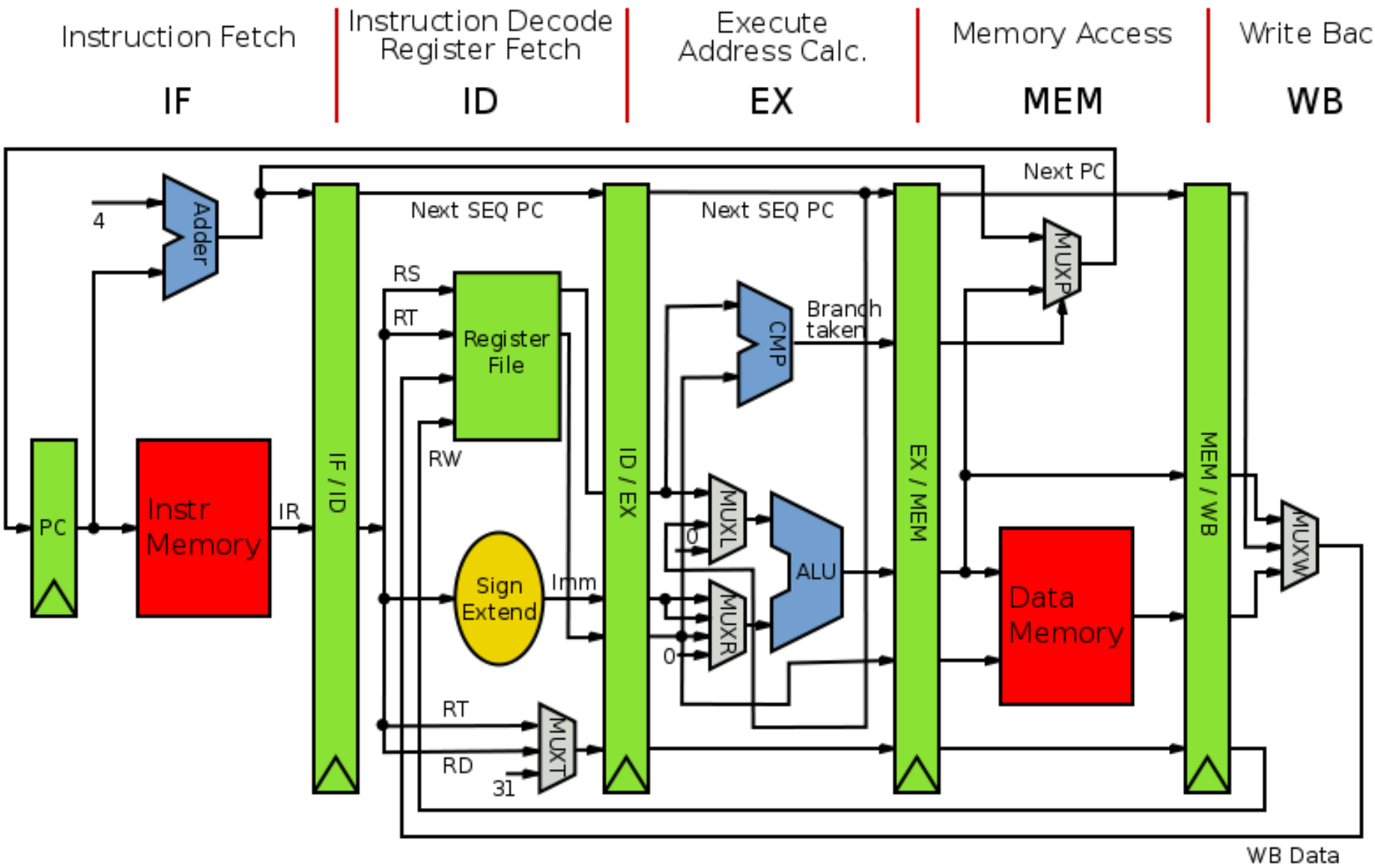
Computer organisation and architecture

Department of Computer Sc & Engg
IIT Kharagpur

November 24, 2010

Pipeline CPU



Pipeline hazards

Maximum benefit from pipelining may not be possible because of pipeline hazards

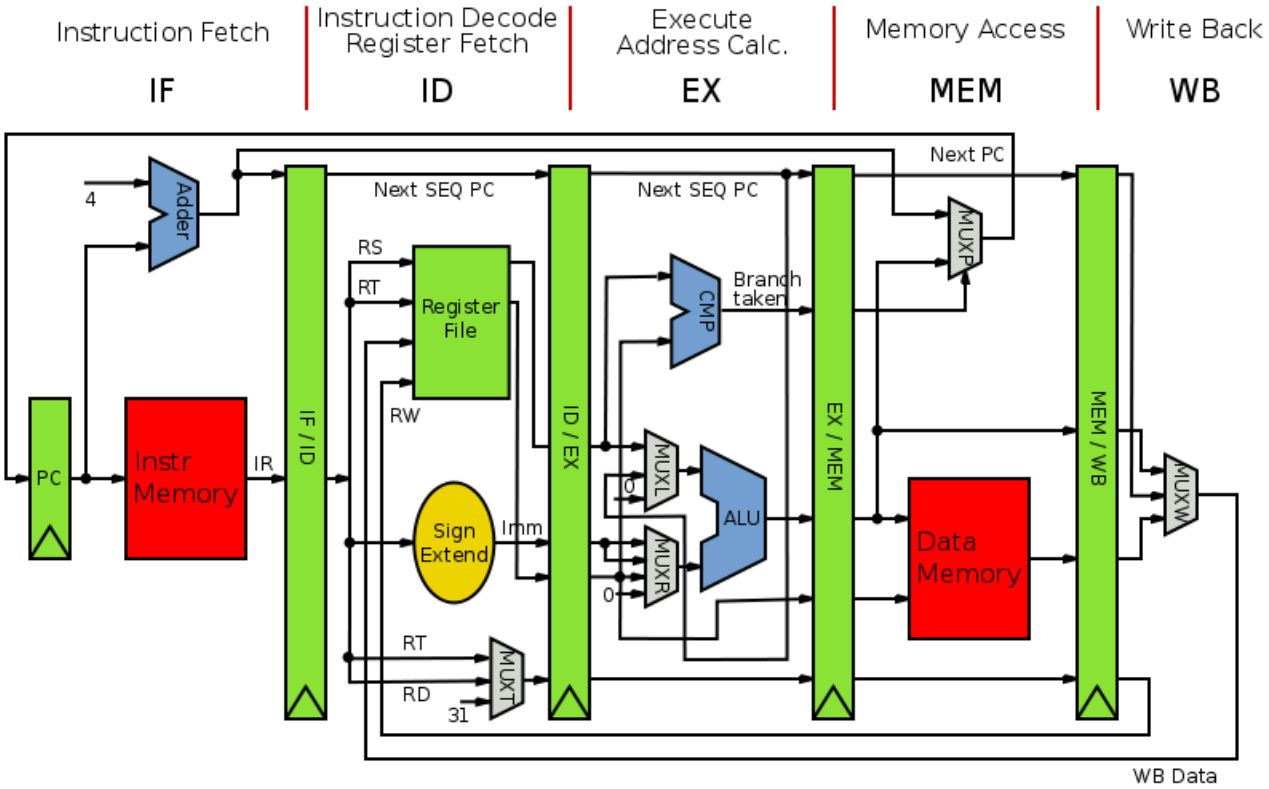
Structural hazards: H/W cannot support certain combinations of instructions which may result in contentious access to the same cpu h/w resource – must be avoided by matching the ISA to the pipeline h/w

Data hazards: There are three generic data hazards

- **Read After Write (RAW)** I_j following I_i depends on a value defined by I_i, but tries to read it before it is defined by I_i, violating the RAW requirement
  This problem can happen in the DLX pipeline (studied here) – avoided by stalling or by forwarding.
- **Write After Read (WAR)** I_j following I_i defines a value used by I_i, but tries to write it before it is read by I_i, violating the WAR requirement – not possible in the DLX pipeline
- **Write After Write (WAW)** I_j follows I_i and both define a value but $I - j$ tries to write it before it is written by I_i, violating the WAW requirement – not possible in the DLX pipeline

Control hazards: Pipelining of conditional, unconditional jumps, subroutine calls result in invalid (hazardous) instructions in the pipeline – may be avoided by flushing those instructions
Loss of performance due to flushing can be mitigated to a good extent by branch prediction

Instruction fetch



- Instruction fetch may have to be stalled on data hazards or flushed on branching
- Updation of PC and the inter-pipeline register of stalled stages, such as IFID and IDEX, is inhibited during stall
- Flushed instructions are "neutralised"

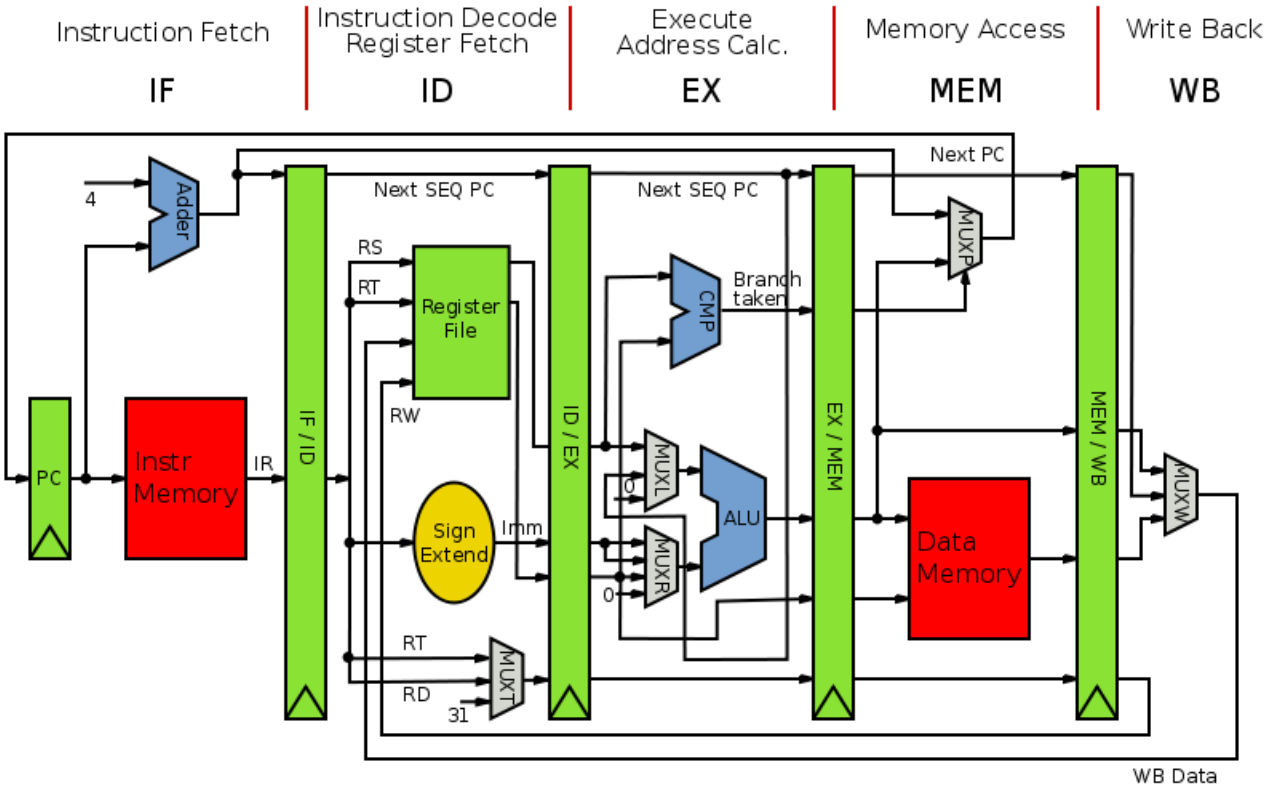
- [!stallF] {defined later}
- IFID[IR]←IM[PC] [IM.R←1]
- PC←NextPC [PC.W←1]
- IFID[nxPC]←PC+4
- IFID.W←1

  {Write to IFID and IDEX only in absence of stall;

  otherwise, inter-pipeline registers written at the end of each clock cycle}

- [brF] {defined later}
- IFID[NOP]←1

Instruction fetch



- Instruction fetch may have to be stalled on data hazards or flushed on branching
- Updation of PC and the inter-pipeline register of stalled stages, such as IFID and IDEX, is inhibited during stall
- Flushed instructions are "neutralised"

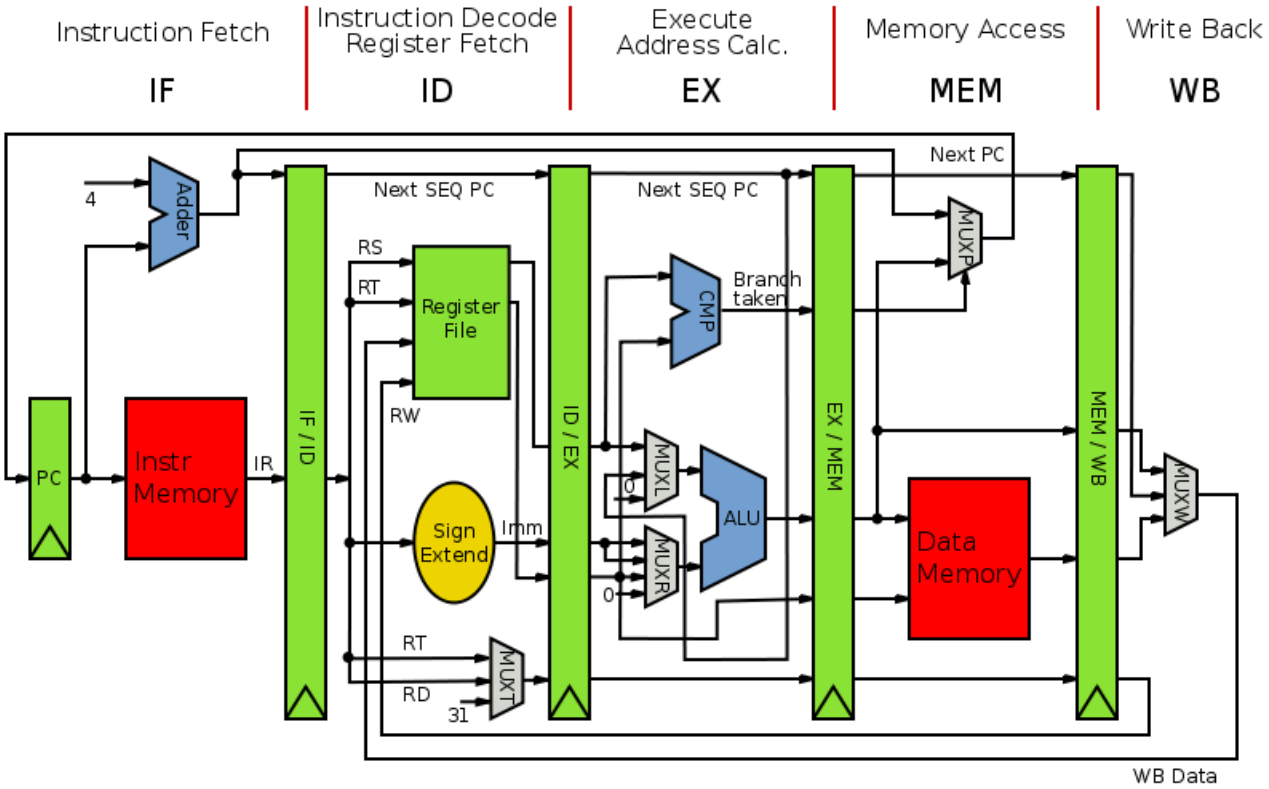
- [!stallF] {defined later}
- IFID[IR]←IM[PC] [IM.R←1]
- PC←NextPC [PC.W←1]
- IFID[nxPC]←PC+4
- IFID.W←1

  {Write to IFID and IDEX only in absence of stall;

  otherwise, inter-pipeline registers written at the end of each clock cycle}

- [brF] {defined later}
- IFID[NOP]←1

Instruction fetch



- Instruction fetch may have to be stalled on data hazards or flushed on branching
- Updation of PC and the inter-pipeline register of stalled stages, such as IFID and IDEX, is inhibited during stall
- Flushed instructions are "neutralised"

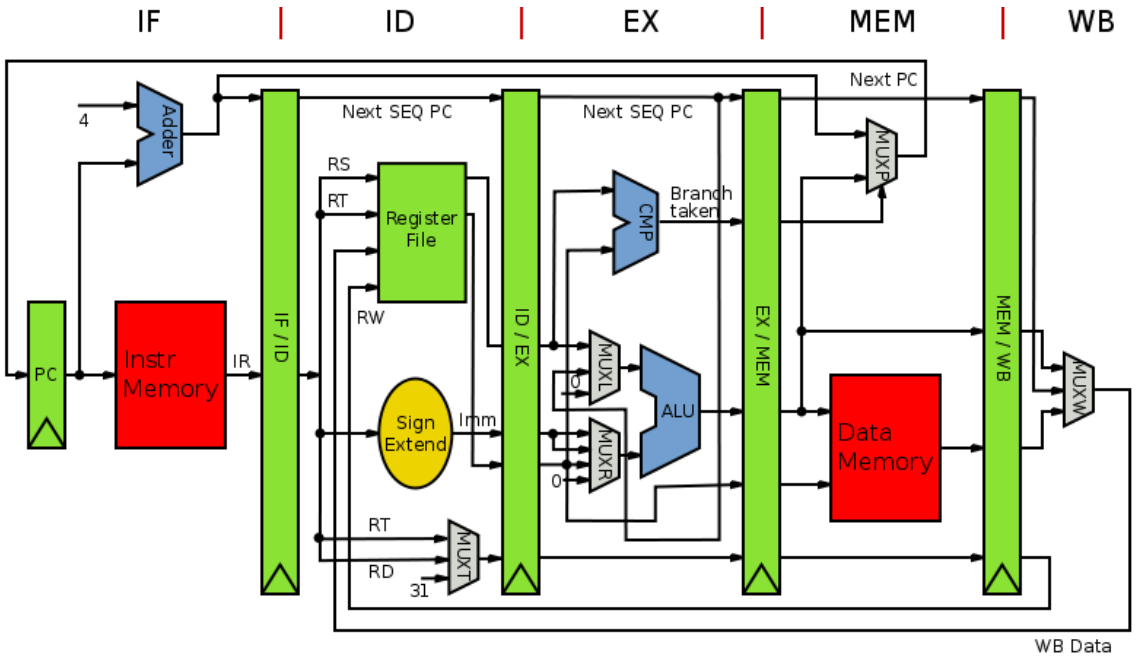
- [!stallF] {defined later}
- IFID[IR]←IM[PC] [IM.R←1]
- PC←NextPC [PC.W←1]
- IFID[nxPC]←PC+4
- IFID.W←1

  {Write to IFID and IDEX only in absence of stall;

  otherwise, inter-pipeline registers written at the end of each clock cycle}

- [brF] {defined later}
- IFID[NOP]←1

Flushing the pipeline on branching



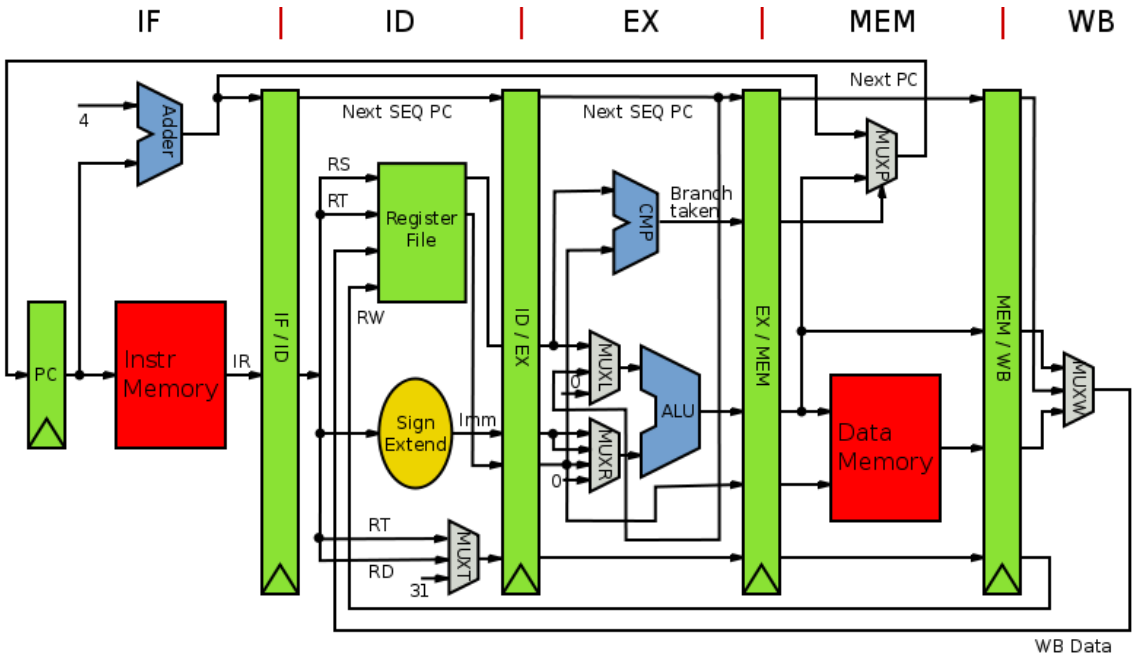
- CMP.out=1 indicates future branching due to I_j now in EX at t (computed early in clock cycle of EX)
- PC←PC+4 at $t+1$; PC←Branch target at $t+2$
- I_{j+1} (in ID phase, i.e. in IFID[IR]) and I_{j+2} (in IF phase, i.e. in M[PC]) must be flushed
- Also to flush: IFID[IR]← I_{j+3} =M[PC+4] at $t+2$
- Always EXM[brT] (at $t+1$) = CMP.out (at t)
- brF←CMP.out∨EXM[brT] {use CMP.out for 2 clks}

- Flush when brF=1
- Set IFID[NOP]←brF
- I_{j+1} and then I_{j+2} fetched into IFID, trivial decoding in ID as IFID[NOP]=1

- Instruction in decoding phase is suspended (to flush) if either CMP.out=1 or IFID[NOP]=1
- flushF←CMP.out∨IFID[NOP]
- $I_{j+1}..I_{j+3}$ handled as NOPs in ID phase – trivial control signals generated and stored in IDEX
- Trivial operations for $I_{j+1}..I_{j+3}$ in EX, MEM and WB

- EXM is loaded as usual at end of current cycle
- PC assigned branch target at end of next cycle ($t+2$), target instruction in IFID at $t+3$

Flushing the pipeline on branching



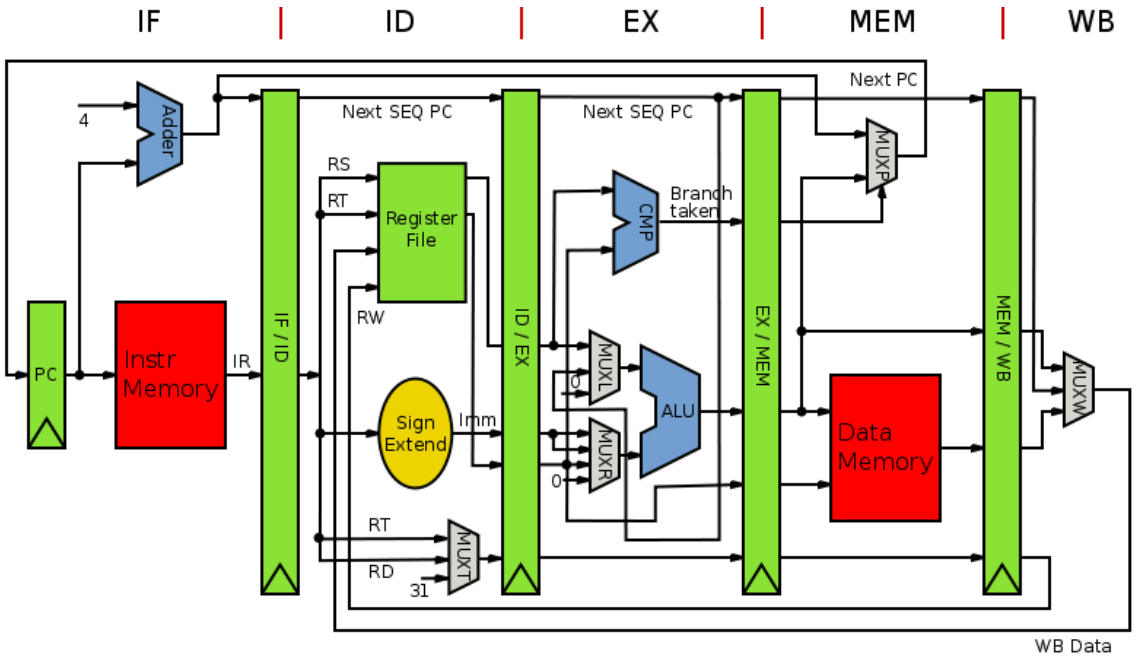
- CMP.out=1 indicates future branching due to I_j now in EX at t (computed early in clock cycle of EX)
- PC←PC+4 at $t+1$; PC←Branch target at $t+2$
- I_{j+1} (in ID phase, i.e. in IFID[IR]) and I_{j+2} (in IF phase, i.e. in M[PC]) must be flushed
- Also to flush: IFID[IR]← I_{j+3} =M[PC+4] at $t+2$
- Always EXM[brT] (at $t+1$) = CMP.out (at t)
- brF←CMP.out∨EXM[brT] {use CMP.out for 2 clks}

- Flush when brF=1
- Set IFID[NOP]←brF
- I_{j+1} and then I_{j+2} fetched into IFID, trivial decoding in ID as IFID[NOP]=1

- Instruction in decoding phase is suspended (to flush) if either CMP.out=1 or IFID[NOP]=1
- flushF←CMP.out∨IFID[NOP]
- $I_{j+1}..I_{j+3}$ handled as NOPs in ID phase – trivial control signals generated and stored in IDEX
- Trivial operations for $I_{j+1}..I_{j+3}$ in EX, MEM and WB

- EXM is loaded as usual at end of current cycle
- PC assigned branch target at end of next cycle ($t+2$), target instruction in IFID at $t+3$

Flushing the pipeline on branching



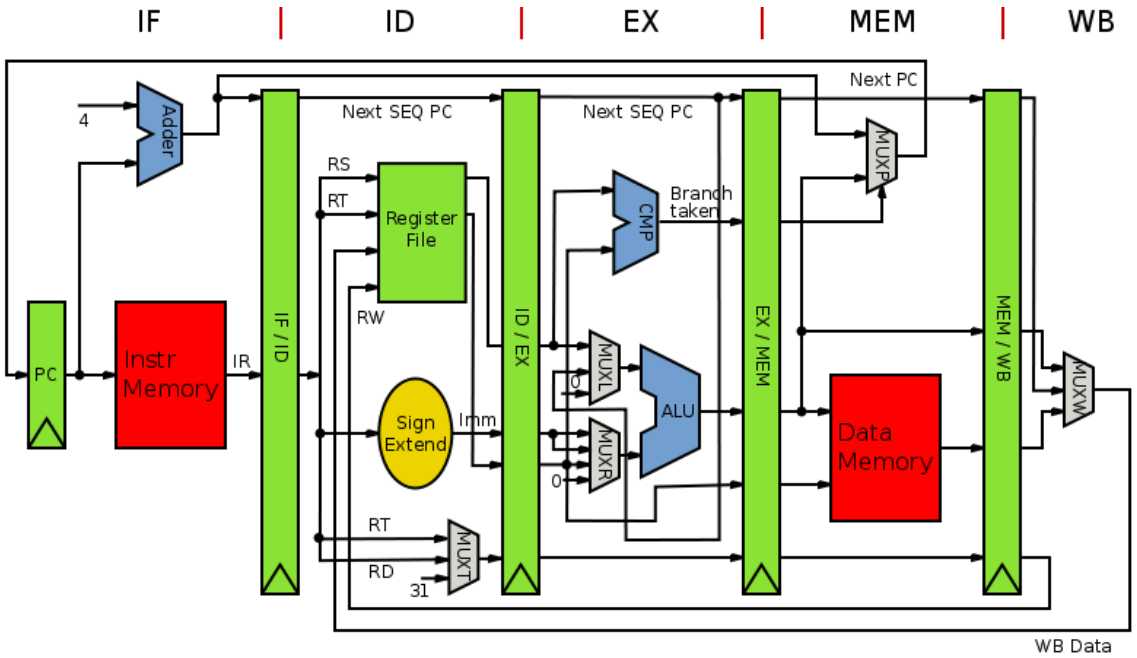
- CMP.out=1 indicates future branching due to I_j now in EX at t (computed early in clock cycle of EX)
- PC←PC+4 at $t+1$; PC←Branch target at $t+2$
- I_{j+1} (in ID phase, i.e. in IFID[IR]) and I_{j+2} (in IF phase, i.e. in M[PC]) must be flushed
- Also to flush: IFID[IR]← I_{j+3} =M[PC+4] at $t+2$
- Always EXM[brT] (at $t+1$) = CMP.out (at t)
- brF←CMP.out∨EXM[brT] {use CMP.out for 2 clks}

- Flush when brF=1
- Set IFID[NOP]←brF
- I_{j+1} and then I_{j+2} fetched into IFID, trivial decoding in ID as IFID[NOP]=1

- Instruction in decoding phase is suspended (to flush) if either CMP.out=1 or IFID[NOP]=1
- flushF←CMP.out∨IFID[NOP]
- $I_{j+1}..I_{j+3}$ handled as NOPs in ID phase – trivial control signals generated and stored in IDEX
- Trivial operations for $I_{j+1}..I_{j+3}$ in EX, MEM and WB

- EXM is loaded as usual at end of current cycle
- PC assigned branch target at end of next cycle ($t+2$), target instruction in IFID at $t+3$

Flushing the pipeline on branching



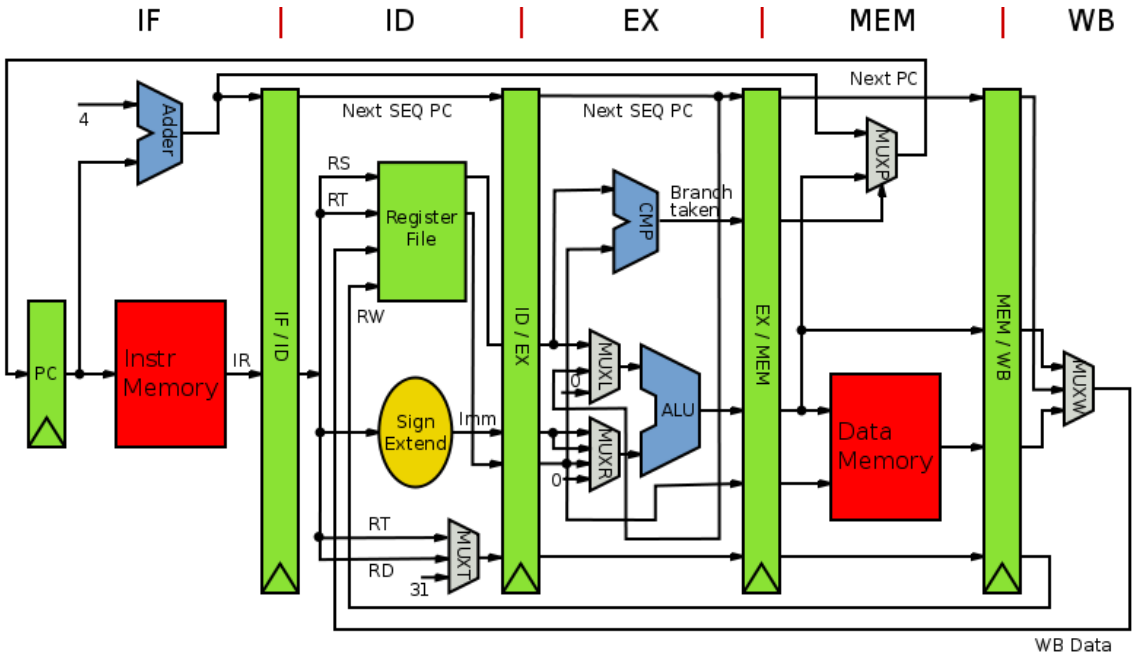
- CMP.out=1 indicates future branching due to I_j now in EX at t (computed early in clock cycle of EX)
- PC←PC+4 at $t+1$; PC←Branch target at $t+2$
- I_{j+1} (in ID phase, i.e. in IFID[IR]) and I_{j+2} (in IF phase, i.e. in M[PC]) must be flushed
- Also to flush: IFID[IR]← I_{j+3} =M[PC+4] at $t+2$
- Always EXM[brT] (at $t+1$) = CMP.out (at t)
- brF←CMP.out∨EXM[brT] {use CMP.out for 2 clks}

- Flush when brF=1
- Set IFID[NOP]←brF
- I_{j+1} and then I_{j+2} fetched into IFID, trivial decoding in ID as IFID[NOP]=1

- Instruction in decoding phase is suspended (to flush) if either CMP.out=1 or IFID[NOP]=1
- flushF←CMP.out∨IFID[NOP]
- $I_{j+1}..I_{j+3}$ handled as NOPs in ID phase – trivial control signals generated and stored in IDEX
- Trivial operations for $I_{j+1}..I_{j+3}$ in EX, MEM and WB

- EXM is loaded as usual at end of current cycle
- PC assigned branch target at end of next cycle ($t+2$), target instruction in IFID at $t+3$

Instruction decode and register fetch: R-type



R-type (add, sub, and, slt, ...): rd ← rs funct rt					
0	rs	rt	rd	shamt	funct
31-26	25-21	20-16	15-11	10-6	5-0

Hardwired: flushF←CMP.out∨IFID[NOP], RF.RS←IFID[IR].rs, RF.RT←IFID[IR].rt, RF.RW←MUXW.out, MUXT.0←IFID[IR].rt, MUXT.1←IFID[IR].rd, MUXL.0←IDEX[RSD], MUXR.2←IDEX[RTD]

- [!stallF ∧ !flushF ∧ IFID[IR].op=0 ∧ IFID[IR].funct∉[0x09..0x09]] {excluding jr, jalr}

Setup operands

- IDEX[RSD]←RF[IFID[IR].rs]
- IDEX[RTD]←RF[IFID[IR].rt]
- IDEX[RW]←IFID[IR].rd [xMUXT←1]

ALU setup

- IDEX[ALUOP]←f_1(IFID[IR].funct, IFID[IR].shamt)
- IDEX[xMUXL]←0 {default}
- IDEX[xMUXR]←1

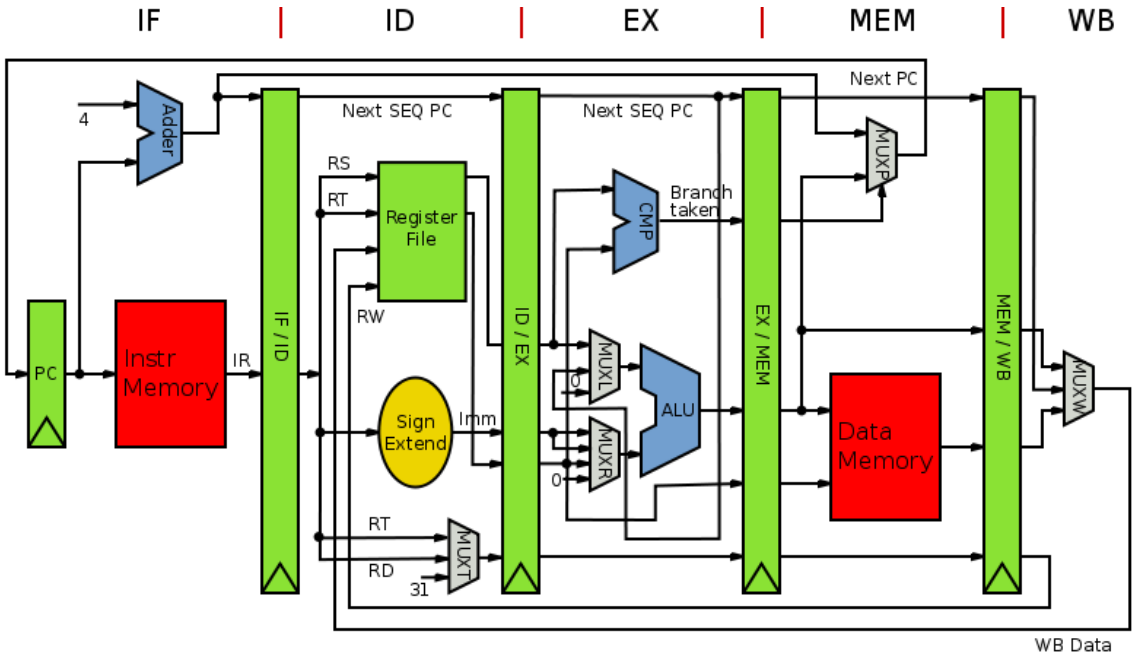
No branch No Mem setup (default)

- IDEX[DMRF]←0
- IDEX[DMWF]←0
- IDEX[CMPEn]←0

WB setup

- IDEX[WB]←1
- IDEX[xMUXW]←0 {default}

Instruction decode and register fetch: R-type



R-type (add, sub, and, slt, ...): rd ← rs funct rt					
0	rs	rt	rd	shamt	funct
31-26	25-21	20-16	15-11	10-6	5-0

Hardwired: flushF←CMP.out∨IFID[NOP], RF.RS←IFID[IR].rs, RF.RT←IFID[IR].rt, RF.RW←MUXW.out, MUXT.0←IFID[IR].rt, MUXT.1←IFID[IR].rd, MUXL.0←IDEX[RSD], MUXR.2←IDEX[RTD]

- [!stallF ∧ !flushF ∧ IFID[IR].op=0 ∧ IFID[IR].funct∉[0x09..0x09]] {excluding jr, jalr}

Setup operands

- IDEX[RSD]←RF[IFID[IR].rs]
- IDEX[RTD]←RF[IFID[IR].rt]
- IDEX[RW]←IFID[IR].rd [xMUXT←1]

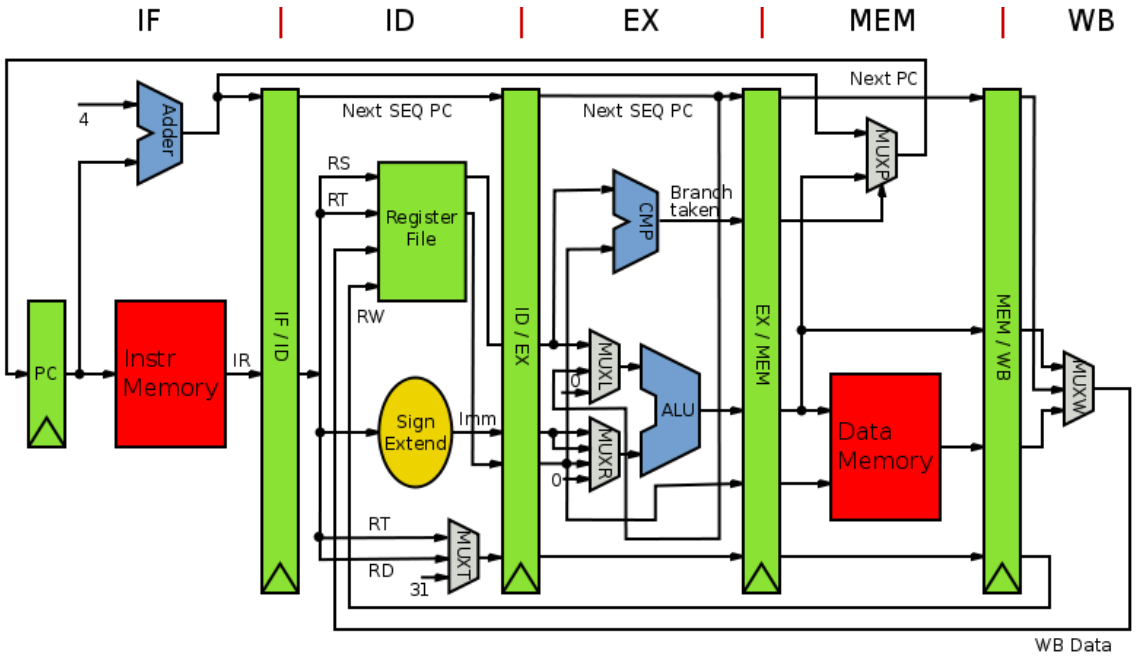
ALU setup

- IDEX[ALUOP]← f_1(IFID[IR].funct, IFID[IR].shamt)
- IDEX[xMUXL]←0 {default}
- IDEX[xMUXR]←1

No branch No Mem WB setup (default)

- IDEX[DMRF]←0
- IDEX[DMWF]←0
- IDEX[CMPEn]←0
- IDEX[WB]←1
- IDEX[xMUXW]←0 {default}

Instruction decode and register fetch: R-type



R-type (add, sub, and, slt, ...): rd ← rs funct rt					
0	rs	rt	rd	shamt	funct
31-26	25-21	20-16	15-11	10-6	5-0

Hardwired: flushF←CMP.out∨IFID[NOP], RF.RS←IFID[IR].rs, RF.RT←IFID[IR].rt, RF.RW←MUXW.out, MUXT.0←IFID[IR].rt, MUXT.1←IFID[IR].rd, MUXL.0←IDEX[RSD], MUXR.2←IDEX[RTD]

- [!stallF ∧ !flushF ∧ IFID[IR].op=0 ∧ IFID[IR].funct∉[0x09..0x09]] {excluding jr, jalr}

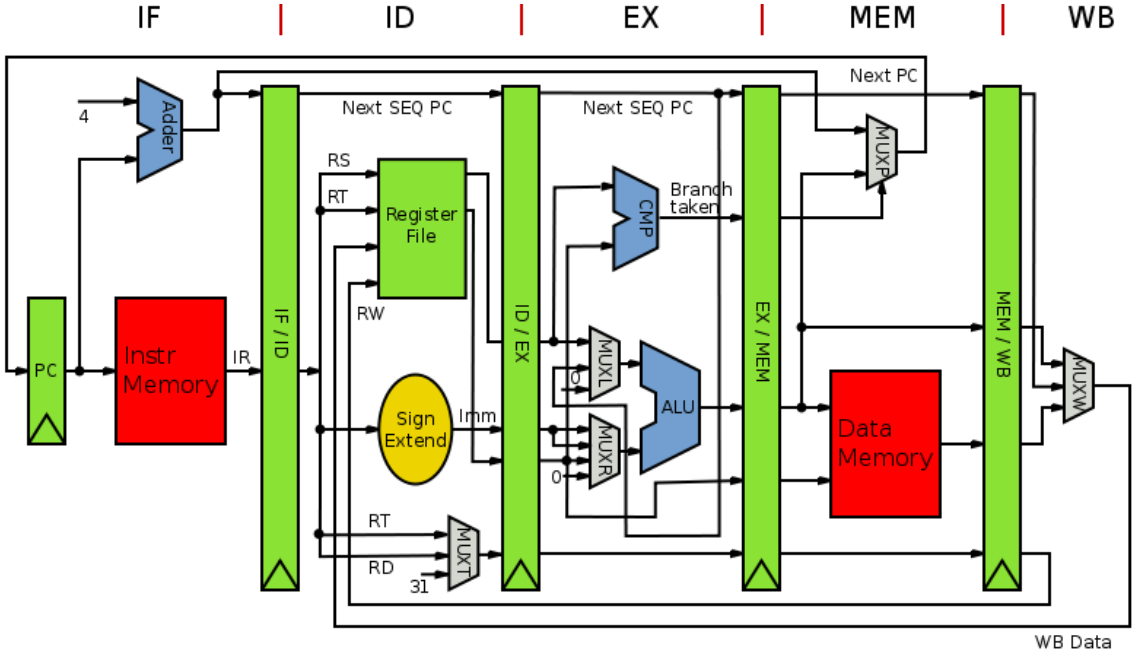
Setup operands
- IDEX[RSD]←RF[IFID[IR].rs]
- IDEX[RTD]←RF[IFID[IR].rt]
- IDEX[RW]←IFID[IR].rd [xMUXT←1]

ALU setup
- IDEX[ALUOP]← f_1(IFID[IR].funct, IFID[IR].shamt)
- IDEX[xMUXL]←0 {default}
- IDEX[xMUXR]←1

No branch No Mem (default)
- IDEX[DMRF]←0
- IDEX[DMWF]←0
- IDEX[CMPEn]←0

WB setup
- IDEX[WB]←1
- IDEX[xMUXW]←0 {default}

Instruction decode and register fetch: R-type



R-type (add, sub, and, slt, ...): rd ← rs funct rt					
0	rs	rt	rd	shamt	funct
31-26	25-21	20-16	15-11	10-6	5-0

Hardwired: flushF←CMP.out∨IFID[NOP], RF.RS←IFID[IR].rs, RF.RT←IFID[IR].rt, RF.RW←MUXW.out, MUXT.0←IFID[IR].rt, MUXT.1←IFID[IR].rd, MUXL.0←IDEX[RSD], MUXR.2←IDEX[RTD]

- [!stallF ∧ !flushF ∧ IFID[IR].op=0 ∧ IFID[IR].funct∉[0x09..0x09]] {excluding jr, jalr}

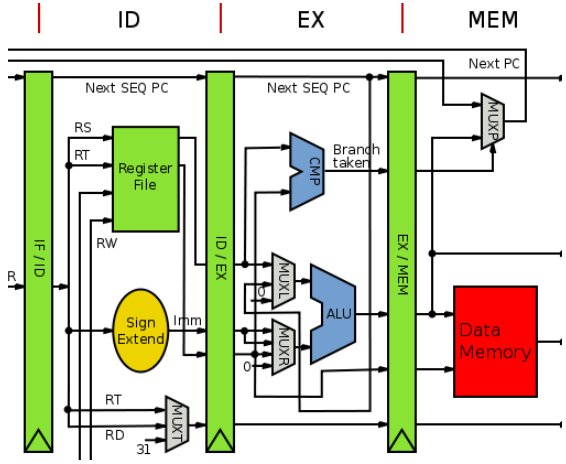
Setup operands
- IDEX[RSD]←RF[IFID[IR].rs]
- IDEX[RTD]←RF[IFID[IR].rt]
- IDEX[RW]←IFID[IR].rd [xMUXT←1]

ALU setup
- IDEX[ALUOP]← f_1(IFID[IR].funct, IFID[IR].shamt)
- IDEX[xMUXL]←0 {default}
- IDEX[xMUXR]←1

No branch No Mem (default)
- IDEX[DMRF]←0
- IDEX[DMWF]←0
- IDEX[CMPEn]←0

WB setup
- IDEX[WB]←1
- IDEX[xMUXW]←0 {default}

Stalling of instruction decode and register fetch: R-type



Stalling of ID/RF due to data hazards

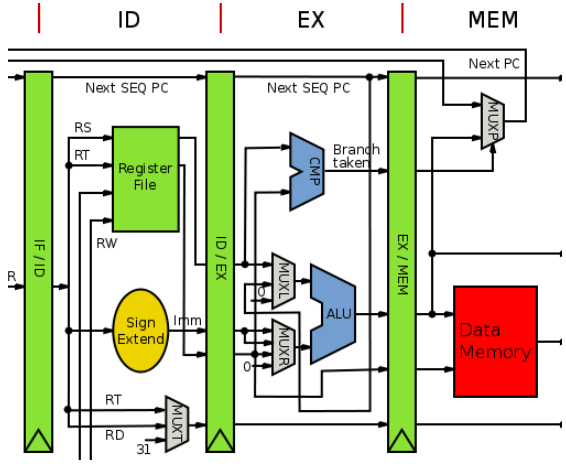
Cannot proceed with register read if:

- One of "rs" or "rt" is defined by instruction in the EX phase: IDEX[WB] $\wedge$ (IDEX[RW]=IFID[IR].rs $\vee$ IDEX[RW]=IFID[IR].rt)
- One of "rs" or "rt" is defined by instruction in the MEM phase: EXM[WB] $\wedge$ (EXM[RW]=IFID[IR].rs $\vee$ EXM[RW]=IFID[IR].rt)

Assert stallF if the above conditions are satisfied

- Stalling of instruction I_j is always identified at ID phase
- Hazard lasts at most 2 cycles
- If instruction I_j is stalled (in ID), it must not be overwritten by I_{j+1} (in IF)
- I_{j+1} not to be skipped over by incrementing PC without fetching (and storing) I_{j+1}
- Writing to both PC and IFID must be inhibited on stalling
- $I_{j-1}, I_{j-2}, I_{j-3}$ must proceed, to be replaced by bubbles
- Instruction decoding proceeds non-trivially only when (!stallF $\wedge$!flushF) is satisfied, otherwise behaves as NOP introducing bubbles after ID
- Stalling does not come in conflict with flushing

Stalling of instruction decode and register fetch: R-type



- Stalling of instruction I_j is always identified at ID phase
- Hazard lasts at most 2 cycles
- If instruction I_j is stalled (in ID), it must not be overwritten by I_{j+1} (in IF)
- I_{j+1} not to be skipped over by incrementing PC without fetching (and storing) I_{j+1}
- Writing to both PC and IFID must be inhibited on stalling
- $I_{j-1}, I_{j-2}, I_{j-3}$ must proceed, to be replaced by bubbles
- Instruction decoding proceeds non-trivially only when (!stallF $\wedge$!flushF) is satisfied, otherwise behaves as NOP introducing bubbles after ID
- Stalling does not come in conflict with flushing

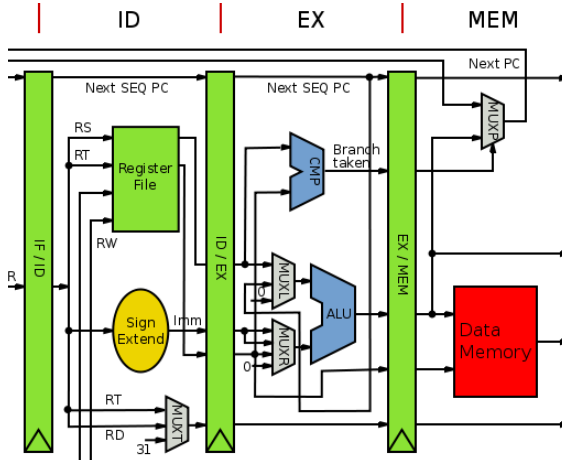
Stalling of ID/RF due to data hazards

Cannot proceed with register read if:

- One of "rs" or "rt" is defined by instruction in the EX phase: IDEX[WB] $\wedge$ (IDEX[RW]=IFID[IR].rs $\vee$ IDEX[RW]=IFID[IR].rt)
- One of "rs" or "rt" is defined by instruction in the MEM phase: EXM[WB] $\wedge$ (EXM[RW]=IFID[IR].rs $\vee$ EXM[RW]=IFID[IR].rt)

Assert stallF if the above conditions are satisfied

Stalling of instruction decode and register fetch: R-type



Stalling of ID/RF due to data hazards

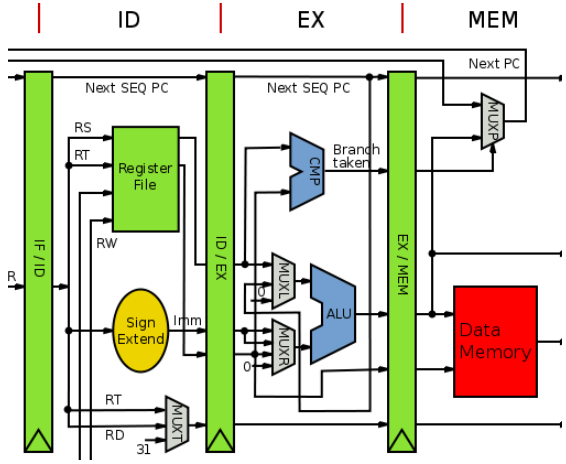
Cannot proceed with register read if:

- One of "rs" or "rt" is defined by instruction in the EX phase: IDEX[WB] $\wedge$ (IDEX[RW]=IFID[IR].rs $\vee$ IDEX[RW]=IFID[IR].rt)
- One of "rs" or "rt" is defined by instruction in the MEM phase: EXM[WB] $\wedge$ (EXM[RW]=IFID[IR].rs $\vee$ EXM[RW]=IFID[IR].rt)

Assert stallF if the above conditions are satisfied

- Stalling of instruction I_j is always identified at ID phase
- Hazard lasts at most 2 cycles
- If instruction I_j is stalled (in ID), it must not be overwritten by I_{j+1} (in IF)
- I_{j+1} not to be skipped over by incrementing PC without fetching (and storing) I_{j+1}
- Writing to both PC and IFID must be inhibited on stalling
- $I_{j-1}, I_{j-2}, I_{j-3}$ must proceed, to be replaced by bubbles
- Instruction decoding proceeds non-trivially only when (!stallF $\wedge$!flushF) is satisfied, otherwise behaves as NOP introducing bubbles after ID
- Stalling does not come in conflict with flushing

Stalling of instruction decode and register fetch: R-type



Stalling of ID/RF due to data hazards

Cannot proceed with register read if:

- One of "rs" or "rt" is defined by instruction in the EX phase: IDEX[WB] $\wedge$ (IDEX[RW]=IFID[IR].rs $\vee$ IDEX[RW]=IFID[IR].rt)
- One of "rs" or "rt" is defined by instruction in the MEM phase: EXM[WB] $\wedge$ (EXM[RW]=IFID[IR].rs $\vee$ EXM[RW]=IFID[IR].rt)

Assert stallF if the above conditions are satisfied

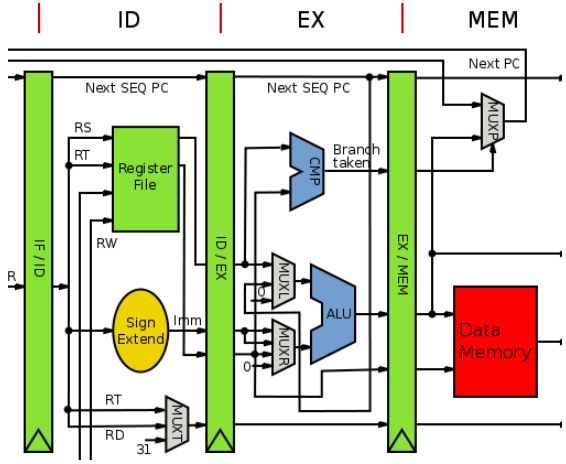
- Stalling of instruction I_j is always identified at ID phase
- Hazard lasts at most 2 cycles

- If instruction I_j is stalled (in ID), it must not be overwritten by I_{j+1} (in IF)
- I_{j+1} not to be skipped over by incrementing PC without fetching (and storing) I_{j+1}
- Writing to both PC and IFID must be inhibited on stalling

- $I_{j-1}, I_{j-2}, I_{j-3}$ must proceed, to be replaced by bubbles
- Instruction decoding proceeds non-trivially only when (!stallF $\wedge$!flushF) is satisfied, otherwise behaves as NOP introducing bubbles after ID

- Stalling does not come in conflict with flushing

Stalling of instruction decode and register fetch: R-type



Stalling of ID/RF due to data hazards

Cannot proceed with register read if:

- One of "rs" or "rt" is defined by instruction in the EX phase: IDEX[WB] $\wedge$ (IDEX[RW]=IFID[IR].rs $\vee$ IDEX[RW]=IFID[IR].rt)
- One of "rs" or "rt" is defined by instruction in the MEM phase: EXM[WB] $\wedge$ (EXM[RW]=IFID[IR].rs $\vee$ EXM[RW]=IFID[IR].rt)

Assert stallF if the above conditions are satisfied

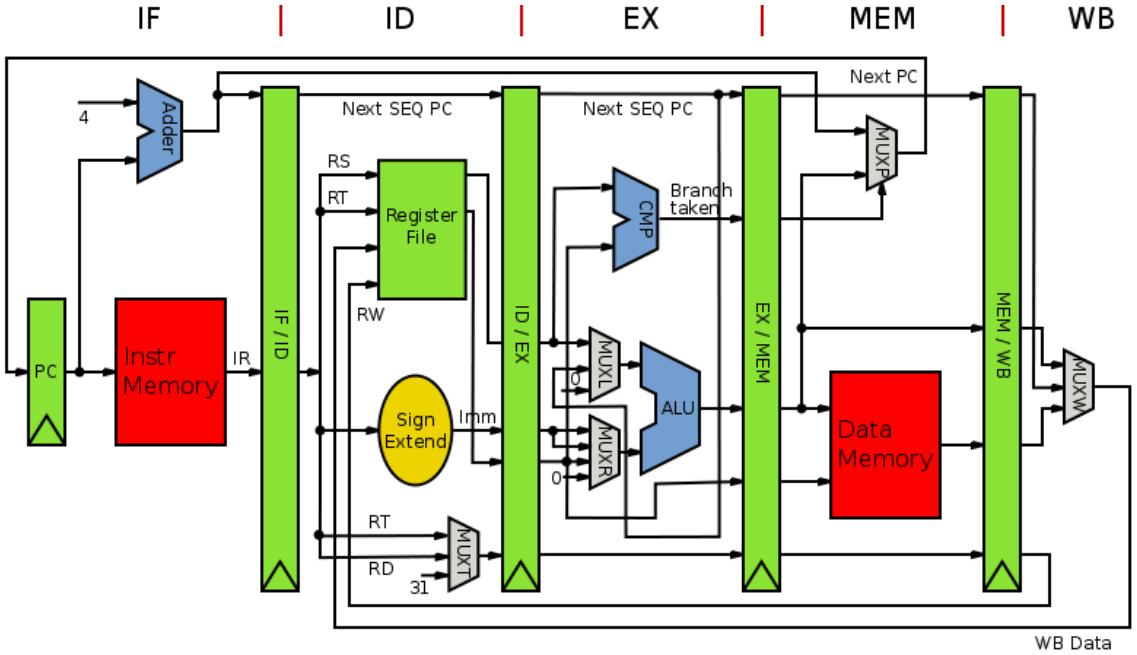
- Stalling of instruction I_j is always identified at ID phase
- Hazard lasts at most 2 cycles

- If instruction I_j is stalled (in ID), it must not be overwritten by I_{j+1} (in IF)
- I_{j+1} not to be skipped over by incrementing PC without fetching (and storing) I_{j+1}
- Writing to both PC and IFID must be inhibited on stalling

- $I_{j-1}, I_{j-2}, I_{j-3}$ must proceed, to be replaced by bubbles
- Instruction decoding proceeds non-trivially only when (!stallF $\wedge$!flushF) is satisfied, otherwise behaves as NOP introducing bubbles after ID

- Stalling does not come in conflict with flushing

Instruction decode and register fetch: RI-type



RI-type (addi, addiu, ...): rt ← rs op imm			
ri-op	rs	rt	imm
31-26	25-21	20-16	15-0

Opcodes in the range 0x08..0x0e

Hardwired: flushF←CMP.out∨IFID[NOP], MUXR.0←IDEX.imm, SignExt.input←IFID[IR].imm26Bit, SignExt.cntrl←jxtF {jxtF asserted only for jumps}

Setup rs, imm, rt (dest)

- [!stallF ∧ !flushF ∧ IDEX[IR].op∈[0x08..0x0e]]

- IDEX[RSD]←RF[IFID[IR].rs]
- IDEX[IMM]←SignExt (RF[IFID[IR].imm], jxtF) [jxtF←0] (default)
- IDEX[RW]←IFID[IR].rt [xMUXT←0]

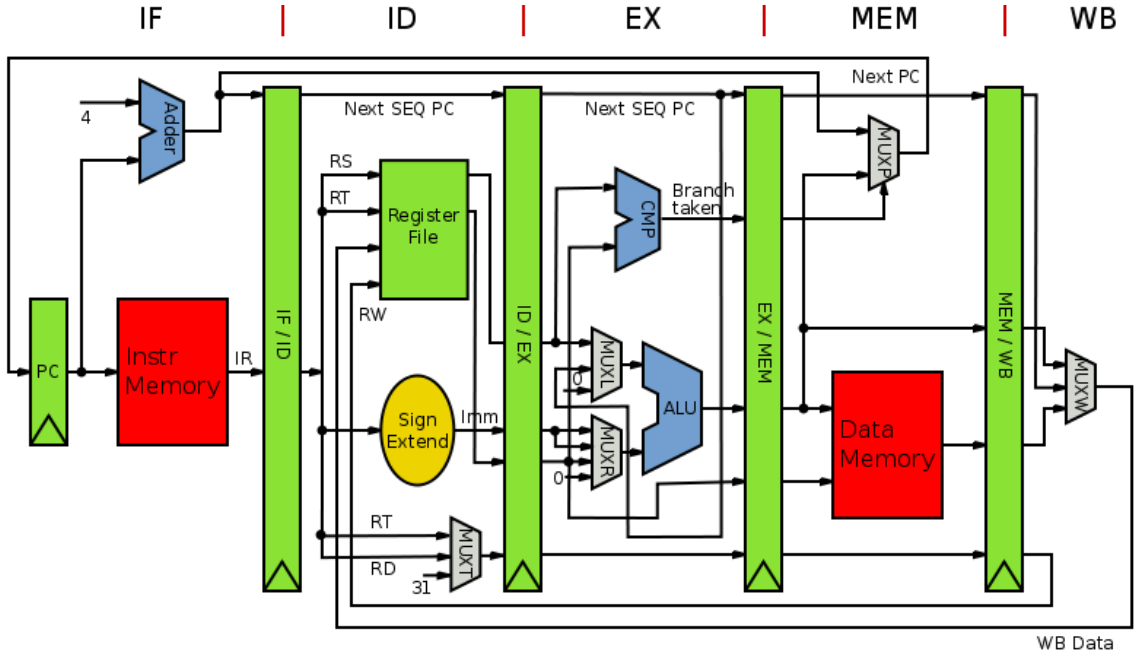
ALU setup

- IDEX[ALUOP]←f_2(IFID[IR].op)
- IDEX[xMUXL]←0
- IDEX[xMUXR]←0

- IDEX[DMRF]←0
- IDEX[DMWF]←0
- IDEX[CMPEn]←0

- IDEX[WB]←1
- IDEX[xMUXW]←0

Instruction decode and register fetch: RI-type



RI-type (addi, addiu, ...): rt ← rs op imm			
ri-op	rs	rt	imm
31-26	25-21	20-16	15-0

Opcodes in the range 0x08..0x0e

Hardwired: flushF←CMP.out∨IFID[NOP], MUXR.0←IDEX.imm, SignExt.input←IFID[IR].imm26Bit, SignExt.cntrl←jxtF {jxtF asserted only for jumps}

- [!stallF ∧ !flushF ∧ IDEX[IR].op∈[0x08..0x0e]]

Setup rs, imm, rt (dest)

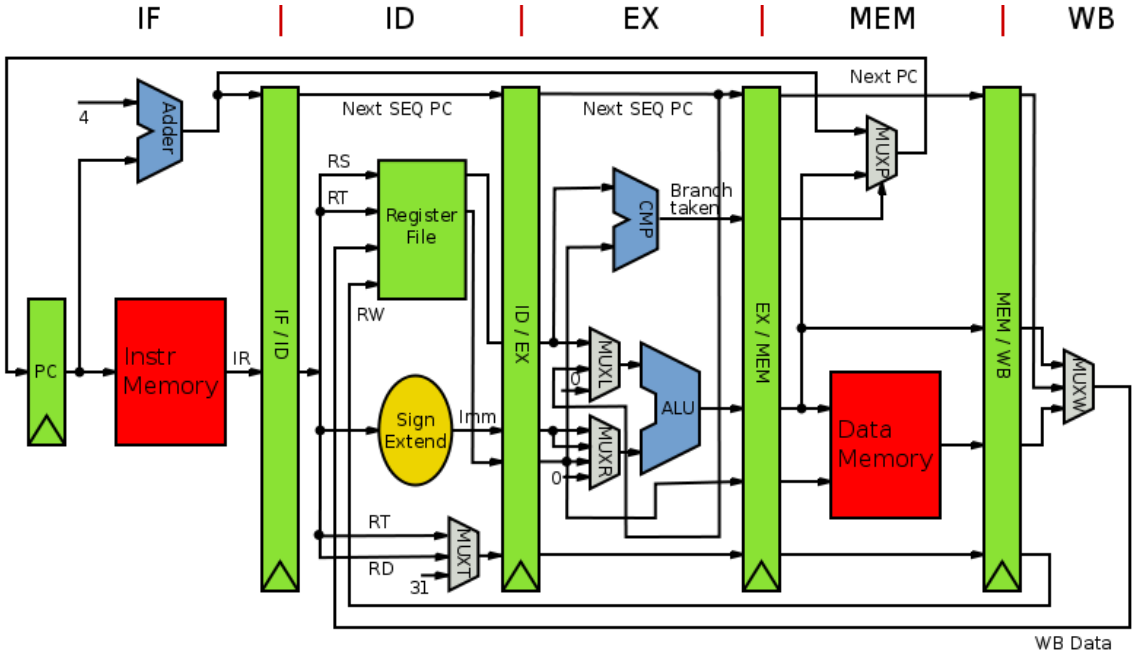
- IDEX[RSD]←RF[IFID[IR].rs]
- IDEX[IMM]←SignExt (RF[IFID[IR].imm], jxtF) [jxtF←0] (default)
- IDEX[RW]←IFID[IR].rt [xMUXT←0]

ALU setup

- IDEX[ALUOP]← f_2(IFID[IR].op)
- IDEX[xMUXL]←0
- IDEX[xMUXR]←0

- IDEX[DMRF]←0
- IDEX[DMWF]←0
- IDEX[CMPEn]←0
- IDEX[WB]←1
- IDEX[xMUXW]←0

Instruction decode and register fetch: RI-type



RI-type (addi, addiu, ...): rt ← rs op imm			
ri-op	rs	rt	imm
31-26	25-21	20-16	15-0

Opcodes in the range 0x08..0x0e

Hardwired: flushF←CMP.out∨IFID[NOP], MUXR.0←IDEX.imm, SignExt.input←IFID[IR].imm26Bit, SignExt.cntrl←jxtF {jxtF asserted only for jumps}

- [!stallF ∧ !flushF ∧ IDEX[IR].op∈[0x08..0x0e]]

Setup rs, imm, rt (dest)

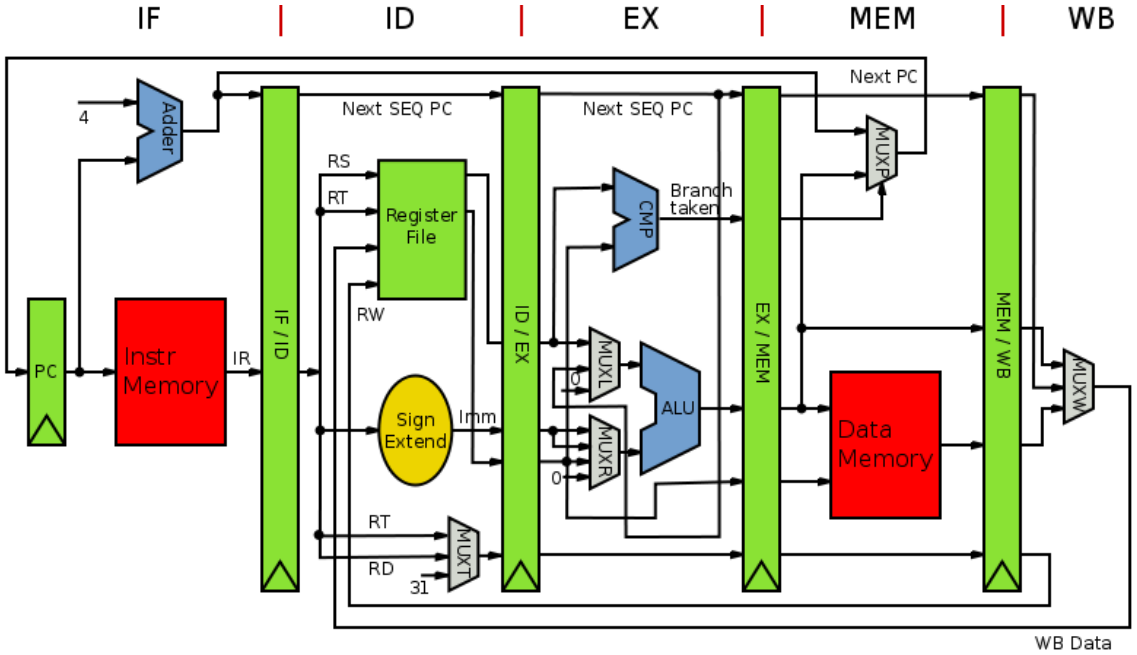
- IDEX[RSD]←RF[IFID[IR].rs]
- IDEX[IMM]←SignExt (RF[IFID[IR].imm], jxtF) [jxtF←0] (default)
- IDEX[RW]←IFID[IR].rt [xMUXT←0]

ALU setup

- IDEX[ALUOP]← f_2(IFID[IR].op)
- IDEX[xMUXL]←0
- IDEX[xMUXR]←0

- IDEX[DMRF]←0
- IDEX[DMWF]←0
- IDEX[CMPEn]←0
- IDEX[WB]←1
- IDEX[xMUXW]←0

Instruction decode and register fetch: RI-type



RI-type (addi, addiu, ...): rt ← rs op imm			
ri-op	rs	rt	imm
31-26	25-21	20-16	15-0

Opcodes in the range 0x08..0x0e

Hardwired: flushF←CMP.out∨IFID[NOP], MUXR.0←IDEX.imm, SignExt.input←IFID[IR].imm26Bit, SignExt.cntrl←jxtF {jxtF asserted only for jumps}

- [!stallF ∧ !flushF ∧ IDEX[IR].op∈[0x08..0x0e]]

Setup rs, imm, rt (dest)

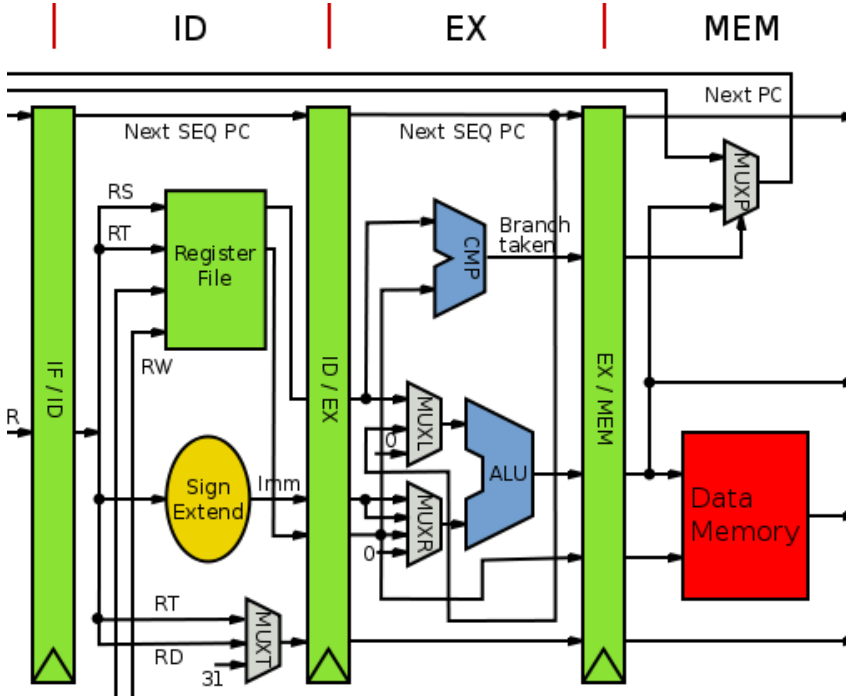
- IDEX[RSD]←RF[IFID[IR].rs]
- IDEX[IMM]←SignExt (RF[IFID[IR].imm], jxtF) [jxtF←0] (default)
- IDEX[RW]←IFID[IR].rt [xMUXT←0]

ALU setup

- IDEX[ALUOP]← f_2(IFID[IR].op)
- IDEX[xMUXL]←0
- IDEX[xMUXR]←0

- IDEX[DMRF]←0
- IDEX[DMWF]←0
- IDEX[CMPEn]←0
- IDEX[WB]←1
- IDEX[xMUXW]←0

Stalling of instruction decode and register fetch: RI-type



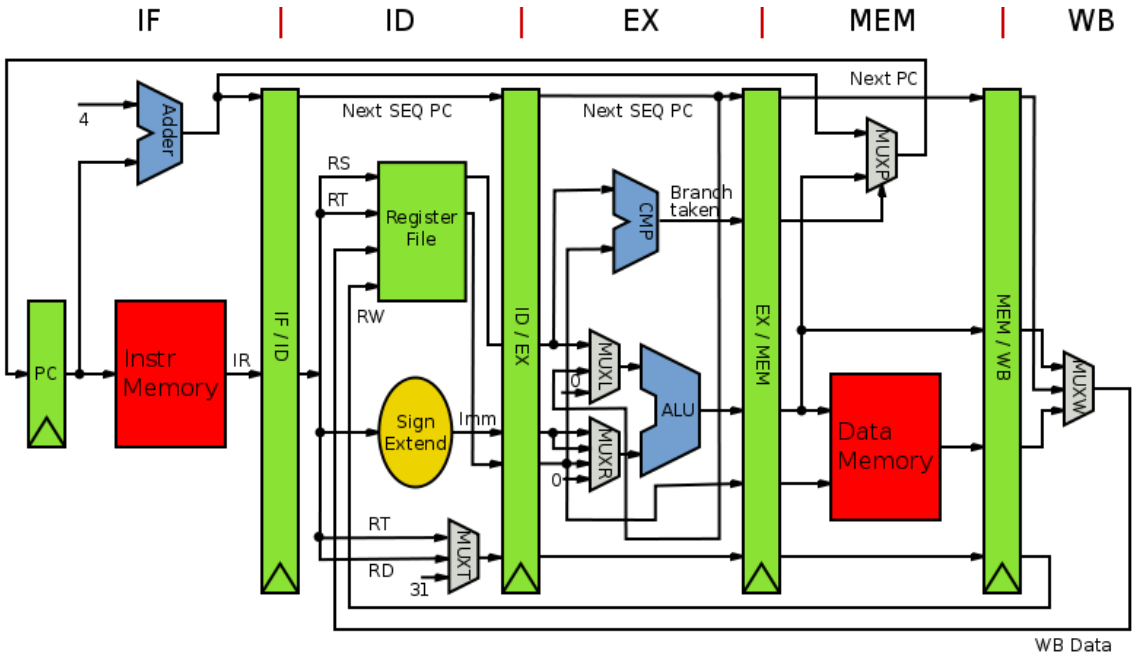
Stalling of ID/RF due to data hazards

Cannot proceed with register read if:

- Register "rs" is defined by instruction in the EX phase: IDEX[WB] $\wedge$ IDEX[RW]=RFIFID[IR].rs
- Register "rs" is defined by instruction in the MEM phase: EXM[WB] $\wedge$ EXM[RW]=RFIFID[IR].rs

Assert stallF if the above conditions are satisfied

Instruction decode and register fetch: Conditional branching



Branching (beq, bne): [CMP] ∧ PC ←PC+4+imm×4			
br-op	rs	rt	imm
31-26	25-21	20-16	15-0

Opcodes: 0x04 (beq), 0x05 (bne)

Hardwired: flushF←CMP.out∨IFID[NOP], MUXR.1←4×IDEX.imm, CMP.inv←IDEX[CMPInv] {to invert the sense of the comparison for bne}, CMP.en←IDEX[CMPEn]

CMP setup

- [!stallF ∧ !flushF ∧ IDEX[IR].op∈[0x04..0x05]] {Stall on rs or rt?}
- IDEX[RSD]←RF[IFID[IR].rs]
- IDEX[RSX]←RF[IFID[IR].rt]
- IDEX[CMPInv]← !stallF ∧ !flushF ∧ IFID[IR].op=0x05 {bne}
- IDEX[CMPEn]←1

Setup br addresses

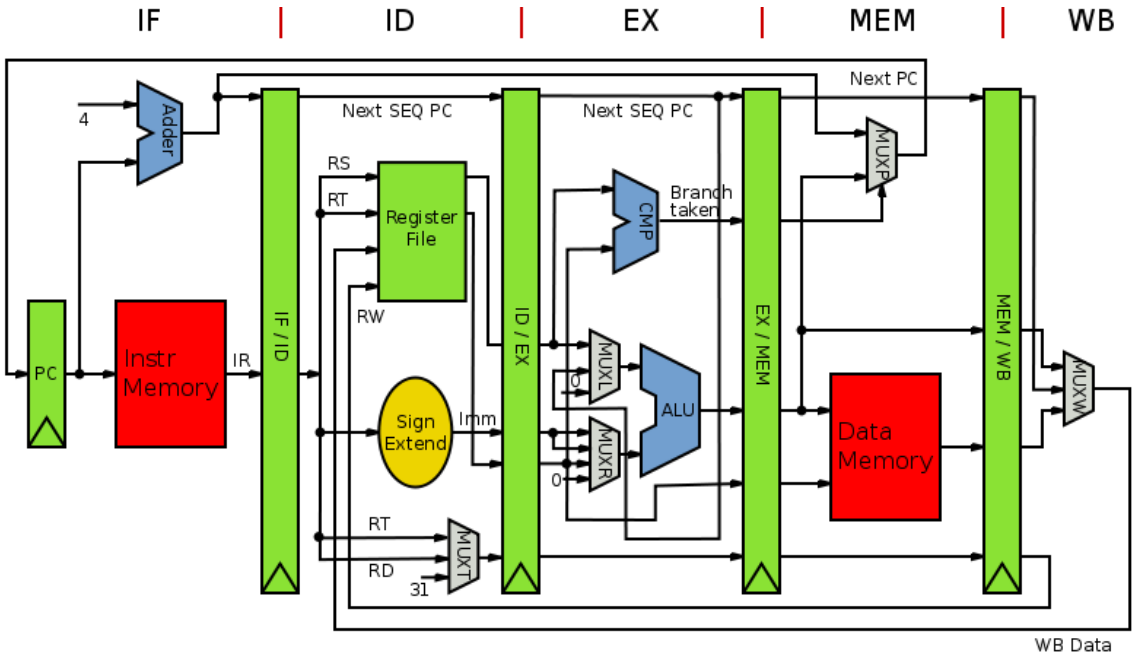
- IDEX[IMM]←SignExt (RF[IFID[IR].imm], jxtF) [jxtF←0 (default)]
- IDEX[nxPC]←IFID[nxPC]

Setup for address arith

To achieve: ALU.out←PC+4+imm×4

- IDEX[ALUOP]←ADD
- IDEX[xMUXL]←1
- IDEX[xMUXR]←1

Instruction decode and register fetch: Conditional branching



Branching (beq, bne): [CMP] ∧ PC ←PC+4+imm×4			
br-op	rs	rt	imm
31-26	25-21	20-16	15-0

Opcodes: 0x04 (beq), 0x05 (bne)

Hardwired: flushF←CMP.out∨IFID[NOP], MUXR.1←4×IDEX.imm, CMP.inv←IDEX[CMPInv] {to invert the sense of the comparison for bne}, CMP.en←IDEX[CMPEn]

CMP setup

- [!stallF ∧ !flushF ∧ IDEX[IR].op∈[0x04..0x05]] {Stall on rs or rt?}
- IDEX[RSD]←RF[IFID[IR].rs]
- IDEX[RSX]←RF[IFID[IR].rt]
- IDEX[CMPInv]← !stallF ∧ !flushF ∧ IFID[IR].op=0x05 {bne}
- IDEX[CMPEn]←1

Setup br addresses

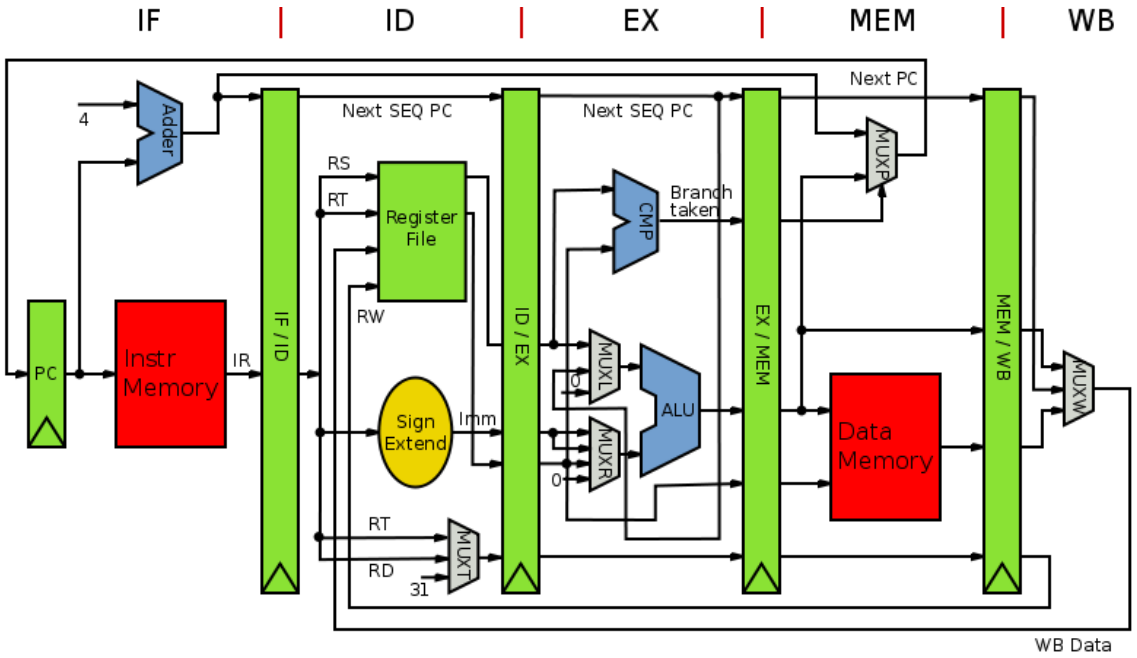
- IDEX[IMM]←SignExt (RF[IFID[IR].imm], jxtF) [jxtF←0 (default)]
- IDEX[nxPC]←IFID[nxPC]

Setup for address arith

To achieve: ALU.out←PC+4+imm×4

- IDEX[ALUOP]←ADD
- IDEX[xMUXL]←1
- IDEX[xMUXR]←1

Instruction decode and register fetch: Conditional branching



Branching (beq, bne): [CMP] ∧ PC ←PC+4+imm×4			
br-op	rs	rt	imm
31-26	25-21	20-16	15-0

Opcodes: 0x04 (beq), 0x05 (bne)

Hardwired: flushF←CMP.out∨IFID[NOP], MUXR.1←4×IDEX.imm, CMP.inv←IDEX[CMPInv] {to invert the sense of the comparison for bne}, CMP.en←IDEX[CMPEn]

CMP setup

- [!stallF ∧ !flushF ∧ IDEX[IR].op∈[0x04..0x05]] {Stall on rs or rt?}

Setup br addresses

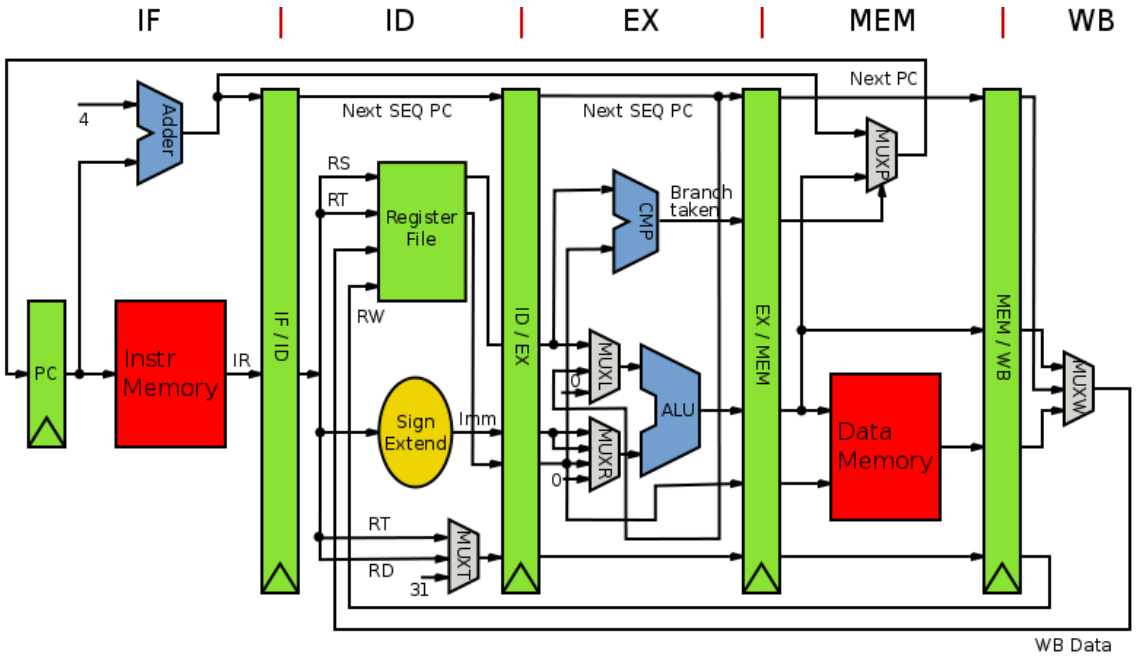
- IDEX[RSD]←RF[IFID[IR].rs]
- IDEX[RSX]←RF[IFID[IR].rt]
- IDEX[CMPInv]← !stallF ∧ !flushF ∧ IFID[IR].op=0x05 {bne}
- IDEX[CMPEn]←1
- IDEX[IMM]←SignExt (RF[IFID[IR].imm], jxtF) [jxtF←0 (default)]
- IDEX[nxPC]←IFID[nxPC]

Setup for address arith

To achieve: ALU.out←PC+4+imm×4

- IDEX[ALUOP]←ADD
- IDEX[xMUXL]←1
- IDEX[xMUXR]←1

Instruction decode and register fetch: Jump [and link]



Jumps (j, jal): PC←JAddr, [jal]∧R[31]←PC+4	
j/jal-op	imm
31-26	25-0

Opcodes: 0x02 (j), 0x03 (jal)

Hardwired: flushF←CMP.out∨IFID[NOP], MUXT.2←31 {for R[31]}, MUXL.2←0, MUXR.1←4×IDEX.imm, CMP.true←IDEX[CMPTrue] {make CMP true}

Setup jump addresses

- [!flushF ∧ IDEX[IR].op∈[0x02..0x03]]
- IDEX[IMM]←SignExt (RF[IFID[IR].imm], jxtF) [jxtF←1] {Note use of 26-bit addr}

Setup for address arith

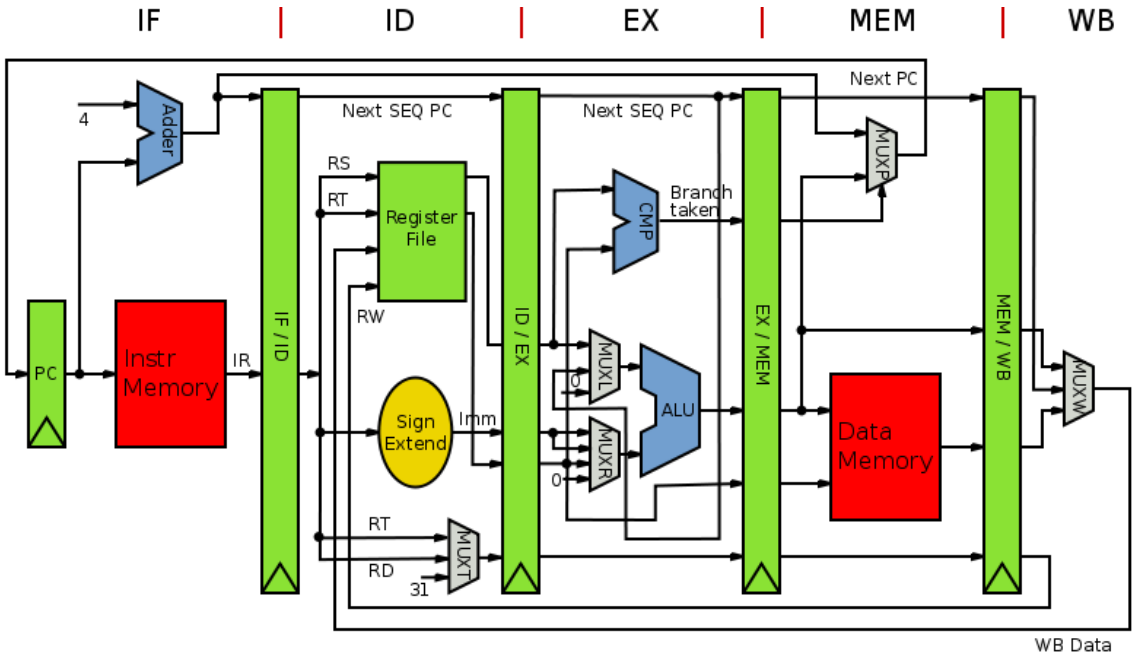
To unconditionally perform: NxPC←0+imm×4

- IDEX[ALUOP]←ADD
- IDEX[xMUXL]←2
- IDEX[xMUXR]←1
- IDEX[CMPTrue]←1

Setup for jal instr

- [!flushF ∧ IFID[IR].op=0x03] {extra for jal instruction}
- IDEX[nxPC]←IFID[nxPC]
- xMUXT←2 {for RW←31}
- IDEX[xMUXW]←1
- IDEX[WB]←1

Instruction decode and register fetch: Jump [and link]



Jumps (j, jal): PC←JAddr, [jal]∧R[31]←PC+4	
j/jal-op	imm
31-26	25-0

Opcodes: 0x02 (j), 0x03 (jal)

Hardwired: flushF←CMP.out∨IFID[NOP], MUXT.2←31 {for R[31]}, MUXL.2←0, MUXR.1←4×IDEX.imm, CMP.true←IDEX[CMPTrue] {make CMP true}

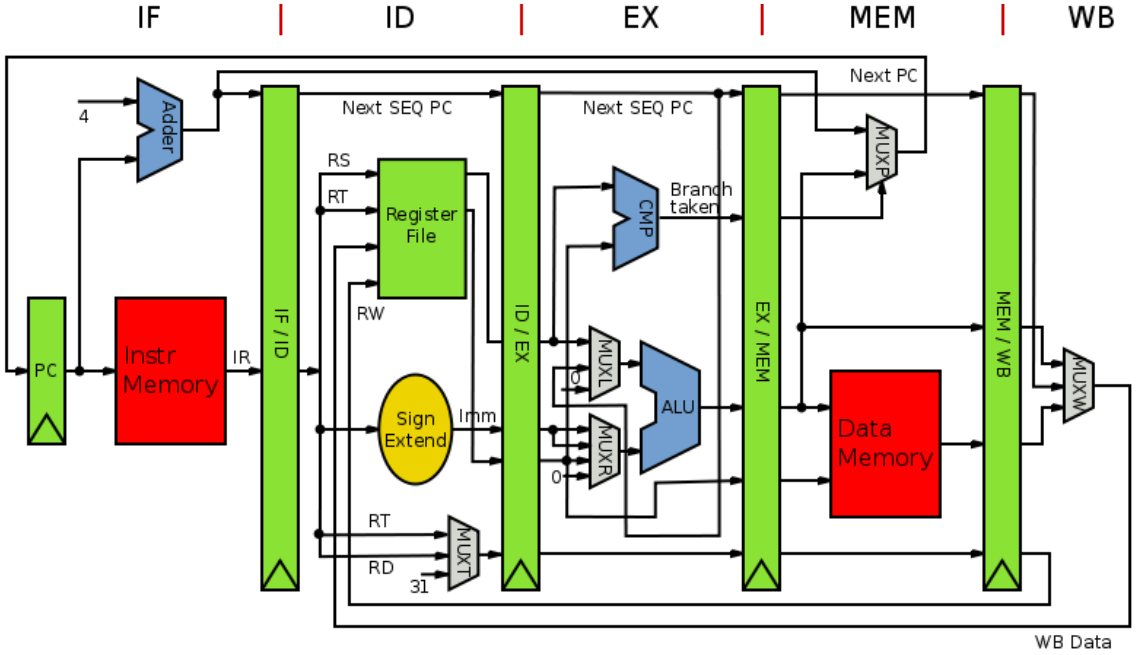
Setup jump addresses:
- [!flushF ∧ IDEX[IR].op∈[0x02..0x03]]
- IDEX[IMM]←SignExt (RF[IFID[IR].imm], jxtF) [jxtF←1] {Note use of 26-bit addr}

To unconditionally perform: NxPC←0+imm×4

Setup for address arith:
- IDEX[ALUOP]←ADD
- IDEX[xMUXL]←2
- IDEX[xMUXR]←1
- IDEX[CMPTrue]←1

Setup for jal instr:
- [!flushF ∧ IFID[IR].op=0x03] {extra for jal instruction}
- IDEX[nxPC]←IFID[nxPC]
- xMUXT←2 {for RW←31}
- IDEX[xMUXW]←1
- IDEX[WB]←1

Instruction decode and register fetch: Jump [and link]



Jumps (j, jal): PC←JAddr, [jal]∧R[31]←PC+4	
j/jal-op	imm
31-26	25-0

Opcodes: 0x02 (j), 0x03 (jal)

Hardwired: flushF←CMP.out∨IFID[NOP], MUXT.2←31 {for R[31]}, MUXL.2←0, MUXR.1←4×IDEX.imm, CMP.true←IDEX[CMPTrue] {make CMP true}

- [!flushF ∧ IDEX[IR].op∈[0x02..0x03]]

Setup jump addresses

- IDEX[IMM]←SignExt (RF[IFID[IR].imm], jxtF) [jxtF←1] {Note use of 26-bit addr}

Setup for address arith

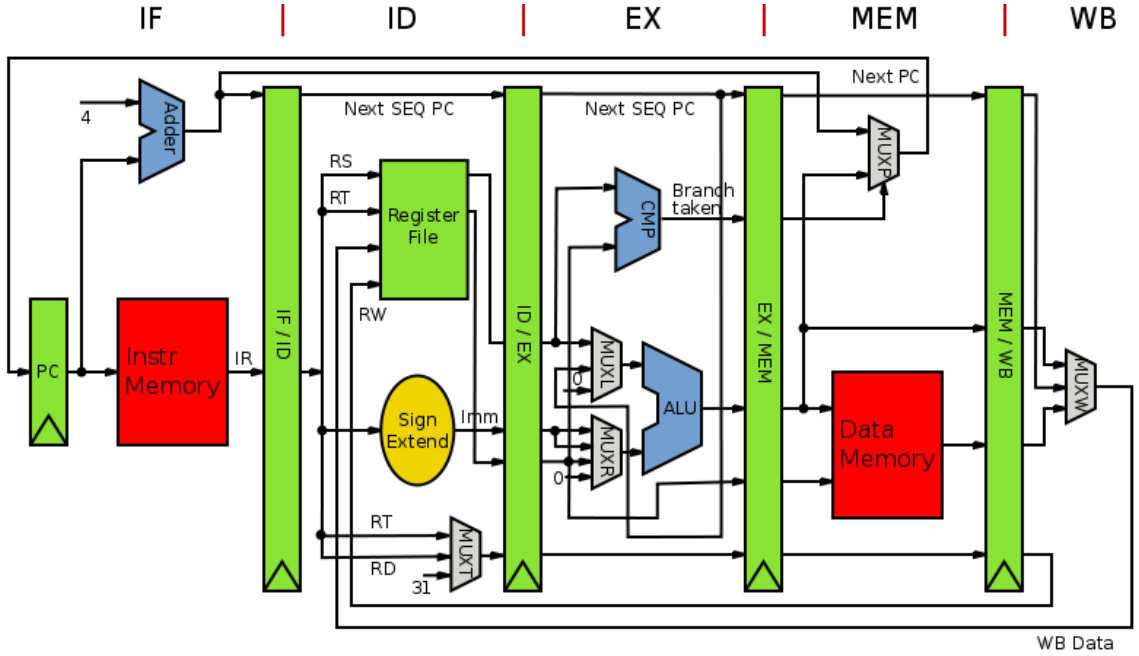
To unconditionally perform: NxPC←0+imm×4

- IDEX[ALUOP]←ADD
- IDEX[xMUXL]←2
- IDEX[xMUXR]←1
- IDEX[CMPTrue]←1

Setup for jal instr

- [!flushF ∧ IFID[IR].op=0x03] {extra for jal instruction}
- IDEX[nxPC]←IFID[nxPC]
- xMUXT←2 {for RW←31}
- IDEX[xMUXW]←1
- IDEX[WB]←1

Instruction decode and register fetch: Jump [and link] register



Jumps w/reg (jr, jalr): PC←rs, [jalr]∧R[31]←PC+4			
0	rs	...	0x08/0x09
31-26	25-21	...	5-0

Opcodes: 0/0x08 (jr), 0/0x09 (jalr)

Hardwired: flushF←CMP.out∨IFID[NOP]

- [!stallF ∧ !flushF ∧ IFID[IR].op=0 ∧ [IFID[IR].funct∈[0x08..0x09]]
  {Stall on rs?}

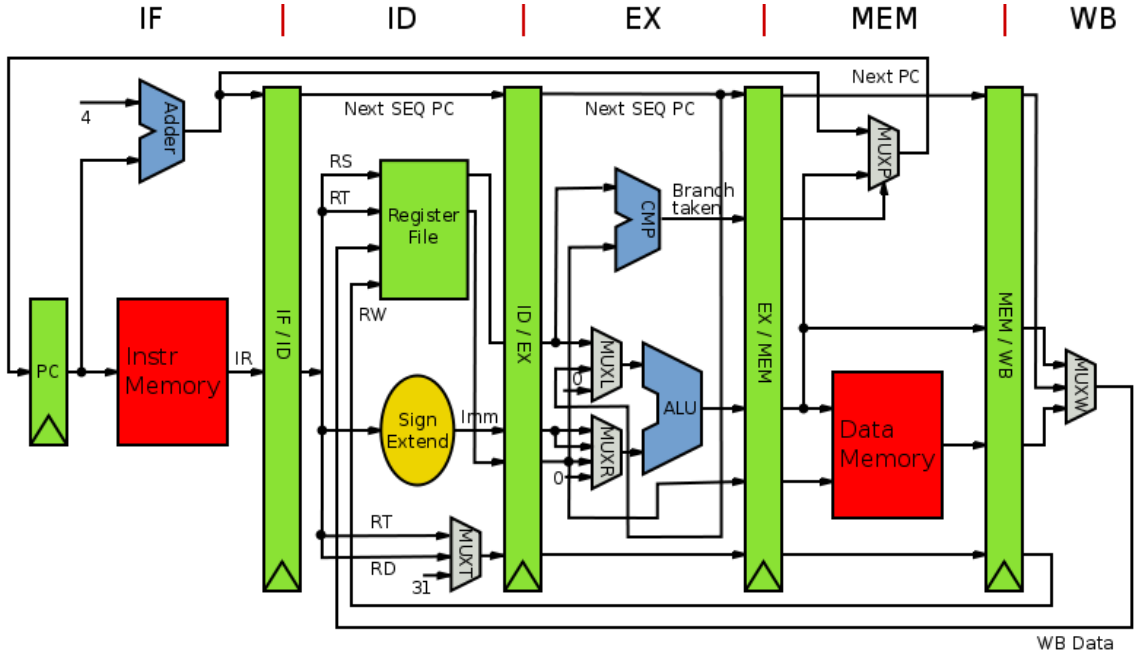
Setup for address arith

- To achieve: NxPC←rs, unconditionally
- IDEX[ALUOP]←ADD
- IDEX[xMUXL]←1
- IDEX[xMUXR]←3
- IDEX[CMPTrue]←1

Setup for jalr instr

- [IFID[IR].funct=0x09] {extra for jalr instruction}
- IDEX[nxPC]←IFID[nxPC]
- xMUXT←2 {for RW←31}
- IDEX[xMUXW]←1
- IDEX[WB]←1

Instruction decode and register fetch: Jump [and link] register



Jumps w/reg (jr, jalr): PC←rs, [jalr]∧R[31]←PC+4			
0	rs	...	0x08/0x09
31-26	25-21	...	5-0

Opcodes: 0/0x08 (jr), 0/0x09 (jalr)

Hardwired: flushF←CMP.out∨IFID[NOP]

- [!stallF ∧ !flushF ∧ IFID[IR].op=0 ∧ [IFID[IR].funct∈[0x08..0x09]]
  {Stall on rs?}

Setup for address arith

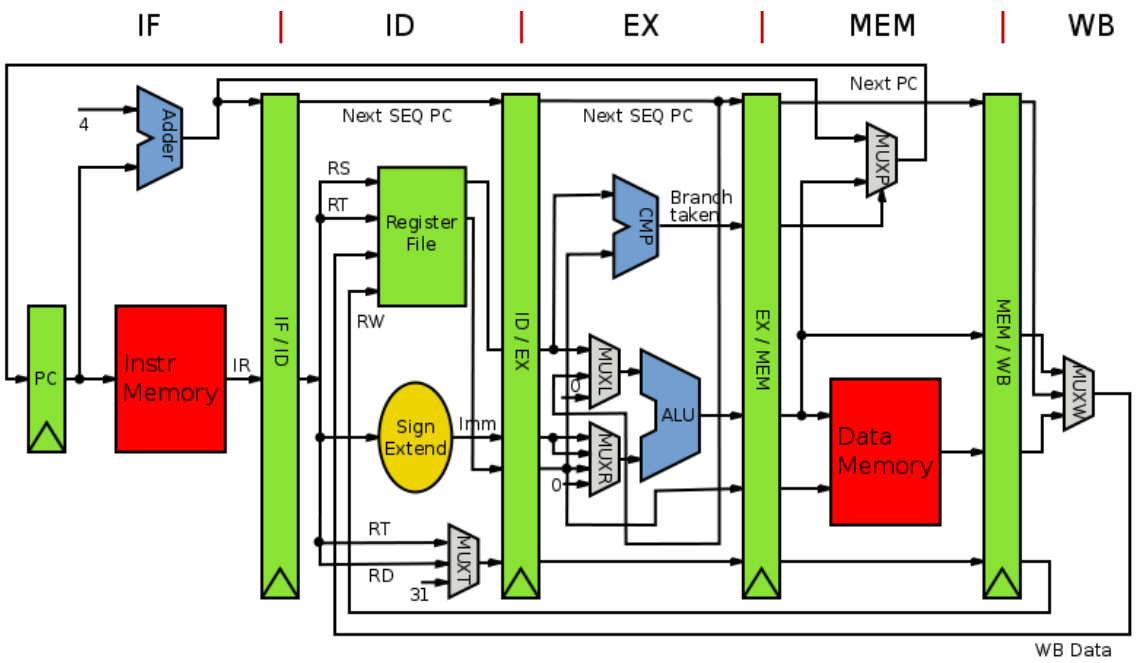
- To achieve: NxPC←rs, unconditionally
- IDEX[ALUOP]←ADD
- IDEX[xMUXL]←1
- IDEX[xMUXR]←3
- IDEX[CMPTrue]←1

- [IFID[IR].funct=0x09]
  {extra for jalr instruction}

Setup for jalr instr

- IDEX[nxPC]←IFID[nxPC]
- xMUXT←2 {for RW←31}
- IDEX[xMUXW]←1
- IDEX[WB]←1

Instruction decode and register fetch: LW-type



LW-type (lw): rt ← M[rs+imm]			
0x23	rs	rt	imm
31-26	25-21	20-16	15-0

Opcode: 0x23

- [!flushF ∧ IFID[IR].op=0x23] {Stall on rs?}

Setup rs, imm, rt (dest)

- IDEX[RSD]←RF[IFID[IR].rs]
- IDEX[IMM]←SignExt (RF[IFID[IR].imm], jxtF) [jxtF←0]
- IDEX[RW]←IFID[IR].rt [xMUXT←0]

ALU setup

- IDEX[ALUOP]←ADD
- IDEX[xMUXL]←0
- IDEX[xMUXR]←0

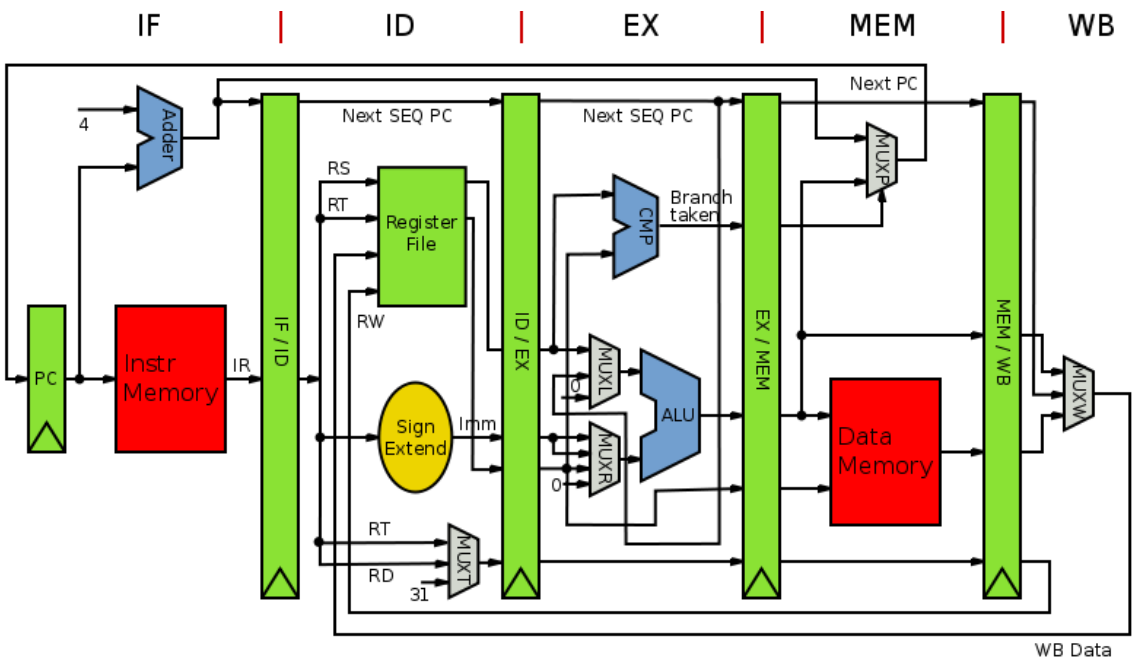
DM read setup

- IDEX[DMRF]←1
- IDEX[DMWF]←0
- IDEX[CMPEn]←0

WB setup

- IDEX[WB]←1
- IDEX[xMUXW]←2

Instruction decode and register fetch: LW-type



LW-type (lw): rt ← M[rs+imm]			
0x23	rs	rt	imm
31-26	25-21	20-16	15-0

Opcode: 0x23

- [!flushF ∧ IFID[IR].op=0x23] {Stall on rs?}

Setup rs, imm, rt (dest)

- IDEX[RSD]←RF[IFID[IR].rs]
- IDEX[IMM]←SignExt (RF[IFID[IR].imm], jxtF) [jxtF←0]
- IDEX[RW]←IFID[IR].rt [xMUXT←0]

ALU setup

- IDEX[ALUOP]←ADD
- IDEX[xMUXL]←0
- IDEX[xMUXR]←0

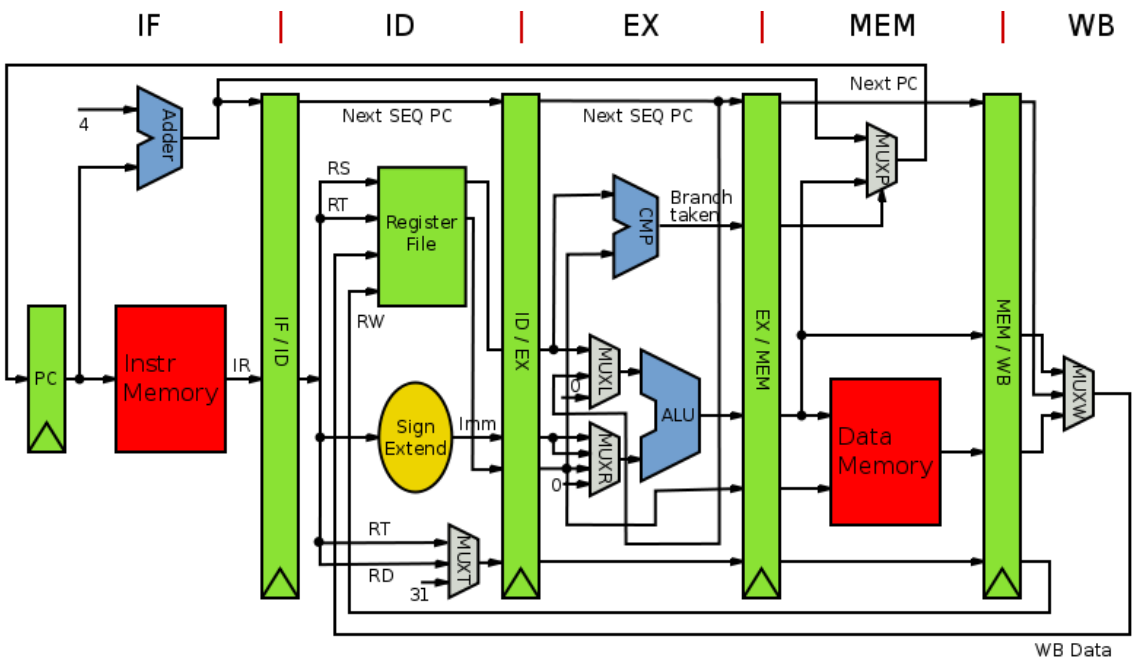
DM read setup

- IDEX[DMRF]←1
- IDEX[DMWF]←0
- IDEX[CMPEn]←0

WB setup

- IDEX[WB]←1
- IDEX[xMUXW]←2

Instruction decode and register fetch: LW-type



LW-type (lw): rt ← M[rs+imm]			
0x23	rs	rt	imm
31-26	25-21	20-16	15-0

Opcode: 0x23

- [!flushF ∧ IFID[IR].op=0x23] {Stall on rs?}

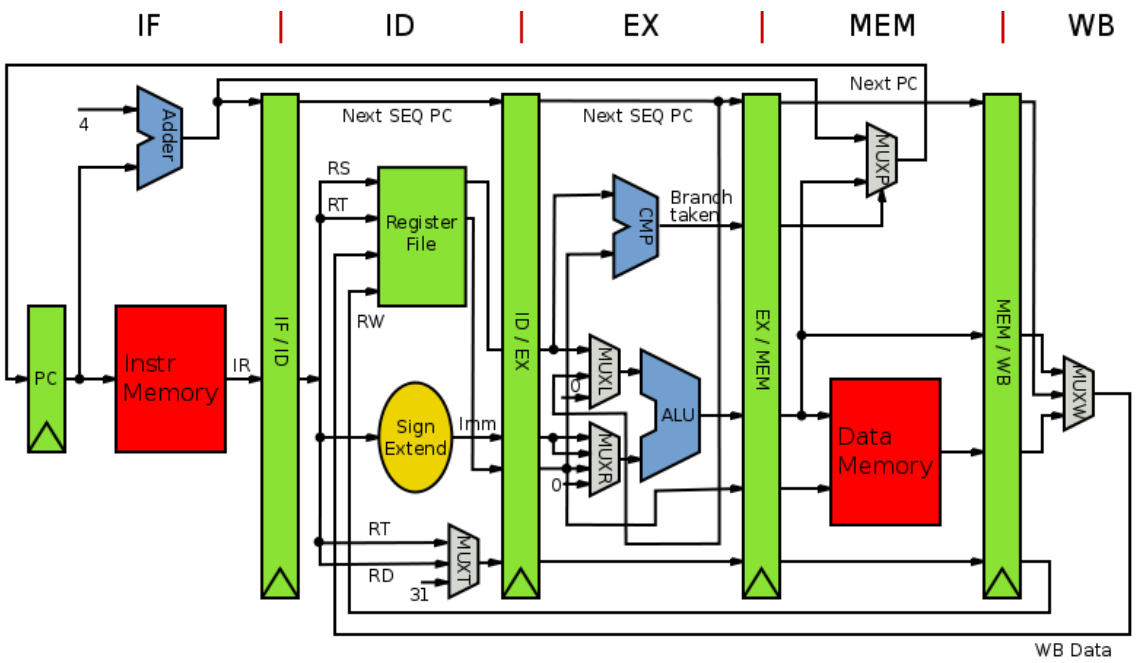
Setup rs, imm, rt (dest)
- IDEX[RSD]←RF[IFID[IR].rs]
- IDEX[IMM]←SignExt (RF[IFID[IR].imm], jxtF) [jxtF←0]
- IDEX[RW]←IFID[IR].rt [xMUXT←0]

ALU setup
- IDEX[ALUOP]←ADD
- IDEX[xMUXL]←0
- IDEX[xMUXR]←0

DM read setup
- IDEX[DMRF]←1
- IDEX[DMWF]←0
- IDEX[CMPEn]←0

WB setup
- IDEX[WB]←1
- IDEX[xMUXW]←2

Instruction decode and register fetch: LW-type



LW-type (lw): rt ← M[rs+imm]			
0x23	rs	rt	imm
31-26	25-21	20-16	15-0

Opcode: 0x23

- [!flushF ∧ IFID[IR].op=0x23] {Stall on rs?}

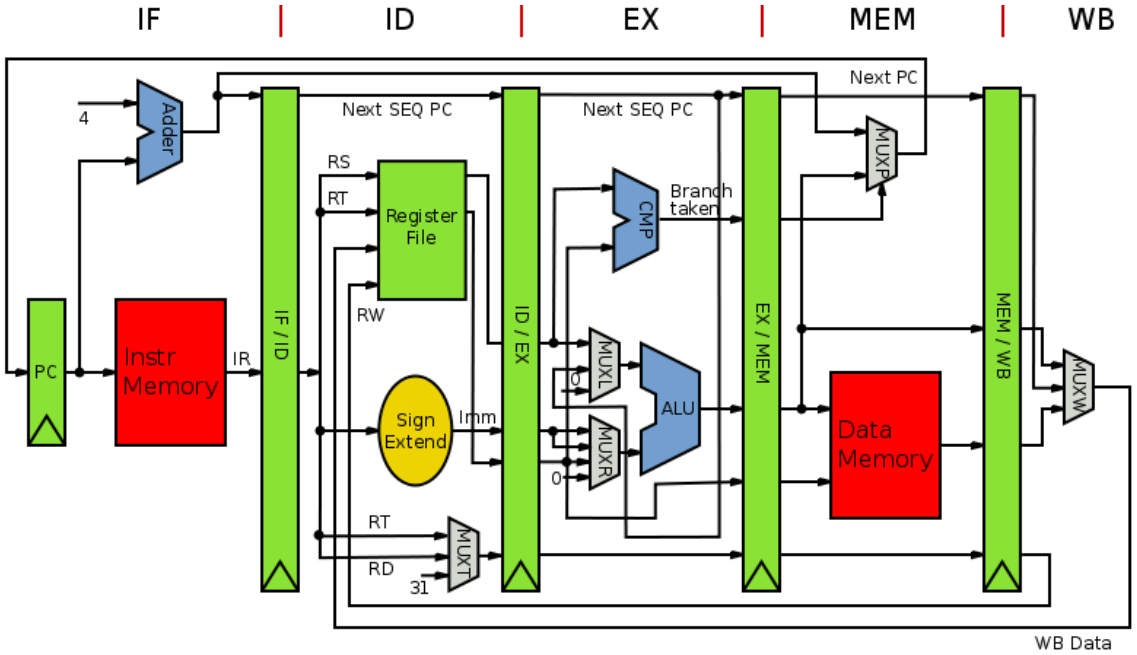
Setup rs, imm, rt (dest)
- IDEX[RSD]←RF[IFID[IR].rs]
- IDEX[IMM]←SignExt (RF[IFID[IR].imm], jxtF) [jxtF←0]
- IDEX[RW]←IFID[IR].rt [xMUXT←0]

ALU setup
- IDEX[ALUOP]←ADD
- IDEX[xMUXL]←0
- IDEX[xMUXR]←0

DM read setup
- IDEX[DMRF]←1
- IDEX[DMWF]←0
- IDEX[CMPEn]←0

WB setup
- IDEX[WB]←1
- IDEX[xMUXW]←2

Instruction decode and register fetch: SW-type



SW-type (sw): M[rs+imm] ← rt			
0x2b	rs	rt	imm
31-26	25-21	20-16	15-0

Opcode: 0x2b

- [!stallF ∧ !flushF ∧ IFID[IR].op=0x2b]

  {Stall on rs or rt?}

Setup rs, imm rt (source)

- IDEX[RTD]←RF[IFID[IR].rt]
- IDEX[RSD]←RF[IFID[IR].rs]
- IDEX[IMM]←SignExt (RF[IFID[IR].imm], jxtF) [jxtF←0]
- IDEX[RW]←IFID[IR].rt [xMUXT←0]

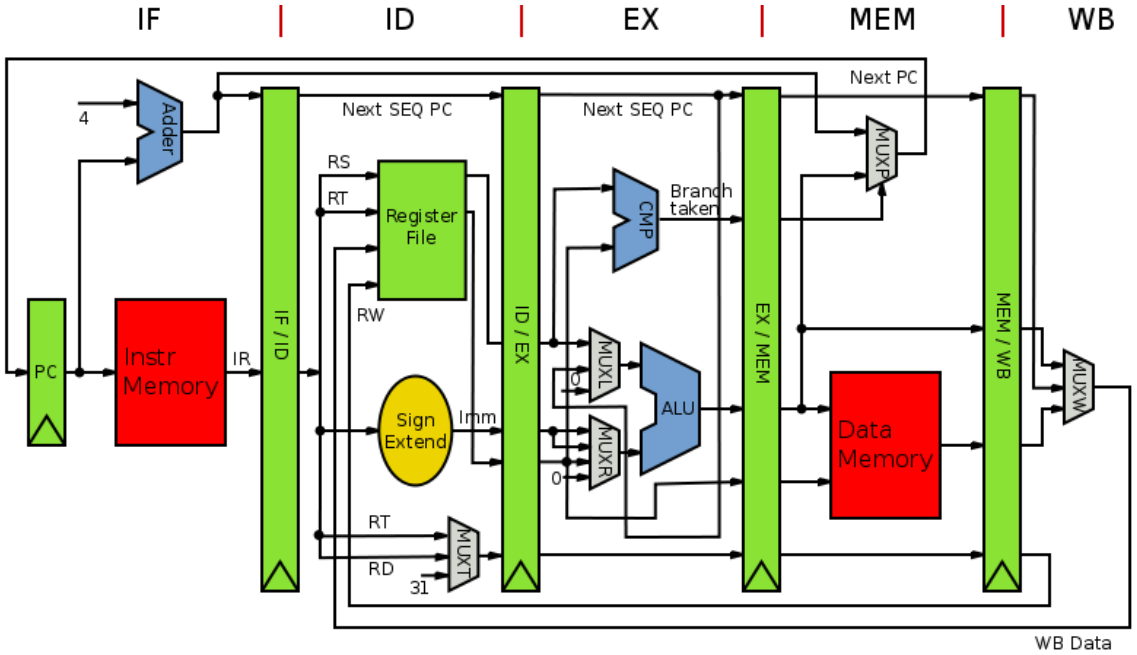
ALU setup

- IDEX[ALUOP]←ADD
- IDEX[xMUXL]←0
- IDEX[xMUXR]←0

DM write setup

- IDEX[DMRF]←0
- IDEX[DMWF]←1
- IDEX[CMPEn]←0

Instruction decode and register fetch: SW-type



SW-type (sw): M[rs+imm] ← rt			
0x2b	rs	rt	imm
31-26	25-21	20-16	15-0

Opcode: 0x2b

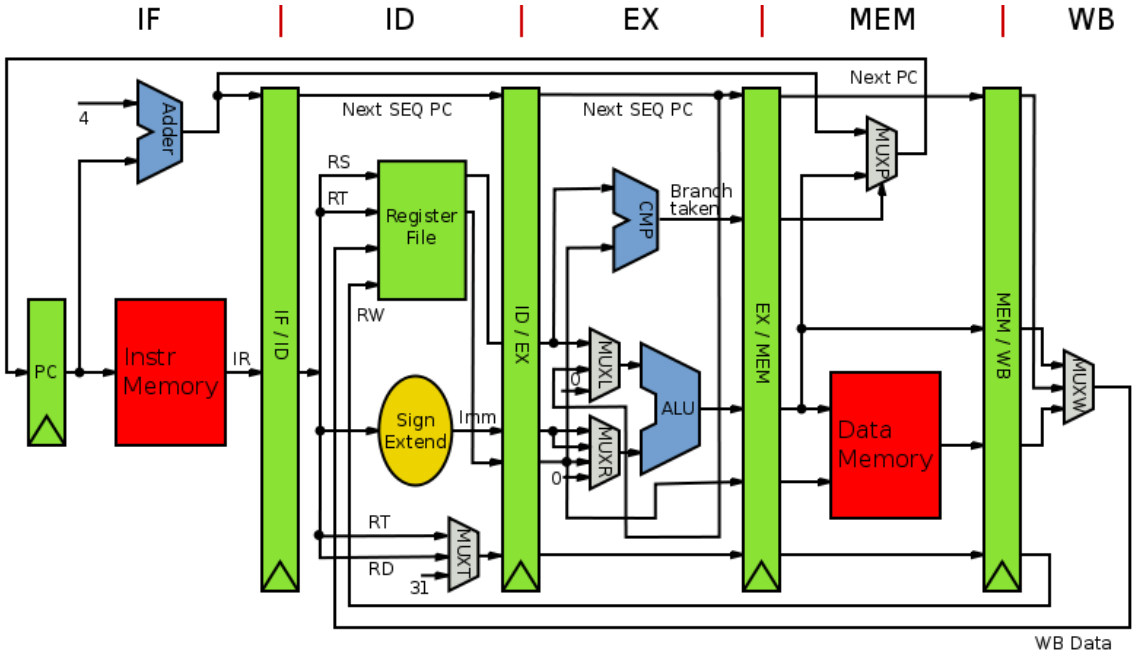
- [!stallF ∧ !flushF ∧ IFID[IR].op=0x2b]
  {Stall on rs or rt?}

Setup rs, imm rt (source)
- IDEX[RTD]←RF[IFID[IR].rt]
- IDEX[RSD]←RF[IFID[IR].rs]
- IDEX[IMM]←SignExt (RF[IFID[IR].imm], jxtF) [jxtF←0]
- IDEX[RW]←IFID[IR].rt [xMUXT←0]

ALU setup
- IDEX[ALUOP]←ADD
- IDEX[xMUXL]←0
- IDEX[xMUXR]←0

DM write setup
- IDEX[DMRF]←0
- IDEX[DMWF]←1
- IDEX[CMPEn]←0

Instruction decode and register fetch: SW-type



SW-type (sw): M[rs+imm] ← rt			
0x2b	rs	rt	imm
31-26	25-21	20-16	15-0

Opcode: 0x2b

- [!stallF ∧ !flushF ∧ IFID[IR].op=0x2b]

  {Stall on rs or rt?}

Setup rs, imm rt (source)

- IDEX[RTD]←RF[IFID[IR].rt]
- IDEX[RSD]←RF[IFID[IR].rs]
- IDEX[IMM]←SignExt (RF[IFID[IR].imm], jxtF) [jxtF←0]
- IDEX[RW]←IFID[IR].rt [xMUXT←0]

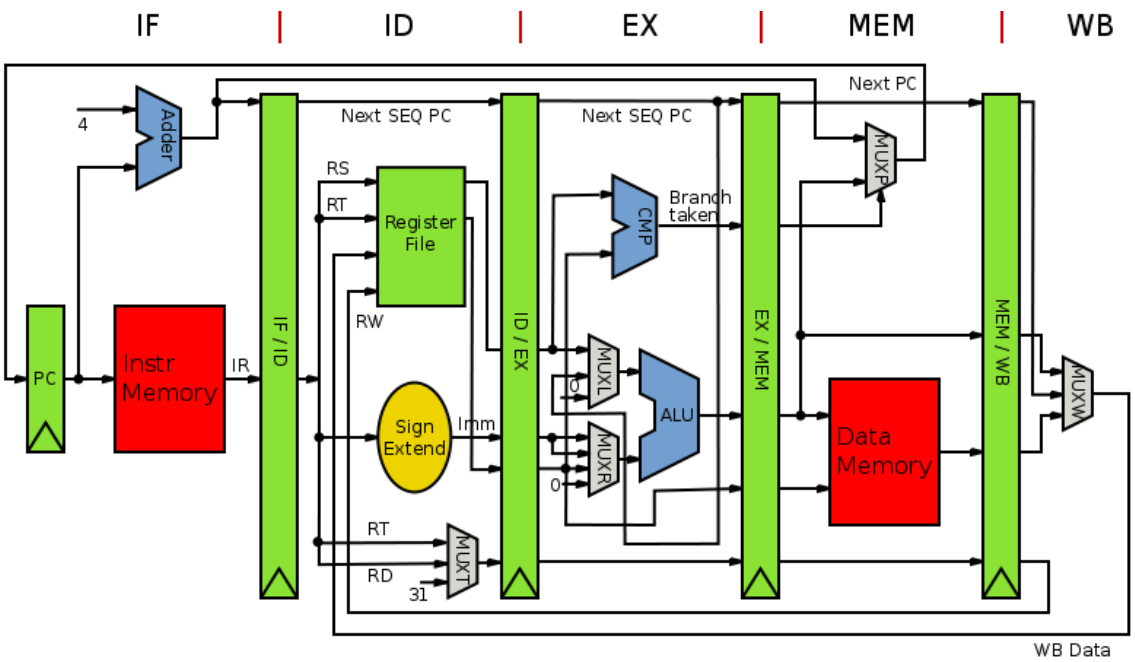
ALU setup

- IDEX[ALUOP]←ADD
- IDEX[xMUXL]←0
- IDEX[xMUXR]←0

DM write setup

- IDEX[DMRF]←0
- IDEX[DMWF]←1
- IDEX[CMPEn]←0

Execute

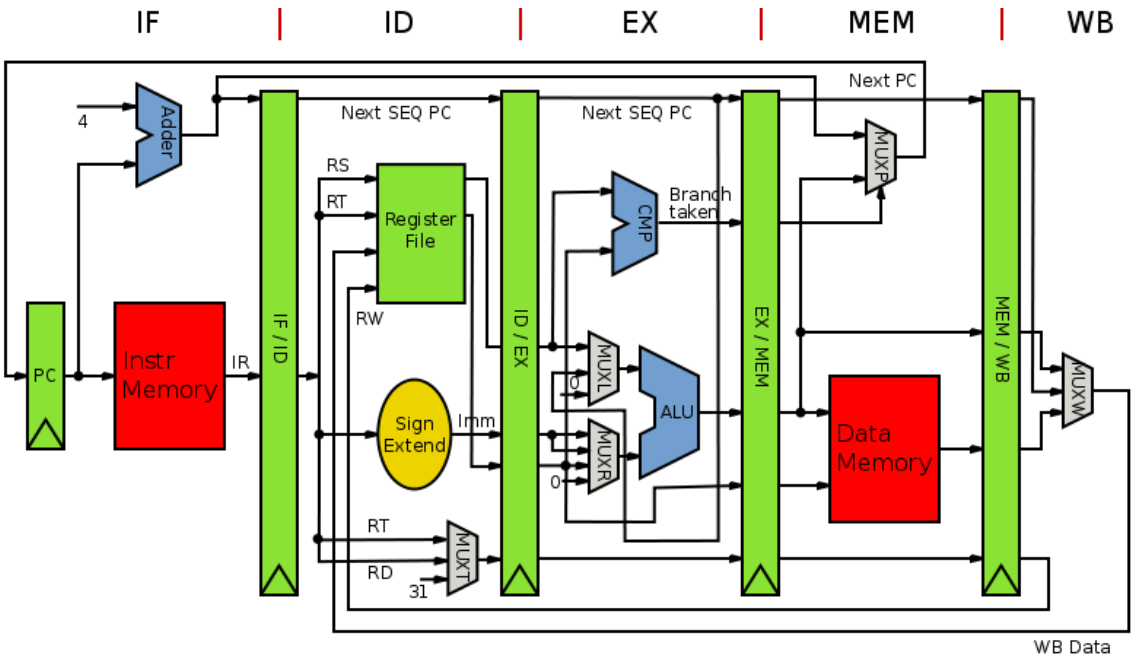


No controller is needed for this stage, as all necessary signals are derived from EXM

Hardwired:

- ALU.left←MUXL.out
- ALU.right←MUXR.out
- ALU.op←IDEX[ALUOP]
- EXM[brT]←CMP.out
- EXM[ALUOut]←ALU.out
- EXM[DMRF]←IDEX[DMRF]
- EXM[DMWF]←IDEX[DMWF]
- EXM[WB]←IDEX[WB]
- EXM[RW]←IDEX[RW]
- EXM[xMUXW]←IDEX[xMUXW]
- CMP.inv←IDEX[CMPInv]
- CMP.en←IDEX[CMPEn]
- MUXL.0←IDEX[RSD]
- MUXL.2←0
- MUXR.0←IDEX.imm
- MUXR.1←4×IDEX.imm
- MUXR.2←IDEX[RTD]

Memory access

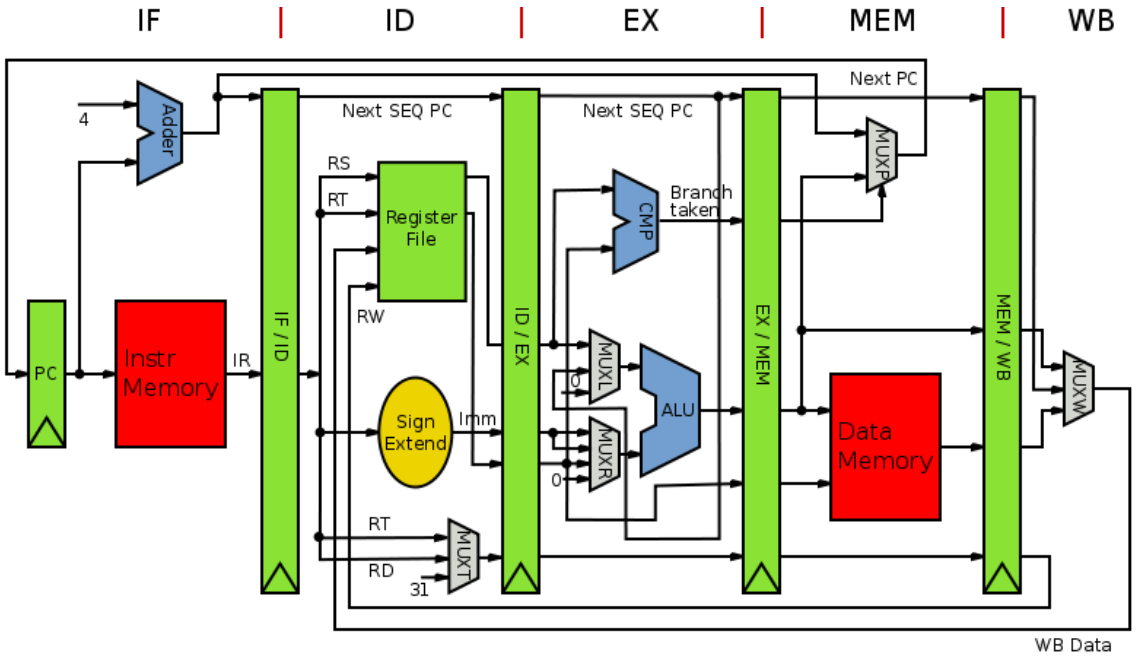


No controller is needed for this stage, as all necessary signals are derived from EXM

Hardwired:

- DM.Addr←EXM[DMAddr]
- DM.WIn←EXM[ALUOut]
- MWB[DMROut]←DM.ROut
- DM.R←EXM[DMRF]
- DM.W←EXM[DMWF]
- MWB[ALUOut]←EXM[ALUOut]
- MUXP.1←EXM[ALUOut]
- MUXP.0←Adder.out {PC+4}
- MUXP.cntl←EXM[brT]
- NxPC←MUXP.out
- MWB[NxPCSeq]←EXM[NxPCSeq]
- MWB[WB]←EXM[WB]
- MWB[RW]←EXM[RW]
- MWB[xMUXW]←EXM[xMUXW]

Write back

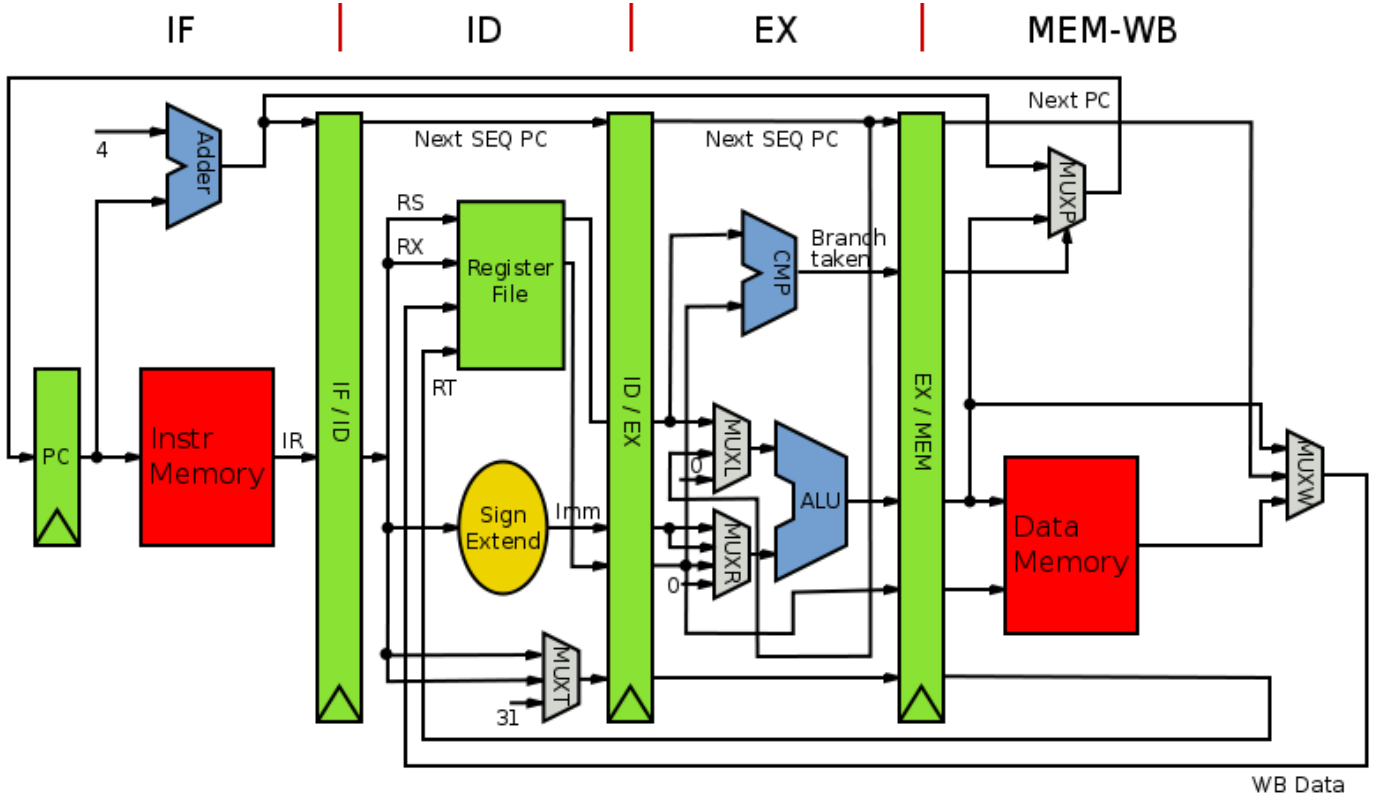


No controller is needed for this stage, as all necessary signals are derived from EXM

Hardwired:

- MUXW.cntl←MWB[xMUXW]
- MUXW.0←MWB[ALUOut]
- MUXW.1←MWB[NxPCSeq]
- MUXW.2←MWB[DMROut]
- RF.RW←MWB[RW]
- RF.RWD←MUXW.out {WB Data}

Contraction to a four stage pipeline



- MEM and WB stages can be merged, dropping the MWB inter pipeline register
- Wires and buses are directly connected
- Unlike before, writing to RF happens late in the clock cycle, opening another possibility of RAW hazard
- Clock cycle for ID will have to be extended to avoid this hazard, leading to lower performance

Pipeline performance

- A k-stage pipelined CPU can ideally process n instructions in $k + n - 1$ cycles
- k cycles are needed to complete the first task
- $n - 1$ cycles are needed to complete the remaining $n - 1$ tasks
- Ideal speedup of a k-stage pipeline over serial execution, assuming non-pipeline CPU takes k cycles per instruction (assuming same fabrication techlology for both CPUs):
  $S_k = \frac{\text{Serial execution in cycles}}{\text{Pipelined execution in cycles}} = \frac{nk}{k+n-1}$, $S_k \rightarrow k$ for large n
- In practice, the speed up is hard to predict on account of stalls and flushes that slow down the pipeline

Pipeline performance considering delays

- If $\tau_i \equiv$ time delay in stage S_i, clock cycle $\tau_k^p = \max(\tau_i)$ is the maximum stage delay of the k-stage pipelined CPU
- The time delay of the non-pipeline CPU (assuming single cycle operation) will be $\tau^u \leq \sum_i \tau_i$ (inter-pipeline stage registers absent)
- $S_k = \frac{\text{Serial execution time}}{\text{Pipelined execution time}} = \left(\frac{n}{k+n-1}\right)\left(\frac{\tau^u}{\tau_k^p}\right)$, ideal sequence of n operations
- $\frac{\tau^u}{\tau_k^p} \leq k$; $S_k \leq \frac{nk}{k+n-1}$; $S_k \rightarrow \frac{\tau^u}{\tau_k^p}$ for large n

Example

Consider a 5-stage pipeline CPU with stage delays as: IF = EX = MEM = 200 ps; RF read = RF write = 150 ps; decoding = 50ps
Decoding is mostly concurrent with RF read, additional delay is neglected.
Clock cycle for single-cycle non-pipelined CPU:
$200 + 150 + 200 + 200 + 150 = 900$ps
Clock cycle for pipelined CPU: $\max(200, 150) = 200$ps
Speedup of pipelined CPU $= \frac{900}{200} = 4.5$