# Processor technology

Riferimenti bibliografici

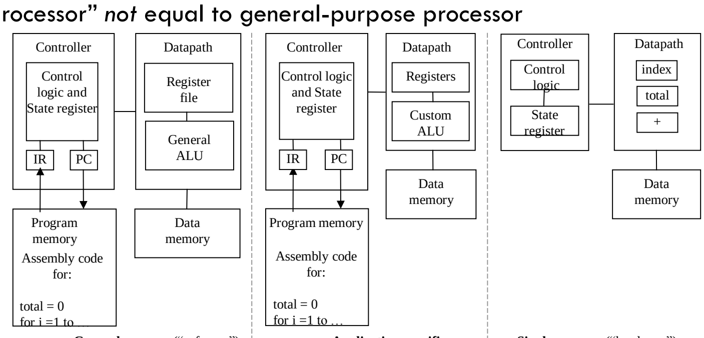
*“Embedded System Design: A Unified Hardware/Software Introduction”* , Frank Vahid, Tony Givargis, John Wiley & Sons Inc., ISBN:0-471-38678-2, 2002.

“Computer architecture, a quantitative approach”, Hennessy & Patterson: (Morgan Kaufmann eds.)



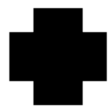
# Processor technology

- The architecture of the computation engine used to implement a system’s desired functionality
- Processor does not have to be programmable
  - “Processor” *not* equal to general-purpose processor



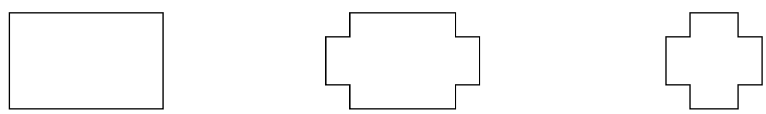
# Processor technology

- Processors vary in their customization for the problem at hand



```
total = 0;
for (i = 0; i< N; i++)
  total += M[i];
```

Desired functionality



General-purpose processor

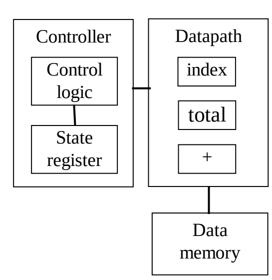
Application-specific processor

Single-purpose processor



# Single-purpose processors

- Digital circuit designed to execute exactly one program
  - a.k.a. coprocessor, accelerator or peripheral
- Features
  - Contains only the components needed to execute a single program
  - No program memory
- Benefits
  - Fast
  - Low power
- Drawbacks
  - No flexibility, high time-to-market, high NRE cost



# Basic logic gates

**A) Problem description**

y is 1 if a is to 1, or b and c are 1. z is 1 if b or c is to 1, but not both, or if all are 1.

**B) Truth table**

| Inputs | | | Outputs | |
|---|---|---|---|---|
| a | b | c | y | z |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |

**C) Output equations**

y = a'bc + ab'c' + ab'c + abc' + abc

z = a'b'c + a'bc' + ab'c + abc' + abc

**D) Minimized output equations**

y

| a \ bc | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

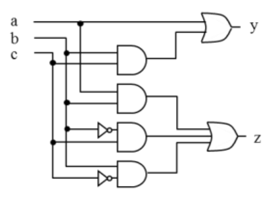
y = a + bc

z

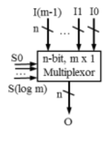
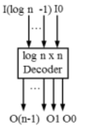
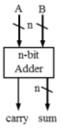
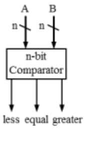
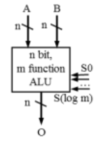
| a \ bc | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 1 |

z = ab + b'c + bc'

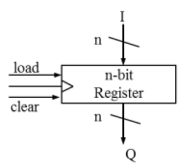
**E) Logic Gates**



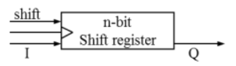

# Combinational components

| | | | | |
|---|---|---|---|---|
|  |  |  |  |  |
| O =<br>I0 if S=0..00<br>I1 if S=0..01<br>…<br>I(m-1) if S=1..11 | O0 =1 if I=0..00<br>O1 =1 if I=0..01<br>…<br>O(n-1) =1 if I=1..11 | sum = A+B<br>(first n bits)<br>carry = (n+1)'th<br>bit of A+B | less = 1 if A<B<br>equal =1 if A=B<br>greater=1 if A>B | O = A *op* B<br>*op* determined<br>by S. |
| | With enable input e →<br>all O's are 0 if e=0 | With carry-in input Ci→<br>sum = A + B + Ci | | May have status outputs<br>carry, zero, etc. |

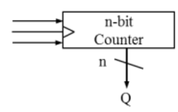
# Sequential components




Q =
0 if clear=1,
I if load=1 and clock=1,
Q(previous) otherwise.




Q = lsb
- Content shifted
- I stored in msb



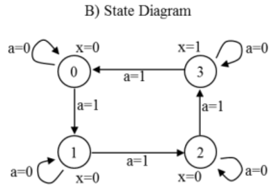

Q =
0 if clear=1,
Q(prev)+1 if count=1 and clock=1.

# Sequential Logic Design

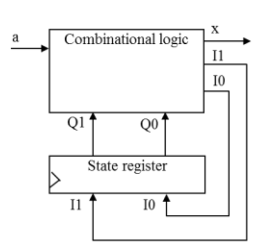
A) Problem Description

You want to construct a clock divider. Slow down your pre-existing clock so that you output a 1 for every four clock cycles

B) State Diagram




C) Implementation Model

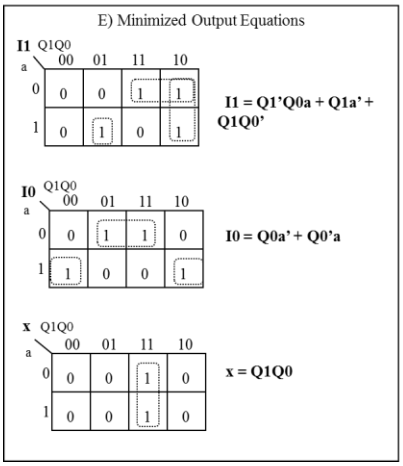


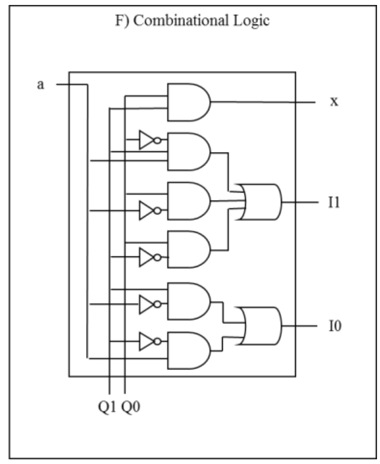

D) State Table (Moore-type)

| Inputs | | | Outputs | | |
|---|---|---|---|---|---|
| Q1 | Q0 | a | I1 | I0 | x |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | |
| 1 | 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | |

- Given this implementation model
  - Sequential logic design quickly reduces to combinational logic design

# Sequential Logic Design

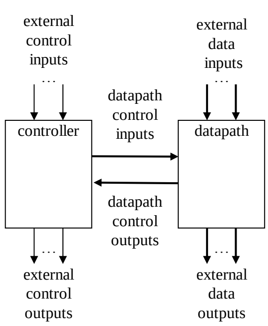



# Single-purpose processor design

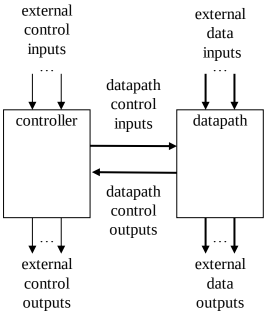
Can be viewed as the design of a system with 2 components:

- Datapath, which executes operations required to the system
- Control Unit, which generates commands for datapath on the basis of data inputs and conditions

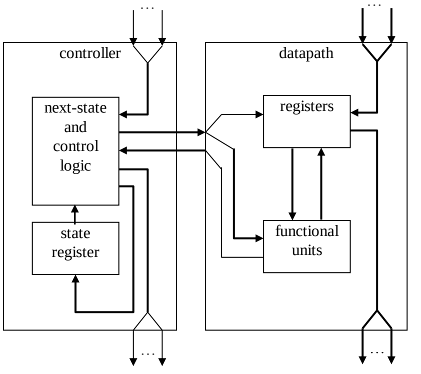



controller and datapath

# Sigle-purpose processor design




controller and datapath




a view inside the controller and datapath

# Single-purpose processor design flow

1. Processor Specifications ( algorithmic description)
2. Convert algorithm to “complex” state machine
   - Known as FSMD: finite-state machine with datapath
   - Can use templates to perform such conversion
3. Datapath design
4. Control unit design

## Datapath design

Datapath design uses a library of components

- Multiplexer
- Decoder
- Comparators
- ALUs
- Registers

## Datapath Design

- The design the datapath requires, starting from the specifications of the system, the realization of a schematic that defines
  - the necessary components;
  - as components are connected;
  - the conditions and the results produced;
  - the control signals which must be produced by the control unit;
- In designing the datapath is necessary to take account of some project constraints such as:
  - maximum latency
  - maximum area
  - maximum power

## Datapath design

- Create a register for any declared variable
- Create a functional unit for each arithmetic operation
- Connect the ports, registers and functional units
  - Based on reads and writes
  - Use multiplexors for multiple sources
- Create unique identifier
  - for each datapath component control input and output

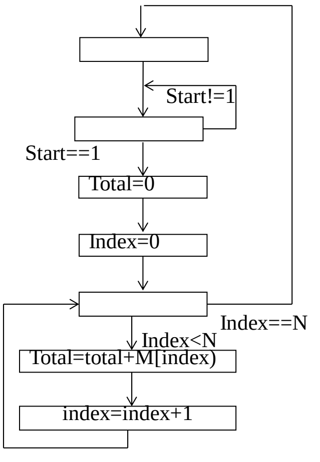
## Control Unit Design

- Designing the control unit is equivalent to designing a finite state machine (FSM)
- Identified states and control signals for the datapath, the design of the control unit can be realized using the methods of synthesis of synchronous sequential circuits

# Example

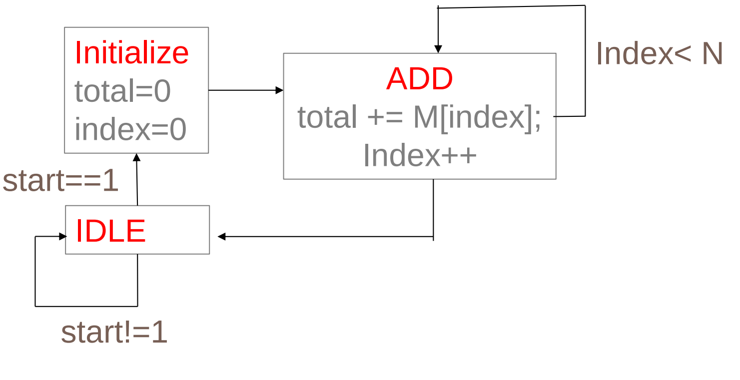
### Specification:

```
while(1)
{while(start!=1);
 {total = 0;
   for (index = 0; index< N; index++)
   total += M[index];
  }
}
```

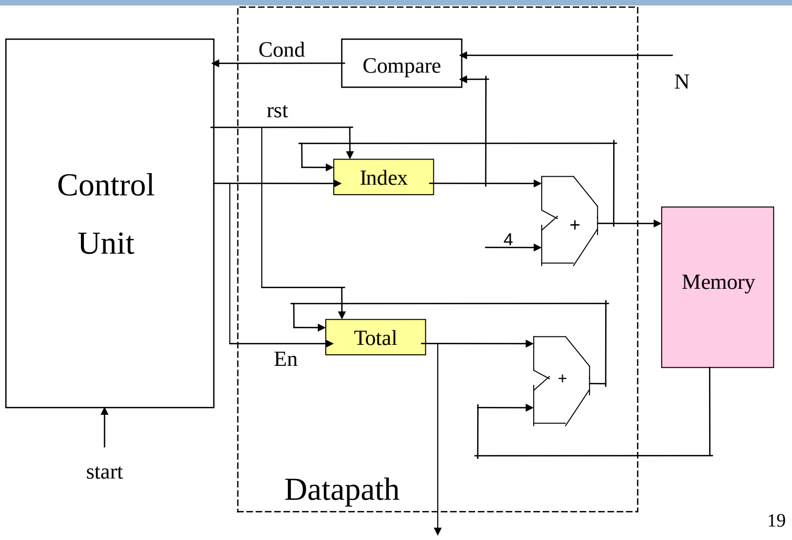



# Example

### FSM:



# Single-purpose processors






# Control Unit Design

| State | rst | en |
|---|---|---|
| IDLE | 0 | 0 |
| INIT | 1 | 0 |
| ADD | 0 | 1 |

# Example: Least common multiple

Specification

```
while(true)
 { Ready='1';
   do
   while(start!='1');
   ma=A; mb=B; Ready='0';
   while(ma!=mb)
    if(ma<mb)
        ma=ma+A;
      else
          mb=mb+B;
      Ris=ma;
   }
```

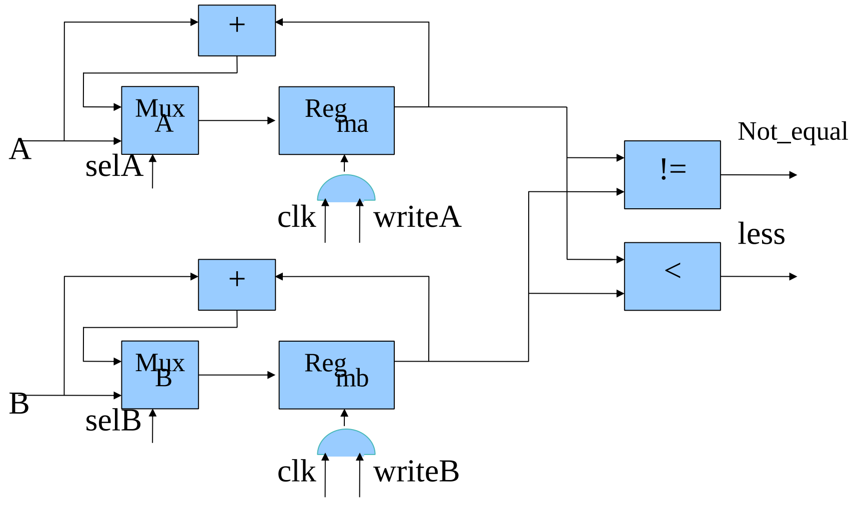
# Example: Least common multiple

To design the datapath the following blocks are required:

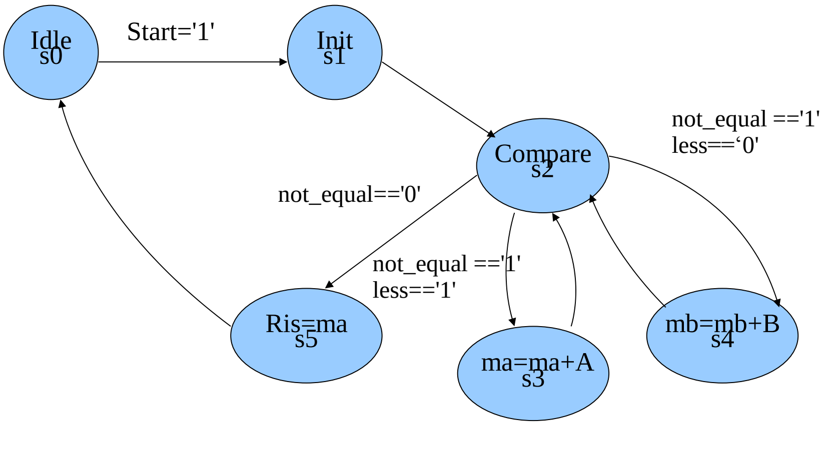
- Registers (ma, mb and Ris)
- Comparatores for conditions (A!=B) and (A<B)
- Adders for ma=ma+A and for mb=mb+B
- Multiplexer for selecting inputs of registers ma ( A or ma+A) using SelA or mb (B or mb+B) using SelB

AND port for clock and a write enable for registers ma (WriteA), mb (writeB) and Ris (WriteR)

## Datapath Least common multiple




## FSM(Moore): Least common multiple

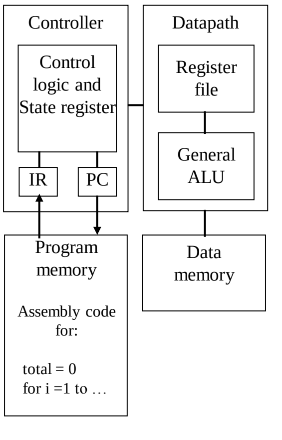


# FSM: outputs

| | S0 | S1 | S2 | S3 | S4 | S5 |
|---|---|---|---|---|---|---|
| SelA | - | 0 | 1 | 1 | 1 | 1 |
| SelB | - | 0 | 1 | 1 | 1 | 1 |
| WriteA | 0 | 1 | 0 | 1 | 0 | 0 |
| WriteB | 0 | 1 | 0 | 0 | 1 | 0 |
| WriteR | 0 | 0 | 0 | 0 | 0 | 1 |
| Ready | 1 | 0 | 0 | 0 | 0 | 0 |

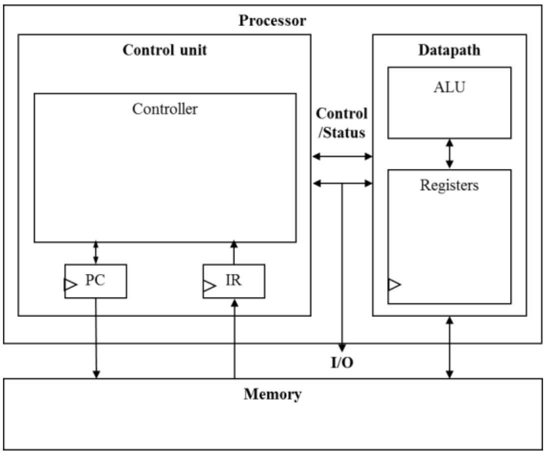
# General-purpose processors

- Programmable device used in a variety of applications
  - Also known as "microprocessor"
- Features
  - Program memory
  - General datapath with large register file and general ALU
- User benefits
  - Low time-to-market and NRE costs
  - High flexibility
- Drawbacks
  - High unit cost
  - Low Performance



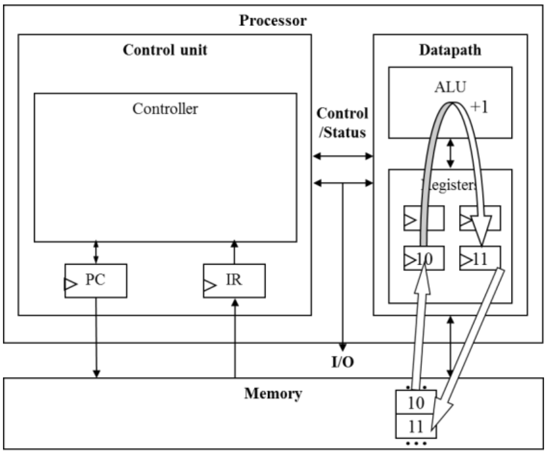
# Basic architecture

- Control unit and datapath
  - Note similarity to single-purpose processor
- Key differences
  - Datapath is general
  - Control unit doesn't store the algorithm – the algorithm is "programmed" into the memory



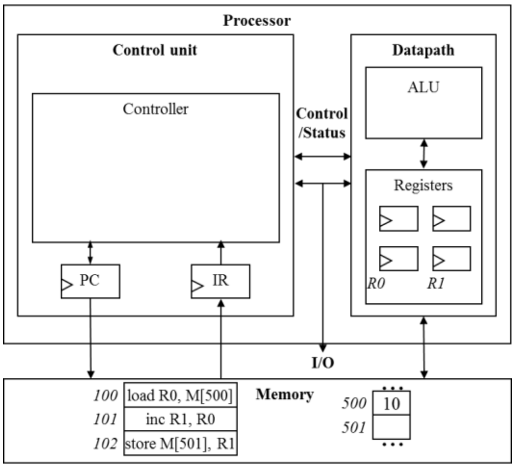

# Datapath

- Load
  - Read memory location into register
- ALU operation
  - Input certain registers through ALU, store back in register
- Store
  - Write register to memory location



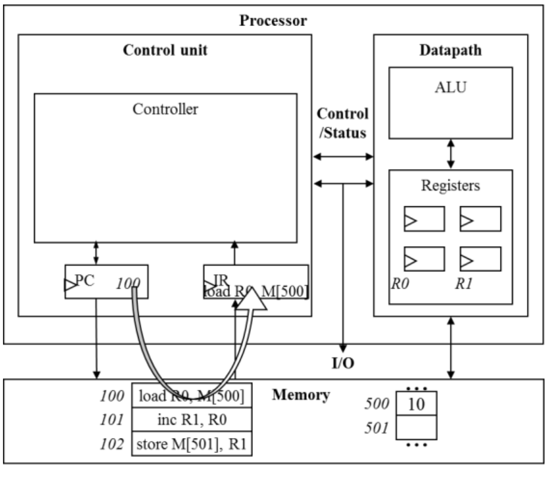
# Control Unit

- Control unit: configures the datapath operations
  - Sequence of desired operations ("instructions") stored in memory – "program"
- Instruction cycle – broken into several sub-operations, each one clock cycle, e.g.:
  - Fetch: Get next instruction into IR
  - Decode: Determine what the instruction means
  - Fetch operands: Move data from memory to datapath register
  - Execute: Move data through the ALU
  - Store results: Write data from register to memory



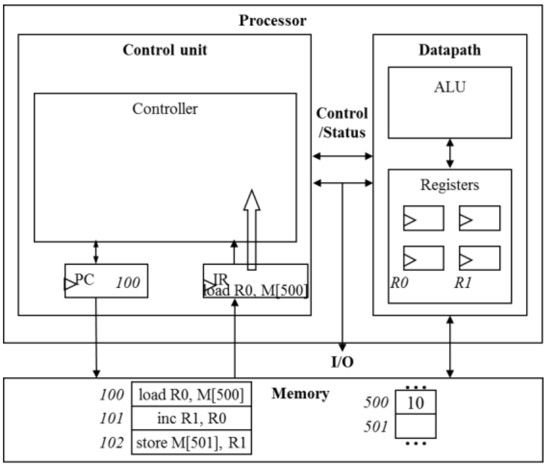

# Control Unit sub - operation

- Fetch
  - Get next instruction into IR
  - PC: program counter, always points to next instruction
  - IR: holds the fetched instruction



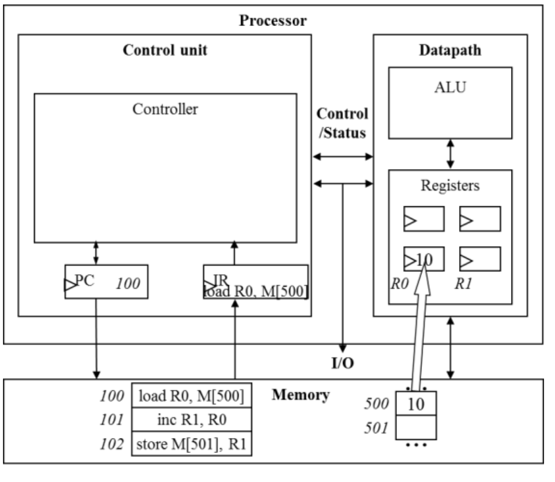
## Control Unit  sub - operation

- Decode
  - Determine what the instruction means



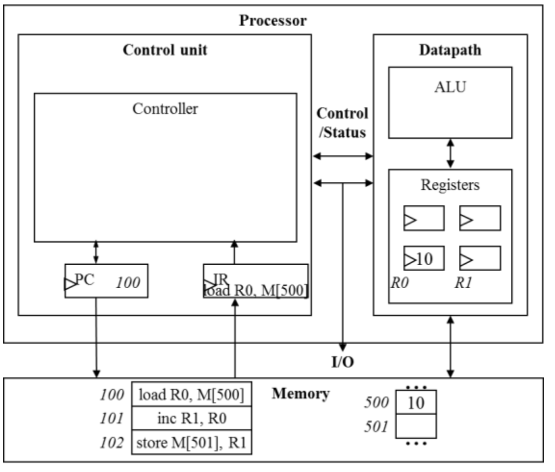

## Control Unit sub - operation

- Fetch operands
  - Move data from memory to datapath register



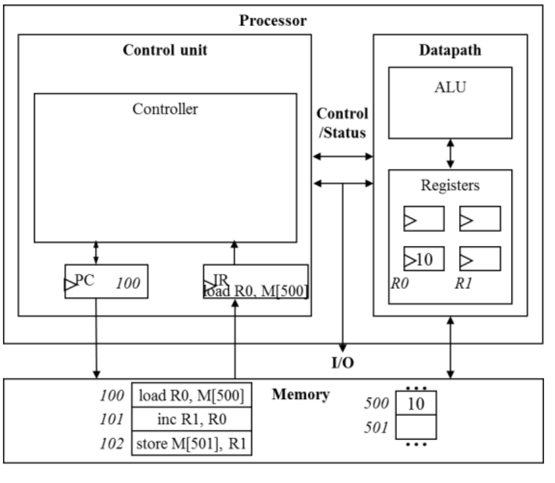
## Control Unit - sub operation

- Execute
  - Move data through the ALU
  - This particular instruction does nothing during this sub-operation

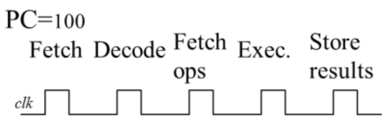


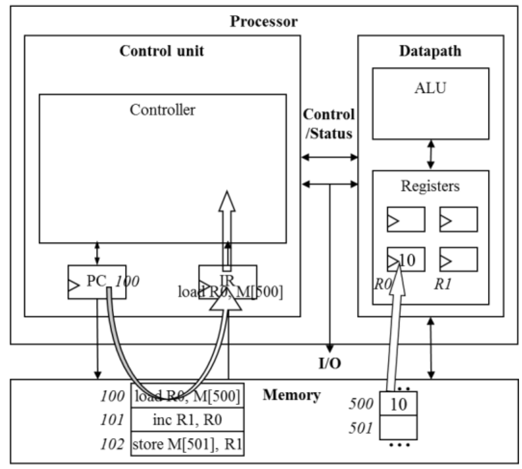

## Control Unit  sub - operation

- Store results
  - Write data from register to memory
  - This particular instruction does nothing during this sub-operation

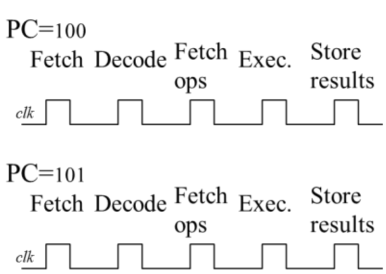


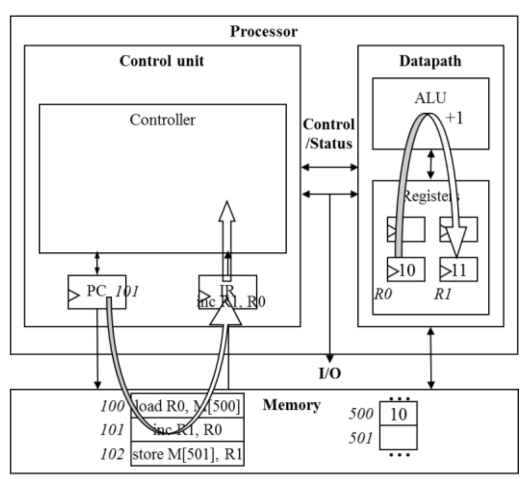
# Instruction Cycles



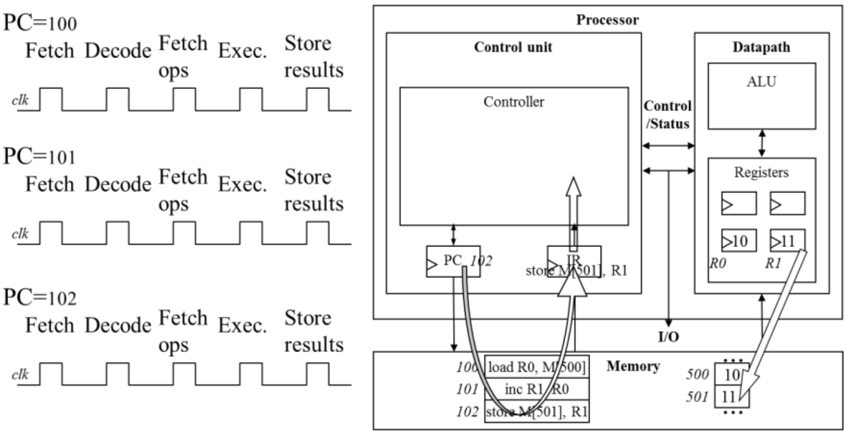




# Instruction Cycles



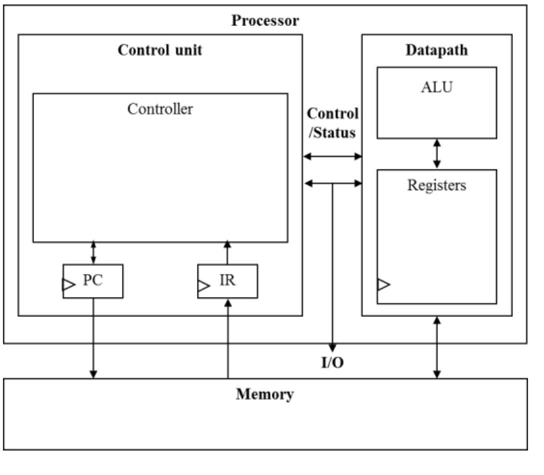



## Instruction Cycles



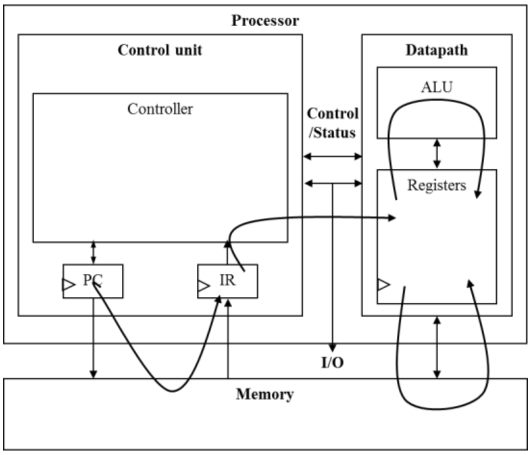

## Architectural Considerations

- *N-bit* processor
  - N-bit ALU, registers, buses, memory data interface
  - Embedded: 8-bit, 16-bit, 32-bit common
  - Desktop/servers: 32-bit, even 64
- PC size determines address space

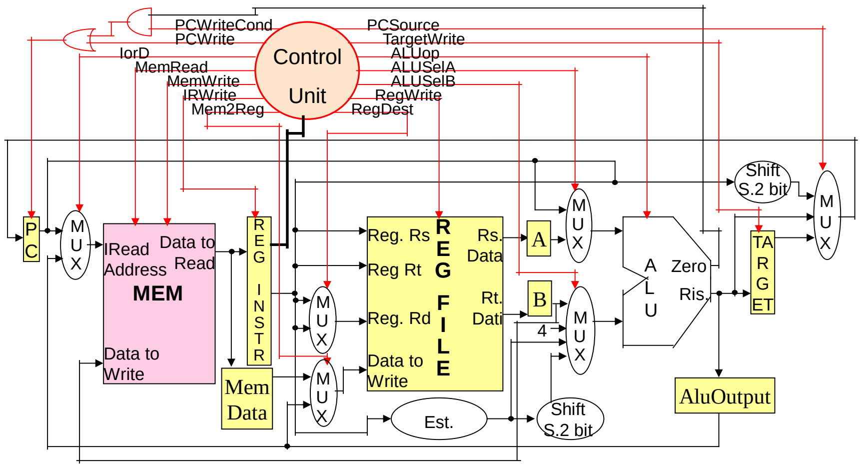


# Architectural Considerations

- Clock frequency
  - Inverse of clock period
  - Must be longer than longest register to register delay in entire processor
  - Memory access is often the longest




# General-purpose processors Sequential DLX



# General-purpose processors

```
total = 0;
for (i = 0; i< N; i++)
  total += M[i];
```

```
       addi r1, r0, 0
       addi r3, r0, 0
Loop:  lw r4,M(r1)
       addi r1,r1,4
       slti r2, r1, 40
       add r3,r3,r4
       bnez r2, loop
```



# How to improve performance

- Improve frequency ( depends on IC technology)
- They increase the number of instructions/data executed in the same clock cycle
  - Temporal parallelism (pipeline)
  - Spatial parallelism
    - Instruction Level Parallelism (Superscalar, VLIW, ..)
    - Data level Parallelism (SIMD processors)

# Pipelining

Performance optimization technique based on the overlap of the execution of multiple instructions deriving from a sequential execution flow.

• Pipelining exploits the parallelism among instructions in a sequential instruction stream.

• Basic idea:

The execution of an instruction is divided into different phases (pipelines stages), requiring a fraction of the time necessary to complete the instruction.

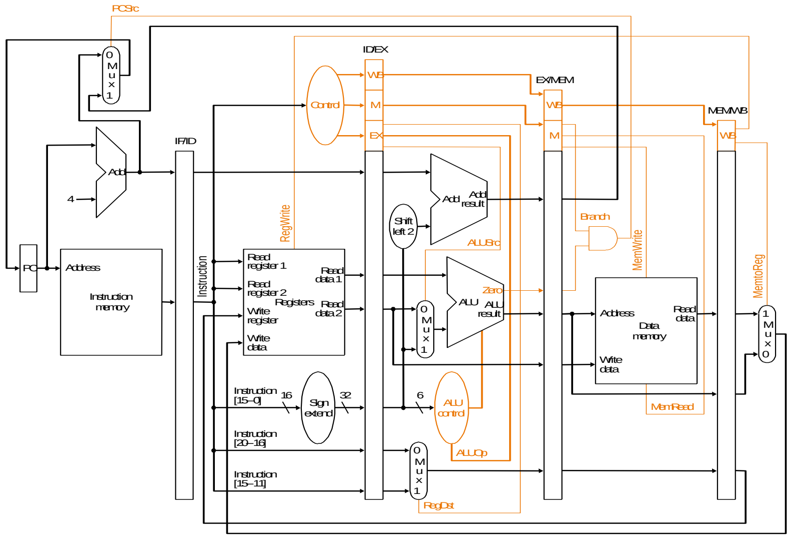
• The stages are connected one to the next to form the pipeline: instructions enter in the pipeline at one end, progress through the stages, and exit from the other end, as in an assembly line.

# Pipelining: Increasing Instruction *Throughput*

| T0 | T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 | T9 | T10 | T11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| IFetch 0 | IDec 0 | IExe 0 | IMem 0 | IWrB 0 | IFetch 1 | IDec 1 | IExe 1 | IMem 1 | IWrB 1 | IFetch 2 | IDec 3 |

| T0 | T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 | T9 | T10 | T11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| IFetch 0 | IDec 0 | IExe 0 | IMem 0 | IWrB 0 | | | | | | | |
| | IFetch 1 | IDec 1 | IExe 1 | IMem 1 | IWrB 1 | | | | | | |
| | | IFetch 2 | IDec 2 | IExe 2 | IMem 2 | IWrB 2 | | | | | |
| | | | IFetch 3 | IDec 3 | IExe 3 | IMem 3 | IWrB 3 | | | | |
| | | | | IFetch 4 | IDec 4 | IExe 4 | IMem 4 | IWrB 4 | | | |
| | | | | | IFetch 5 | IDec 5 | IExe 5 | IMem 5 | IWrB 5 | | |
| | | | | | | IFetch 6 | IDec 6 | IExe 6 | IMem 6 | IWrB 6 | |
| | | | | | | | IFetch 7 | IDec 7 | IExe 7 | IMem 7 | IWrB 7 |

# General-purpose processors pipeline DLX




45

# The Problem of Hazards

- A hazard is created whenever there is a dependence between instructions, and instructions are close enough that the overlap caused by pipelining would change the order of access to the operands involved in the dependence.
- Hazards prevent the next instruction in the pipeline from executing during its designated clock cycle.
- Hazards reduce the performance from the ideal speedup gained by pipelining.

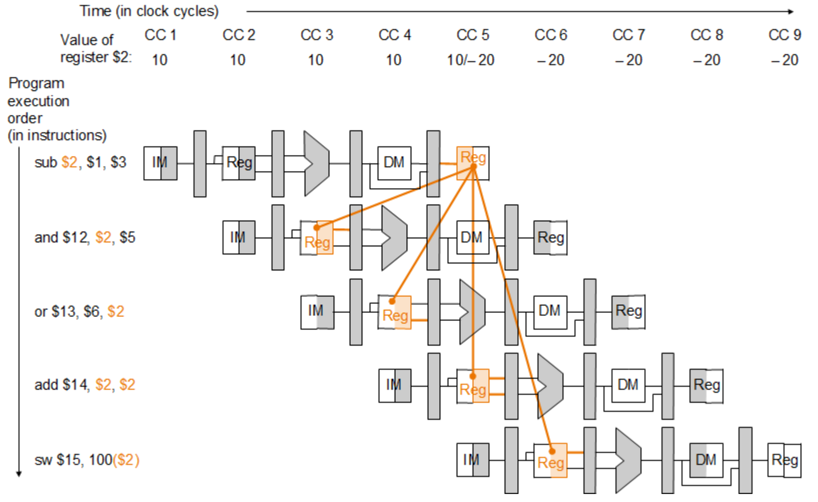
## Three Classes of Hazards

- **Structural Hazards**: Attempt to use the same resource from different instructions simultaneously
  - Example: Single memory for instructions and data
- **Data Hazards**: Attempt to use a result before it is ready
  - Example: Instruction depending on a result of a previous instruction still in the pipeline
- **Control Hazards**: Attempt to make a decision on the next instruction to execute before the condition is evaluated
  - Example: Conditional branch execution

## Structural hardware

- Two solutions
  - Hardware duplication
  - Insertion of *"bubbles"* or *stalls* in the pipeline

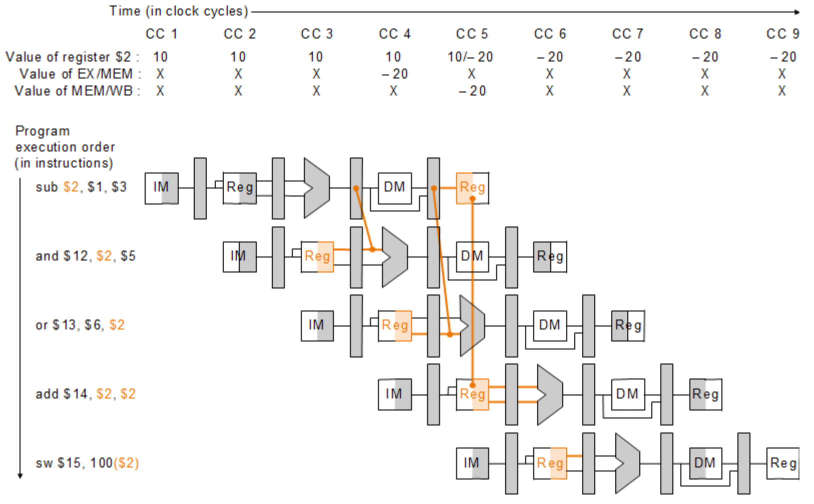
# Data Hazards



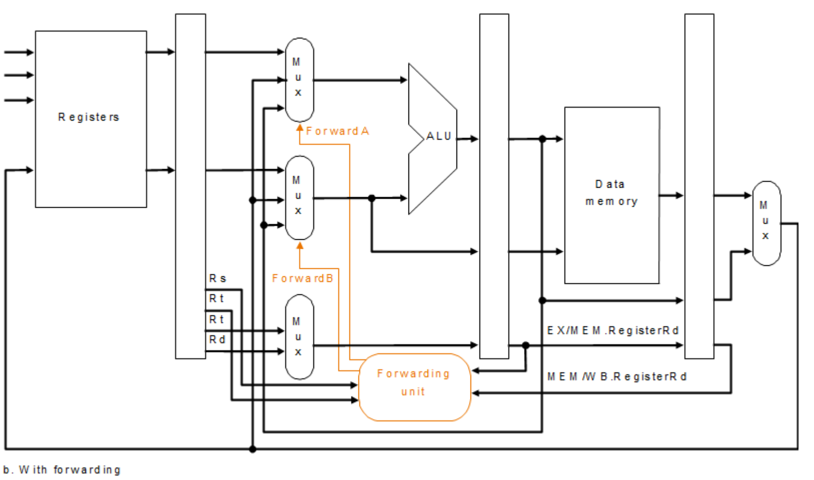

# Data Hazards

- Compilation techniques
  - Insertion of nop (*no operation)* instructions
  - Instructions Scheduling to avoid that correlating instructions are too close
    - The compiler tries to insert independent instructions among correlating instructions
    - When the compiler does not find independent instructions, it Insert nops.
- Hardware techniques
  - Insertion of "*bubbles*" or *stalls* in the pipeline
  - Data Forwarding or Bypassing

# Data Forwarding

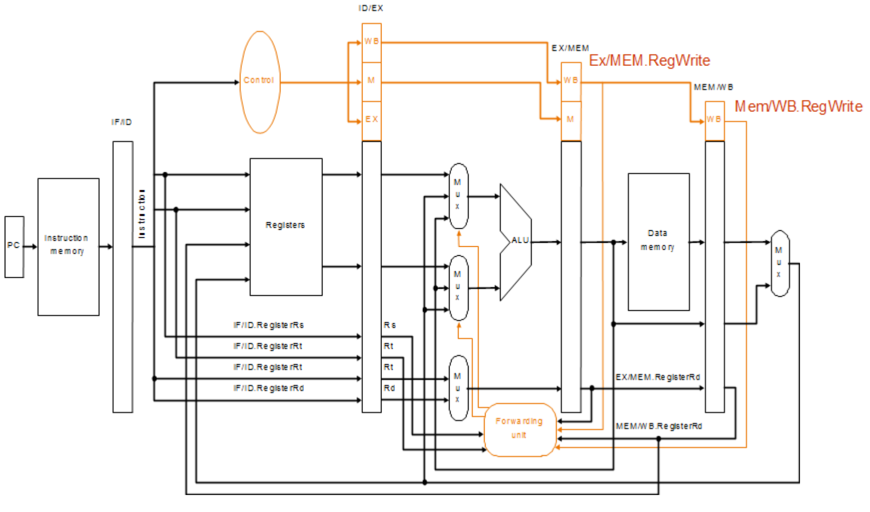



# Forwarding implementation

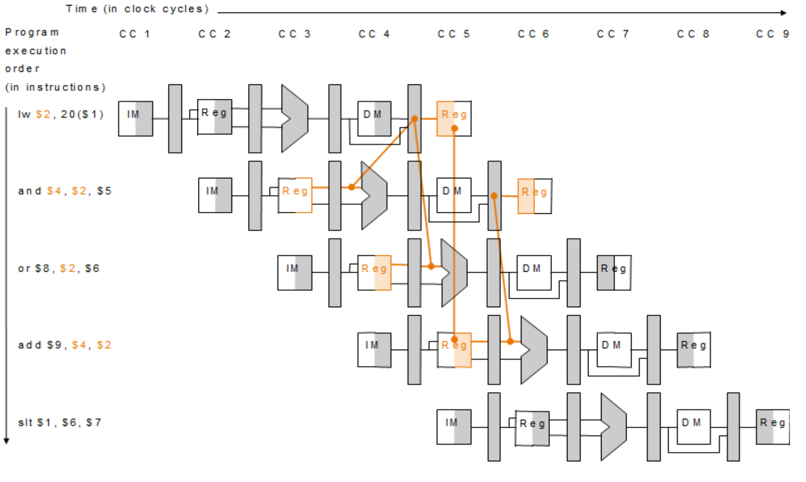



b. With forwarding

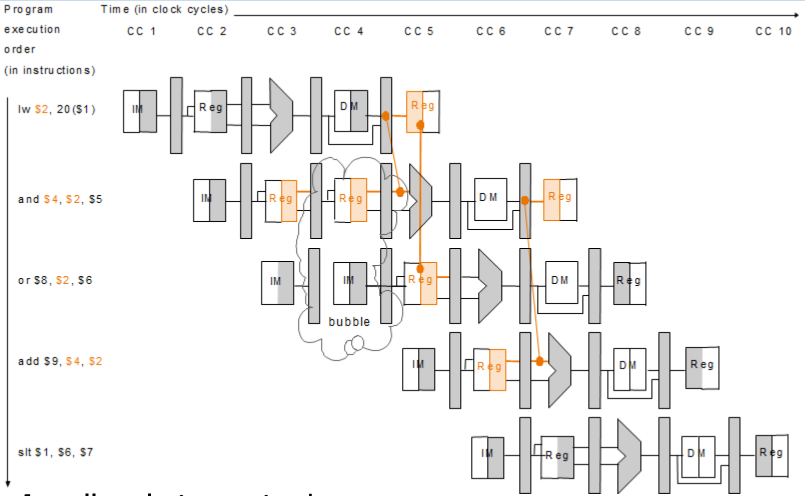
# Forwarding implementation




# Data hazard with lw

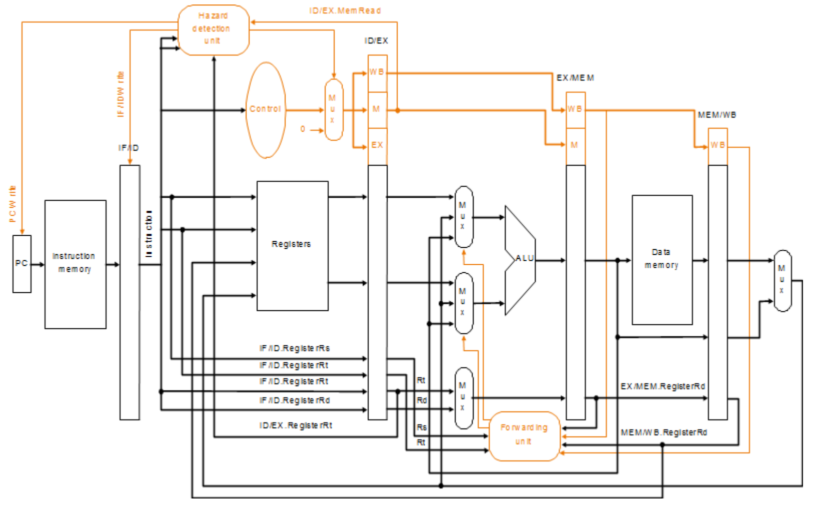


# Data hazard with lw




1 stall cycle is required

# Hazard Detection Unit



# Data Hazards

Data hazards analyzed up to now are:
– RAW (READ AFTER WRITE) hazards:
instruction *n+1* tries to read a source register before the previous instruction *n* has written it in the RF.

Example:

```
add $r1, $r2, $r3
sub $r4, $r1, $r5
```

• By using forwarding, it is always possible to solve this conflict without introducing stalls, except for the load/use hazards where it is necessary to add one stall

# Data Hazards

- **Other types of data hazards in the pipeline:**
  - WAW (WRITE AFTER WRITE)
  - WAR (WRITE AFTER READ)

# Data Hazard: Write After Write

n: lw $r1, 0($r2)

n+1: add $r1,$r2,$r3

- Instruction *n+1* tries to write a destination operand before it has been written by the previous instruction *n*

⇒ write operations executed in the wrong order

- This type of hazards could not occur in the MIPS pipeline because all the register write operations occur in the WB stage and instructions are completed in order

# Data Hazard: Write After Write

- Example: If we assume the register write in the ALU instructions occurs in the fourth stage and that load instructions require two stages (MEM1 and MEM2) to access the data memory, we can have:

| | T0 | T1 | T2 | T3 | T4 | T5 |
|---|---|---|---|---|---|---|
| lw $r1, 0($r2) | IFetch | IDec | IExe | IMem 1 | IMem 2 | IWB |
| add $r1,$r2,$r3 | | IFetch | IDec | IExe | IWB | |

## Data Hazard: Write After Read

n: sw $r1, 0($r2)

n+1: add $r2, $r3, $4

- Instruction *n+1* tries to write a destination operand before it has been read from the previous instruction *n*
- ⇒ instruction *n* reads the wrong value.
- This type of hazards could not occur in the MIPS pipeline because the operand read operations occur in the ID stage and the write operations in the WB stage.

## Data Hazard: Write After Read

- As before, if we assume the register write in the ALU instructions occurs in the fourth stage and that we need two stages to access the data memory, some instructions could read operands too late in the pipeline.
- Example: Instruction sw reads $r2 in the second half of MEM2 stage and instruction add writes $r2 in the first half of WB stage ⇒ sw reads the new value of $r2.

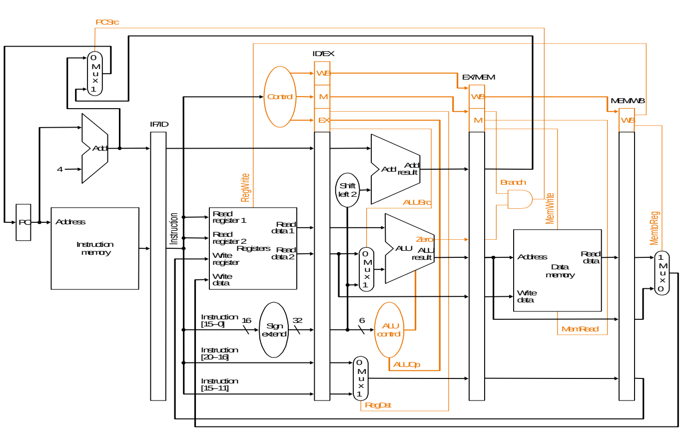
| | T0 | T1 | T2 | T3 | T4 | T5 |
|---|---|---|---|---|---|---|
| sw $r1, 0($r2) | IFetch | IDec | IExe | IMem 1 | IMem 2 | IWB |
| add $r2, $r3, $4 | | IFetch | IDec | IExe | IWB | |

# Control Hazards

- Control hazards: Attempt to make a decision on the next instruction to fetch before the branch condition is evaluated.
- Control hazards arise from the pipelining of conditional branches and other instructions changing the PC.
- Control hazards reduce the performance from the ideal speedup gained by the pipelining since they can make it necessary to stall the pipeline.

# Branch hazards

- To feed the pipeline we need to fetch a new instruction at each clock cycle, but the branch decision (to change or not change the PC) is taken during the MEM stage.



# Branch hazards

- This delay to determine the correct instruction to fetch is called Control Hazard or Conditional Branch Hazard
- If a branch changes the PC to its target address, it is a taken branch
- If a branch falls through, it is not taken or untaken.

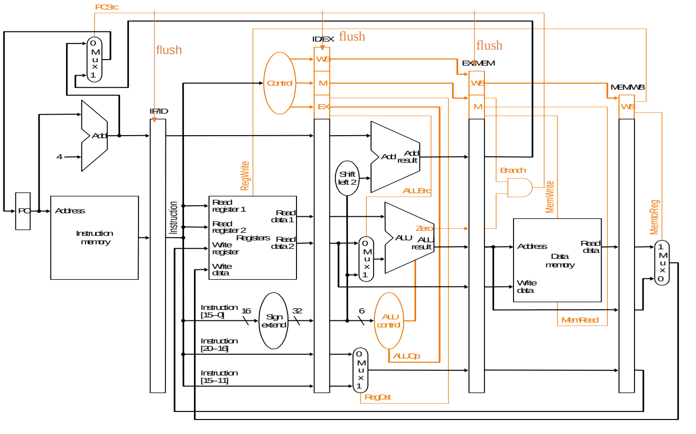
# Branch hazards: solutions

- To stall the pipeline until the branch decision is taken (stalling until resolution) and then fetch the correct instruction flow.
- If the branch is *not taken*, the three cycles penalty is not justified ⇒ throughput reduction.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Branch instruction | IF | ID | EX | MEM | WB | | | | | |
| Branch successor | | IF | ***stall*** | ***stall*** | IF | ID | EX | MEM | WB | |
| Branch successor + 1 | | | | | | IF | ID | EX | MEM | WB |
| Branch successor + 2 | | | | | | | IF | ID | EX | MEM |
| Branch successor + 3 | | | | | | | | IF | ID | EX |
| Branch successor + 4 | | | | | | | | | IF | ID |
| Branch successor + 5 | | | | | | | | | | IF |

# Branch hazards: solutions

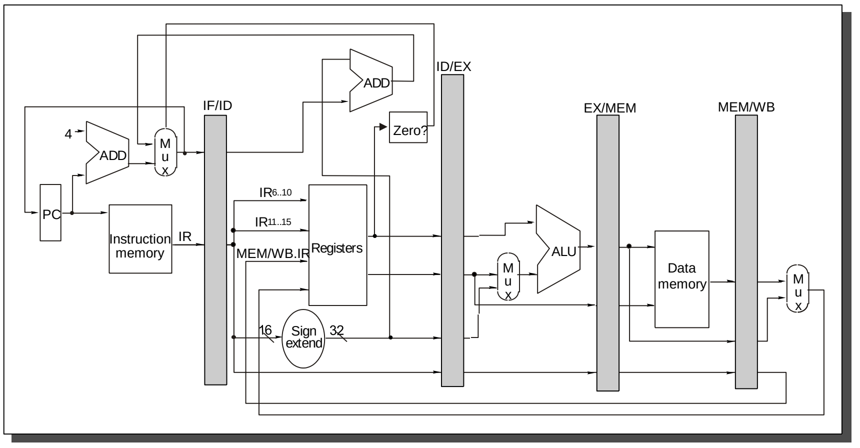
- We can assume the *branch not taken*, and flush the next 3 instructions in the pipeline only if the branch will be taken.



# Early Evaluation of the PC

- To improve performance in case of branch hazards, we need to add hardware resources to:
  - Compare registers
  - Compute branch target address
  - Update the PC register as soon as possible in the pipeline.
- MIPS processor compares registers, computes branch target address and updates PC during ID stage.

# Early Evaluation of the PC




# Branch Prediction Techniques

- Main goal of branch prediction techniques: try to predict ASAP the result of a branch instruction.
- In general, the problem of the branch prediction becomes worse for deeply pipelined processors because the cost of incorrect predictions increases
- The performance of a branch prediction technique depends on:
  - Accuracy measured in terms of percentage of incorrect predictions.
  - Cost of a incorrect prediction measured in terms of time lost to execute useless instructions (misprediction penalty).
- We also need to consider branch frequency: the importance of accurate branch prediction is higher in programs with higher branch frequency.

## Branch Prediction Techniques

- There are many methods to deal with the performance loss due to branch hazards:
  - **Static Branch Prediction Techniques**: The actions for a branch are fixed for each branch during the entire execution. The actions are fixed at compile time.
  - **Dynamic Branch Prediction Techniques**: The decision causing the branch prediction can change during the program execution.
- In both cases, care must be taken not to change the processor state until the branch is definitely known.

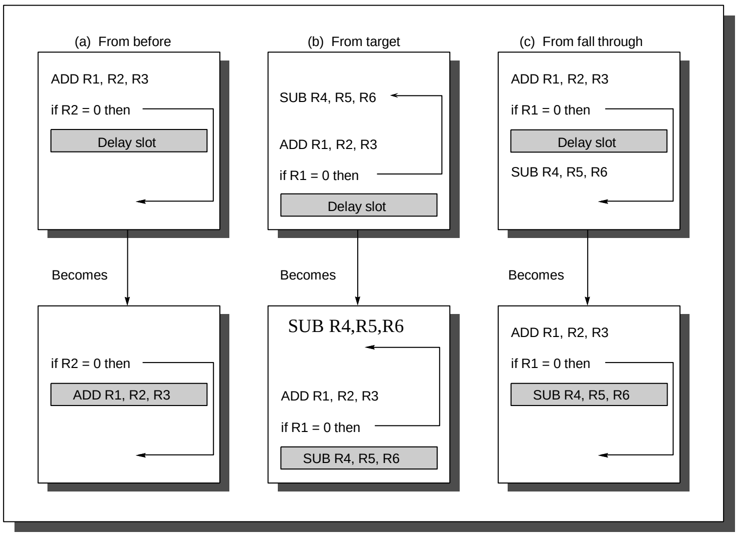
## Static Branch Prediction Techniques

- Branch Always Not Taken (Predicted-Not-Taken)
  - Execute successor instructions in sequence
  - "Squash" instructions in pipeline if branch actually taken
  - Advantage of late pipeline state update
  - 47% DLX branches not taken on average
- Branch Always Taken (Predicted-Taken)
  - 53% DLX branches taken on average
  - But haven't calculated branch target address in MIPS
  - DLX still incurs 1 cycle branch penalty
  - Other machines: branch target known before outcome
- Backward Taken Forward Not Taken (BTFNT)

# Static Branch Prediction Techniques

- Delayed Branch
  - The instruction in the branch delay slot is executed whether or not the branch is taken.
  - The compiler statically schedules an independent instruction in the branch delay slot.

# Branch delay slot



# Dynamic Branch Prediction

- **Basic Idea**: To use the past branch behavior to predict the future.
- We use hardware to *dynamically* predict the outcome of a branch: the prediction will depend on the behavior of the branch at run time and will change if the branch changes its behavior during execution.

# Dynamic Branch Prediction

- Dynamic branch prediction is based on two interactive mechanism:
  - Branch Outcome Predictor:
    - To predict the direction of a branch (i.e. taken or not taken).
  - Branch Target Predictor:
    - To predict the branch target address in case of taken branch.
- These modules are used by the Instruction Fetch Unit to predict the next instruction to read in the I-cache.
  - If branch is not taken $\Rightarrow$ PC is incremented.
  - If branch is taken $\Rightarrow$ BTP gives the target address

# Branch Prediction Buffers

- *The simplest thing to do with a branch is to predict whether or not it is taken.*
- *This helps in pipelines where the branch delay is longer than the time it takes to compute the possible target PCs .*
  - *If we can save the decision time, we can branch sooner.*
- *Note that this scheme does NOT help with the MIPS we studied.*
  - *Since the branch decision and target PC are computed in ID, assuming there is no hazard on the register tested.*

# Branch-Prediction Buffers

**One-bit Prediction Scheme**

- Is a buffer (cache) (BHT - Branch History Table) indexed by the lower portion of the address of the branch instruction
  - The memory contains a bit that says whether the branch was recently taken or not
  - It has no tag
    - ✓ It may have been put there by another branch (that has the same low-order address bits)
  - The prediction is a hint that is presumed to be correct, and fetching begins in the predicted direction
    - ✓ If the hint turns out to be wrong, the prediction bit is inverted and stored back
- The branch direction could be incorrect because:
  - misprediction
  - Instruction mismatch
  - In either case, the worst that happens is that you have to pay the full latency for the branch.

# Branch-Prediction Buffers

## One-bit Prediction Scheme

- Consider a loop branch whose behavior is taken nine times in a row, then not taken once. What is the prediction accuracy for this branch, assuming the prediction bit for this branch remains in the prediction buffer?

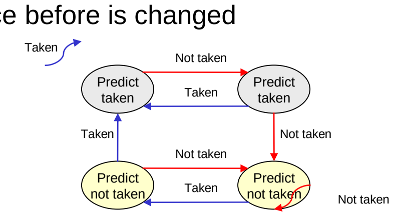
| Branch | Prediction |
|---|---|
| Taken | ? |
| Taken | Taken |
| Taken | Taken |
| … | Taken |
| Taken | Taken |
| Not taken | Taken |
| Taken | Not taken |
| Taken | Taken |
| Taken | Taken |
| … | Taken |
| Taken | Taken |
| Not taken | Taken |
| Taken | Not taken |
| Taken | Taken |
| Taken | Taken |
| … | … |

- The prediction accuracy for this branch that is taken 90% of the time is only **80%** (two incorrect predictions and eight correct ones).

# Branch-Prediction Buffers

## Two-bit Prediction Scheme

- A prediction must miss twice before is changed



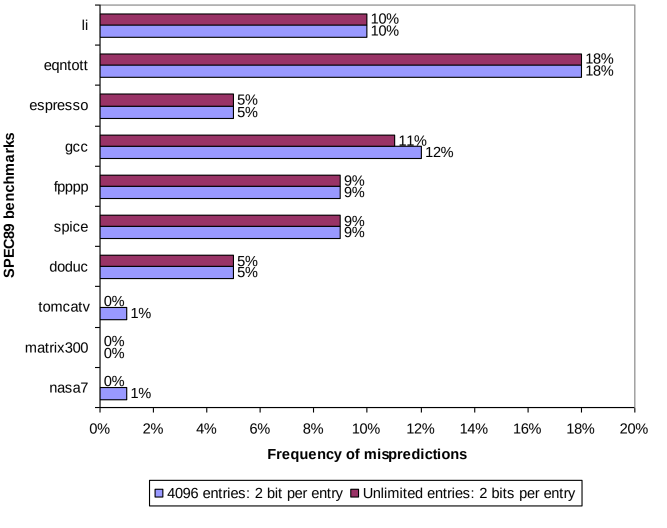

| Branch | Prediction |
|---|---|
| Taken | ? |
| Taken | ? |
| Taken | Taken |
| … | Taken |
| Taken | Taken |
| Not taken | Taken (miss) |
| Taken | Taken |
| Taken | Taken |
| Taken | Taken |
| … | Taken |
| Taken | Taken |
| Not taken | Taken (miss) |
| Taken | Taken |
| Taken | Taken |
| Taken | Taken |
| … | … |

- The prediction accuracy for this branch that is taken 90% of the time is **90%** (one incorrect predictions and nine correct ones)
- The two-bit scheme is actualy a specialization of a more general scheme that has *n*-bit saturating counter for each entry in the prediction buffer
  - ➔ Studies of *n*-bit predictors have shown that two-bit predictors do almost as well, and thus most systems rely on two-bit branch predictors

## Branch Prediction Buffer

- A branch prediction buffer can be implemented as
  - A small special cache accessed with the instruction address during the IF pipe stage
  - A pair of bits attached to each block in the instruction cache and fetched with the instruction
- While this scheme is useful for most pipelines, the DLX pipeline finds out both whether the branch is taken and what the target of the branch is at roughly the same time

## Branch Prediction Buffer

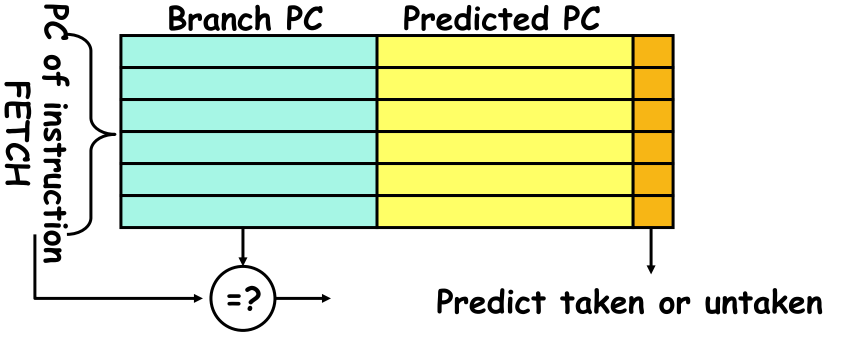


## Branch-Target Buffers

- To reduce the branch penalty on DLX, we need to know from what address to fetch by the end of IF
  - ➔If the instruction is a branch and we know what the next PC should be, we can have a branch penalty of zero!
- **Branch-Target Buffer** (BTB)
  - ➔Is a cache that stores the predicted address for the next instruction after a branch
  - ➔It is accessed during the IF stage using the instruction address of the fetched instruction
  - ➔It only stores the predicted-taken brances

## Branch Target Buffer

- Branch Target Buffer (BTB): Address of branch index to get prediction AND branch address (if taken)
  - Note: must check for branch match now, since can't use wrong branch address



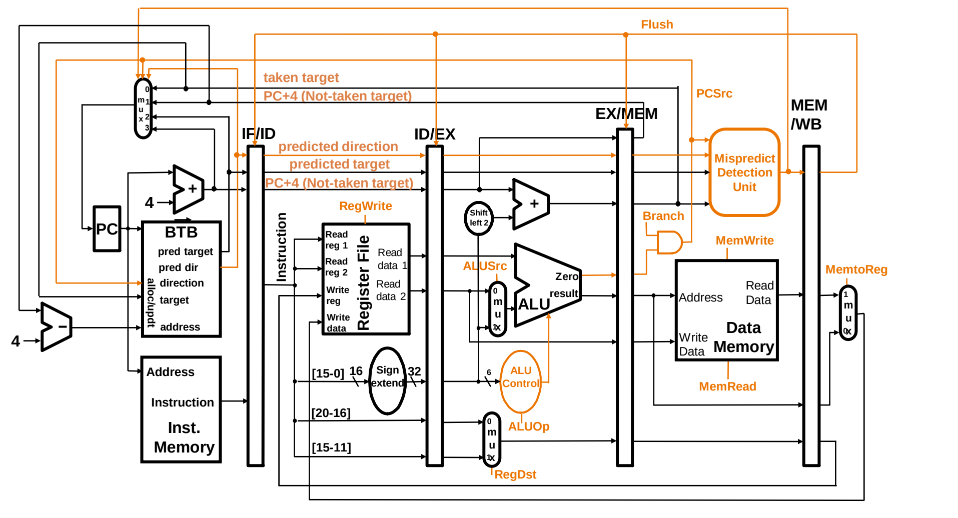
# BTB

- Allocation
  - Allocate instructions identified as branches (after decode)
    - Both conditional and unconditional branches are allocated
  - Not taken branches need not be allocated
    - BTB miss implicitly predicts not-taken
- Prediction
  - BTB lookup is done parallel to IC lookup
  - BTB provides
    - Indication that the instruction is a branch (BTB hits)
    - Branch predicted target
    - Branch predicted direction
    - Branch predicted type (e.g., conditional, unconditional)
- Update (when branch outcome is known)
  - Branch target
  - Branch history (taken / not-taken)

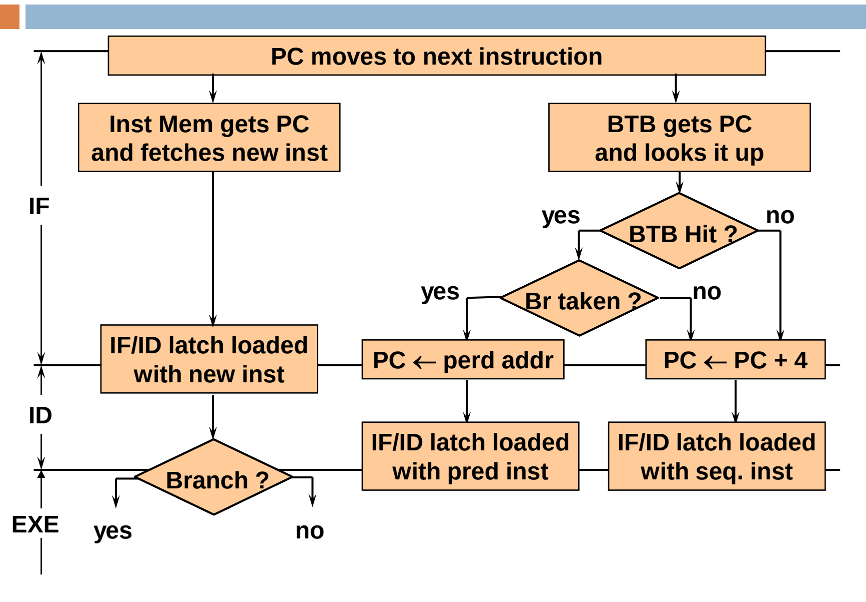
# BTB (cont.)

- Wrong prediction
  - Predict not-taken, actual taken
  - Predict taken, actual not-taken
- In case of wrong prediction – flush the pipeline
  - Reset latches (same as making all instructions to be NOPs)
  - Select the PC source to be from the correct path
    - Need get the fall-through with the branch
  - Start fetching instruction from correct path

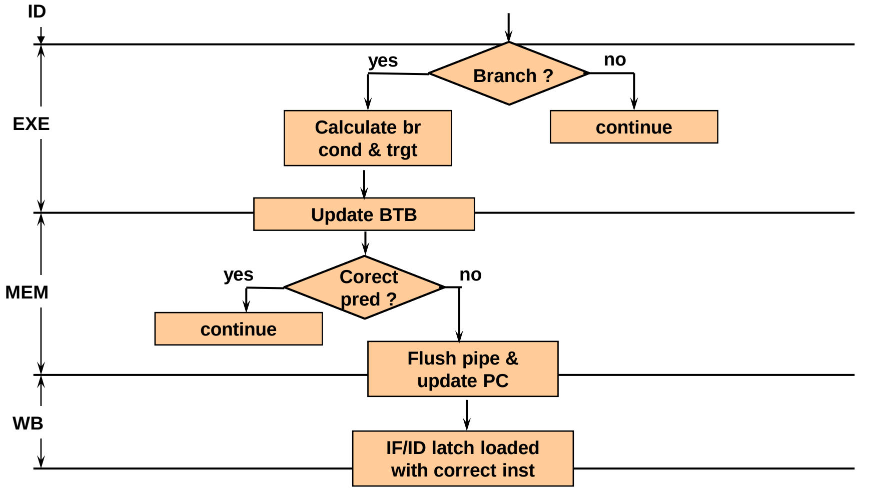
# Adding a BTB to the Pipeline




# Using The BTB



# Using The BTB (cont.)



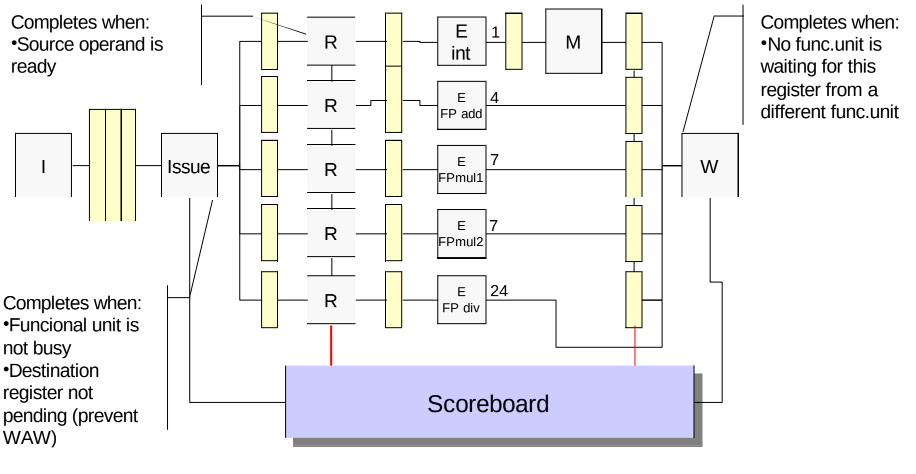

# Performance Improvement

- Performance can be improved by:
  - Faster clock (but there's a limit)
  - Pipelining: slice up instruction into stages, overlap stages
  - *Multiple ALUs* to support more than one instruction stream

# Superscalar

- Multiple ALU which can operate in parallel
- Fetches instructions in batches,
- Executes as many as possible instructions
  - Instructions without hazards can be executed in parallel
  - May require extensive hardware to detect independent instructions (dynamic scheduling)
  - Out of order execution
- Illusion of in order sequential execution ( from the point of view of programmer/compiler
- A superscalar implementation does not change instruction Set Architecture

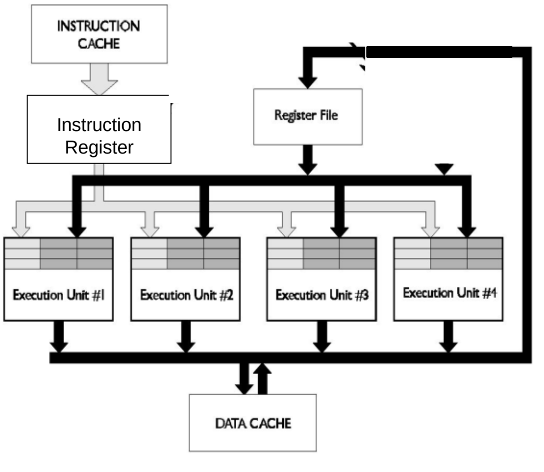
# Superscalar



# VLIW

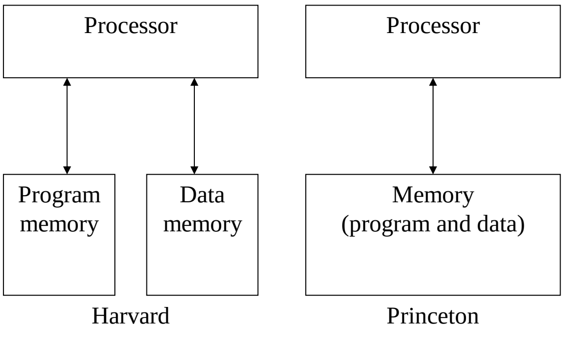
- Each word in memory has multiple independent instructions
- Rely on software for identifying potential parallelism and schedule instructions (static scheduling)
- Processors expect dependency-free code generated by the compiler
- No hardware scheduler, no hardware management of hazards
  - VLIW can be smaller, cheaper, and require less power to operate
- Currently growing in popularity

# VLIW



## Two Memory Architectures

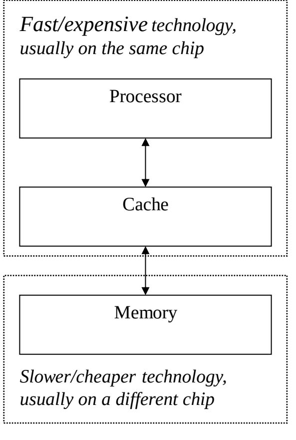
- Princeton
  - Fewer memory wires
- Harvard
  - Simultaneous program and data memory access






## Cache Memory

- Memory access may be slow
- Cache is small but fast memory close to processor
  - Holds copy of part of memory
  - Hits and misses



## Programmer's View

- Programmer doesn't need detailed understanding of architecture
  - Instead, needs to know what instructions can be executed
- Two levels of instructions:
  - Assembly level
  - Structured languages (C, C++, Java, etc.)
- Most development today done using structured languages
  - But, some assembly level programming may still be necessary
  - Drivers: portion of program that communicates with and/or controls (drives) another device
    - Often have detailed timing considerations, extensive bit manipulation
    - Assembly level may be best for these



## Assembly-Level Instructions

| | | | |
|---|---|---|---|
| Instruction 1 | opcode | operand1 | operand2 |
| Instruction 2 | opcode | operand1 | operand2 |
| Instruction 3 | opcode | operand1 | operand2 |
| Instruction 4 | opcode | operand1 | operand2 |
| | | ... | |

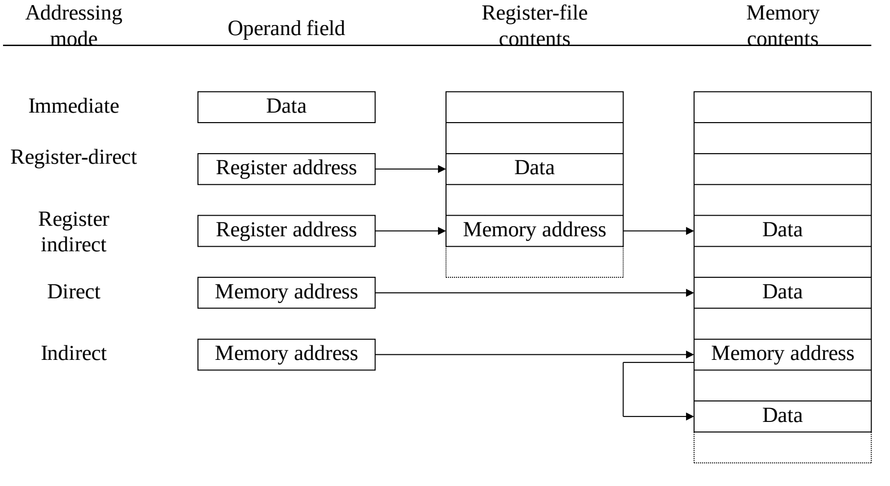
- Instruction Set
  - Defines the legal set of instructions for that processor
    - Data transfer: memory/register, register/register, I/O, etc.
    - Arithmetic/logical: move register through ALU and back
    - Branches: determine next PC value when not just PC+1

# A Simple (Trivial) Instruction Set

| Instruction | Opcode | | Operands | | Meaning |
|---|---|---|---|---|---|
| MOV Rn, direct | 0000 | Rn | direct | | Rn = M(direct) |
| MOV direct, Rn | 0001 | Rn | direct | | M(direct) = Rn |
| MOV @Rn, Rm | 0010 | Rn | Rm | | M(Rn) = Rm |
| MOV Rn, #immed. | 0011 | Rn | immediate | | Rn = immediate |
| ADD Rn, Rm | 0100 | Rn | Rm | | Rn = Rn + Rm |
| SUB Rn, Rm | 0101 | Rn | Rm | | Rn = Rn - Rm |
| JZ Rn, relative | 0110 | Rn | relative | | PC = PC+ relative (only if Rn is 0) |



# Addressing Modes



# Sample Programs

C program

```
int total = 0;
for (int i=10; i!=0; i--)
   total += i;
// next instructions...
```

Equivalent assembly program

```
0       MOV R0, #0;       // total = 0
1       MOV R1, #10;      // i = 10
2       MOV R2, #1;       // constant 1
3       MOV R3, #0;       // constant 0
Loop:   JZ R1, Next;      // Done if i=0
5       ADD R0, R1;       // total += i
6       SUB R1, R2;       // i--
7       JZ R3, Loop;      // Jump always
Next:   // next instructions...
```

- Try some others
  - Handshake: Wait until the value of M[254] is not 0, set M[255] to 1, wait until M[254] is 0, set M[255] to 0 (assume those locations are ports).
  - (Harder) Count the occurrences of zero in an array stored in memory locations 100 through 199.



# Programmer Considerations

- Program and data memory space
  - Embedded processors often very limited
    - e.g., 64 Kbytes program, 256 bytes of RAM (expandable)
- Registers: How many are there?
  - Only a direct concern for assembly-level programmers
- I/O
  - How communicate with external signals?
- Interrupts

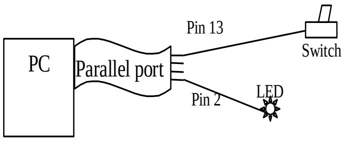
# Microprocessor Architecture Overview

- If you are using a particular microprocessor, now is a good time to review its architecture



# Example: parallel port driver

| LPT Connection Pin | I/O Direction | Register Address |
|---|---|---|
| 1 | Output | $0^{th}$ bit of register #2 |
| 2-9 | Output | $0^{th}$ bit of register #2 |
| 10,11,12,13,15 | Input | 6,7,5,4,$3^{th}$ bit of register #1 |
| 14,16,17 | Output | 1,2,$3^{th}$ bit of register #2 |

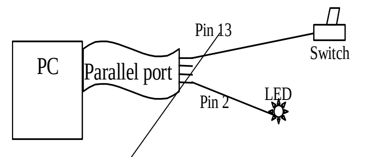



- Using assembly language programming we can configure a PC parallel port to perform digital I/O
  - write and read to three special registers to accomplish this table provides list of parallel port connector pins and corresponding register location
  - Example : parallel port monitors the input switch and turns the LED on/off accordingly

# Parallel Port Example

```
; This program consists of a sub-routine that reads
; the state of the input pin, determining the on/off state
; of our switch and asserts the output pin, turning the LED
; on/off accordingly
       .386

CheckPort     proc
       push   ax             ; save the content
       push   dx                    ; save the content
       mov    dx, 3BCh + 1   ; base + 1 for register #1
       in     al, dx         ; read register #1
       and    al, 10h        ; mask out all but bit # 4
       cmp    al, 0          ; is it 0?
       jne    SwitchOn       ; if not, we need to turn the LED on

SwitchOff:
       mov    dx, 3BCh + 0   ; base + 0 for register #0
       in     al, dx         ; read the current state of the port
       and    al, f7h        ; clear first bit (masking)
       out    dx, al         ; write it out to the port
       jmp    Done           ; we are done

SwitchOn:
       mov    dx, 3BCh + 0   ; base + 0 for register #0
       in     al, dx         ; read the current state of the port
       or     al, 01h        ; set first bit (masking)
       out    dx, al         ; write it out to the port

Done:  pop    dx             ; restore the content
       pop    ax             ; restore the content
CheckPort     endp
```

```
extern "C" CheckPort(void);        // defined in
                                   // assembly
void main(void) {
       while( 1 ) {
              CheckPort();
       }
}
```




| LPT Connection Pin | I/O Direction | Register Address |
|---|---|---|
| 1 | Output | 0th bit of register #2 |
| 2-9 | Output | 0th bit of register #2 |
| 10,11,12,13,15 | Input | 6,7,5,4,3th bit of register #1 |
| 14,16,17 | Output | 1,2,3th bit of register #2 |



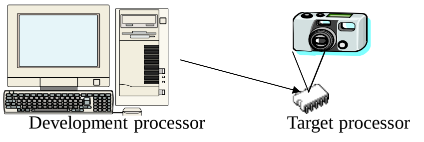
# Operating System

- Optional software layer providing low-level services to a program (application).
  - File management, disk access
  - Keyboard/display interfacing
  - Scheduling multiple programs for execution
    - Or even just multiple threads from one program
  - Program makes system calls to the OS

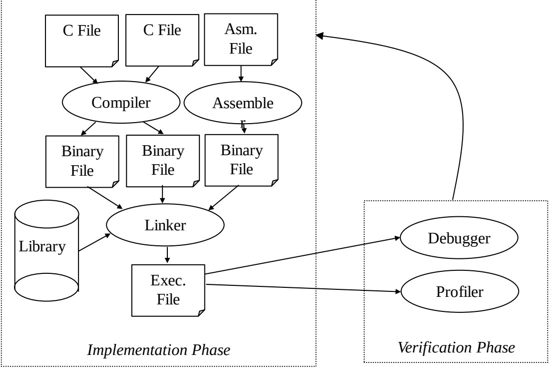
```
DB file_name "out.txt" -- store file name

MOV R0, 1324           -- system call "open" id
MOV R1, file_name      -- address of file-name
INT 34                 -- cause a system call
JZ  R0, L1             -- if zero -> error

   . . . read the file
JMP L2                 -- bypass error cond.
L1:
   . . . handle the error

L2:
```

# Development Environment

- Development processor
  - The processor on which we write and debug our programs
    - Usually a PC
- *Target processor*
  - The processor that the program will run on in our embedded system
    - Often different from the development processor




# Software Development Process




- Compilers
  - Cross compiler
    - Runs on one processor, but generates code for another
- Assemblers
- Linkers
- Debuggers
- Profilers

# Running a Program

- If development processor is different than target, how can we run our compiled code? Two options:
  - Download to target processor
  - Simulate
- Simulation
  - One method: Hardware description language
    - But slow, not always available
  - Another method: *Instruction set simulator (ISS)*
    - Runs on development processor, but executes instructions of target processor

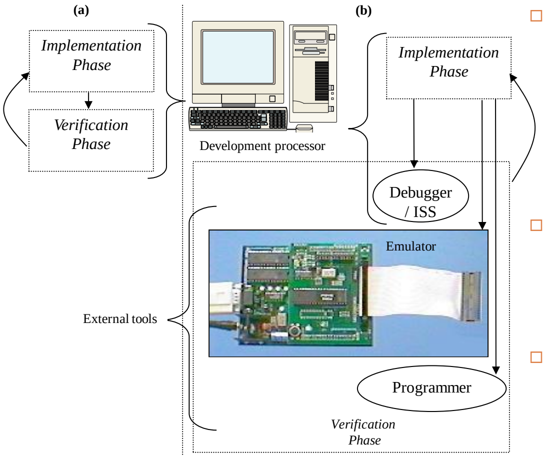


# Instruction Set Simulator For A Simple Processor

```
#include <stdio.h>
typedef struct {
   unsigned char first_byte, second_byte;
} instruction;

instruction program[1024];  //instruction memory
unsigned char memory[256];  //data memory

void run_program(int num_bytes) {

   int pc = -1;
   unsigned char reg[16], fb, sb;

   while( ++pc < (num_bytes / 2) ) {
      fb = program[pc].first_byte;
      sb = program[pc].second_byte;
      switch( fb >> 4 ) {
         case 0: reg[fb & 0x0f] = memory[sb]; break;
         case 1: memory[sb] = reg[fb & 0x0f]; break;
         case 2: memory[reg[fb & 0x0f]] =
                 reg[sb >> 4];  break;
         case 3: reg[fb & 0x0f] = sb; break;
         case 4: reg[fb & 0x0f] += reg[sb >> 4]; break;
         case 5: reg[fb & 0x0f] -= reg[sb >> 4]; break;
         case 6: pc += sb; break;
         default: return -1;
```

```
}
   }
   return 0;
}

int main(int argc, char *argv[]) {

   FILE* ifs;

   If( argc != 2 ||
       (ifs = fopen(argv[1], “rb”) == NULL ) {
            return -1;
   }
   if (run_program(fread(program,
       sizeof(program) == 0) {
      print_memory_contents();
      return(0);
   }
   else return(-1);
}
```

# Testing and Debugging

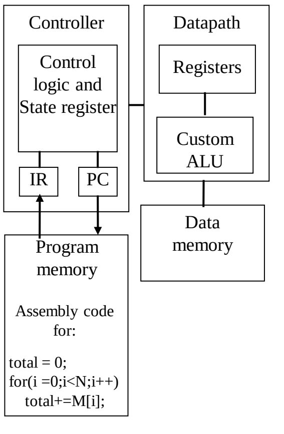



- ISS
  - Gives us control over time – set breakpoints, look at register values, set values, step-by-step execution, ...
  - But, doesn't interact with real environment
- Download to board
  - Use device programmer
  - Runs in real environment, but not controllable
- Compromise: emulator
  - Runs in real environment, at speed or near
  - Supports some controllability from the PC



# Application-specific processors

- Programmable processor optimized for a particular class of applications having common characteristics
  - Compromise between general-purpose and single-purpose processors
- Features
  - Program memory
  - Optimized datapath
  - Special functional units
- Benefits
  - Some flexibility, good performance, size and power
- Drawbacks
  - High NRE cost (processor and compiler)
- Examples: Microcontroller, DSP



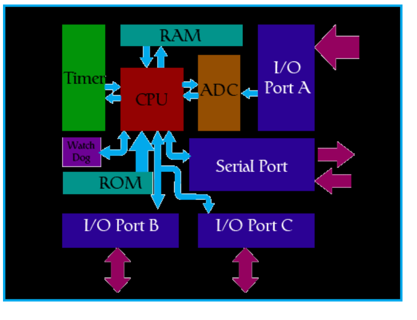
# Application-Specific Instruction-Set Processors (ASIPs)

- General-purpose processors
  - Sometimes too general to be effective in demanding application
    - e.g., video processing – requires huge video buffers and operations on large arrays of data, inefficient on a GPP
  - But single-purpose processor has high NRE, not programmable
- ASIPs – targeted to a particular domain
  - Contain architectural features specific to that domain
    - e.g., embedded control, digital signal processing, video processing, network processing, telecommunications, etc.
  - Still programmable



# A Common ASIP: Microcontroller

- For embedded control applications
  - Reading sensors, setting actuators
  - Mostly dealing with events (bits): data is present, but not in huge amounts
  - e.g., VCR, disk drive, digital camera (assuming SPP for image compression), washing machine, microwave oven




- Microcontroller features
  - On-chip peripherals
    - Timers, analog-digital converters, serial communication, etc.
    - Tightly integrated for programmer, typically part of register space
  - On-chip program and data memory
  - Direct programmer access to many of the chip's pins
  - Specialized instructions for bit-manipulation and other low-level

## Digital Signal Processors (DSP)

- For signal processing applications
  - Large amounts of digitized data, often streaming
  - Data transformations must be applied fast
  - e.g., cell-phone voice filter, digital TV, music synthesizer
- DSP features
  - Several instruction execution units
  - Multiple-accumulate single-cycle instruction, other instrs.
  - Efficient vector operations – e.g., add two arrays
    - Vector ALUs, loop buffers, etc.

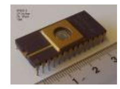


## Trend: Even More Customized ASIPs

- In the past, microprocessors were acquired as chips
- Today, we increasingly acquire a processor as Intellectual Property (IP)
  - e.g., synthesizable VHDL model
- Opportunity to add a custom datapath hardware and a few custom instructions, or delete a few instructions
  - Can have significant performance, power and size impacts
  - Problem: need compiler/debugger for customized ASIP
    - Remember, most development uses structured languages
    - One solution: automatic compiler/debugger generation
      - e.g., www.tensillica.com
    - Another solution: retargettable compilers
      - e.g., www.improvsys.com (customized VLIW architectures)

# Microcontroller: ST6

- 8-bit Microcontroller



  - **Memories**
    Up to 4 Kbytes of program memory OTP/ROM
    Up to 64 bytes of RAM
  - **I/O Ports**
    Up to 20 I/O lines
    Multifunctional, bi-directional I/O pins
    Up to 4 high current capability I/O line
  - **Clock, Reset and Power Supply**
    Power supply operating range: 3.0V to 6V
    Maximum external frequency: 8 MHz
    Oscillator Safeguard (OSG) and Backup oscillator (LFAO)
    Low Voltage Detector (LVD)
    2 power saving modes: WAIT and STOP
  - **Interrupts**
    4 interrupt vectors plus NMI and RESET
    Software programmable for each I/O
- **I/O Ports**
  Up to 20 I/O lines
  Multifunctional, bi-directional I/O pins
  Up to 4 high current capability I/O line
- **Peripherals**
  Watchdog timer
  8-bit timer
  ADC
- **Instruction Set**
  8-bit accumulator-based architecture
  40 instructions
  9 addressing modes



# Microcontroller: STR7(ARM7TDMI® core)

- STR710F Flash Microcontrollers from STMicroelectronics combine the industry standard ARM7TDMI® RISC microprocessor with embedded Flash and powerful peripheral functions including, USB and CAN.