Instruction Set, Pipelining, and Instruction Level Parallelism (ILP)

- We have dealt with memory and cache issues
- For cache performance we looked at
  - Loop interchange, loop fusion, and blocking
- Now we will get in the CPU and see how registers work with instruction sets
- We will
  - Explain pipeline architectures
  - Study Instruction Level Parallelism (ILP)
  - How knowledge of architecture, instruction and ILP can be used
- Sometimes increase in ILP and decrease in cache miss can <u>contradict</u> each other

- Current architectures are called load-store or register-register architecture implying on can only access memory (via cache) only with load and store instructions (also called general-purpose register – GPR – architectures)
- Reasons for this :

1. Registers like other form of storage internal to the CPU – are faster than memory
2. Registers can be used easily by a compiler than other forms of internal storage
3. Registers can be used to hold variables. When variables are allocated to registers the memory traffic is reduced and hence program speeds up

- The code sequence C = A + B looks like following in assembly language :

```
load    R1, A
load    R2, B
add     R3, R1, R2
store   C, R3
```

- Load instruction loads A into register R1 and B into register R2, add instruction adds content of register R2 and R1 into register R3, and store instruction stores content of register R3 into C
- Experience (for e.g. running a collection of 12 integer programs on Intel 80x86) shows that 10 simple instructions can account for 96% of total instructions executed
- Implementors would make those fast since those are common cases

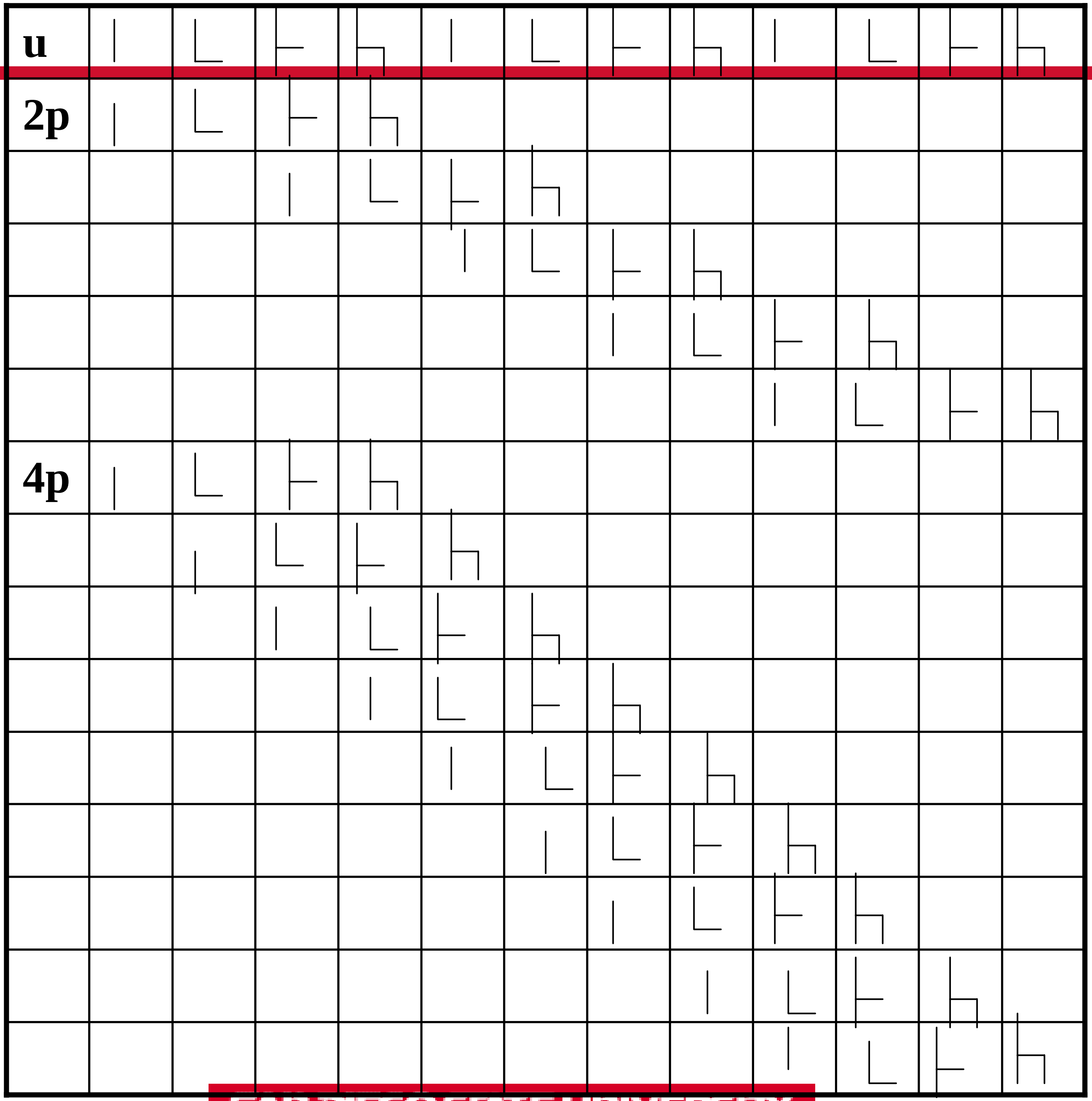
Types and Numbers of Registers

- Types and numbers of registers vary from machine to machine
- Registers on a typical hypothetical machine:
  - Thirty two 32-bit integer registers named R0, R1, ….., R31
  - Additionally there is a set of floating point registers (FPRs) which can be used as 32 single precision (32 bit) registers or as even-odd pairs holding double precision values. This 64 bit FPRs are named F0, F2, F4, …… , F30
  - Some machines have 64 bit registers
- Registers on the SUN HPC10000
- V9 architecture (64 bit architecture)
  - (at least) 64 64-bit integer register (g,o,l,i registers)
  - 32 32-bit (f0,f1,…,f31)
    32 64-bit (f0, f2, …, f62)
    16 128-bit (f0, f4,…,f60)
  - Other kinds of registers that is used internally (i.e. not directly for user code)

Pipelining

- Pipelining is an implementation technique whereby multiple instructions are overlapped in execution. This implementation technique exploits parallelism among the instructions in a sequential instruction stream. All current CPUs uses pipelining
- Pipelining is equivalent to automobile assembly line. In a computer pipeline each step in the pipeline completes a part of an instruction. Like the assembly line different steps are completing different parts of different instructions in parallel
- Each of the steps are called a pipe stage. The stages are connected one to another to form a pipe. Instructions enter at one end, progress through stages, and exit at the other end.

Chair building process :unpipeline and pipeline



- Goal is to keep the time taken by each pipe line stage balanced i.e. more or less same as the time taken by any other pipeline stage
- If the stages are perfectly balanced then the time per instruction on the pipeline machine, in ideal case, is equal to :

Time per instruction on an unpipelined machine
--
Number of pipe stages

- There is initial latency for the the first instruction but for large number of instructions this is negligible

- Speedup from pipelining equals the **number of pipe stages**
- Problems :
  - Stages will not be perfectly balanced
  - Pipelining does involve some overhead
  - Time per instruction on the pipeline machine will not have its possible minimum value, but can be close
- We will look at pipelining as yielding a reduction in the average execution time per instruction or equivalently it can be looked upon as decreasing the average clock cycle per instruction (CPI)

Simple Implementation of Instructions

- First look at how instruction is implemented without pipilening
- In this implementation (different computers will have different implementations) every instruction takes 5 clock cycles
- We will extend this to pipeline version resulting in lower CPI
- The five clock cycles are (for our example computer architecture):
  1. Instruction fetch (IF) cycle : Send out program counter and fetch the instruction from memory into instruction register
  2. Instruction decode (ID) cycle : Decode the instruction and access the register file to read registers
  3. Execution cycle (EX) : The Arithmetic or Logic Unit (ALU) operates on the operands prepared in previous cycle
  4. Memory access/branch completion (ME) : load, stores, and branches
  5. Write back (WB) cycle :Write the results to the register file whether it comes from memory operation or from ALU

The Basic Pipeline

clock cycle number

Instr #	1	2	3	4	5	6	7	8	9
I	IF	ID	EX	ME	WB				
I+1		IF	ID	EX	ME	WB			
I+2			IF	ID	EX	ME	WB		
I+3				IF	ID	EX	ME	WB	
I+4					IF	ID	EX	ME	WB

Observations

- Each instruction takes 5 clock cycles to complete. During each clock cycle hardware will initiate a new instruction
- Latency is 5 clock cycles, after which one instruction gets executed every clock cycle. Registers read in ID and data written to register in WB. Separate instruction and data cache will allow this.
- Pipeline increases the CPU instruction throughput i.e. the number of instructions completed per unit time but it does not reduce the execution time of an individual instruction.
- In reality it increases slightly the execution time of each instruction due to overhead required for control of pipeline
- Some of the operations (like EX) can take more than one clock cycle which causes imbalance among pipeline stages

Example: Consider a unpipelined machine. It has 10 ns clock cycles, it uses 4 clock cycles for ALU ops, and branches, and 5 cycles for memory ops. Assume relative frequency of these are 40%, 20%, and 40% respectively. Assume due to pipeline overheads the pipelined machine requires 1 ns overhead to the clock. Ignoring latency how much speedup in the instruction execution rate will we gain from pipeline.

Answer : Average instruction execution time on the unpipelined machine = clock cycle * average CPI

= 10ns * (4 * 40% + 4 * 20% + 5 * 40%)

= 10ns * (160 + 80 + 200) / 100

= 44 ns

On the pipeline machine we get one instruction per clock cycle ignoring latency. Hence average instruction execution time on the pipelined machine is = 10 + 1 = 11 ns

Speedup = 44/11 = 4 times

Major Hurdles of Pipelining

- There are situations, called **hazards**, that prevent the next instruction in the instruction stream from executing during its designated clock cycle. Hazards reduce the performance from the ideal speedup gained by pipelining.
- Kinds of hazards :
  1. Structural Hazards : This arise from resource conflicts when the hardware cannot support all the possible combinations of instructions in simultaneous overlapped execution.
  2. Data Hazards : Arise when an instruction depends on the results of a previous instruction.
  3. Control Hazards : Arise from the pipelining of branches and other instruction that change the PC. (we are not going to talk about this much more)
- Hazards in pipeline causes it to stall like CPU stalls in case of cache miss.

Performance of Pipelines with Stalls

- Stall causes the pipeline performance to degrade from the ideal performance
- Speedup from pipelining =

average instruction time unpipelined
-- =
average instruction time pipelined

CPI unpipelined * clock cycle unpipelined
-- =
CPI piplined * clock cycle pipelined

CPI unpipelined clock cycle unpipelined
------------------- * ----------------------------
CPI pipelined clock cycle pipelined

Pipelining can be thought of as decreasing the CPI. The ideal CPI on a pipelined machine is almost always 1.

Hence,

CPI pipelined =
Ideal CPI + Pipeline stall clock cycles per instruction =
1 + Pipeline stall clock cycles per instruction

If we ignore the cycle time overhead of pipelining and assume the stages are perfectly balanced then the cycle time of the two machines can be equal, leading to

$$\textbf{Speedup} \quad \frac{\textbf{CPI unpipelined}}{\textbf{1 + Pipeline stall clock cycles per instruction}}$$

One important simple case is where all the instructions take the same number of cycles, which must equal the number of pipeline stages (also called depth of the pipeline). Here the unpipelined CPI is equal to the depth of the pipeline, leading to

$$\textbf{Speedup} = \frac{\textbf{Pipeline depth}}{\textbf{1 + Pipeline stall clock cycles per instruction}}$$

If there are no pipeline stalls then the speedup is equal to the intuitive result i.e. the pipeline depth as expected in the ideal case.

Structural Hazards

- For a pipelined machine overlapped execution of instructions occurs
- Requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline
- If some combination of instructions cannot be accommodated because of resource conflicts, the machine is said to have structural hazard
- Cases :
  - When some functional unit is not fully pipelined then a sequence of instructions using that unpipelined unit cannot proceed at the rate of one per clock cycle
  - When some resource has not been duplicated enough (e.g. only one register-file write port but pipeline may want to perform two writes in a clock cycle) to allow all combinations of instructions in the pipeline to execute
- A stall is commonly called a **pipeline bubble** or **bubble** since no useful work gets done in that cycle

Structural Hazard in the Basic Pipeline

- A machine with only one memory port will generate a conflict whenever a memory reference occurs e.g. load and IF of I3 conflict

clock cycle number

Instr #	1	2	3	4	5	6	7	8	9
load	IF	ID	EX	**ME**	WB				
I1		IF	ID	EX	ME	WB			
I2			IF	ID	EX	ME	WB		
I3				**IF**	ID	EX	ME	WB	
I4					IF	ID	EX	ME	WB

Bubble Caused by Structural Hazard

- No instruction completes in clock cycle 8 (Instruction 1 is assumed not to be a load or store so that there is no conflict with I3)

clock cycle number

Instr #	1	2	3	4	5	6	7	8	9
load	IF	ID	EX	**ME**	WB				
I1		IF	ID	EX	ME	WB			
I2			IF	ID	EX	ME	WB		
stall				☁	☁	☁	☁	☁	
I3					**IF**	ID	EX	ME	WB
I4						IF	ID	EX	ME

Data Hazards

- Pipelining changes relative timing of instructions by overlapping their execution
- Data hazards occur when pipeline changes the order of read/write accesses to operands from what seen by sequentially executing instructions on an unpipelined machine
- Consider following instructions on our machine with 5 clock cycle instructions and 5 stage pipeline :

```
ADD     R1, R2, R3
SUB     R4, R1, R5
AND     R6, R1, R7
OR      R8, R1, R9
XOR     R10, R1, R11
```

- All instructions after the ADD use the result of the ADD instruction clock cycle number

Instr#	1	2	3	4	5	6	7	8	9
Add r1,r2,r3	IF	ID	EX	ME	WB				
Sub r4,r1,r5		IF	ID	EX	ME	WB			
AND r6,r1,r7			IF	ID	EX	ME	WB		
OR r8, r1,r9				IF	ID	EX	ME	WB	
XOR r10,r1,r11					IF	ID	EX	ME	WB

- The ADD instr writes value of R1 in 5th clock cycle but SUB reads it in 3rd clock cycle. This is called data hazard clock cycle number

Instr#	1	2	3	4	5	6	7	8	9
Add r1,r2,r3	IF	ID	EX	ME	WB				
Sub r4,r1,r5		IF	ID	EX	ME	WB			
AND r6,r1,r7			IF	ID	EX	ME	WB		
OR r8, r1,r9				IF	ID	EX	ME	WB	
XOR r10,r1,r11					IF	ID	EX	ME	WB

- Data hazard is minimized with a hardware technique called **forwarding**
- The result is moved from where ADD produces it, EX/ME register, to where SUB needs it in the
- Forwarding can be generalized to include passing a result directly to the functional unit that requires it

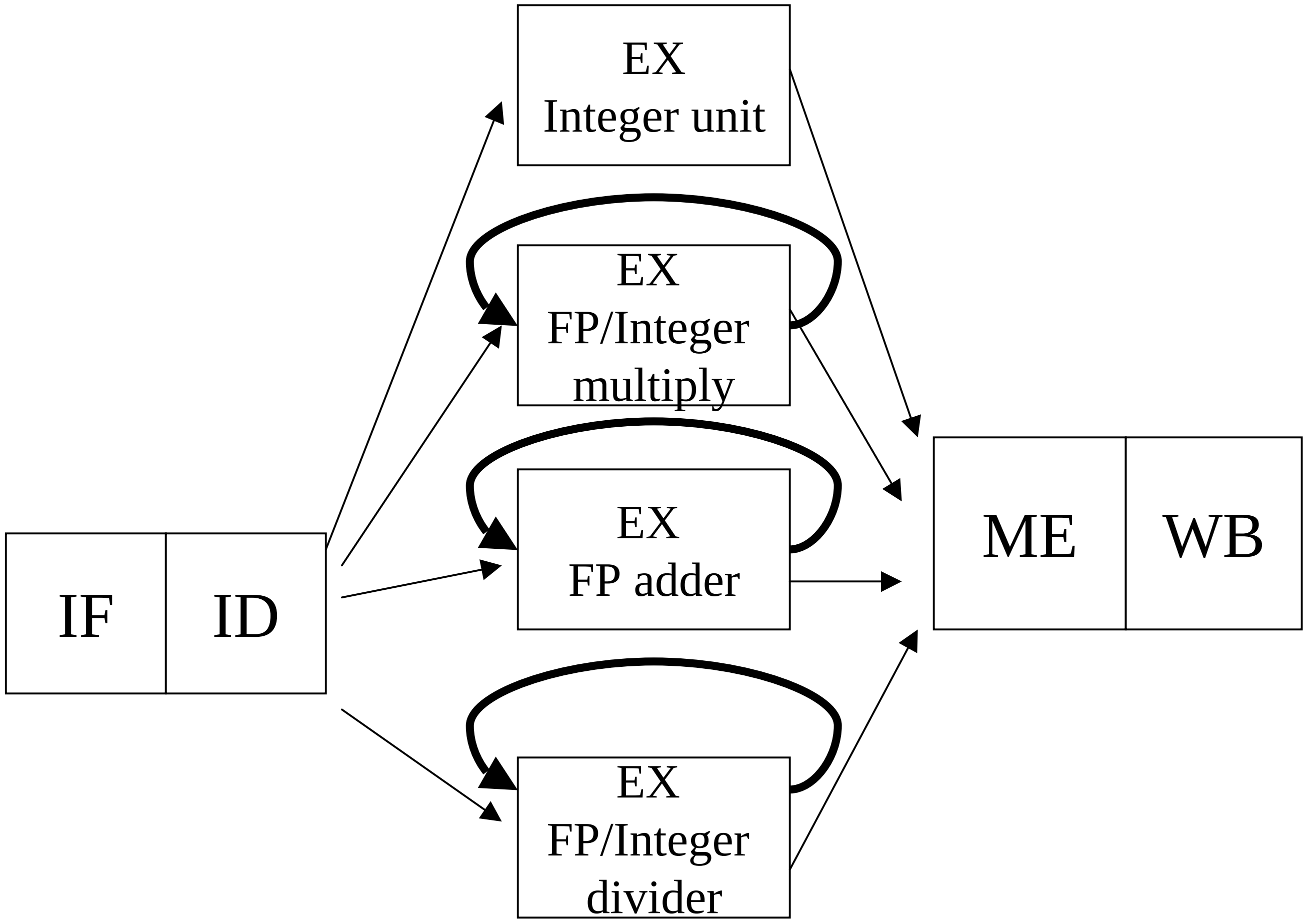
- Three kinds of data hazards depending on the order of read and write accesses in the instructions
  - RAW (read after write) : one instruction tries to read source before another writes to it (forwarding is used to avoid it)
  - WAW (write after write) : Instruction I+1 tries to write an operand before it is written by instruction I. The writes end up being performed in the wrong order, leaving the value written by I rather than the value written by I+1. (happens only in pipelines that write in more than one pipe stage)
  - WAR (write after read) : Instruction I+1 tries to write before instruction I reads, so I incorrectly gets the new value. (this cannot happen in our example pipeline since all reads (in ID) are early than writes (in WB))
- Not all data hazards can be avoided and like structural hazards require stalls

Lets now look at floating point operations

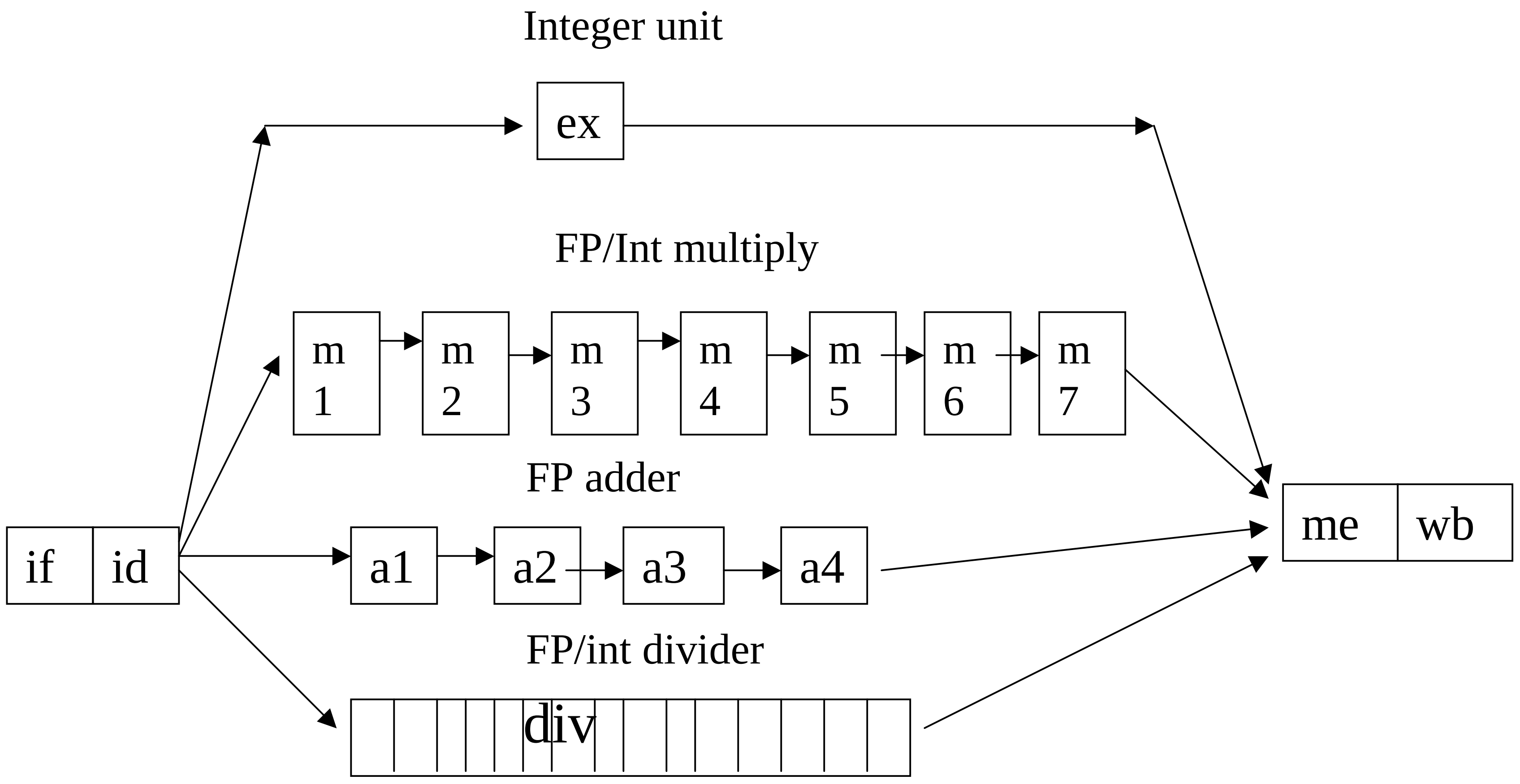
- Impractical to require all FP ops to complete in one or even two clock cycles since that would make slower clock or using enormous logic in FP units
- Instead the FP pipeline will allow for a longer latency for operations
- In FP pipelines
  - The EX cycle can be repeated many times and the number of repetitions vary for different FP operations
  - There may be multiple FP functional units

- Lets assume that there are four separate functional units
  1. Main integer unit handles loads, stores, integer ALU ops, and branches
  2. FP and integer multiplier
  3. FP adder that handles FP add, subtract, and conversion
  4. FP and integer divider
- The execution stages of these functional units are not pipelined

Resulting structure of previous machine with three additional unpipelined FP functional units



Lets build a pipelined FP functional units out of the last one


Integer unit
ex
FP/Int multiply
m 1
m 2
m 3
m 4
m 5
m 6
m 7
FP adder
a1
a2
a3
a4
FP/int divider
div
if
id
me
wb

- The pipelined machine supports multiple outstanding FP operations:
  - Up to four outstanding FP adds
  - Up to seven outstanding FP/integer multiplies
  - One divide since divide is not pipelined
- Complications :
  - Requires introduction of additional pipeline registers (e.g. a1, a2, a3, a4)
  - The id/ex register must be expanded to connect id to ex, div, m1, and a1

Pipeline timing of a set of independent FP operations

MULT	IF	ID	M1	M2	M3	M4	M5	M6	M7	ME	WB
ADD		IF	ID	A1	A2	A3	A4	ME	WB		
LD			IF	ID	EX	ME	WB				
SD				IF	ID	EX	ME	WB			

Stall occurs when the ADD depends on MULT

MULT	IF	ID	M1	M2	M3	M4	M5	M6	M7	ME	WB
ADD		IF	ID	A1 stall	A2 stall	A3 stall	A4 stall	ME stall	WB		
ADD		IF	ID	stall	stall	stall	stall	stall	A1	A2	A3