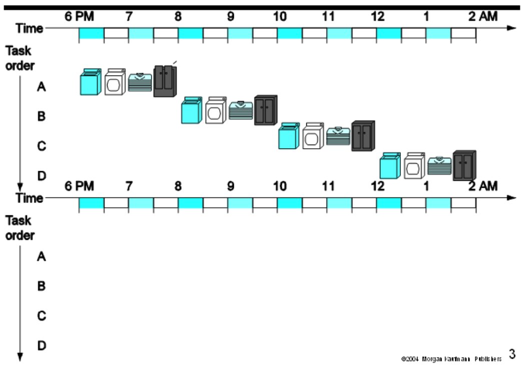
Chapter Six

Motivação

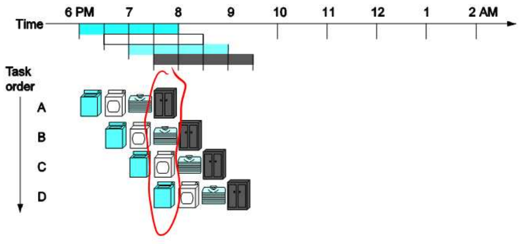


Motivação: Você pode fazer melhor?



Motivação: Quais são as restrições?

Motivação



Você consegue extrapolar para um processador?

- **Período do clock de 200ps**
- **Esqueçam, temporariamente, as dependências entre instruções**
- **Quantas fases seriam?**

- **Quais seriam as fases?**
  - **Dica: lembre-se do controle do datapath multiciclo**
  - **Todas as fases gastam o mesmo tempo?**

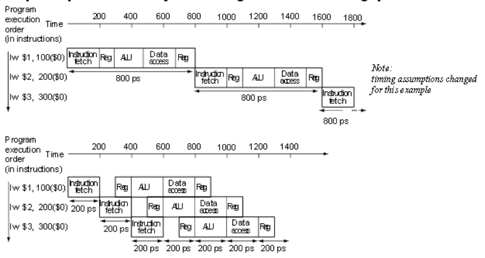
Monte um diagrama da execução das instruções

Tempo

- lw $1, 100($2)
- lw $3, 100($4)
- lw $5, 100($6)
- lw $7, 100($8)
- lw $9, 100($10)

Pipelining

- Improve performance by increasing instruction throughput

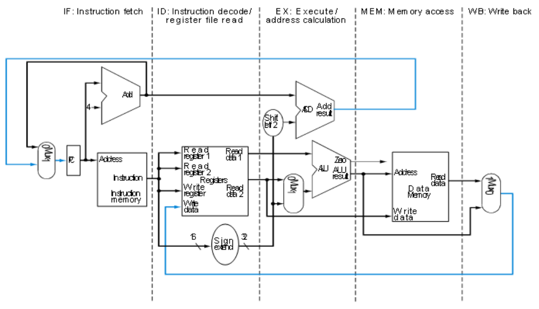


Ideal speedup is number of stages in the pipeline. Do we achieve this?

Pipelining

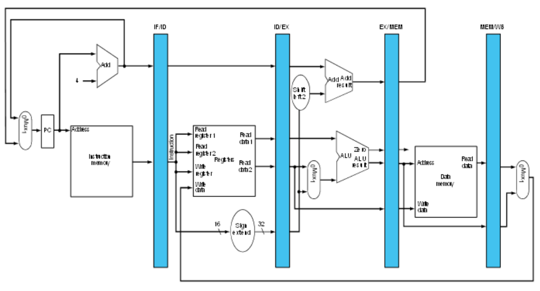
- What makes it easy
  - all instructions are the same length
  - just a few instruction formats
  - memory operands appear only in loads and stores
- What makes it hard?
  - structural hazards: suppose we had only one memory
  - control hazards: need to worry about branch instructions
  - data hazards: an instruction depends on a previous instruction
- We'll build a simple pipeline and look at these issues
- We'll talk about modern processors and what really makes it hard:
  - exception handling
  - trying to improve performance with out-of-order execution, etc.

Basic Idea



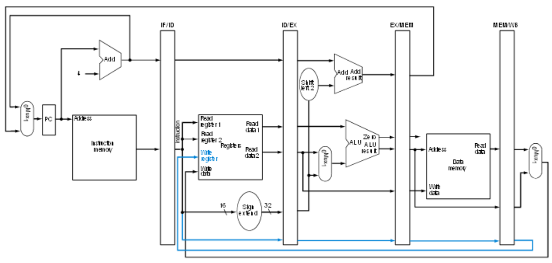
- *What do we need to add to actually split the datapath into stages?*

Pipelined Datapath

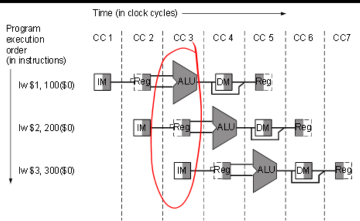


Can you find a problem even if there are no dependencies?
What instructions can we execute to manifest the problem?

Corrected Datapath



Graphically Representing Pipelines

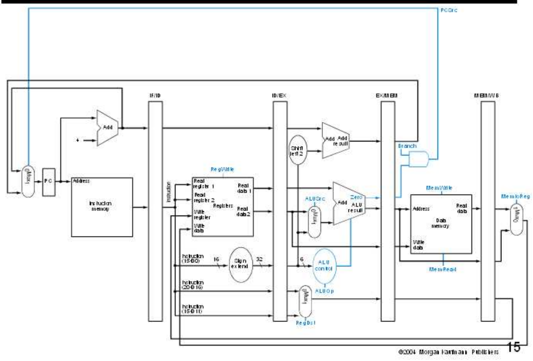


- Can help with answering questions like:
  - how many cycles does it take to execute this code?
  - what is the ALU doing during cycle 4?
  - use this representation to help understand datapaths

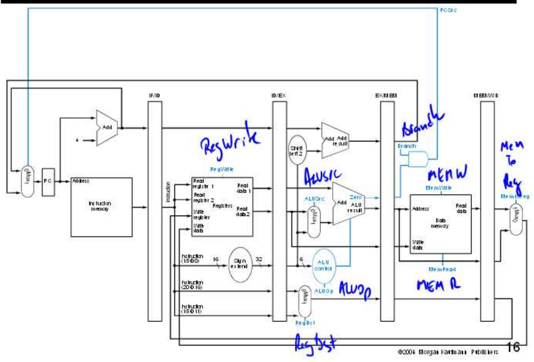
Pipeline control

- We have 5 stages. What needs to be controlled in each stage?
  - Instruction Fetch and PC Increment
  - Instruction Decode / Register Fetch
  - Execution
  - Memory Stage
  - Write Back
- How would control be handled in an automobile plant?
  - a fancy control center telling everyone what to do?
  - should we use a finite state machine?

Pipeline Control



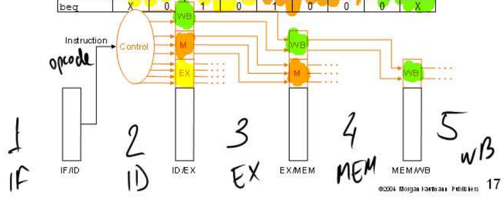
Como você geraria os sinais de controle?



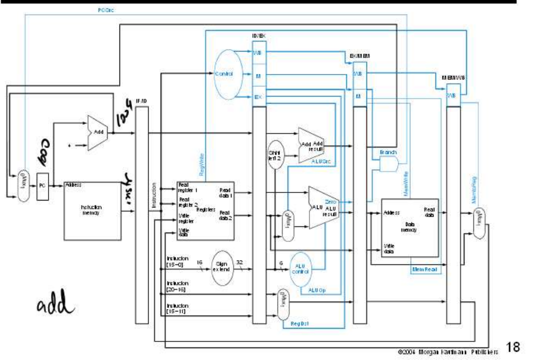
Pipeline Control

- Pass control signals along just like the data

Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

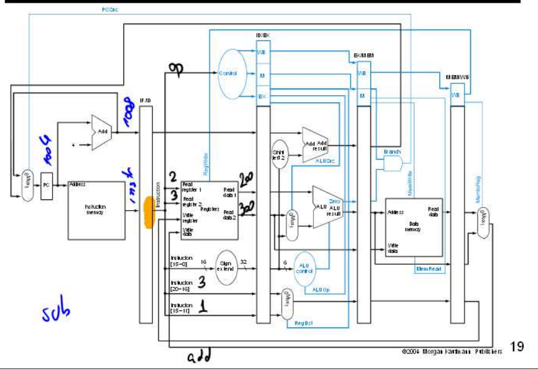


Datapath with Control (1) 1000 add $1,$2,$3



Datapath with Control (2)

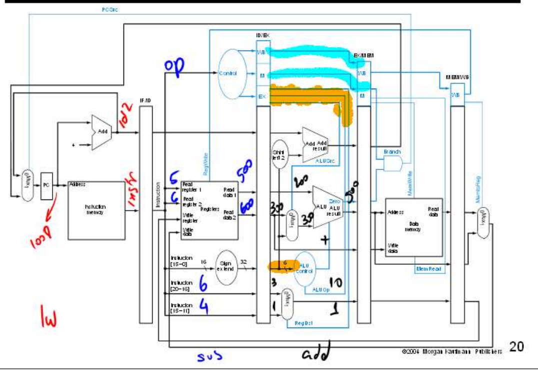
1004 sub $4,$5,$6



©2004 Morgan Kaufmann Publishers 19

Datapath with Control (2)

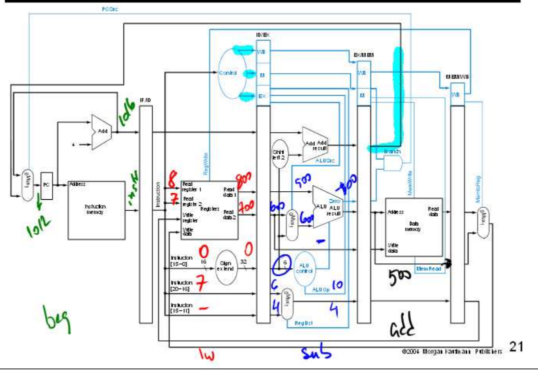
1008 lw $7,0($8)



©2004 Morgan Kaufmann Publishers 20

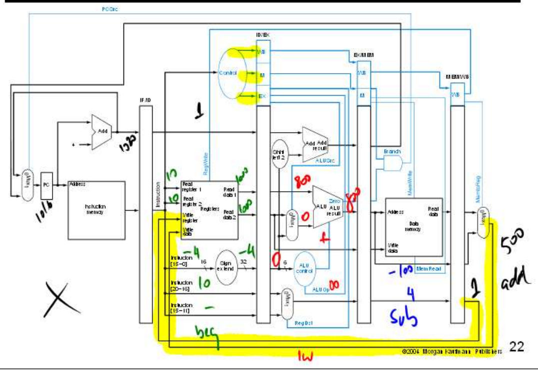
Datapath with Control (4)

1012 beq $10,$10, 1000 LABEL



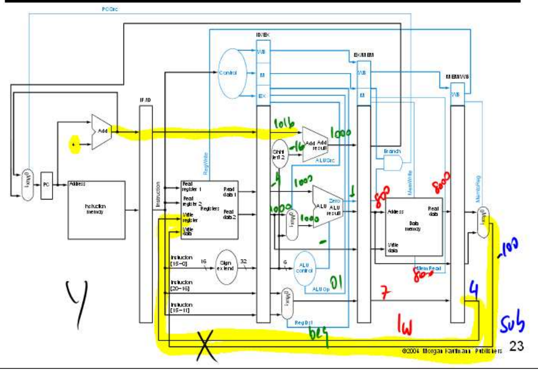
©2004 Morgan Kaufmann Publishers 21

Datapath with Control (5)



©2004 Morgan Kaufmann Publishers 22

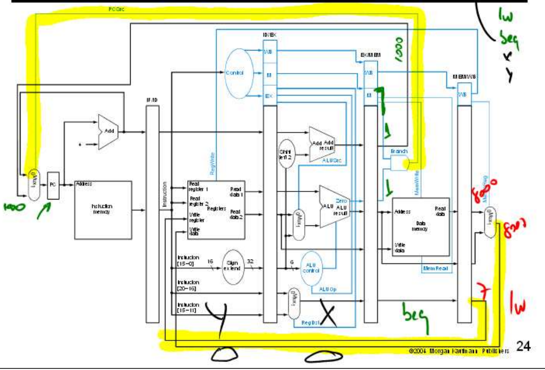
Datapath with Control (6)



©2004 Morgan Kaufmann Publishers 23

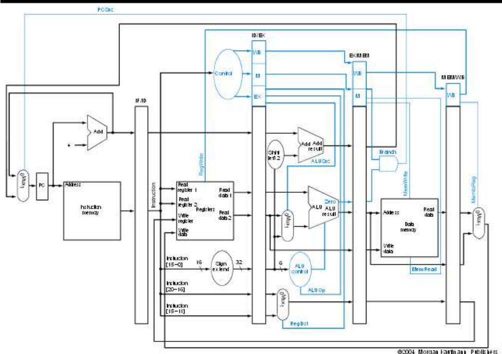
Datapath with Control (7)

add
sub
lw
beq
x
y



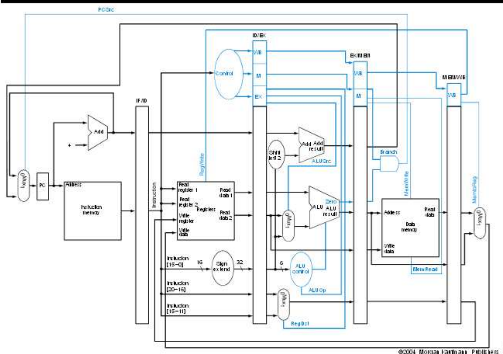
©2004 Morgan Kaufmann Publishers 24

Datapath with Control (8)



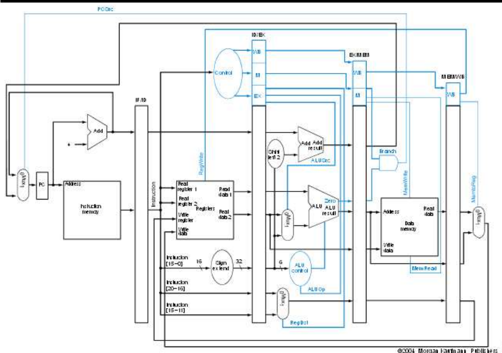
©2004 Morgan Kaufmann Publishers 25

Datapath with Control (9)



©2004 Morgan Kaufmann Publishers 26

Datapath with Control (10)



©2004 Morgan Kaufmann Publishers 27

E se existirem dependências de dados?

- Faça o diagrama de tempo abaixo ignorando as dependências
- Indique as dependências e mostre por que elas atrapalham

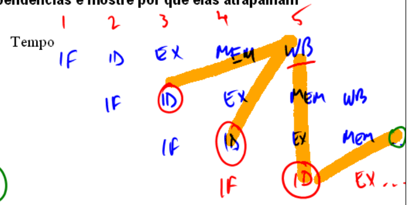
Tempo

1 • add $1, $2, $3

2 • add $4, $1, $2

3 • add $5, $1, $4

4 • add $6, $1, $5



©2004 Morgan Kaufmann Publishers 28

Como resolver por software?

- Modifique o código para que as dependências não atrapalhem
  - Não dá para trocar os registradores!!!
- Qual o impacto desta solução?

- add $1, $2, $3

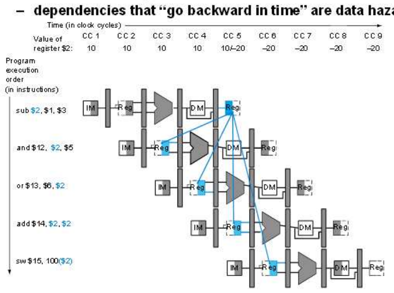
NOP / NOP

- add $4, $1, $2
- add $5, $1, $4
- add $6, $1, $5

©2004 Morgan Kaufmann Publishers 29

Dependencies

- Problem with starting next instruction before first is finished
  - dependencies that "go backward in time" are data hazards



©2004 Morgan Kaufmann Publishers 30

Software Solution

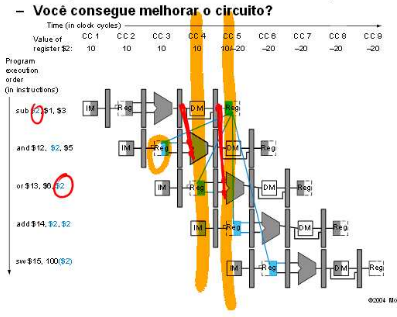
- Have compiler guarantee no hazards
- Where do we insert the "nops" ?

```
sub   $2,  $1,  $3   → NOP/NOP
and   $12, $2,  $5
or    $13, $6,  $2
add   $14, $2,  $2
sw    $15, 100($2)
```

- Problem: this really slows us down!

Solução por hardware

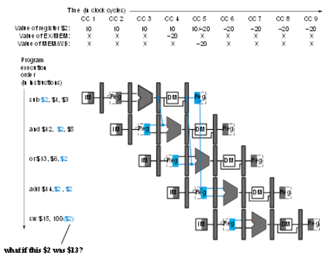
- Quando que os resultados das instruções ficam prontos?
  - Você consegue melhorar o circuito?



Time (in clock cycles)
CC 1 CC 2 CC 3 CC 4 CC 5 CC 6 CC 7 CC 8 CC 9
Value of register $2: 10 10 10 10 10/–20 –20 –20 –20 –20
Program execution order (in instructions)
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
IM
Reg
DM

Forwarding

- Use temporary results, don't wait for them to be written
  - register file forwarding to handle read/write to same register
  - ALU forwarding



Time (in clock cycles)
CC 1 CC 2 CC 3 CC 4 CC 5 CC 6 CC 7 CC 8 CC 9
Value of register $2: 10 10 10 10 10/–20 –20 –20 –20 –20
Value of EX/MEM: X X X –20 X X X X X
Value of MEM/WB: X X X X –20 X X X X
Program execution order (in instructions)
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
what if this $2 was $13?
IM
Reg
DM

Forwarding

- De onde para onde?

MEM → EX

WB → EX

- Funciona sempre?

Não

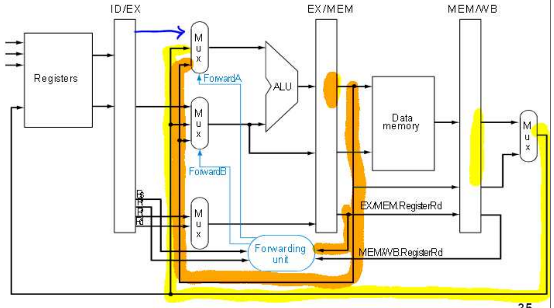
lw $3, 0($2) → NOP

add $1, $3, $4

Forwarding

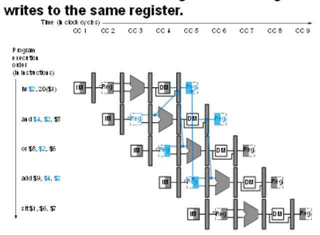
$0

- The main idea (some details not shown)



Can't always forward

- Load word can still cause a hazard:
  - an instruction tries to read a register following a load instruction that writes to the same register.



- Thus, we need a hazard detection unit to "stall" the load instruction

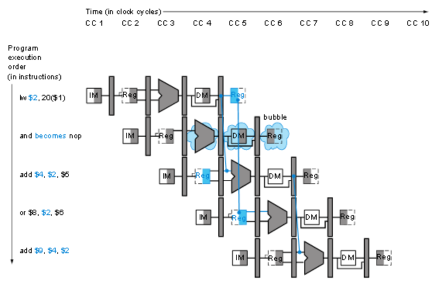
Como resolver quando não funciona?

- Software
- Hardware

©2004 Morgan Kaufmann Publishers 37

Stalling

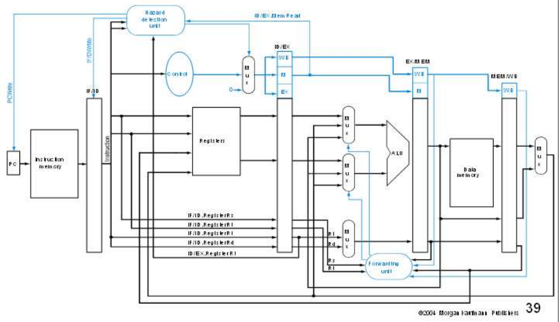
- We can stall the pipeline by keeping an instruction in the same stage



©2004 Morgan Kaufmann Publishers 38

Hazard Detection Unit

- Stall by letting an instruction that won't write anything go forward



©2004 Morgan Kaufmann Publishers 39

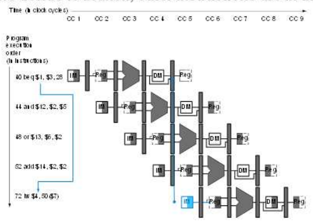
E os saltos?

- O que acontece quando um salto ocorre?
- Quando você descobre que uma instrução precisa saltar?
- E aí?

©2004 Morgan Kaufmann Publishers 40

Branch Hazards

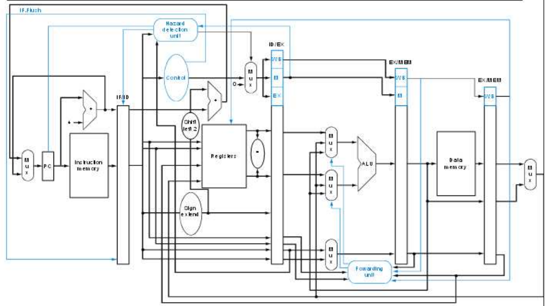
- When we decide to branch, other instructions are in the pipeline!



- We are predicting "branch not taken"
  - need to add hardware for flushing instructions if we are wrong

©2004 Morgan Kaufmann Publishers 41

Flushing Instructions



Note: we've also moved branch decision to ID stage

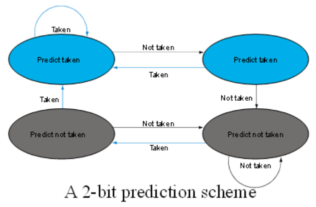
©2004 Morgan Kaufmann Publishers 42

Processadores advinham branches

- Você consegue descobrir como?
  - Dica: Lembre-se que os programas têm laços e as instruções estão em lugares fixos da memória

Branches

- If the branch is taken, we have a penalty of one cycle
- For our simple design, this is reasonable
- With deeper pipelines, penalty increases and static branch prediction drastically hurts performance
- Solution: dynamic branch prediction



A 2-bit prediction scheme

Branch Prediction

- Sophisticated Techniques:
  - A "branch target buffer" to help us look up the destination
  - Correlating predictors that base prediction on global behavior and recently executed branches (e.g., prediction for a specific branch instruction based on what happened in previous branches)
  - Tournament predictors that use different types of prediction strategies and keep track of which one is performing best.
  - A "branch delay slot" which the compiler tries to fill with a useful instruction (make the one cycle delay part of the ISA)
- Branch prediction is especially important because it enables other more advanced pipelining techniques to be effective!
- Modern processors predict correctly 95% of the time!

Improving Performance

- Try and avoid stalls! E.g., reorder these instructions:

```
lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t2, 0($t1)
sw $t0, 4($t1)
```

- Dynamic Pipeline Scheduling
  - Hardware chooses which instructions to execute next
  - Will execute instructions out of order (e.g., doesn't wait for a dependency to be resolved, but rather keeps going!)
  - Speculates on branches and keeps the pipeline full (may need to rollback if prediction incorrect)

- Trying to exploit instruction-level parallelism

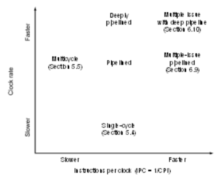
Advanced Pipelining

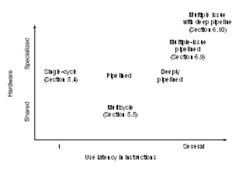
- Increase the depth of the pipeline
- Start more than one instruction each cycle (multiple issue)
- Loop unrolling to expose more ILP (better scheduling)
- "Superscalar" processors
  - DEC Alpha 21264: 9 stage pipeline, 6 instruction issue
- All modern processors are superscalar and issue multiple instructions usually with some limitations (e.g., different "pipes")
- VLIW: very long instruction word, static multiple issue (relies more on compiler technology)

- This class has given you the background you need to learn more!

Chapter 6 Summary

- Pipelining does not improve latency, but does improve throughput


Clock rate
Faster
Slower
Deeply pipelined
Multiple issue with deep pipeline (Section 6.10)
Multicycle (Section 5.5)
Pipelined
Multiple-issue pipelined (Section 6.9)
Single-cycle (Section 5.4)
Slower
Faster
Instructions per clock (IPC = 1/CPI)


Hardware
Specialized
Shared
Multiple issue with deep pipeline (Section 6.10)
Multiple-issue pipelined (Section 6.9)
Single-cycle (Section 5.4)
Pipelined
Deeply pipelined
Multicycle (Section 5.5)
Several
Use latency in instructions