

ADS1 Zusammenfassung

Matthias Jung

9. September 2009

1 Number Systems:

Number System - Definition:

$$S = (b, Z, \delta)$$

b : Basis of the number ($b \geq 2$)

Z : Set consisting of $|b|$ digits

$\delta : Z \rightarrow \{0, 1, \dots, b-1\}$ is a transfer which assigns a natural number

Examples:

Decimal	$b = 10$	$Z = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
Binary	$b = 2$	$Z = \{0, 1\}$
Octal	$b = 8$	$Z = \{0, 1, 2, 3, 4, 5, 6, 7\}$
Hexadecimal	$b = 16$	$Z = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f\}$

Fixed Point Numbers:

$$\underbrace{d_{n-1} \ d_{n-2} \ \dots \ d_1 \ d_0}_{n \text{ digits left}}, \underbrace{d_{-1} \ \dots \ d_{-k}}_{k \text{ digits right}}$$

Binary to Decimal:

$$\langle d \rangle = \sum_{i=-k}^{n-1} 2^i \cdot d_i$$

Decimal to Binary:

Left:

$$\frac{\langle d \rangle}{2} = \sum_{i=0}^{n-2} 2^i \cdot d_{i+1} \quad \text{Remainder} = d_0 \quad \dots$$

Right:

$$\langle d \rangle \cdot 2 = d_{-1} + \sum_{i=-k}^{-2} 2^i \cdot d_{i+1}$$

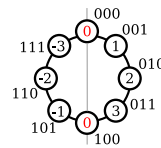
Alignment in Memory: 0110₂ in a 16bit word:

| 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 |
MSB.....LSB

Signed Fixed Point Numbers:

Sign and Magnitude:

$$\langle d_{n-1} \dots d_0, d_{-1} \dots d_{-k} \rangle = (-1)^{d_{n-1}} \cdot \sum_{i=-k}^{n-2} 2^i d_i$$



Problem: double 0 representation

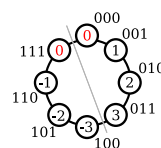
Symmetric Range:

Smallest: $-(2^{n-1} - 2^{-k})$

Greatest: $2^{n-1} - 2^{-k}$

One's Complement:

$$\langle d_{n-1} \dots, \dots d_{-k} \rangle = \sum_{i=-k}^{n-2} 2^i d_i - d_{n-1} \cdot (2^{n-1} - 2^{-k})$$



Problem: double 0 representation

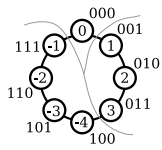
Symmetric Range:

Smallest: $-(2^{n-1} - 2^{-k})$

Greatest: $2^{n-1} - 2^{-k}$

Two's Complement:

$$\langle d_{n-1} \dots, \dots d_{-k} \rangle = \sum_{i=-k}^{n-2} 2^i d_i - d_{n-1} \cdot (2^{n-1})$$



Good: No more double 0

Asymmetric Range:

Smallest: $-(2^{n-1})$

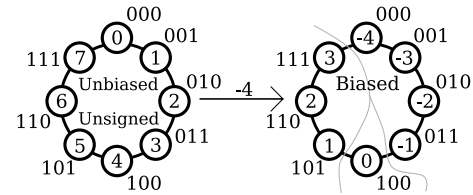
Greatest: $2^{n-1} - 2^{-k}$

Benefit: Simple HW performing (add,sub...)

Biased:

$$\langle d_{n-1} \dots, \dots d_{-k} \rangle = \sum_{i=-k}^{n-2} 2^i d_i - BIAS$$

Typically: $BIAS = 2^{n+k-1}$ or $BIAS = 2^{n+k-1} - 1$



Good: No more double 0

Asymmetric Range:

Smallest: $-BIAS$

Greatest: $2^n - 2^{-k} - BIAS$

Benefit: Sorting can be done as for unsigned

Floating Point Numbers

$$(-1)^S \cdot M \cdot 2^E \quad M: \text{Mantissa}, S: \text{Sign}, E: \text{Exponent}$$

IEEE 754 Standard:

Single precision:

31	30	29	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	Exponent										Mantissa																			

Double precision:

63	62	61	..	..	..	..	..	..	..	53	52	51	50	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	1	0
S	Exponent										Mantissa																					

Mantissa:

The number is Normalized if $M = 1.m_{-1} \dots m_{-k}$ and $1 \leq M < 2$

The 1. is not stored (hidden bit) $\rightarrow M = 1 + \sum_{i=-1}^{-k} 2^i \cdot m_i$

Exponent:

The exponent is biased (32bit=127 and 64bit? 1023) $\rightarrow E = \sum_{i=0}^{n-1} 2^i \cdot e_i - BIAS$ Range -126 to 127

Special Values:

- $E = 2^n - 1$ (111... = 128):

$M = 0 \rightarrow \pm\infty$

Some M bits are 1 $\rightarrow$ NaN

- $E = 0$ (-127)

$M = 0 \rightarrow 0$

Else: denormalized Number, hidden bit not used:

$$N = (-1)^S \cdot \left(\sum_{i=-k}^{-1} 2^{i+1} \cdot m_i \right) 2^{-127}$$

Addition of Floating Point Numbers:

1. Align the Smaller exponent to larger Exponent (Shift Mantissas and round...)
2. Add Mantissas
3. Normalize and round (if necessary)

Multiplying of Floating Point Numbers:

1. Multiply signs ($S_a \oplus S_b$)
2. Multiply mantissas
3. Add exponents
4. Normalize and round (if necessary)

2 Computer Arithmetics:

2.1 Basic Components:

Shifting: (e.g. Barrel Shifter)

Shift operand by k digits to the left/right.

Shift Right/Left Logical: (srl, sll)

Fill empty digits with 0

Shift Right/Left Arithmetic: (sra, sla)

Preserve sign bit:

sra: copy most significant bit into empty digits

sla: do not shift most significant bit,
fill empty with 0

Features:

$$\text{sra}(a) \hat{=} \left\lfloor \left(\frac{a}{2} \right) \right\rfloor$$

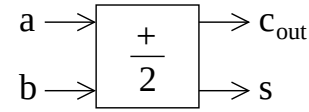
(Division by 2 without remainder)

$$\text{sla}(a) \hat{=} 2 \cdot a$$

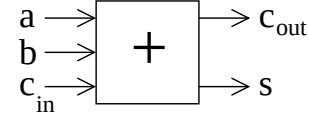
(Multiplication by 2 if $a_{n-1} = a_{n-2}$ else overflow)

1-Bit Adder:

Half Adder:



Full Adder:



Calculation of result:

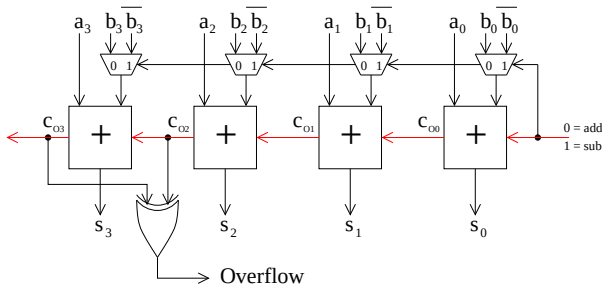
$$s = (a \oplus b) \oplus c_{in}$$

Calculation of carry out:

$$c_{out} = a \cdot b + c_{in}(a \oplus b)$$

2.2 Compound of basic Components:

Ripple Carry Adder:



Properties of Ripple Carry Adder:

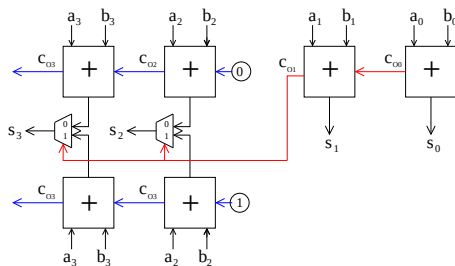
Critical path: carry chain (carry ripples through the adders). The delay increases linearly with number of adders

Overflow detection:

Unsigned: if $c_{o3} = 1$

Signed: if $c_{o3} \neq c_{o2}$

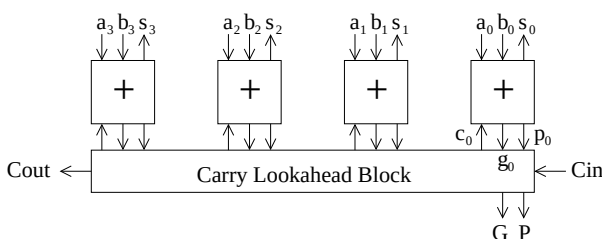
Carry Select Adder:



Properties of the Carry Select Adder:

Split number of adders into blocks and perform the addition in parallel, assuming the carry out of the first block. (More splits are possible but the area grows)

Carry Lookahead Adder:



Properties of Carry Lookahead Adder:

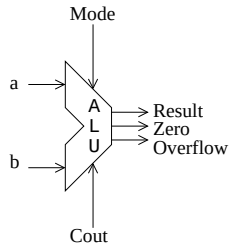
The idea is to calculate the carry signals directly from the inputs and not for the predecessor states.

$$c_{out} = \underbrace{a \cdot b}_{\text{Generation}} + c_{in} \cdot \underbrace{(a \oplus b)}_{\text{Propagation}}$$

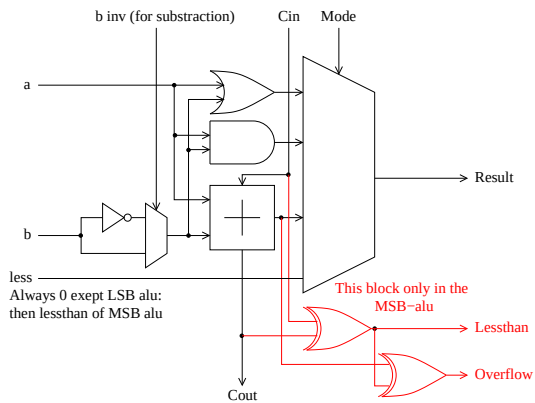
Insert recursive the equation in itself to get the equation for each carry signal.

(In general no more than 4 bits are added in parallel to prevent CL-Block become to large.)

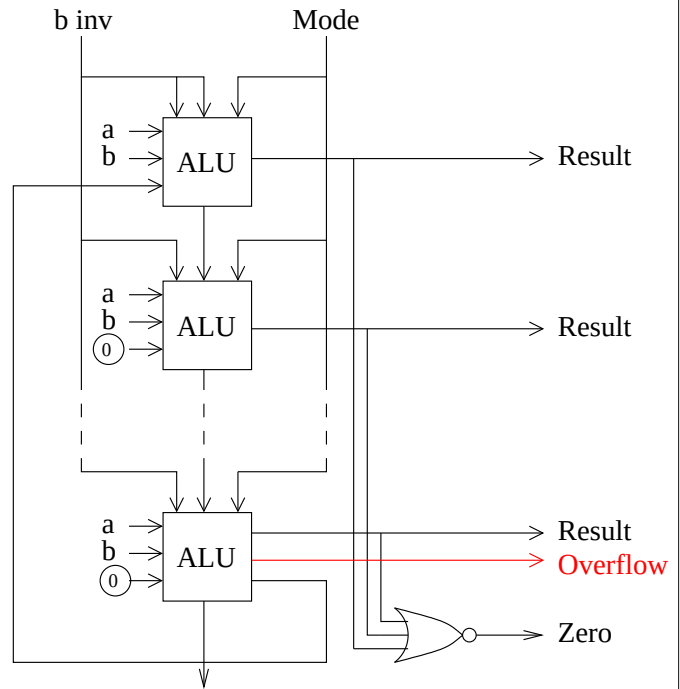
Arithmetic Logical Unit:



Symbol



1-Bit Alu



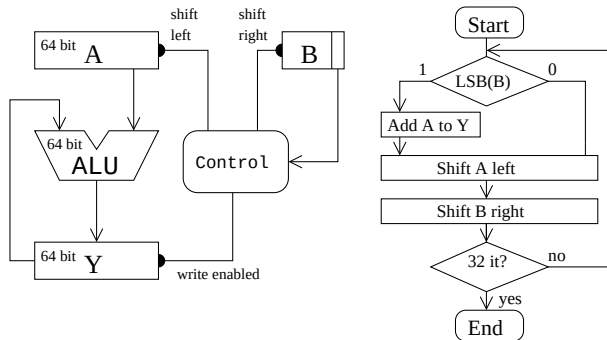
N-Bit Ripple Carry Alu

Comparison: $sign \oplus overflow \hat{=} a < b$

Signed Multiplier: See script

Unsigned Multiplier: See script

Sequential Multiplier:



Properties of Sequential Multiplier:

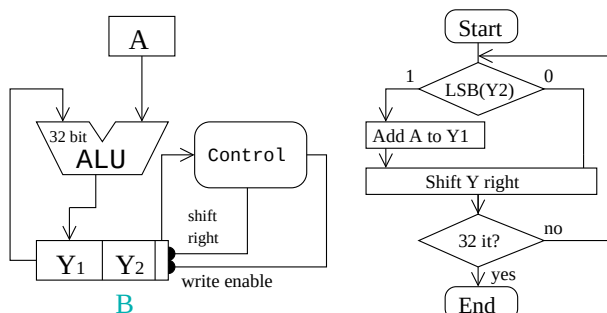
- Use add operation of ALU for summing up the partial products
- Distribute multiplication over several clock cycles
- Store multiplicand, multiplier and intermediate result in registers

$$\underbrace{Y}_{\text{Product}} = \underbrace{A}_{\text{Multiplicand}} \cdot \underbrace{B}_{\text{Multiplier}}$$

For each bit b_i of the multiplier B (from right to left):

- If $b_i == 1$ add A to intermediate sum Y
- Shift A left

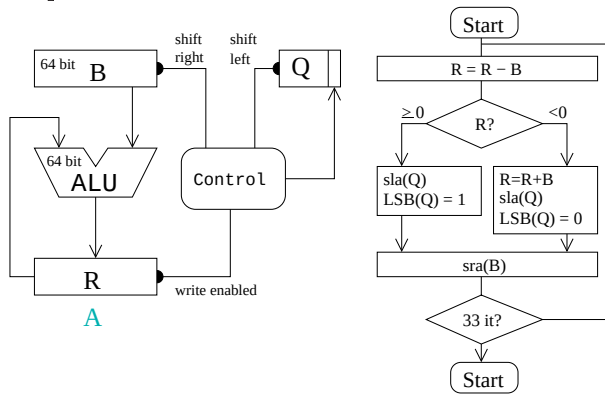
Sequential Multiplier IMPROVED:



IMPROVED Sequential Multiplier:

- Shift Y right instead A left $\rightarrow$ only a 32 bit ALU is needed.
- Store B initial in Y_2

Sequential Divisor:



Properties of Sequential Divisor:

$$\underbrace{\overbrace{A}^{\text{Divident}} : \overbrace{B}^{\text{Divisor}}}_{\text{Quotient}} = \underbrace{X}_{\text{Value of Quotient}}$$

$$\underbrace{\overbrace{A}^{\text{Divident}}}_{\text{Quotient}} = \underbrace{\frac{A}{B}}_{\text{Quotient}} \cdot \underbrace{\overbrace{B}^{\text{Divisor}}}_{\text{Quotient}} + \text{Remainder}$$

Both Method:

Implentation of multiplication with shift & add/subtract
Example:

$$88 \cdot 126 = 88 \cdot (128 - 2) = (88 \ll 7) - (88 \ll 1)$$

While stepping through a block containing only 1s perform for each bit only a shift insted of an add & shift add the multiplicand after having reached the end of the block. Remove the "error" that has been made by subtracting the multiplicand in advance.

Current and previous LSB(B):

00	0 → 0	shift
11	1 → 1	shift
01	1 → 0	add A to Y & shift
00	0 → 0	sub A from Y & shift

We save add/sub operations but we need still n clockcycles

Improvement:

Use a barrelshifter and priority encoder.

3 Instruction Set

Definition:

The instruction set defines the basic operations a processor is able to execute. All computations have to be mapped onto a sequence of basic instructions.

- Less instructions lead to larger machine language programmes
- More instructions require more sophisticated compilers

To simplify the hardware design all instructions with the same length.

Reduced Instruction Set Computing (RISC):

- Operands are stored in Registers
- Refisters corresponding to variables in high level language
- MIPS:
 - \$sX Variable
 - \$tX Auxillary registers for compiler

R-Type: Register - Register Instruction

Opcode	Source	Source	Target	Shift	Func
6Bit	5Bit	5Bit	5Bit	5Bit	6Bit

Shift: shift amount for shifting operations

Func: Subtype of Operatoion

(add & sub have same Opcode but different Func)

MIPS example:

add \$s1,\$s2,\$s3 # S3 = S1 + S2

I-Type: Immediate Instruction

Opcode	Source	Target	Immidiade Value
6Bit	5Bit	5Bit	16Bit

Use for operations use a constant value being known at compile time

MIPS example:

addi \$s1,\$s2, 4 # S2 = S1 + 4

Data Transfer Instructions:

Many Programs use more variables than registers are available → Variables are stored in Memory and copied into registers when they are needed. The process of moving the content of registers being less frequently used to memory is called *register spilling*.

Alignment Restriction: 32-Bit word always starts at addresses being multiple of 4.

Big Endian:

x+0	x+1	x+2	x+3
-----	-----	-----	-----

Little Endian:

x+3	x+2	x+1	x+0
-----	-----	-----	-----

Load/Store-Type:

Opcode 6Bit	Source 5Bit	Target 5Bit	Address Offset 16Bit
----------------	----------------	----------------	-------------------------

MIPS example:

lw \$t0, 32(\$s3)

Take the content of \$s3 as a memory address and load the data being stored there into \$t0

Branching Instructions:

(Conditional and unconditional jumps, call subprogram, return from subprogram)

Branches: evaluate a certain condition and jump to a certain target when the condition holds

Jumps: jump always to a certain target

Branch Formats:

Jump:

Opcode 6Bit	Jump Address 26Bit
----------------	-----------------------

Branch:

Opcode 6Bit	Source 5Bit	Target 5Bit	Branch Address 16Bit
----------------	----------------	----------------	-------------------------

More than 2^{16} Words → PC=PC+Branch Address

Further Instructions:**Jump and Link:**

Stores address of the instruction to be executed immediately after returning.

Shift and Rotation:

e.g. Logic & arithmetic shift

String Instructions:

e.g. copy and compare bytearrays

Processor Control Instructions:

e.g. RESET, STOP, TRAP, NOP

Stack:

For saving register contents of main routine a stack is used (LIFO). The stackpointer is a register which points to the top of the stack. The framepointer is a register which points to the first stack entry belonging to the running procedure

Stack operations:

Push: SP− and insert word at SP

Pop: Remove word at SP and SP++

3.1 Addressing modes:

Register addressing: 0-Stage**Explicit:**

Operand is the content of a register. the instruction contains the number of the register explicit.

Implicit:

Instruction always accesses on a register implicitly (label)

Immediate addressing 0-Stage

The instruction contains operand value:

e.g. addi \$s1,\$s1,4

1-Stage:**Direct:**

Instruction contains address direct (j)

Indirect:

Instruction contains number of register which contains the address (jr)

Index Addressing:

The effective address is calculated from a base (in reg. or mem.) and an offset being contained in the instruction or in a register. (lw, sw)

PC-Relative Addressing:

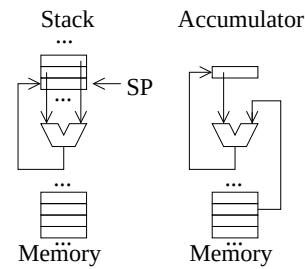
Index addressing with PC as base

3.2 Instruction Set Architectures

Instruction set architectures differ in the way where the operands are located immediately before an instruction is executed. Today Reg-Memory and Reg-Reg are mostly used

Stack and Accumulator Machine:

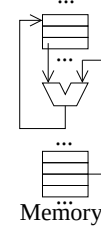
- + Simple Hardware
- Accessing operand is more complicated than for machines using general purpose registers
- Many memory access



Register-Memory Machine:

- + Data can be accessed without using an additional load instruction
- Number of clockcycles used per instruction may vary depending on addressing mode
- Instruction sizes may vary depending on addressing mode

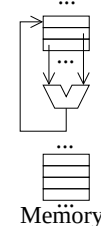
Register-Memory



Register-Register Machine:

- + Simple (only a few instruction formats)
- + Many instructions require similar number of clockcycles
- + 3 register addresses (source, source, target) can be easily implemented into a 32 bit instruction
- + fast
- Larger programs because of additional load/store instructions

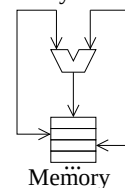
Register-Register



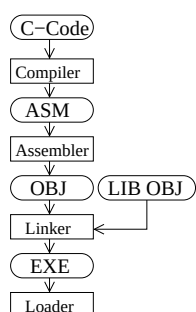
Memory-Memory Machine:

- + Simple Hardware (no registers)
- memory access limit speed

Memory-Memory



3.3 Building a Program



Compiler:

Translates high level language program into assembler language

Assembler:

Translates symbolic ASM program into object files (binary). The assembler assigns addresses to labels (symbol table) and translates pseudo assembler instructions.

Linker:

Many parts of programs rarely change and are stored in a library. The linker binds object and library files and recalculates the addresses of the labels.

Loader:

The loader allocates space in Memory and copies the executable into it.

3.4 Instruction execution

1. Instruction Fetch:

- fetch instruction from memory
- compute the address of the next sequential instruction

2. Decode and Fetch Operands:

- Read operands specified by *register source* and *register target* field in instruction into A and B
- Calculate branch target address (ALU doesn't do anything else)

3. Execution Step:

Action being performed depends on instruction type:

- load / store: calculate address of memory cell being referenced using ALU
- arithmetic / logical: perform operation using A and B and store result in T
- conditional branches: compare A and B using ALU. Depending on the Result write target address to the PC (**3 φ**)
- unconditional jump: replace PC nby the jump target address (**3 φ**)

4. Memory Access / Execution Completion:

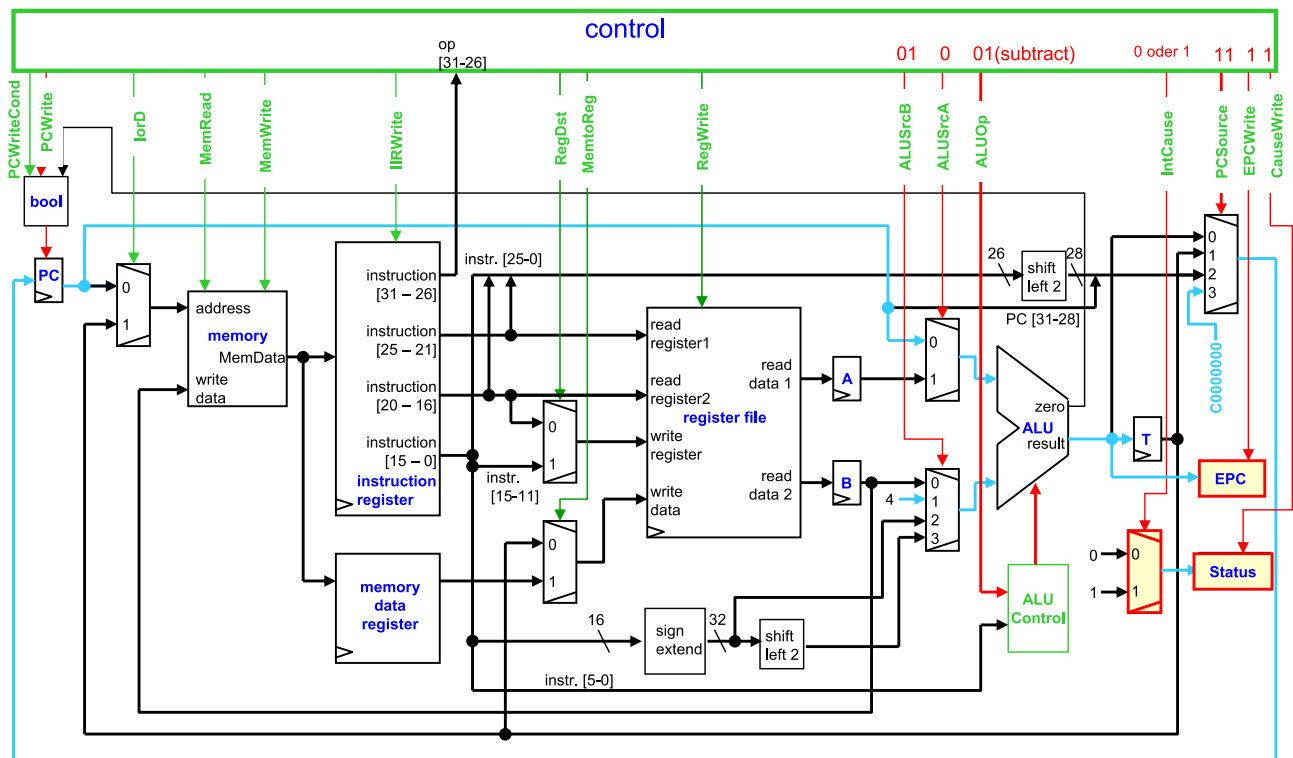
- load / store:
 - load: read data from memory and store in MDR
 - store: write content of operand register B into memory (**4 φ**)
- arithmetic / logical: store the calculated result in the target register (**4 φ**)

5. Memory Read Completion:

- store content of MDR in target register (**5 φ**)

4 Datapath and Control:

For instance: MIPS Datapath (*Lecture slides ADS1 UNI-KL 5-28*)



Instructions may require different execution times (lw 5, j LABEL 2)

Single-Cycle Machine:

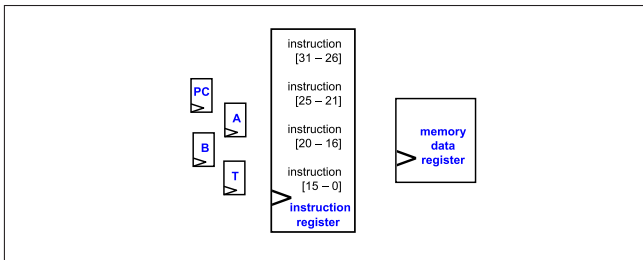
In a single-cycle implementation the cycle time is determined by the most complex instruction
CPI: Clock per instruction

Multy-Cycle Machine:

Each instruction is broken into a series of steps. The execution of each step take one clock cycle

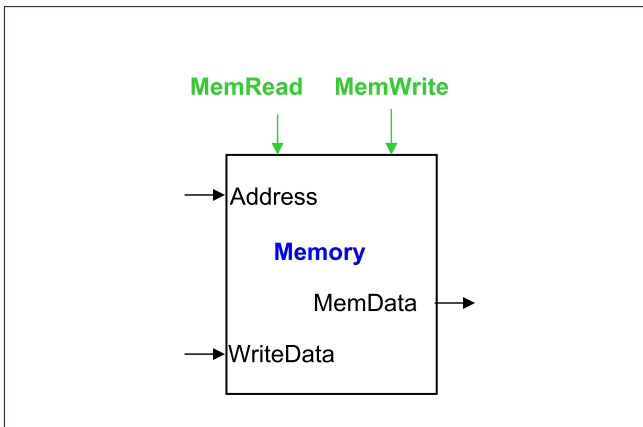
- + The execution time can be adapetet to its complexity
- + Functional units may be shared of the instructions (pipelining)
- Needs additionaly temprary registers for storing intermediate results of the instruction

4.1 Components:



Temporary Registers:

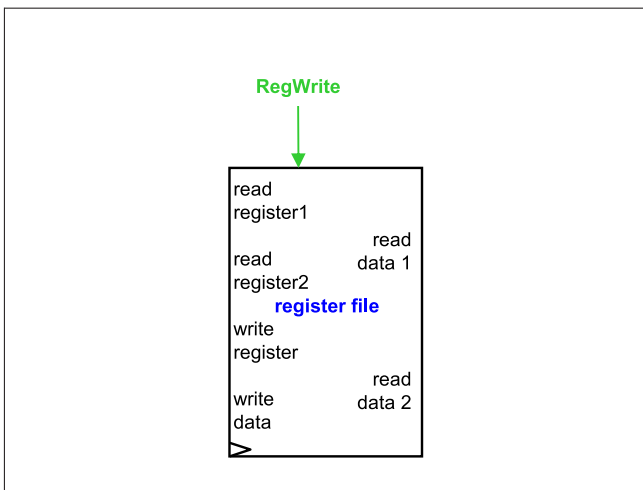
- IR: Instrucion register
- MDR: Memory data register
- A,B: Operand register
- T: Alu target register
- PC: Programm Counter



Memory:

The Memory stores instructions and data.

- Data Signals:
 - Write Data (input)
 - MemData (output)
- Adress Signals:
 - Address: Which cell should be affected?
- Control Signals:
 - MemRead set to 1 to read
 - MemWrite set to 1 to write

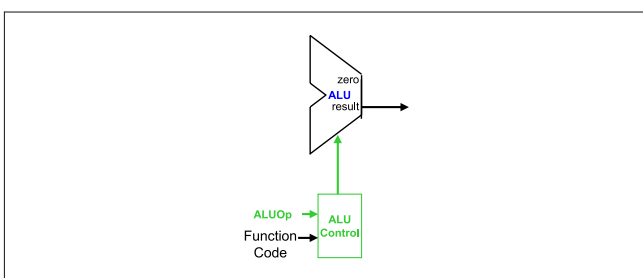


Register File:

The Memory stores instructions and data.

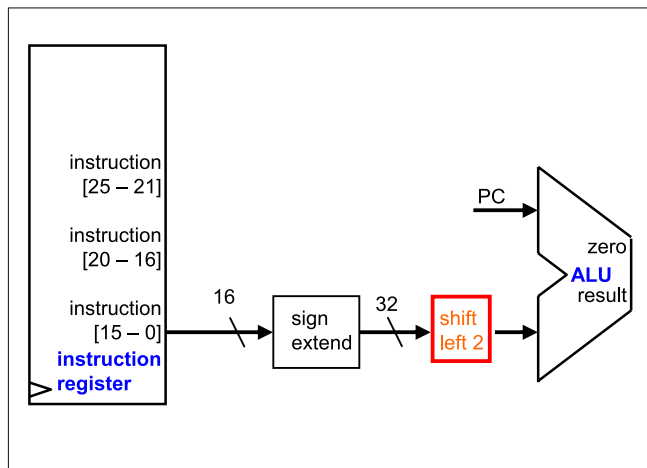
- Data Signals:
 - 1 Data input
 - 2 Data output
- Adress Signals:
 - 3 Address input
- Control Signals:
 - RegWrite (determines whether or not write is performed)

Two register may be read and one register may be written simultaneously



ALU:

- For arithmetic / logical instructions.
- For address calculation, condition testing
- Alucontrol: combinatorial circuit determining the operation being executed



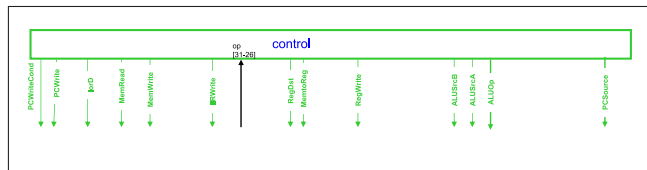
Sign Extension:

Load Store instructions contain a 16 bit signed address offset. This address offset must be transformed into a 32 bit value.

Copy the most significant bit into the left half of the 32 bit word the right half remains unchanged

Shifting:

Conditional branches contains a 16-bit offset being added to content of the PC when the target address of the branch is being calculated. Because the alignment restriction the offset will be multiplied by 4 → shift 2 bits left (also unconditional branches...)



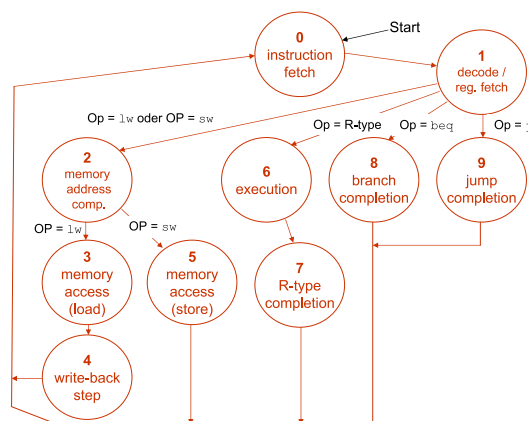
Control Unit:

Implement as a finite state machine.

Input: opcodefield of instruction

Output: controlling signals for modules

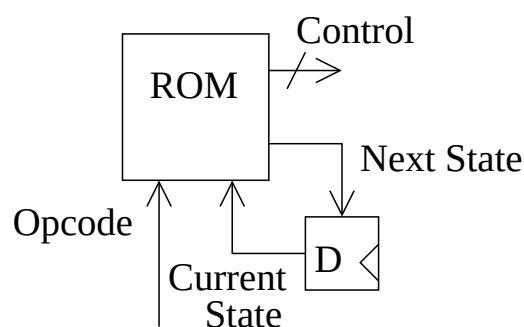
4.2 Control Unit:



Implementation as ROM:

Use a ROM for implementing functions, calculating output and next state. The ROM address consists of the Opcode field and the current state. The corresponding memory word consists of control signals.

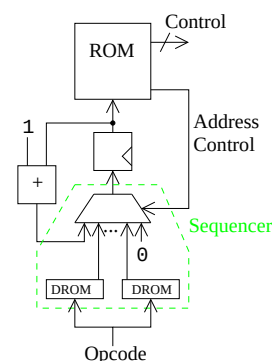
Disadvantage: ROM can become too large for real big instruction sets.



Implementation as ROM with Sequencer:

Datapath control signals depend only on current state → Separate calculation of next state from calculation of the control signals (Sequencer)

For each branch in the state transition diagram there is one dispatch table containing the next states



Implementation as Microprogramm:

The control unit is designed by writing a program that implements each machine instruction as a sequence of microinstructions.

Microprogramm:

Symbolic representation of the behaviour of the control unit.

- Every microinstruction consists of several fields
- Fields control basic actions in the data path
- Each field corresponds to a set of control signals
- Each field can take one of several symbolic values
- Symbolic values are translated by a microassembler into binary for control signals
- Additional field sequencing: determines how to select the next instruction
 - Seq: increment current address by 1
 - Fetch: begin executing a new instruction
 - Dispatch i: determine next microinstruction by using a dispatch table (in ROM)

Exceptions and Interrupts:

Exception: Processor Internal

- Division by zero
- Overflow during arithmetic operation
- Instruction with illegal opcode

Interrupt: Processor External

- Communication request from I/O

Handling:

- Store address of current instruction in exception program counter (EPC)
- store cause of interrupt in status register
- call a dedicated procedure of the operating system (ISR)
- The first instruction of the ISR is stored in the exception address (mostly 0000)

5 Instruction-Level Parallelism

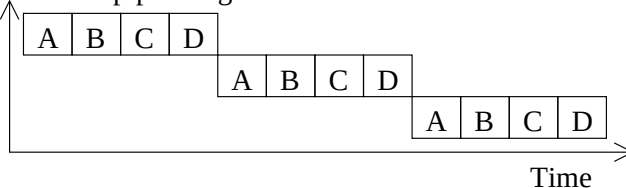
Modern processors apply techniques for executing several instructions in parallel to enhance the computing power

5.1 Pipelining

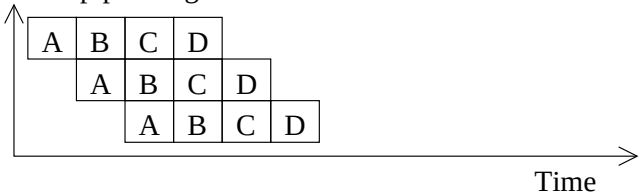
Principle:

The execution of a machine instruction is divided into several steps called *pipeline stages* (same execution time). These stages may be executed in parallel

Without pipelining:



With pipelining:



MIPS 5 Pipeline stages:

- 1 IF: Instruction fetch
- 2 ID: Instruction decode and register file read
- 3 EX: execution / memory address calculation
- 4 MEM: data memory access
- 5 WB: result writeback

Structural Hazard:

Non-pipelined multi-cycle CPU has shared ALU for executing arithmetic/logical instructions and incrementing the PC. With pipelining there is now a problem!

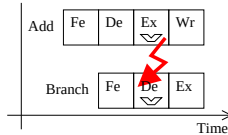
Definition structural hazard:

Two instructions wish to use a certain hardware component in the same clock cycle leading to a *resource conflict*

Solution → use additional hardware

Avoiding Structural Hazards:

- Permit incrementing PC and executing arithmetic / logical instructions concurrently:
→ Use separate adder for incrementing PC
- Permit reading next instruction and reading/writing data from memory:
→ Divide memory into instruction and data parts (Harvard)
- Permit executing an arithmetic / logical instruction followed by a branch:
→ Use a separate adder for branch address calculation



→ Use a separate adder for branch address calculation

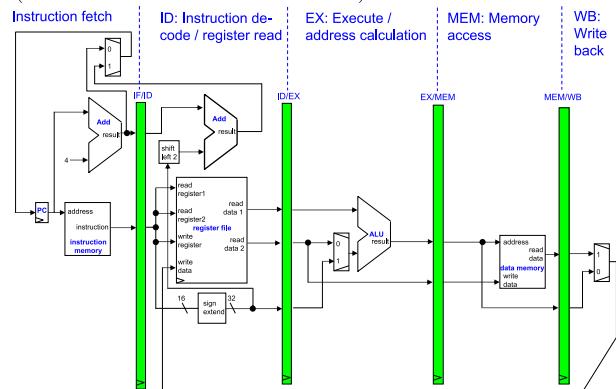
Duplicating hardware components in general leads to less and / or smaller multiplexers

Avoiding Conceptual Bugs:

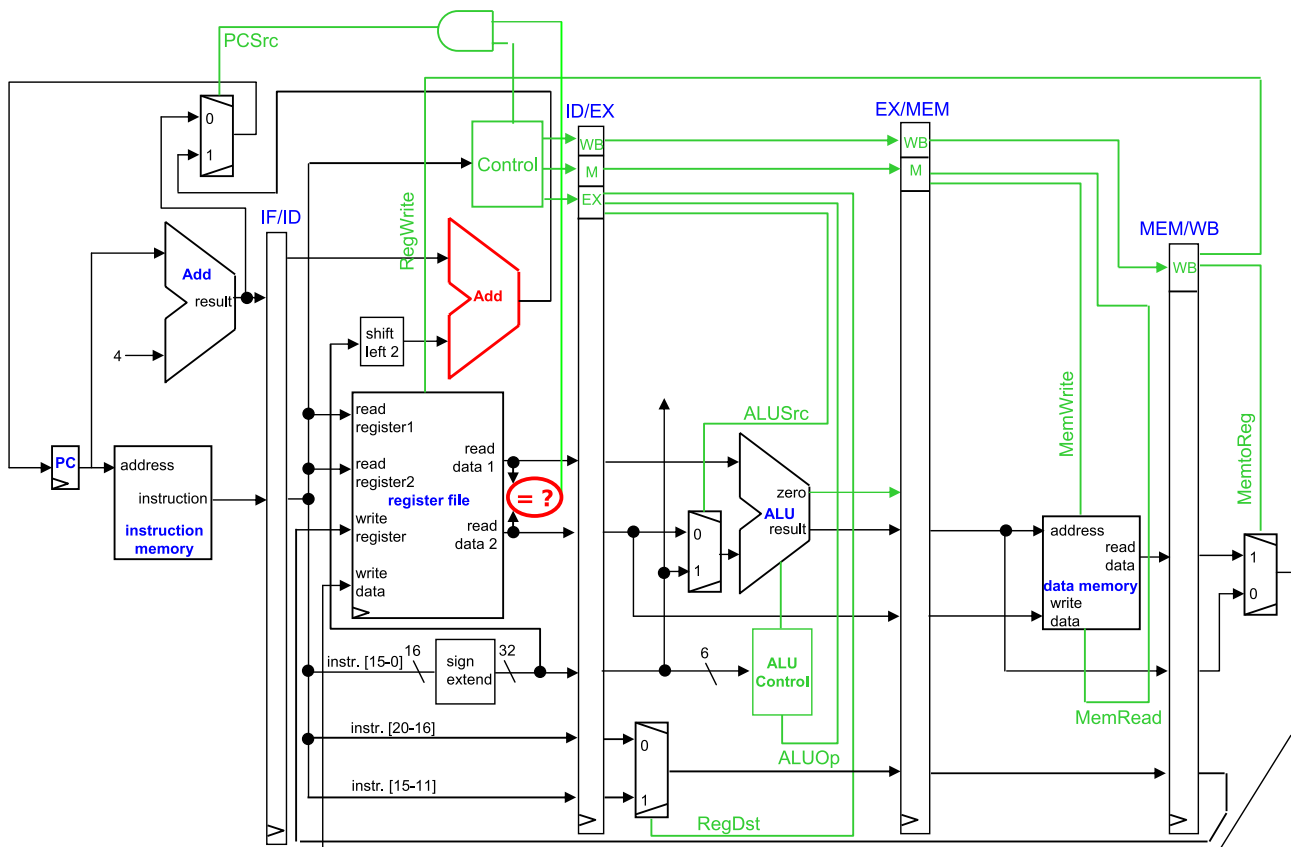
BUG: e.g. load instruction writes result into wrong register: the used register number belongs to the instruction that has just been fed into to pipeline!

Pipelined Datapath 1. Approach

(Lecture Slides ADS1 TU-KL)

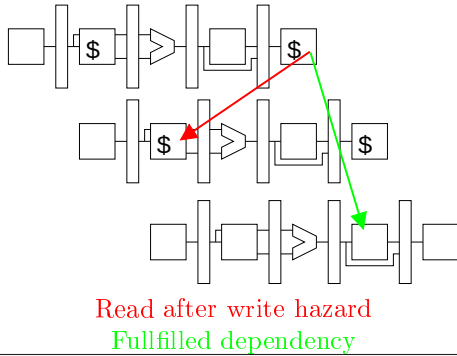


Solution: Hand over control signals to the next stage using the corresponding pipeline register



Consider an instruction a that precedes an instruction b in program order:

Data Dependences - Data Hazard



Dependences:

Data Dependence:

a writes into a register that will be read by b

Antidependence:

b writes into a register that will be read by a

Outputdependence:

a and b write into the same register

Data Hazards:

A data hazard is created whenever the overlapping (pipelined) execution of a and b would change the order of access to the operands which are involved in the dependency

read after write:

b read source before a writes it (b gets the old value)

write after read:

b writes an operand before a can read it (a gets the new value)

write after write:

b writes an operand before it is write by a

Resolving with NOP:

The compiler can resolve the data hazards by test the program for potential data hazards and insert NOP's to eliminate them.

Modern processors are able to detect data hazards during execution by analyzing the register numbers of the instructions using additional control logic. Then the pipeline must be stalled.

Resolving with Forwarding:

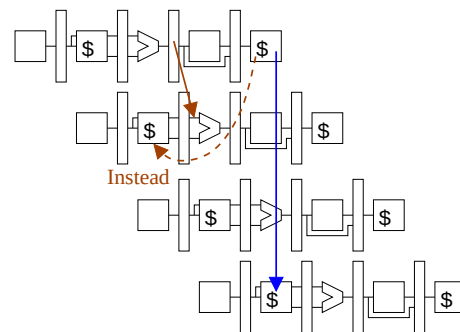
The ALU may read operands from each of the pipeline registers. The correct operands are selected by multiplexers that are controlled by the *forwarding unit*.

Inputs for Forwarding Unit:

- Register operand numbers of instruction in EX stage
- Target register number of instructions being in MEM and WB stage
- Control signals indicating type of instructions being in MEM and WB stage

Register numbers are stored and moved forward in the pipeline registers

Forwarding Example:



Data required by subsequent instructions exists already in pipeline register

If a register is read and written in the same clock cycle, send new data to data output

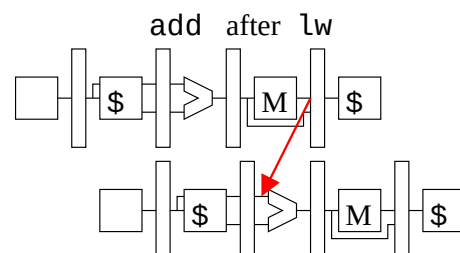
Limitation of Forwarding:

We have to stall the pipeline for combinations of a load followed by an instruction that reads its result. Therefore we need additional hazard detection hardware

Stall:

Stalling the pipeline means to repeat all actions from the previous clock cycle in the corresponding stages. (PC and IF/ID register must be prevented from being overwritten)

Hazard despite forwarding:



Control (or Branch) Hazards:

In pipelining an instruction is fetched at every clock cycle. If there is a Branch we start executing instructions before we know whether they are really part of the program flow! If the branch is not taken we can continue normally. Else we have to discard all instructions from the pipeline.

Avoiding Control Hazards:

The earlier we know whether a branch will be taken, the fewer instructions need to be flushed from the pipeline.

We integrate an additional comparator into the ID-Phase. → Only one instruction needs to be flushed.

Optimization: Branch Delay Slot

Don't flush the instruction from the pipeline! An instruction following immediately after a branch is always executed

Strategy by compiler:

- Place an instruction in the branch delay slot which is not affected by the branch
- If no suitable instruction is found place a NOP there.

Typically the compiler/assembler will fill about 50% of all delay slots with useful instructions

5.2 Concurrent execution

Processors with several functional units:

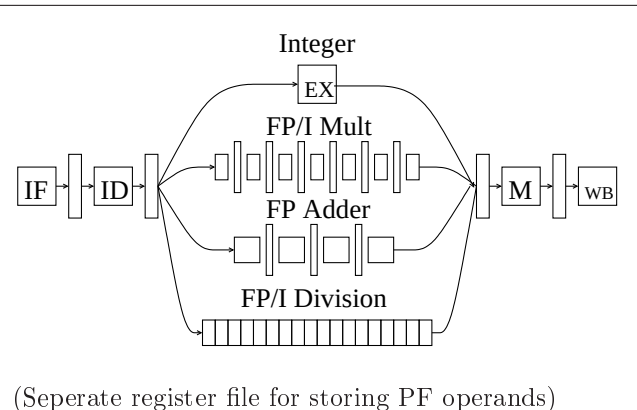
The times required for executing different instructions may differ significantly depending on the type (Integer / Floating Point...).

Making the cycle time long enough so that the slowest instruction can be executed in one cycle would slow down the processor dramatically.

Solution:

Distribute the EX stage of complex operations over several clock cycles and use several functional units in the EX stage. That allows to execute several instructions in parallel.

Functional units which are not pipelined (division) and which require more than one clock cycle for execution may create **structural hazards**! Also *out-of-order* (mult, add) completion may lead to **write after write hazard**



Avoiding Structural Hazards:

Use a shift register to memorize the clock cycle of the writeback phase.

If an instruction detects that a structural hazard will occur, the instruction is stalled for one cycle in the ID stage.

Avoiding Write After Write Hazards:

Because several instructions are longer than others, they are completed after the others are completed (out-of-order).

Solution:

The shorter instruction has to be stalled while the longer one is completed.

Hazard detection logic detects all hazards in the ID stage and resolves them by stalling the corresponding instruction.

Hazard Detection in Decode Phase:

- **Check for structural hazards:**
Make sure that the register writeport is available when it will be needed. (else stall)
- **RAW hazards:**
Make sure that the source register is not a target register in one path of the extended execution phase.
- **WAW hazards:**
Make sure that no instruction has the same target as this instruction.

Dynamic Branch Prediction:

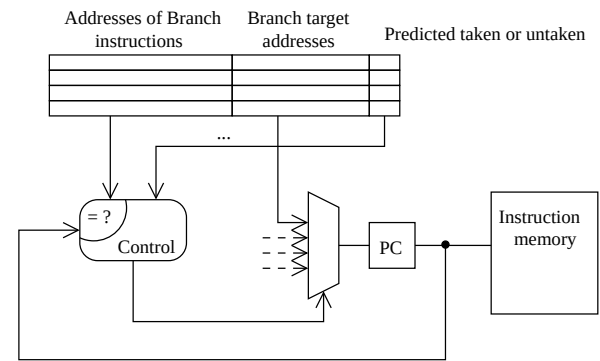
In processors with multiple functional units deep pipelines are used. This may lead to large branch delays if a branch is predicted the wrong way!

Idea: predict branches using the program's past behaviour.

Small memory addressed by the lower bits of the instruction address contains a flag indicating whether the branch has been taken or not. This flag is set or reset at each branch

Improvement: 2 bit prediction scheme

Branch Target Buffer:



6 Dynamic Scheduling:

Static Scheduling:

Execution is started in the order in which the instructions have been fetched (e.g. in the order which the compiler has determined). If a data dependence occurs that can not be resolved by forwarding the pipeline is stalled. No new instructions are fetched until the dependence is cleared

Dynamic Scheduling:

Hardware rearranges instruction executions dynamically to reduce stalls. Dynamic Scheduling takes structural hazards and data hazards into consideration.

To avoid an instruction is stalled because a data hazard delays all subsequent instructions, the ID stage is split into two stages: **issue** (decode, check for hazards), **read operands** (wait until no data hazards, then read operands). This can lead to o-o-o execution and o-o-o completion

Out of Order Execution:

O-O-O execution may lead to write after read hazards.

```
div $f0, $f2, $f4
add $f5, $f0, $f6   Stalled because $f0 is needed RAW
mul $f6, $f6, $f7   If started mul is finished before add, add will read the wrong value from $f8 WAR
```

6.1 Scoreboard:

Idea:

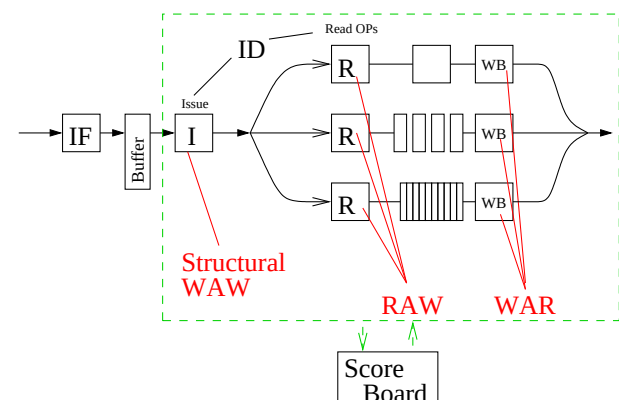
Maintain a execution rate of one instruction per clock cycle by executing an instruction as early as possible. If an instruction needs to be stalled because of a data hazard other instructions can be issued and executed.

- Detect structural and data hazards
- Determines when an instruction may read operands and when its executed
- Determines when an instruction can write its result into the destination register.

A Scoreboard consists of following tables:

- Instruction status
- Functional unit status
- Result register status

Scoreboard and Pipeline:



6.1.1 Scoreboarding Procedure:

1. Issue:

Issue the instruction if and only if:

- a functional unit for the instruction is free.
(**Resolve structural hazards**)
- no other active instruction has the same target register
(**Resolve WAW**)

Then the scoreboard issue the instruction to the functional unit and updates the internal data structure. Else the issue stage stalls. Subsequent instructions are written into a buffer between instruction fetch and issue. If the buffer is filled then the instruction fetch stage stalls.

2. Read Operands:

When all operands are available the scoreboard tells the functional unit to read its operands and to begin execution (may lead to o-o-o execution). A source operand is available when no active instruction issued earlier is going to write it. → stall. (**Resolve RAW**).

3. Execution:

The functional unit executes the instruction (may take several clock cycles) When the result is ready the functional unit notifies the scoreboard.

4. Writeback

After execution the scoreboard checks for **WAR hazards**. If no such hazard occurs the result can be written to the destination register.

6.2 Tomasulo Scheme:

There are no ways to avoid **RAW** hazards. We have to wait until all operands are calculated! (if there is no forwarding) But we can avoid stalls by **WAR** and **WAW** hazards.

Idea:

Rename destination registers of instructions in a way that prevents instructions being executed o-o-o from overwriting operands being still required by other instructions.

Use temporary registers for the remaining.

Example:

```
div $f0, $f2, $f4
add $f6, $f0, $f8    RAW for $f0
sub $f8, $f10, $f14  WAR for $f8
mul $f6, $f10, $f8    WAW for $f6, RAW for $f8
```

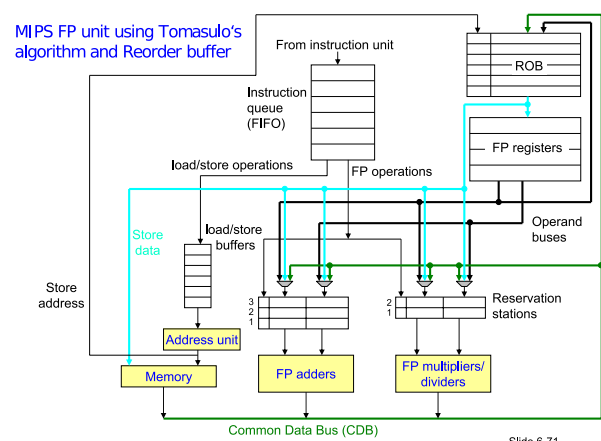
Rename Registers:

```
div $f0, $f2, $f4
add $S, $f0, $f8
sub $T, $f10, $f14
mul $f6, $f10, $T
```

Reservation Station:

- Buffer the operands for instructions waiting for execution.
If an operand is not yet calculated the corresponding reservation station contains the number of functional unit which will deliver the result.
- Renaming of register number to number of reservation station (during issue phase)
- Information about the availability of the operands stored in a reservation determines when the corresponding instruction can be executed.
- As results become available they are sent directly from the reservation station to the waiting FU over the common data bus
- When successive writes to a register overlap in execution only the result of the instruction being issued at last is used to update the register (**resolves WAW/WAR**)

Tomasulo Algorithm:



1. Issue:

Get next instruction from the head of the instruction queue and issue it to a matching reservation station.

Operands are available in registers:

yes: hand over values to reservation station

no: hand over names of those reservation stations that are calculating the values

Buffering operands **resolves WAR hazards**. If no matching reservation station is empty there is a **structural hazard**. → stall.

2 Execution:

1. If one more of the operands are not available, monitor the CDB
2. If an operand becomes available place it in the waiting reservation station(s)
3. Wait until all operands for an instruction are available then start execution

Resolves **RAW hazards**

3. Write result: When the result is available send it to the CDB

From CDB it is sent:

1. To waiting reservation station
2. Only if the instruction is the one being issued last write to the target register

Resolves **WAW hazards**

DS-Branching Simple Solution:

Instructions following a branch are issued only. Execution starts not earlier than the branch prediction has turned out to be correct

→ Can reduce efficiency of the pipeline dramatically!

DS-Branching - Speculative execution:

Write Result Stage is split in two stages:

1. Write results:

- Instructions are executed as operands become available. Results are written to a reorder buffer
- For each active instruction there is one entry in the ROB. Their order corresponds to the order the instructions have been issued (The head of the ROB contains the result of the active instruction being issued first)
- Subsequent instructions can read their operands from this buffer.

Writes going to register file and memory are delayed until branch predictions turn out to be correct.

2. Commit:

When an instruction that writes to memory or register file reaches the head of the ROB its result is written. If the head of the ROB contains an incorrectly predicted branch the ROB is flushed. Results calculated by instructions following the branch are discarded!

7 Multiple Issue Processors

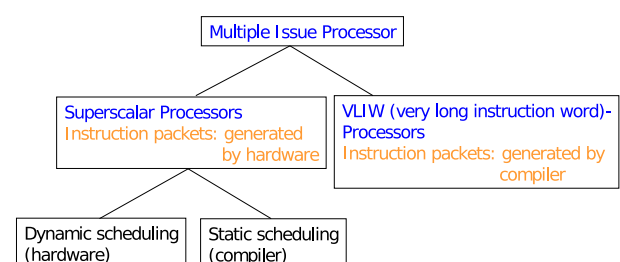
Multiple Issue:

Using multiple FUs, dynamic scheduling, branch prediction and speculation allow to achieve an CPI of nearly one. $CPI < 1$ is not possible because we issue only one instruction per clock cycle

→ Issue multiple instructions in one clock cycle (up to 8 in practice)

The sets of instructions being issued in parallel are called instruction packets or issue packets.

Overview:



7.1 Superscalar (Instruction Packets by Hardware)

Dual-Issue: (static)

In one clock cycle we can issue:

- One integer instruction
- One arithmetic floating point instruction

Only slight extensions of the hardware necessary compared to a single issued implementation.

4-way issuing processor: (dynamic)

In order to enable multiple issues per clock cycle we must be able to fetch multiple instructions per cycle also. We have to fetch the instructions stored at PC, PC+4, PC+8, PC+12 from memory

→ Wide bus to instruction memory required.

Problem:

What if one of these instructions is a branch?

Looking into the branch target buffer would increase cycle time → Only single issue for branches

Statically Scheduled:

- Must detect dependences in instruction packets and resolve them by inserting stalls
- Needs assistance of the compiler for achieving a high amount of parallelism
- Simple Hardware

Dynamically Scheduled:

- Requires less assistance of the compiler
- Hardware is much more complex

Compiler Techniques:

Loops may be unrolled in advance by the compiler.

7.2 VLIW (Instruction Packets by Compiler)

Very Long Instruction Word:

For highly superscalar processors the hardware becomes very complex. With this approach we let the compiler do as much work as possible in advance. The Compiler groups instructions with no dependences into a VILW. This VILW will be executed in parallel.

→ No hardware for hazard detection and scheduling is needed

Layers of Scheduling:

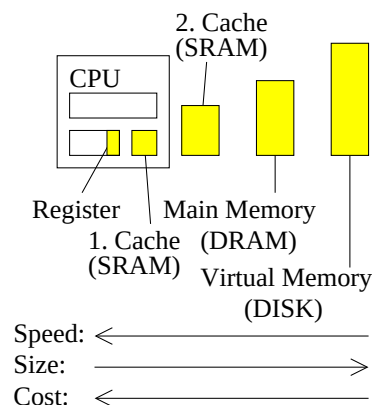
- *local* : scheduling inside list of instructions without branches (basic block)
- *global* : scheduling over several basic blocks

8 Memory

Memory Consistence:

- Memory Cell:
Stores one memory word consisting of k bytes (word size)
- Capacity:
Number of cells $\times$ word size
- Control:
Input for addresses
I/O for data being read/write
R/W signal: read or write?
CS (chip select): activate module?

Memory Hierarchy:



8.1 Principle of Locality:

Temporal Locality: (locality in time)

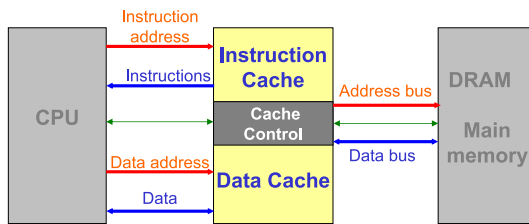
If an item is referenced then it will tend to be referenced soon again. (e.g. loops, or interim result...)

Spatial Locality: (locality in space)

If an item is referenced, items whose addresses are close by will tend to be referenced soon. (PC most of the time is just incremented, iterating over arrays...) → copy those into a small but very fast memory.

8.2 Caches:

Cache:

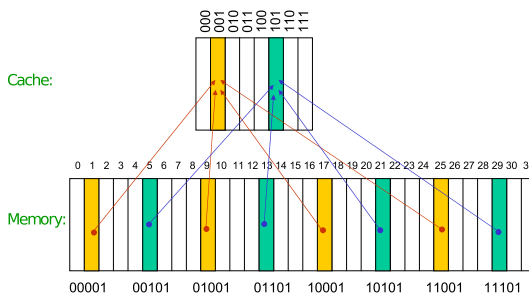


Use separate caches for instructions and data. This allows to read instructions and data item in parallel and the structure of the cache is more simple.

Terminology:

- Upper level = Cache
- Lower level = Memory
- Hit = data is in upper level
- Miss = data is not in upper level
- Hit ratio = $\frac{\text{hit}}{\text{access}}$
- Miss ratio = $\frac{\text{miss}}{\text{access}}$
- Hit time = time necessary for accessing the upper level
- Miss penalty = additional time being required for accessing the lower level (when miss).

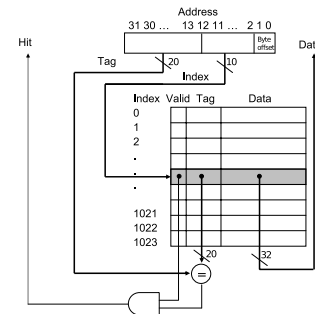
Direct Mapped: (one-way associative)



For each memory cell there exists exactly one location in the cache where it can be stored. Index = (memory address of block) modulo (number of blocks in the cache) But you can't have e.g. 00001 and 01001 in the cache at the same time! (**collision misses**)

Also there is only one comparator needed

Terminology:



Tag: For each block, the part of the upper portion of the address being the same for all memory cells in this block is stored

Valid Bit: indicates if content of block is valid

Reading from the Cache:

Read Hit:

an instruction is executed as usual

Read Miss:

Cache control unit notifies CPU that the item it just has read has been invalid → CPU has to be stalled.

Example for instruction cache:

1. Send original PC (current -4) to memory
2. Instruct memory to perform a read and wait...
3. Store data in cache, store upper portion of address in tag and set valid bit to 1
4. Restart fetching the instruction

Writing to the Cache (1):

Write hit:

Write through method:

The information is written to both: the block in the cache and to the block in the next level memory (easy to implement, no speedup for writes...)

Write back method:

The information is written only to the block in the cache. The block is marked as modified (=dirty) using an additional dirty bit for each block. The data is written back to memory later when the block is replaced. (reduces the working load on system bus, in cases where multiple components are allowed to write there can be a coherency problem: DMA, Multicore..)

Writing to the Cache (2):

Write miss:

Write allocate method:

The corresponding block is allocated first. Then the information is written in the case of a write hit (used in combination with *writeback*)

Write no-allocate method:

The information is written only to the memory. The cache is not affected (used in combination with *write through*)

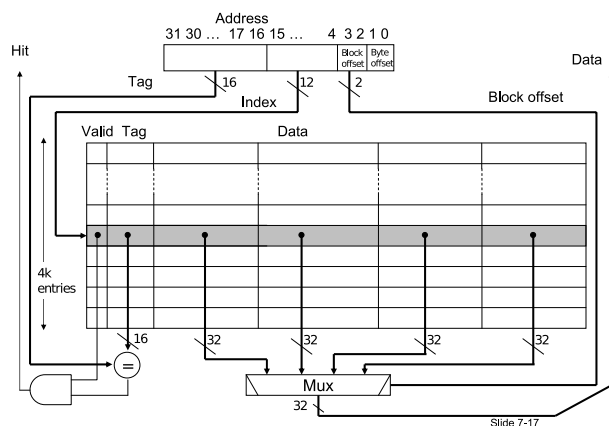
Taking advantage of spatial locality:

In case of a cache miss load multiple words from memory (and not only the one being referenced)

→ Multiple words for each cache block
(e.g. 4 32-bit words)

To take advantage of grouping multiple words into one block the memory must support multi-word memory access in order to increase the memory bandwidth

Cache Using Four-Word Blocks:



One Word Wide Memory Organisation: (With Four-Word Cache)



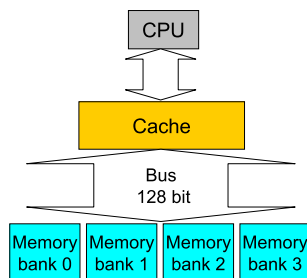
Cache sends address a to memory. The memory sends back the words stored at $a, a+1, a+2, a+3$ one after the other

Miss penalty: 65 clk

Band width: 0.25

1 word wide organisation

Four Word Wide Memory Organisation: (With Four-Word Cache)



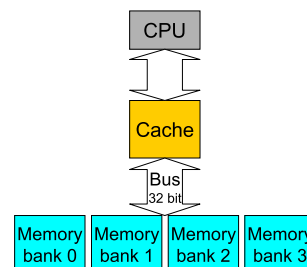
Wide memory organisation

Cache sends address a to memory. The memory sends back the words stored at $a, a+1, a+2, a+3$ in parallel. (But high cost for wide busses)

Miss penalty: 17 clk

Band width: 0.94

Interleaved Memory Organisation:



Interleaved memory organisation

Cache sends address a to memory. The memory reads the words stored at $a, a+1, a+2, a+3$ in parallel but sends them to the cache sequentially

Miss penalty: 20 clk

Band width: 0.8

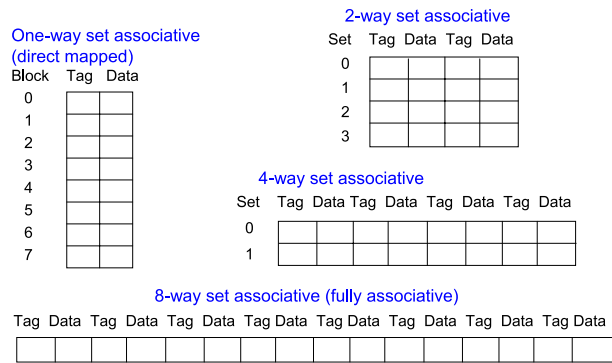
Fully Associative Cache:

Each Block can be placed anywhere in the cache. When accessing the cache the tags of all blocks are compared in parallel against the tag of the address. So there no collision misses but we need one comparator for each cache block and this is very expensive. (Only for small caches)

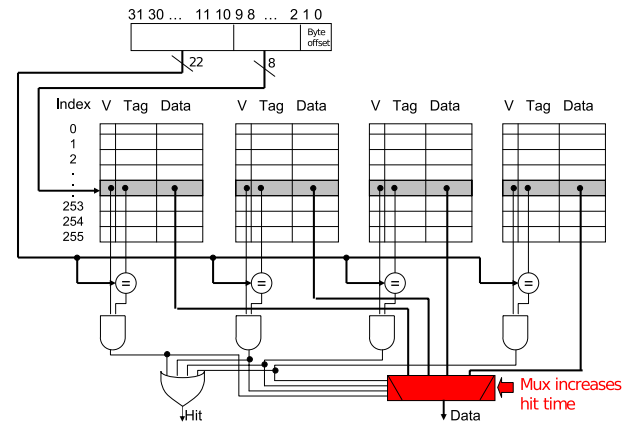
Set Associative Cache:

For each memory block there exists one set of cache blocks where the memory block can be placed. For a n -way set associative cache each set consists of n cache blocks. The index field denotes the number of the sets where a block can be placed: $\text{index} = (\text{block address}) \bmod (\text{number of sets})$. There are n comparators necessary.

Block Placement Strategies:



Example for 4-Way Associative Cache



Replacement Strategies:

1. Random:

Randomly select the block that will be replaced.

2. Least Recently Used (LRU):

Replace the block of a set that has been unused for the longest time. Reduces number of conflict misses. Easy to implement but expensive for full associative caches.

Main Memory:

Modern computer have a very large address range. Using n address bits we can address 2^n memory cells. But we cannot afford to provide computers with memories being as large as the address range! (e.g. 64 Bit = $1,7 \cdot 10^{12}$ €)

9 Multi Tasking Systems

Typical Workstation Scenario:

Many Programms run at once.

- A Programm cann ot use the whole memory for itself
- Programms have to be protected against each other
- **Problem:** Compiler/Linker do not know how much physical memory is available and ehre in memory a programm will be stored

Easy Solutions:

- Loader replaces absolute addresses when loading a programm
- User programms may only use addresses relative to PC or dedicated address base registers
- **Drawbacks:** Fragmentation, Low utilization

Virtual Memory:

- Data that cannot be stored in main memory because of capacity reasons are moved to the hard disk.
→ Capacity of the whole virtual memory may be much larger than main memory
- From the user pont of view all data is stored in the main memory
- Moving the portions of data is done by the operation system, but assistacne by the hardware is required

Virtual Memory:

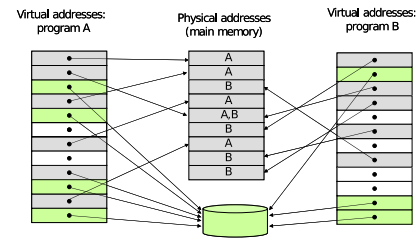
Each program is allowed to use the whole address range (independently on how much memory is actually present and which other programs are running at the same time)!

- Process: unique exection of a programm
- Addresses used by a process are translated into real hardware addresses
- Hardware addresses may point to main memory or to disk
- The address spaces are managed by the operating system
- Translation is done by operationg system and hardware

Paging: Basic Concept:

- **Virtual Memory** (as its seen by a programm) is divided into **pages** of fixed size (e.g. 4kB)
- **Physical Memory** the parts of the main memory where pages can be stored are divided into **page frames**
- **Virtual Addresses** are used by the CPU which are translated by the OS and hardware to **physical addresses**
- The physical address may be assigned to several virtual addresses do allow several programs to share data or part of the program (libs)

Virtual and Physical Addresses:

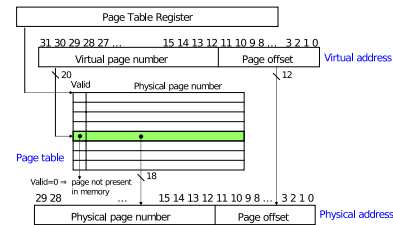


A virtual address consists of a Virtual page number and a page offset (select one memory cell within a page). The Physical Pagenumber points either to a page frame in memory or to a page on the disk.

Benefits of Virtual Addresses:

- Virtual memory simplifies loading of a program for execution:
- A program may be stored anywhere in the physical memory
- The operating system does not have to allocate a contiguous block of memory for a program
- Relocation of a program is possible

Virtual and Physical Addresses:



Page faults:

If the Processor wants to access data being not in main memory the corresponding page has to be loaded from disk. But this is very expensive and may take millions of clock cycles → Avoiding access to disk whenever possible!

If the valid bit of a page being accessed is 0 a page fault occurs. The CPU causes an exception and the operating system performs reloading the page

Page Table: For translating virtual page numbers into physical page numbers a page table is used:

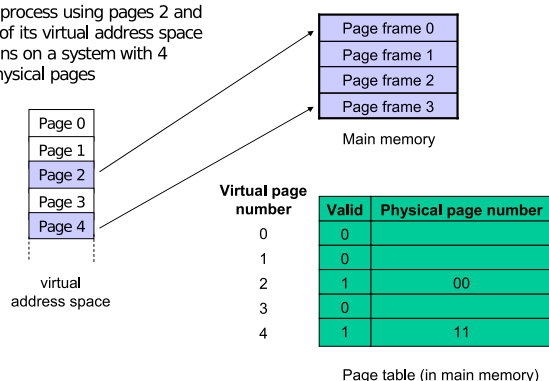
- Contains for each virtual page number the corresponding physical page number and the valid bit
- Page table is stored in main memory
- Each process has a pagetable of its own (allows to provide each process a full address space of 2^{32} addresses of its own)
- Processor needs a page table register that contains the start address of the page table of the program being active

Only the operating system is allowed to modify page tables!

Example:

Example:

A process using pages 2 and 4 of its virtual address space runs on a system with 4 physical pages



Process Switch - Context Switch:

Operating system allows to share CPU and memory with several processes by activating and deactivating them one after another.

A process is activated by loading its state into the processor:

- PC
- Register contents
- Page table register

Starting a Program:

The operating system allocates physical pages in memory (and disk if necessary) and builds an initial page table

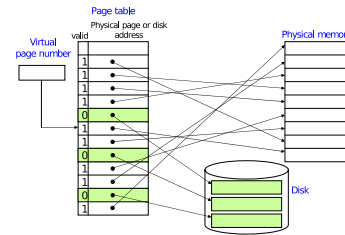
Allocating Memory:

If a process wants to allocate additional memory for storing data during runtime it calls an operating system routine which allocates additional pages and updates the page table. (same with stack...)

Remove Pages:

If the operating system has to remove pages of a non-active process from main memory it has to update its pagetable. So the operating system needs information about which processes use which physical pages! (additional table...)

Where to Store addresses of pages on disk?:



Single-Entry Page Table:

During loading a page from disk the corresponding page table entry is overwritten. The old disk address gets lost. When the page will be removed from memory later it must be written back to disk even if it has not been modified.

Extend Page Table Entries:

To prevent unmodified pages from being written back to disk we have to store both (mem & disk addr)

Replacement Strategies:

Least Recently Used:

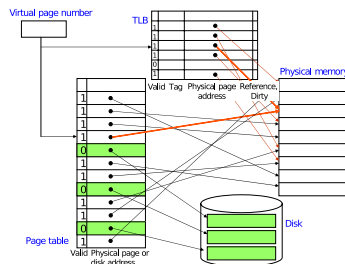
Replace the page that has been unused for the longest time.

Completely accurate: see cache... counter... much too expensive!

Simple: Use one Reference bit for each page: is set by MMU when the page is accessed. The OS clears periodically the bits. Page with non set bit will be replaced first.

Translation Lookaside Buffer:

Each access of a program to its virtual memory requires two accesses to the physical memory (table, data). Use an additional cache to store the physical page addresses of frequently used virtual addresses



Translation with TLB:

Virtual address: lookup physical page number in TLB

TLB hit:
Set reference bit
Set dirty bit if writing

TLB miss: lookup physical page number in page table

Page fault:
CPU causes exception
Operating system reloads page