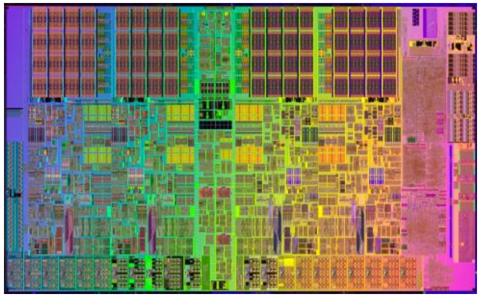
Computer Architecture

CH4 Processor Microarchitecture (II)

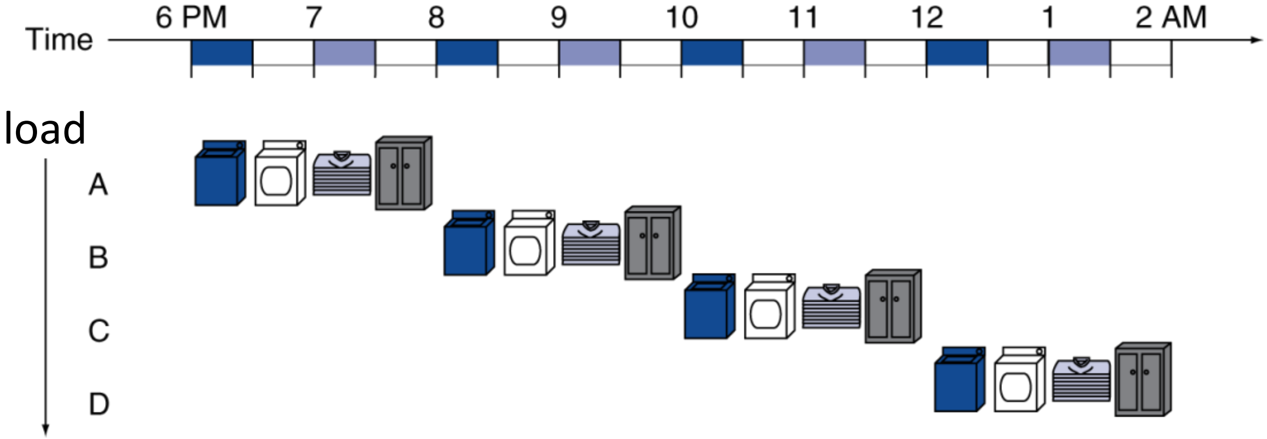
Prof. Ren-Shuo Liu
NTHU EE
Fall 2017



Outline

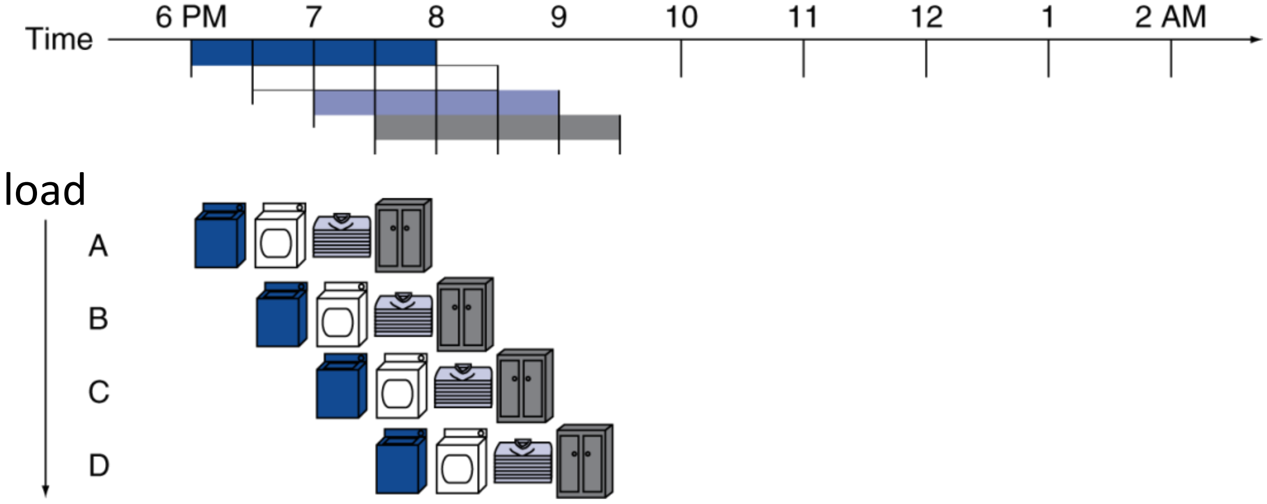
- Background
- Single-cycle design
- Pipelined design
  - Pipeline concepts and MIPS's pipeline
  - Cost and issues of pipelining
  - Detailed pipelined datapath and control

Pipeline Analogy



4 loads of laundry

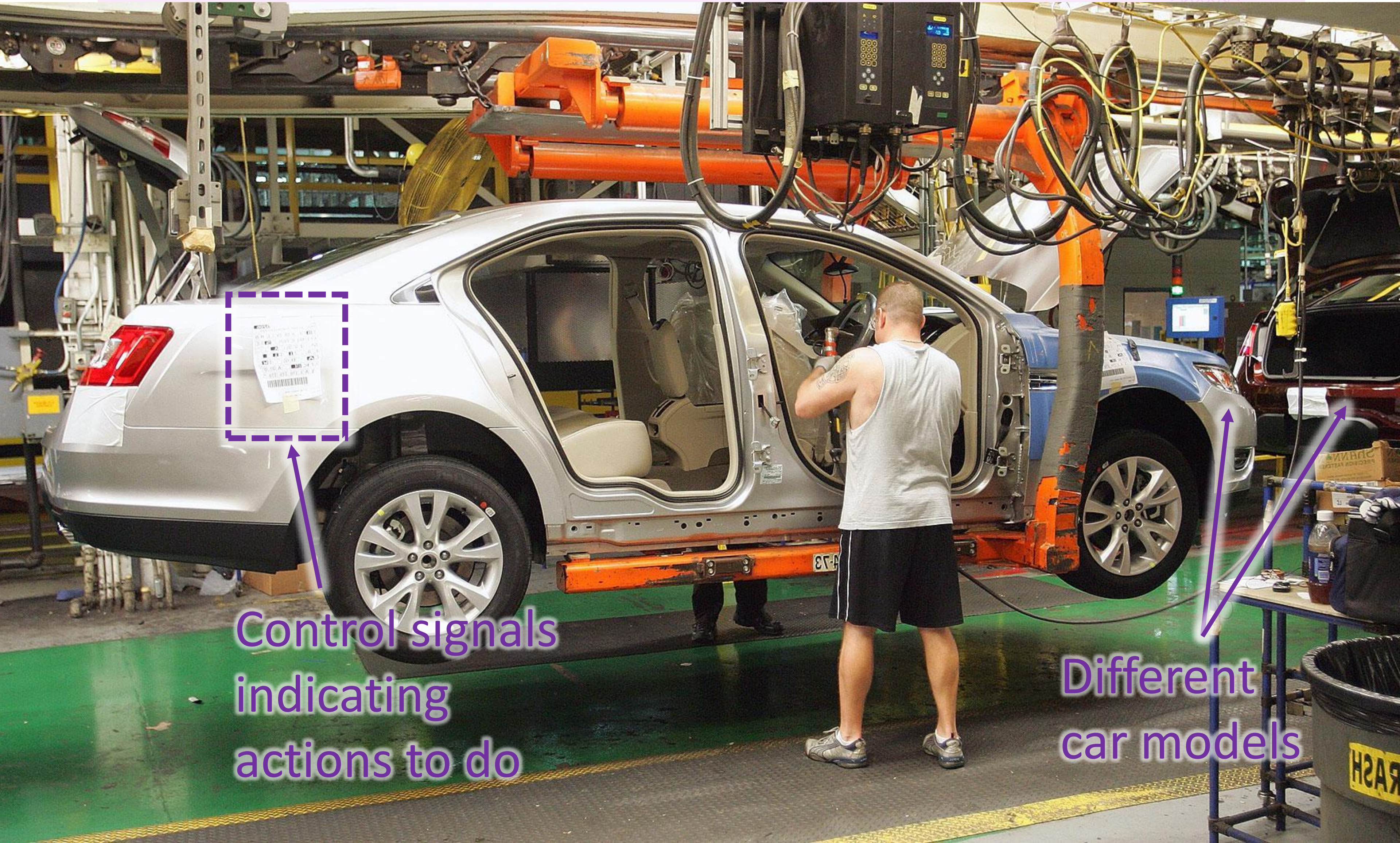
16 hrs



speedup = 16/7

7 hrs

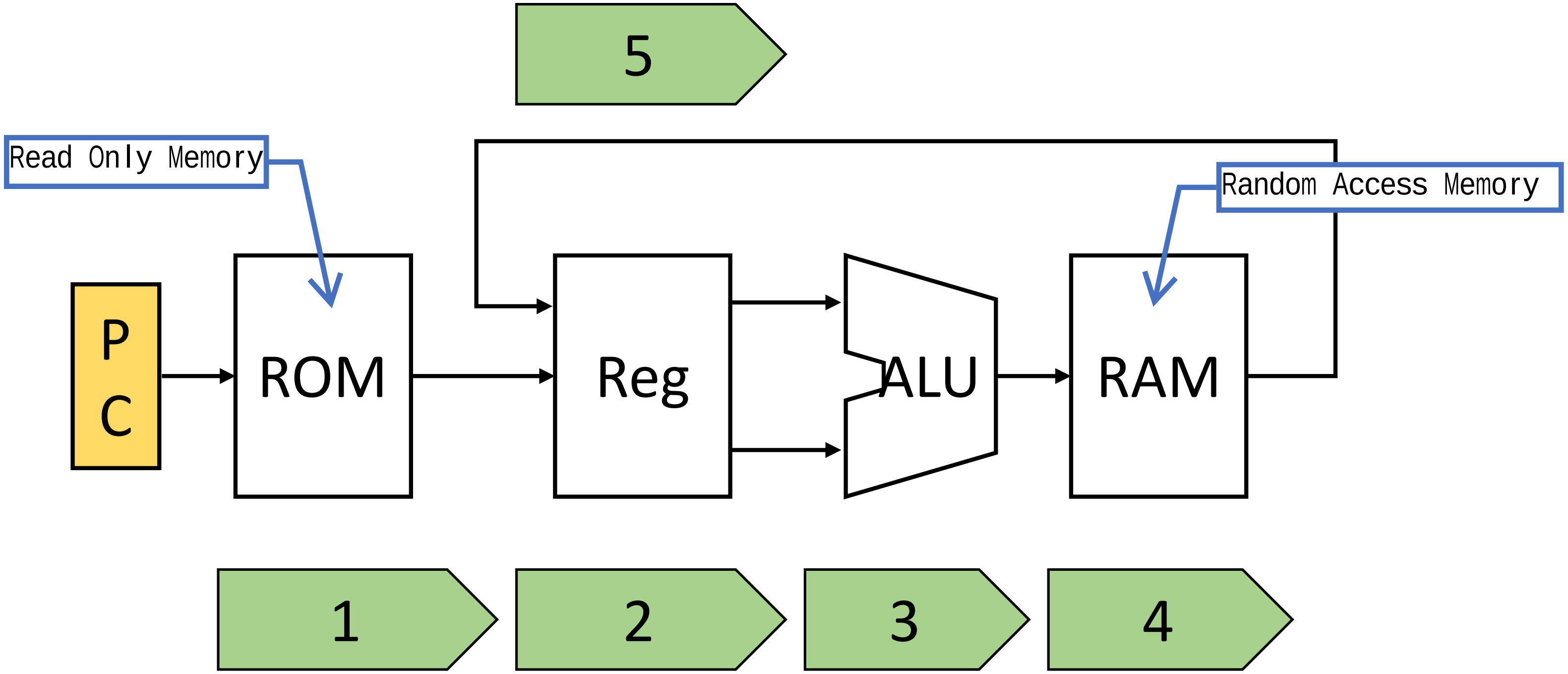
Car Factory Example



Control signals indicating actions to do

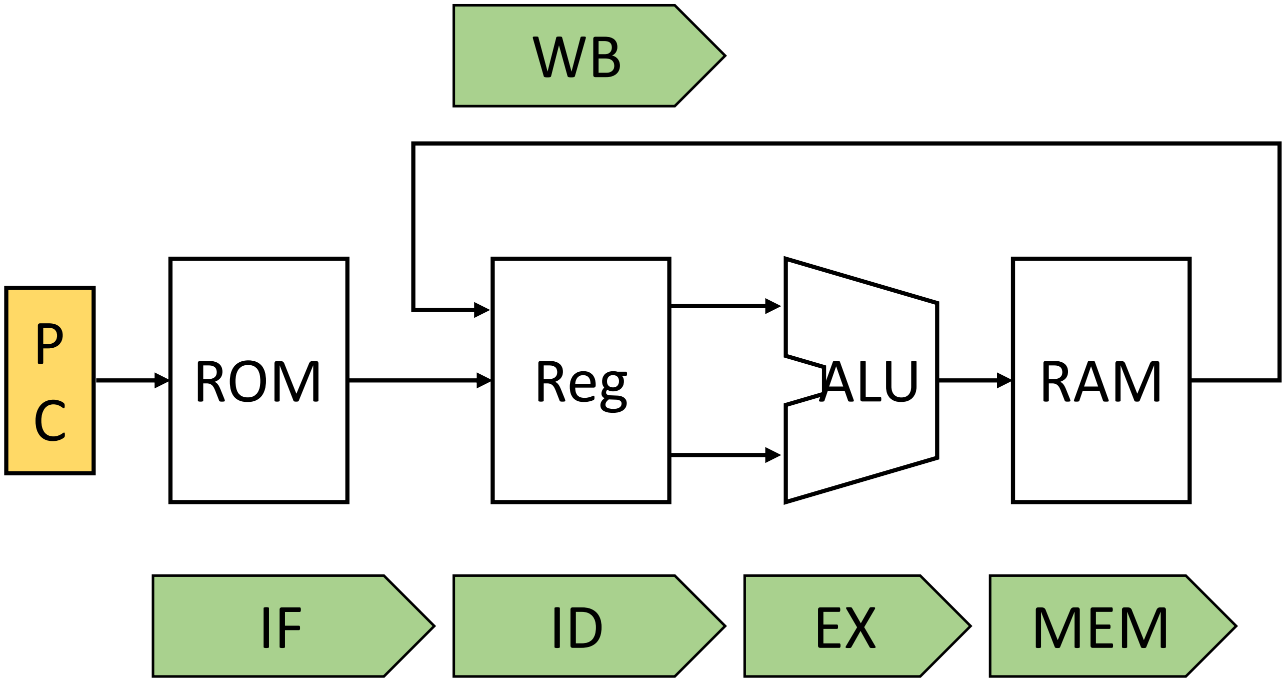
Different car models

5 Stages of Executing an Instruction



5 Stages of Executing an Instruction

Write result back to register



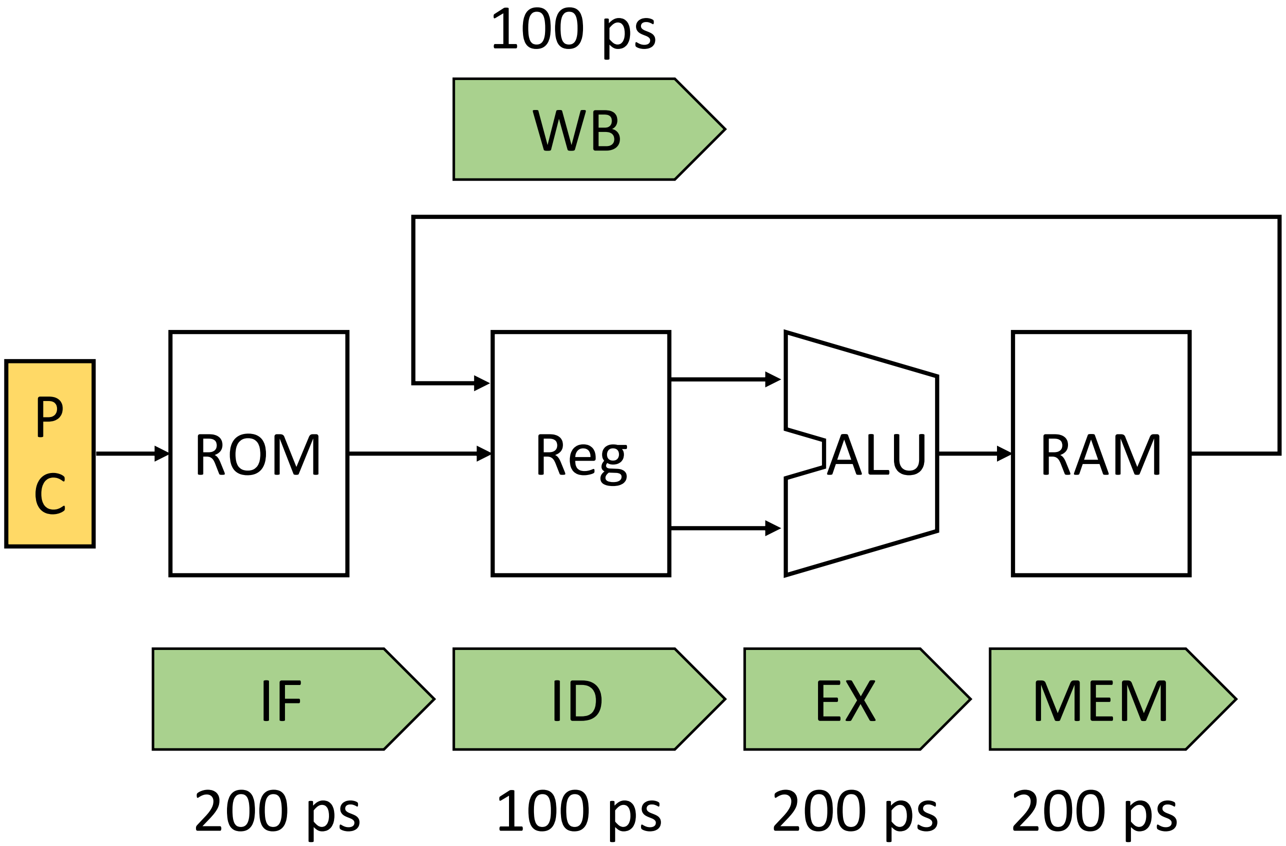
Instruction fetch from memory

Instruction decode & register read

Execute operation or calculate address

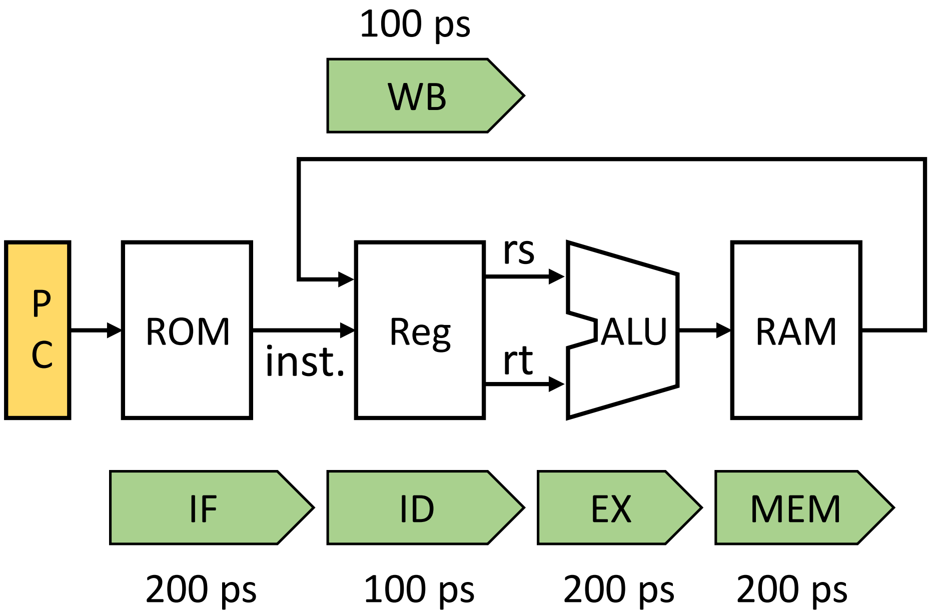
Access memory operand

Exampling Timing



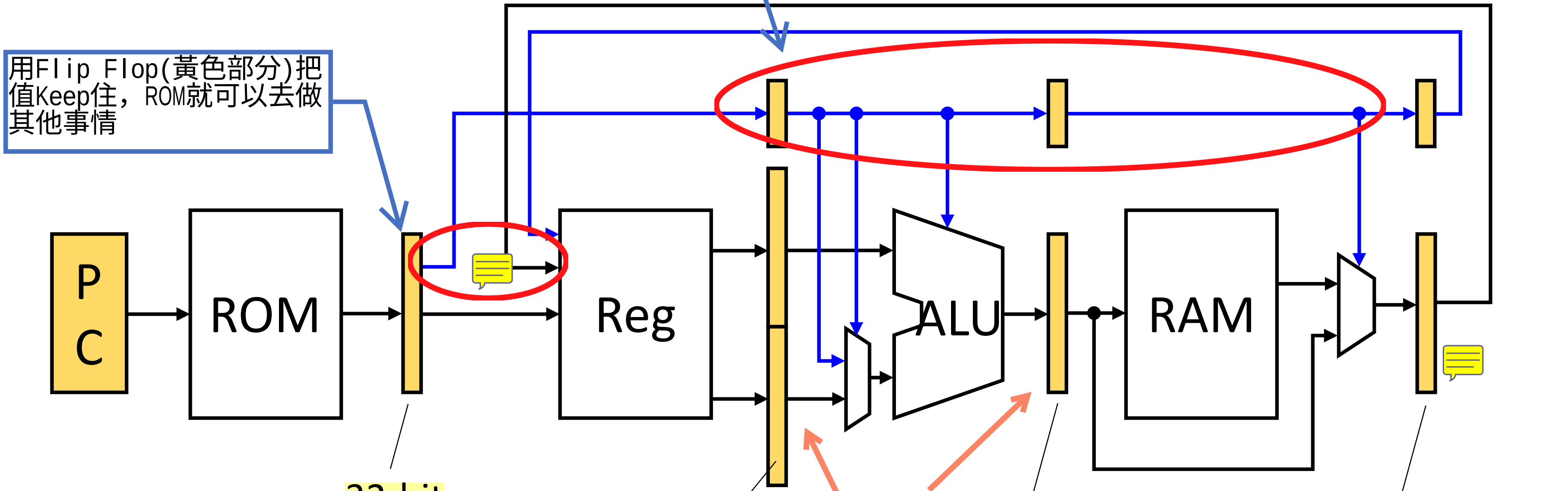
Observations

- 800 ps cycle time
- ROM is idle during 200 - 800 ps
    - Instruction is ready @ 200 ps
    - ROM only keeps its output after 200 ps
    - Keeping value can be done using flip-flops instead
- Reg, ALU, and RAM have similar situations



5-Stage Pipeline

傳送控制訊號(藍色線)，也用FlipFlop去keep住



用Flip Flop(黃色部分)把值Keep住，ROM就可以去做其他事情

跟Register Size大小有關

- 32-bit instruction

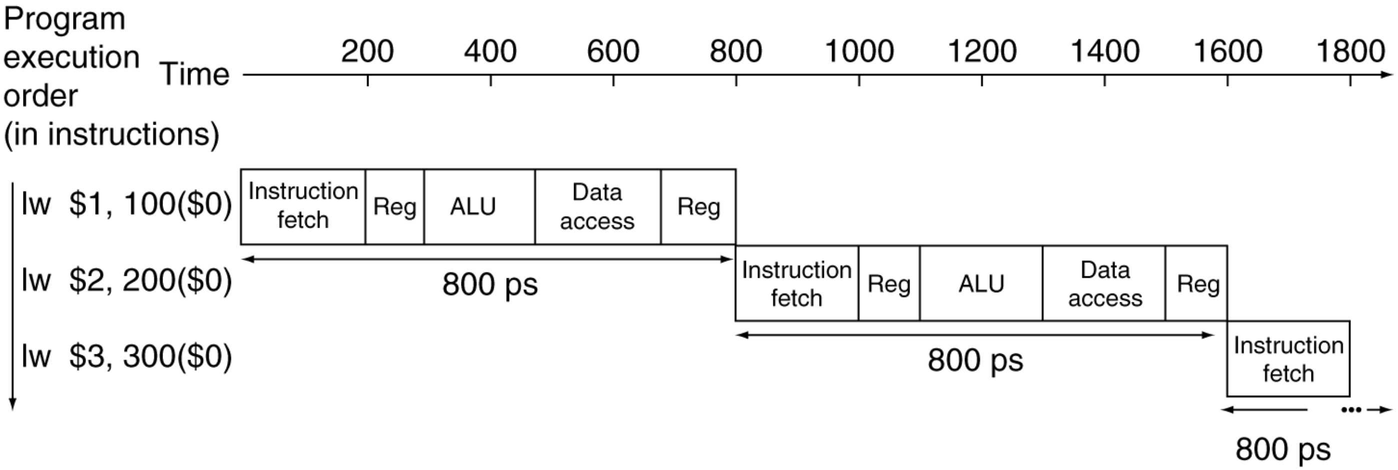
- rs
- rt
- control signals

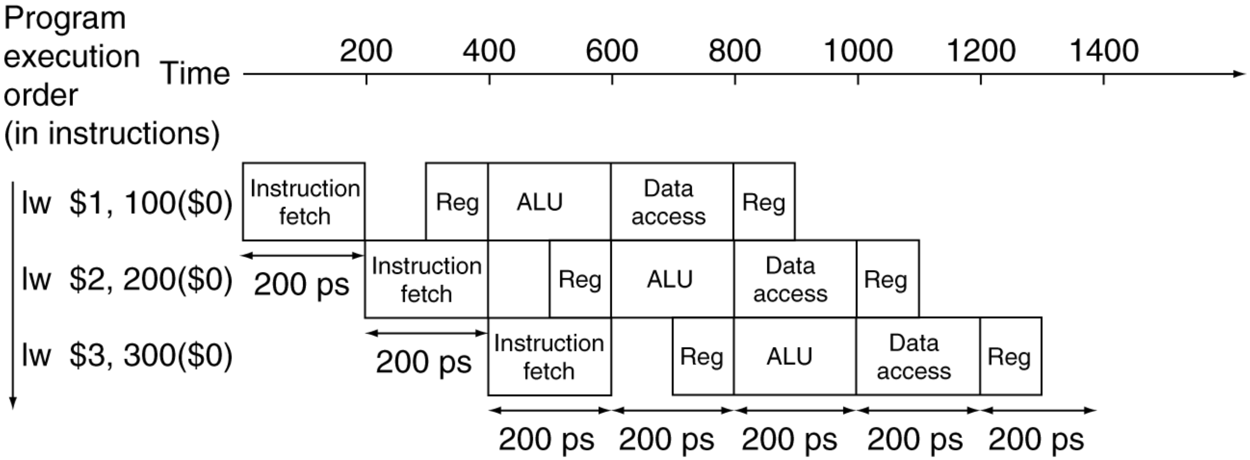
- ALU results
- control signals

- write-back value
- control signals

- Please specify the width and contents of every registers and buses
- 汽車、師傅、出貨單分別在哪裡?

Pipeline Performance



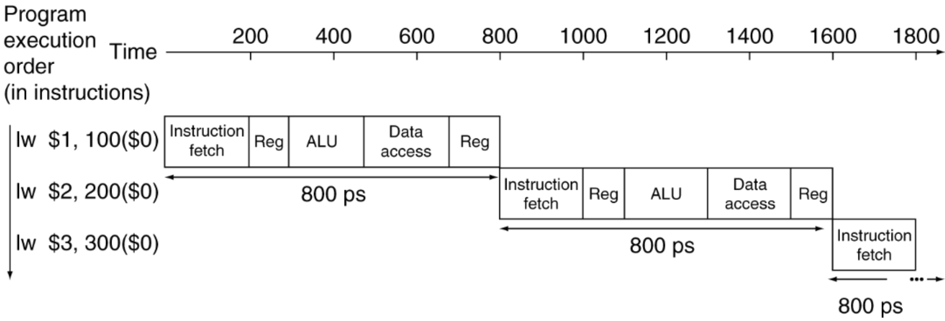


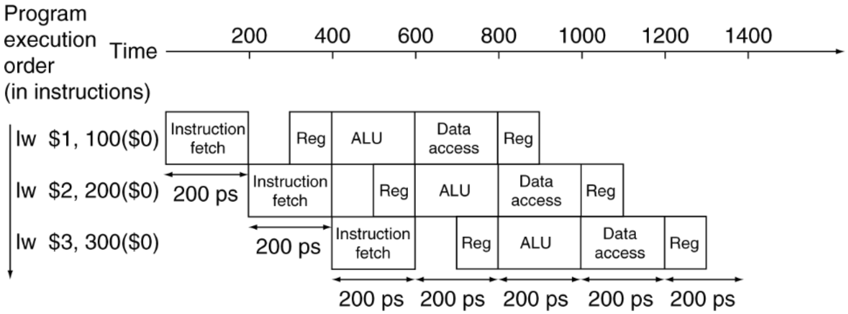
Pipeline Performance Observations

- Latency (time for completing each instruction) does not decrease

Pipeline不會影響到各instruction的latency

- Throughput (number of instructions completed per unit of time) increases
- Ideal case
  - All stages are balanced
  - Speedup can approach #stages





MIPS ISA Design for Pipelining

- All instructions are 32-bit

幫助Pipeline簡化設計

- Regular instruction formats
- Register-register arithmetic

如果Register和Memory做運算會比較複雜，無法使用Pipeline

- Base-offset addressing mode

唯一access address的模式

- Aligned memory access

固定Load、Store的大小規範(Ex:4Bytes)，減少讀取任意值造成時間不一致的問題

  - Load or store instruction cannot access memory spanning two 32-bit words

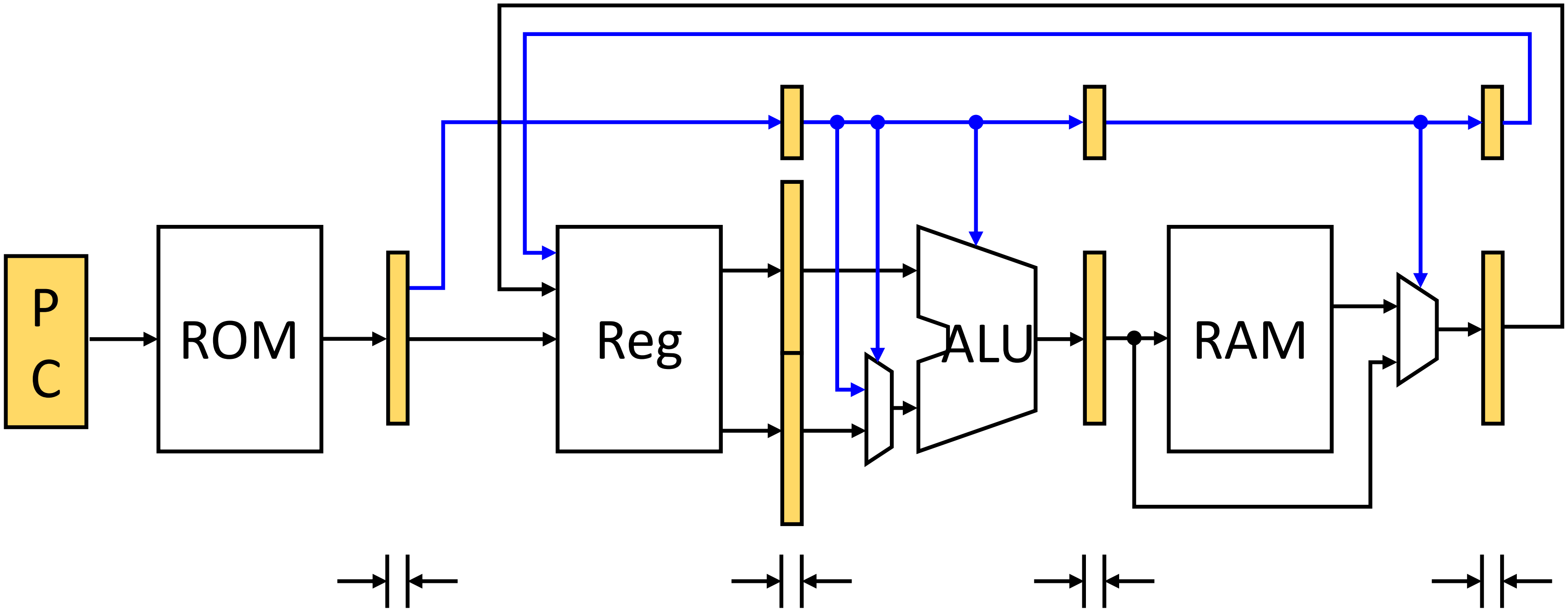
- These design decisions simplify pipeline hardware
  - Though they are not necessary for pipelining to be achievable

Costs and Issues of Pipelining

- Costs
  - Hardware cost
  - Performance cost
- Issues
  - Some instructions cannot be executed in a pipeline fashion due to hazards

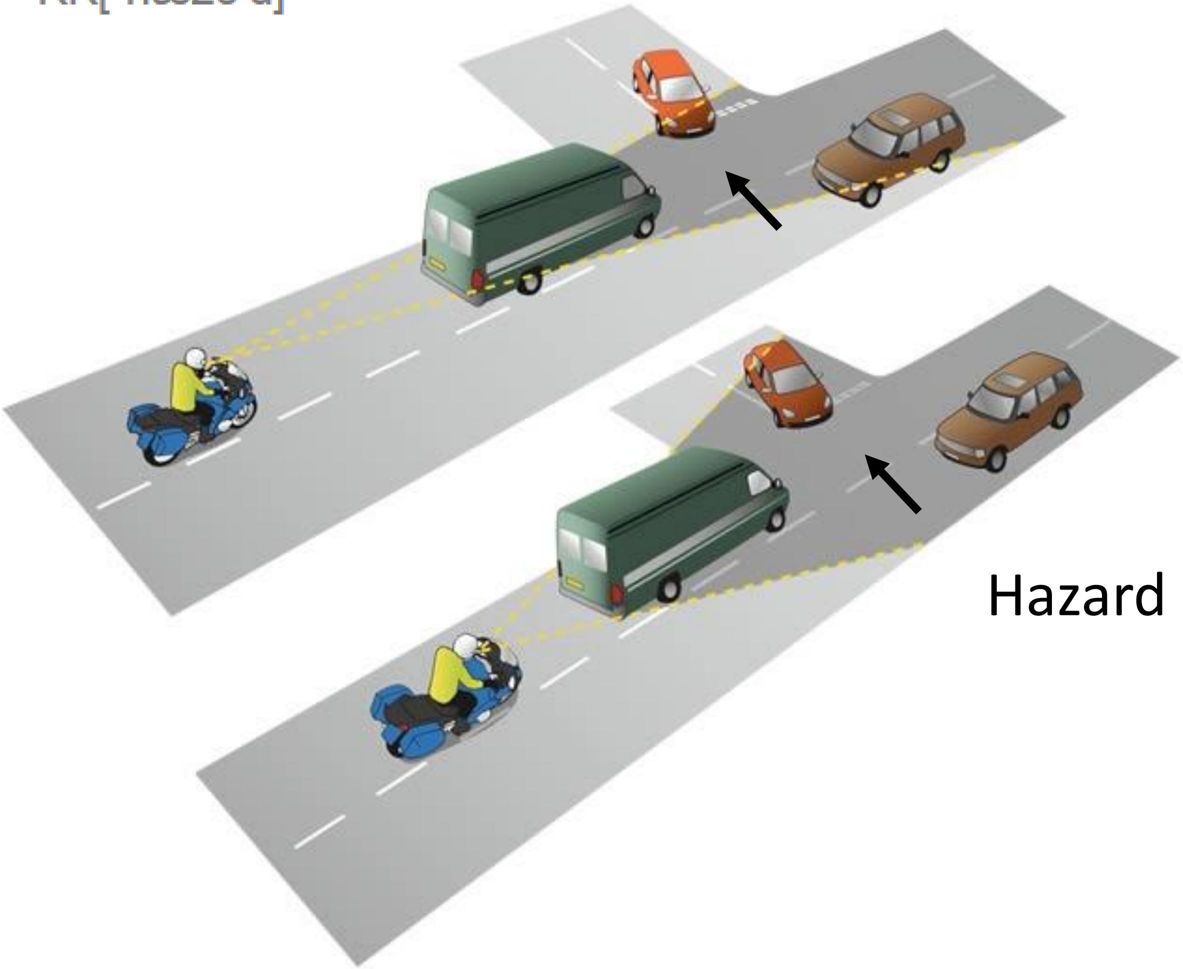
Costs

- Additional flip-flop hardware
- Additional flip-flop latency



Issues

- Hazard KK[ˋhæzɚd]



Hazard

Pipeline Hazards

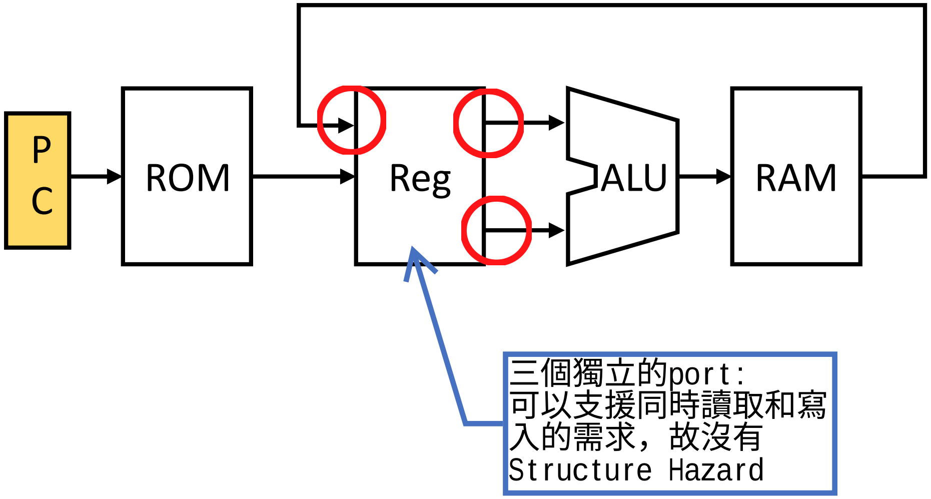
- Situations that prevent a pipeline from starting the next instruction in the next cycle
  - Structure hazard
  - Data hazard
  - Control hazard (branch hazard)

Hazard Examples

- Structure hazard
  - One required resource is busy
  - Take ALU for example: a late instruction needs to use the ALU, but the ALU is still busy doing an early instruction's job
- Data hazard 由Data Dependency造成
  - Late instruction's operand is yet calculated by an early instruction
- Control hazard (branch hazard)
  - Late instruction depends on an early branch instruction

Structure Hazard

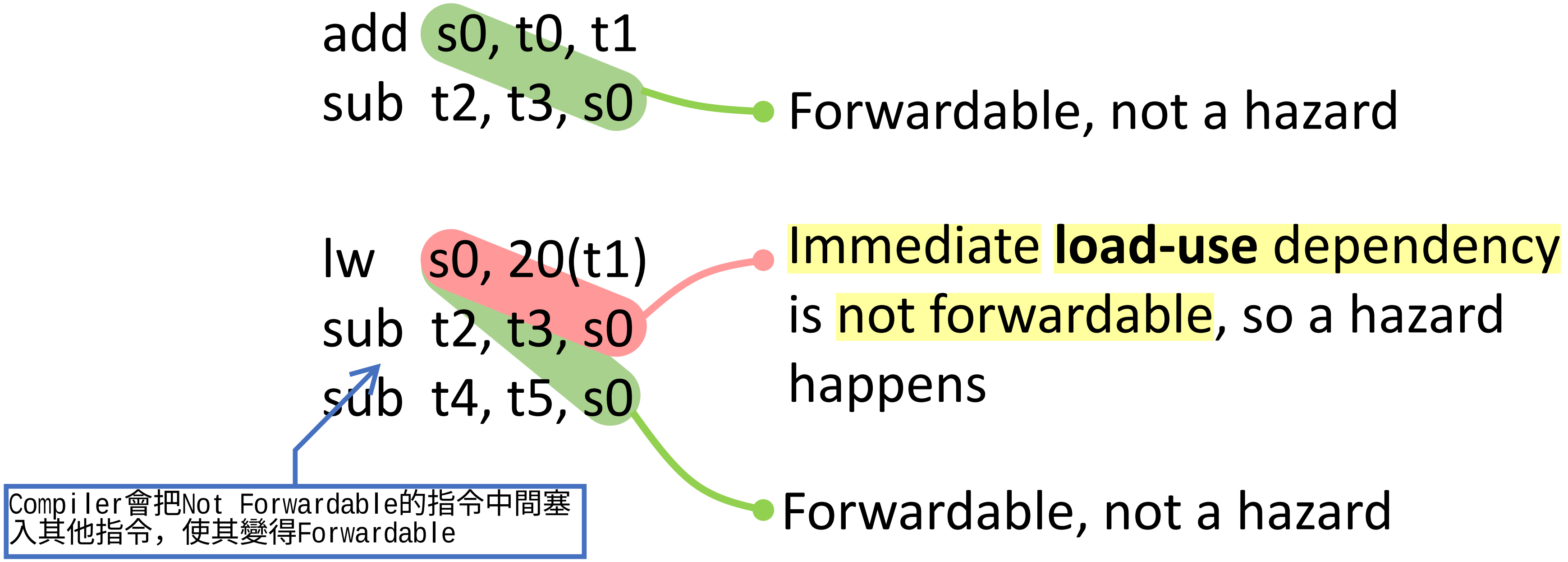
- Our MIPS example exhibits no structure hazards
  - Each instruction only uses ROM, ALU, and RAM for exactly one cycle
- Though each instruction accesses the register twice (IF and WB)
  - Hazard does not actually occur because Reg is multi-ported



三個獨立的port:
可以支援同時讀取和寫入的需求，故沒有Structure Hazard

Data Hazard

- Data hazards are caused by data dependencies
- Not all data dependencies cause data hazard
  - Some of data dependencies are resolved by forwarding (bypassing)
  - The following examples show data dependency



```
add  s0, t0, t1
sub  t2, t3, s0
```

Forwardable, not a hazard

```
lw   s0, 20(t1)
sub  t2, t3, s0
sub  t4, t5, s0
```

Immediate **load-use dependency** is not forwardable, so a hazard happens

Forwardable, not a hazard

Compiler會把Not Forwardable的指令中間塞入其他指令，使其變得Forwardable

Data Hazard

仍會在800ps時存入s0

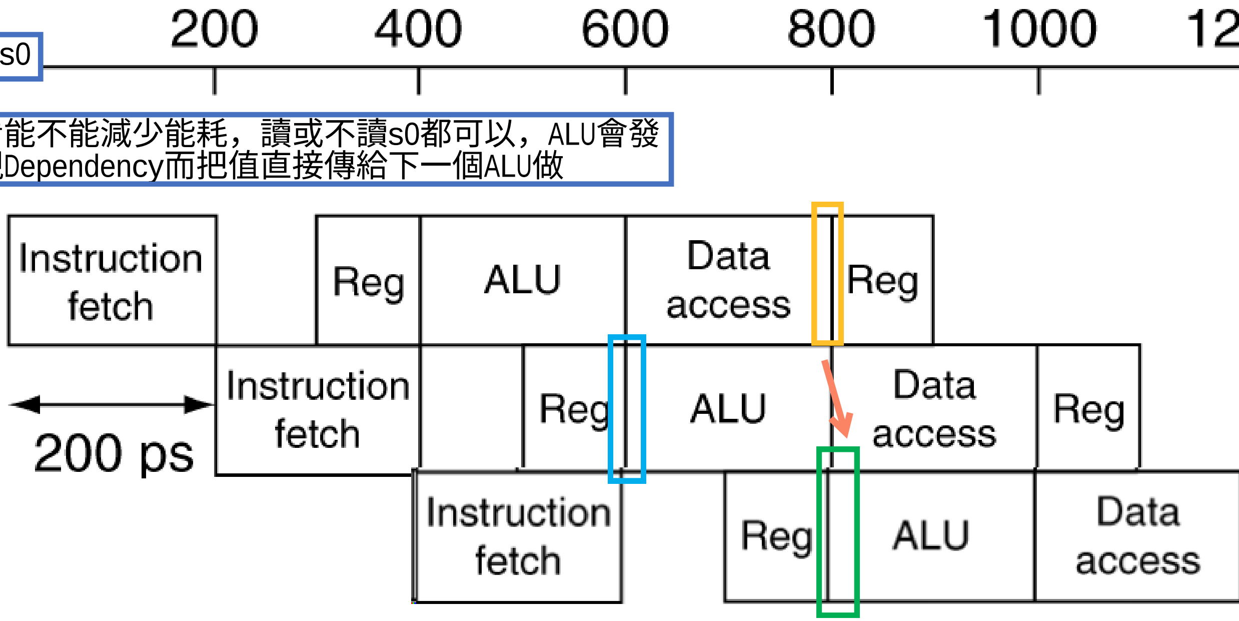
600ps時完成ALU

```
add  s0, t0, t1
sub  t2, t3, s0
```

600ps時開始ALU(需要s0)

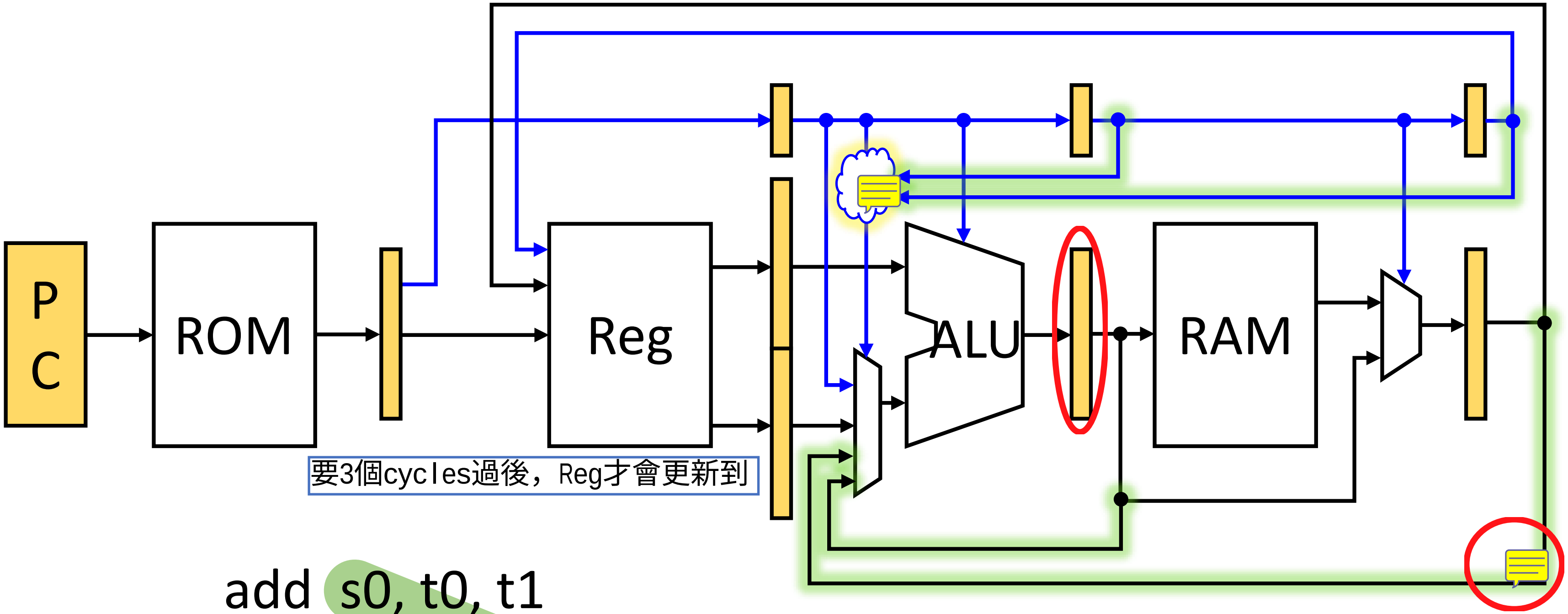
看能不能減少能耗，讀或不讀s0都可以，ALU會發現Dependency而把值直接傳給下一個ALU做

```
lw   s0, 20(t1)
sub  t2, t3, s0
sub  t4, t5, s0
```



硬體要支持立即的輸出導回當作立即的輸入

Forwarding Examples



要3個cycles過後，Reg才會更新到

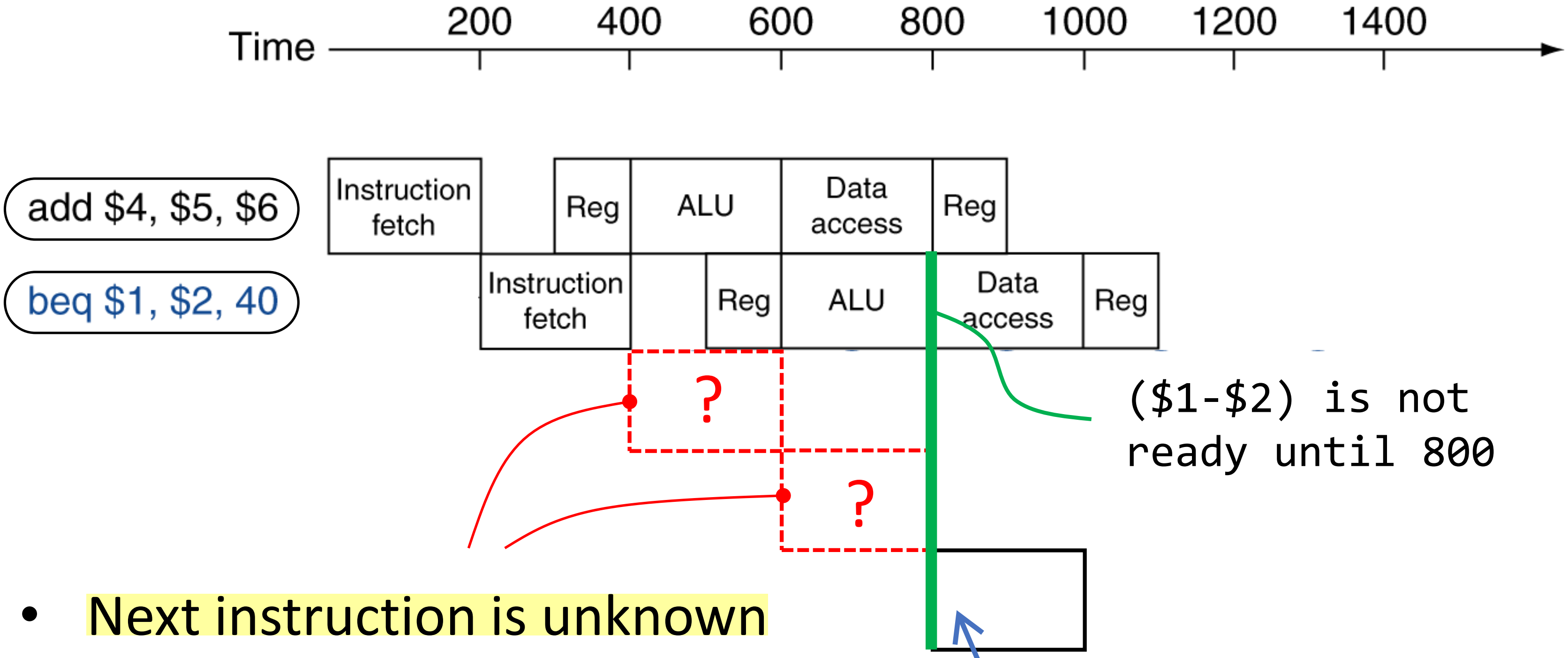
add s0, t0, t1
sub t2, t3, s0

lw s0, 20(t1)
nop
sub t4, t5, s0

Concept: the outcome of an early instruction directly becomes an operand of a late instruction

These are two examples only.
We will discuss more details later.

Control Hazard



($1-$2) is not ready until 800

根據ALU計算結果去決定下一個Instruction

- Next instruction is unknown for two cycles after a branch
- Pipeline is stalled for two cycles per branch

遇到Branch要先停兩個cycles

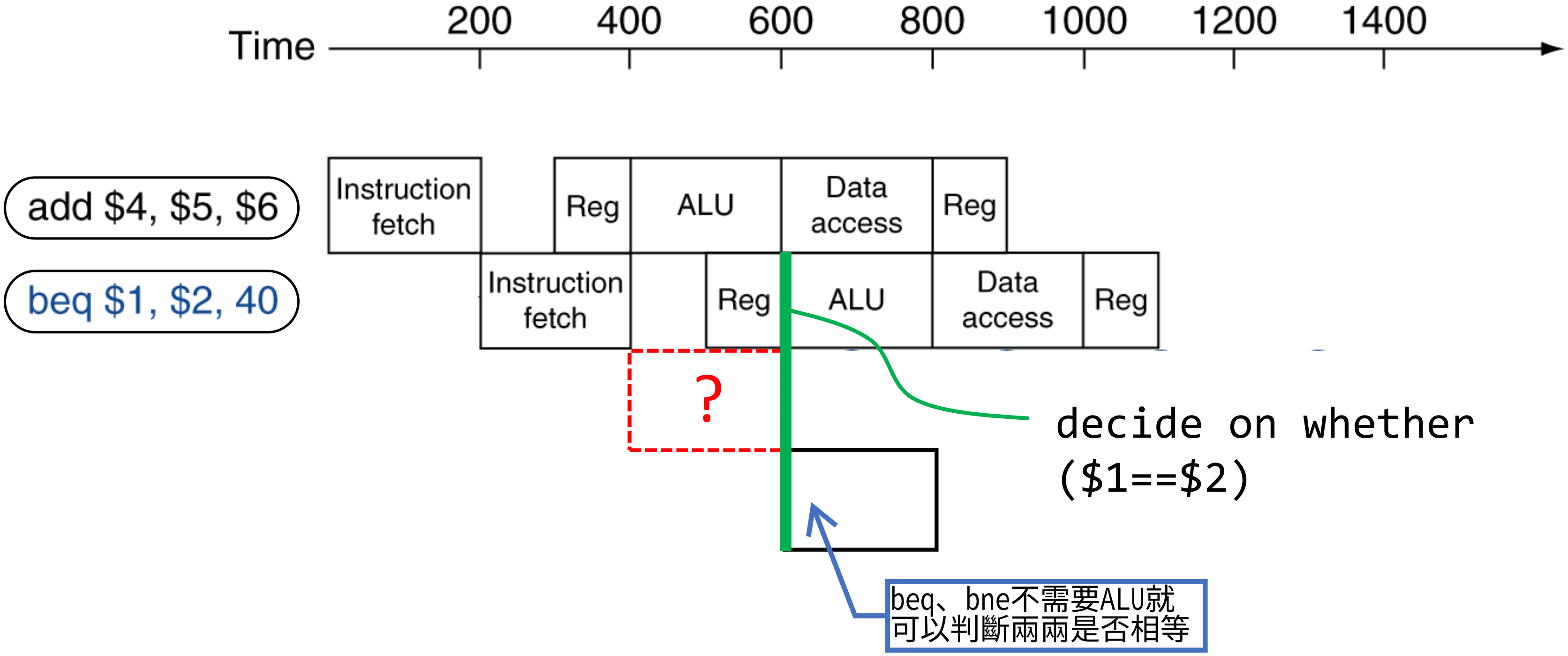
Handling Control Hazard

1
- Reducing stall cycles to one

2
- Delayed branch (branch delay slot)
- Branch prediction + rollback

Handling Control Hazard

- Reducing stall cycles to one
  - Perform branch test right after registers are read

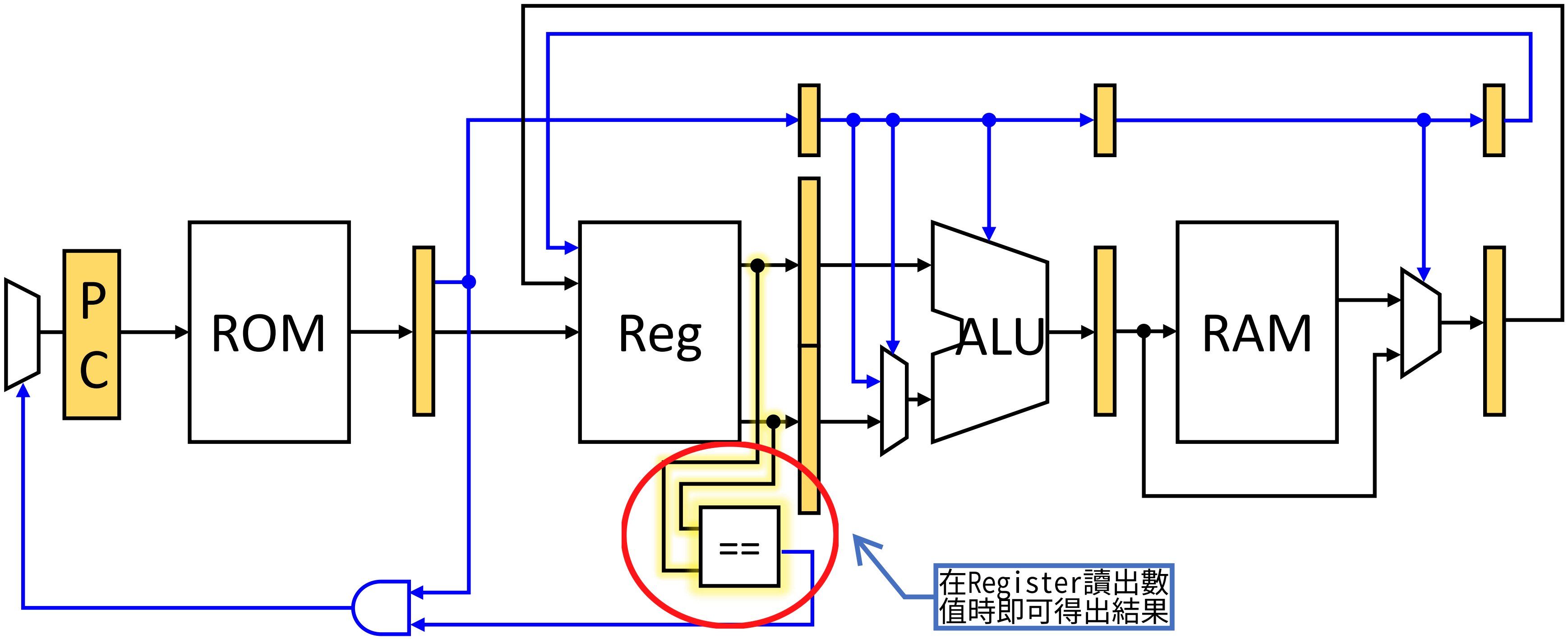


decide on whether ($1==$2)

beq、bne不需要ALU就可以判斷兩兩是否相等

Handling Control Hazard

- Of course, this leads to additional hardware costs to make both register read and equality test fast enough to fit into a cycle



在Register讀出數值時即可得出結果

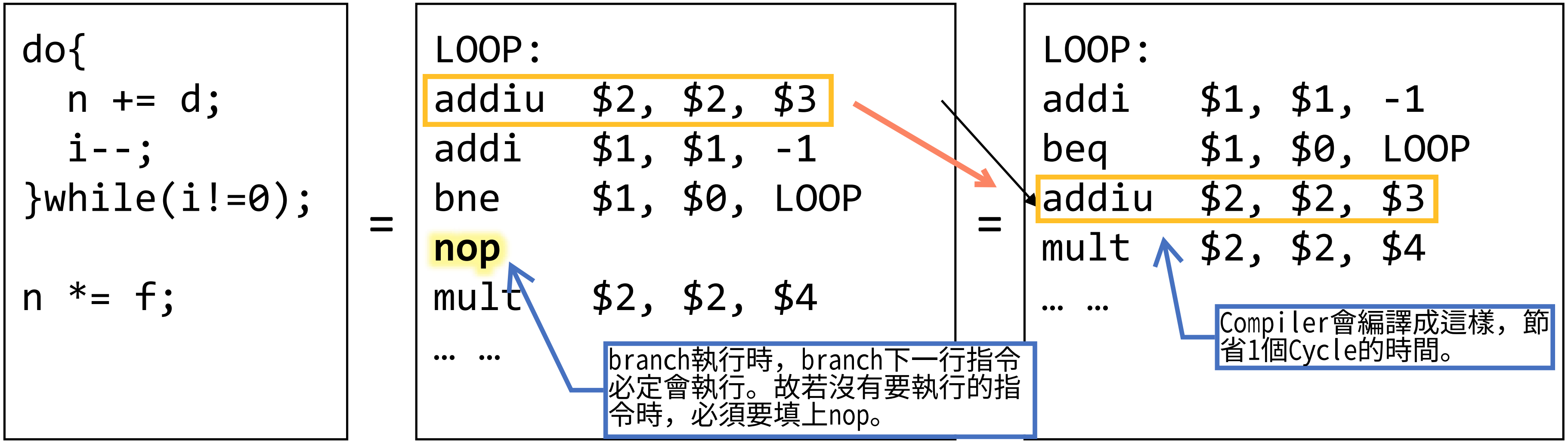
Handling Control Hazard

- Reducing stall cycles to one is usually a must
  - Branch instructions are very common (~10%)
  - Significant loss in performance if every branch stalls the pipeline for two cycles

- If branches only need equality test, reducing stall cycles to one is doable
  - Now we know why MIPS only support beq and bne but not branch instructions for relational operators
    - bge (branch on greater than or equal), bgt (branch on greater than), ble (branch on less than or equal), blt (branch on less than) are pseudo instructions

Handling Control Hazard

- Delayed branch (branch delay slot)
  - MIPS always fetches and executes the instruction following a branch
  - An nop is a must if no other instruction should not be fetched and executed in the slot



```
do{
  n += d;
  i--;
}while(i!=0);

n *= f;
```

=

```
LOOP:
addiu   $2, $2, $3
addi    $1, $1, -1
bne     $1, $0, LOOP
nop
mult    $2, $2, $4
… …
```

=

```
LOOP:
addi    $1, $1, -1
beq     $1, $0, LOOP
addiu   $2, $2, $3
mult    $2, $2, $4
… …
```

branch執行時，branch下一行指令必定會執行。故若沒有要執行的指令時，必須要填上nop。

Compiler會編譯成這樣，節省1個Cycle的時間。

Handling Control Hazard

- Branch prediction + rollback
  - Pipeline speculatively fetch an instruction
  - If the fetched instruction turns out to be on the wrong path, the fetched instruction is discarded and the corrected instruction is then fetched

Predict策略:
(1)遇到往回跳的情況，通常代表是迴圈，故很大機率要往回跳
(2)其餘可能是代表if-else，故50/50

- Basic prediction strategy
  - Always predict taken
    - Branches to an earlier address usually mean a loop and are taken most of the times
    - Branches to an other addresses usually mean if-else, and we guess they are taken for 50% of the times

Backward Taken Forward NotTaken (BTFN)

Outline

- Background
- Single-cycle design
- Pipelined design
  - Pipeline concepts and MIPS's pipeline
  - Cost and issues of pipelining
  - Detailed pipelined datapath and control