Pipelining a RISC Architecture Stepwise Verification of DLX

Egon Börger

Dipartimento di Informatica, Universita di Pisa
http://www.di.unipi.it/~boerger

For details see Chapter 3.3 (Microprocessor Design Case Study) of:

E. Börger, R. Stärk

Abstract State Machines

A Method for High-Level System Design and Analysis

Springer-Verlag 2003

For update info see AsmBook web page:

http://www.di.unipi.it/AsmBook

The Problem

- Prove the pipelined version of a sequential architecture to be semantically correct
  - i.e. show for every program that its execution under the pipelining discipline is equivalent to its step-by-step (so-called sequential) execution
- Tasks:
  - model the two architectures
  - define the equivalence notion
  - state the assumptions on the environment (compiler, hardware, etc.)
  - prove the correctness

The Case Study

- RISC microprocessor architecture DLX (Hennessy/Patterson 1990,1996)
  - typical instruction set architecture comprising instructions for arithmetical and set operations, jumps, interrupts, memory access
- Pipelining techniques resolving structural, data and control hazards:
  - instruction scheduling, forwarding, new hardware links with additional control logic, stalling
- Design and verification technique:
  - A model for datapath and sequential control of DLX is stepwise refined to its pipelined version which is proven to be correct
    - using incremental design and modular proof techniques (based upon locality principle)

Modeling and Verification Steps

- Ground model DLX^{seq} for sequential DLX providing one-instruction-at-a-time RISC processor view
  - correct wrt control graph definition by Hennessy/Patterson 1996
- Three application domain driven refinement steps (concentrated on control aspects)
  - parallelization to DLX^{par} resolving structural hazards
  - resolving data hazards in DLX^{data}
  - resolving control hazards in DLX^{pipe}
- Each refined model proven to be correct wrt the preceding model
  - using inductions and case distinctions which correspond to the pipelining conflict types and standard methods to solve them

Main Theorem

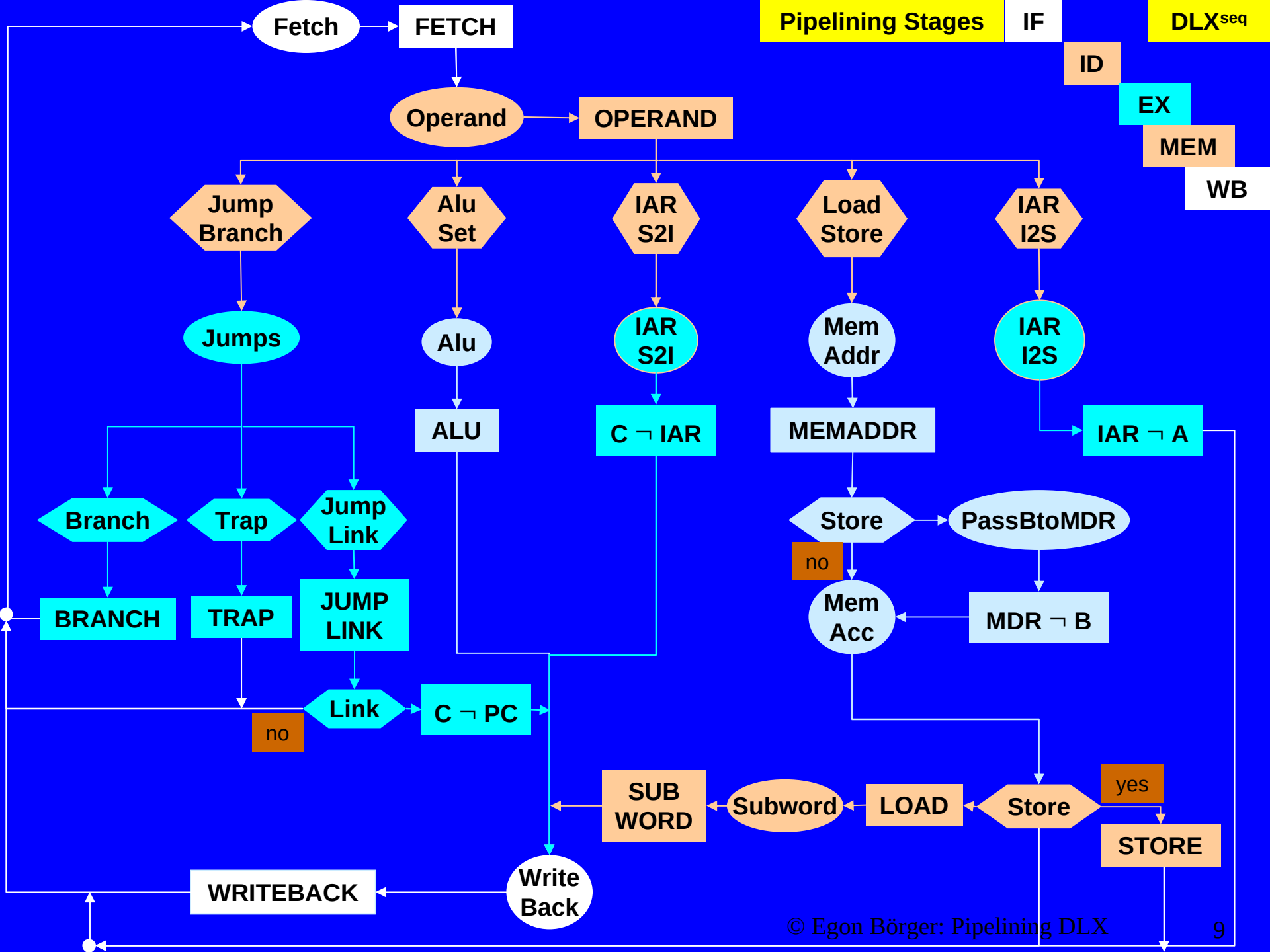
- For each DLX program P, the result of its sequential (one-instruction-at-a-time) execution on DLX^{seq} is the same as the result of its pipelined (up to five instructions at-a-time) execution on DLX^{pipe} .
- Main contribution of the proof:
  - from segments of parallel DLX^{pipe} runs, which perform at each step multiple operations for multiple instructions, extracting for each analyzed instruction an equivalent sequential DLX^{seq} subcomputation, which together form an equivalent DLX^{seq} run
  - defining the appropriate notions of
    - data dependency and control dependency of instructions
    - instruction cycle, result locations, relevant locations (per instruction, supporting instructionwise proofs)

Ground Model DLX^{seq}

- Each instruction is executed by 5 successive steps
  - IF: fetching instr from memory to datapath register IR
  - ID: decoding instr, including extracting its operands from register file registers fstop,scdop into ALU input ports A,B
  - EX: execution proper, computing via ALU
    - ALU operations: value of ALU output port C for given input A,B
    - data memory or branch address:
      - update memory address register MAR using A
      - update memory data register MDR with B
      - update jump address PC
      - update by A or transfer to C interrupt address register IAR
  - MEM: loading from or storing to mem at MAR via MDR
  - WB: writing back C to the destination register
- Phases reflected in ground model by control states

Uniformity of ground model to architectural features

- size of register file using abstract content fct reg
  - width of datapath using abstract registers PC,IR,A,B,C, MAR,MDR,IAR
  - instruction set (in immediate or not immediate form)
    - Alu ∪ Set
    - Jump ∪ Branch Denote JumpLink = Jump - {TRAP}
      - Jump consists of absolute jump instructions of three types
        - » TRAP: system jump (save PC in IAR and update PC)
        - » Link: a set of jump links (save PC in C, to be stored in reg file). The destination register of LINK instructions is assumed to be fixed (in Hennessy/Patterson R31).
        - » a set of other (so-called plain) jumps
    - Mem = Load ∪ Store
    - Interrupt = { MOVS2I, MOVI2S } manipulating IAR
  - instruction format: using abstract parameter functions opcode, fstop, scdop, dest, iop, ival (arg IR suppressed)
  - memory access (bandwith) using abstract function mem



Ground Model DLX^{seq} Macros

- IF: FETCH = IR ← mem(PC), PC ← next(PC)
- ID: OPERAND = A ← fstop, B ← scdop
  - JumpBranch = opcode ∈ Jump ∪ Branch
  - AluSet = opcode ∈ Alu ∪ Set
  - LoadStore = opcode ∈ Load ∪ Store
  - IARS2I/IARI2S = opcode ∈ { MOVS2I, MOVI2S }
- EX: ALU = if iop(opcode) then C ←opcode(A,ival)
  else C ←opcode(A,B)
  - MEMADDR = MAR ← A + ival
  - BRANCH = if reg(A) = 0 then PC ← PC + ival
    - TRAP = IAR ← PC, PC ← ival
    - JUMPLINK = if iop(opcode) then PC ← PC + ival
      else PC ← A
- MEM: LOAD = MDR ← mem(MAR)
  - STORE = mem(MAR) ← MDR
  - SUBWORD = C ←opcode(MDR)
- WB: WRITEBACK = dest ← C

R ← **S** ° reg(R): = reg(S)
standard instruction argument **IR** and **reg** notationally suppressed

Refining DLX^{seq} to DLX^{par}

- control states become guards of instruction phases in rules
- latches for longer living values: IR1,IR2,IR3,PC1,C1 with preservation rules
- resolving structural conflicts by doubling resources
  - code memory mem_{inst} increasing mem bandwith for simultaneous access
  - incrementing PC with function next, avoiding use of ALU
  - separating load/store accesses by splitting MDR into LMDR, SMDR
- Speeding up pipe stages by incorporating SUBWORD into WB

IF, ID macros	DLX^{seq}	DLX^{par}	Pre serve
FETCH	**IR ¬ mem(PC)** **PC ¬ next(PC)**	**IR ¬ mem_{instr}(PC)** **If not jumps then PC ¬ next(PC)**	
OPERAND	**A ¬ fstop** **B ¬ scdop**	**A ¬ fstop** **B ¬ scdop**	

IR1 ¬ IR
PC1 ¬ $next^2$(PC)

jumps = opcode(IR1) ∈ Jumps or (opcode(IR1) ∈ Branch, reg(A)=0)

Update of PC1 for instruction scheduling reasons connected to jump instructions

Refining EX stage rules

ALU	If iop(opcode) then C←opcode(A,ival) else C←opcode(A,B)	If opcode(IR1) ∈ Alu ∪ Set then If iop(opcode(IR1)) then C ← opcode(IR1)(A,ival(IR1)) else C ← opcode(IR1)(A,B)	IR2 ← IR1
MEMADDR	MAR ← A + ival	If opcode(IR1)∈ Load ∪ Store then MAR ← A + ival(IR1)	
PassBto MDR	MDR ← B	If opcode(IR1) ∈ Store then SMDR ← B	
MOVS2I	C ← IAR	If opcode(IR1) = MOVS2I then C ← IAR	
MOVI2S	IAR ← A	If opcode(IR1) = MOVI2S then IAR ← A	
BRANCH	If reg(A)=0 then PC ← PC + ival	If opcode(IR1) ∈ Branch then If reg(A)=0 then PC ← PC1 + ival(IR1)	
TRAP	IAR ← PC PC ← ival	If opcode(IR1) = Trap then IAR←PC1, PC← ival(IR1)	
JUMPLINK	If iop(opcode) then PC ← PC + ival else PC ← A	If opcode(IR1) ∈ JumpLink then If iop(opcode(IR1)) then PC ←PC1+ival(IR1) else PC← A	
LINK	C ← PC	If opcode(IR1) ∈ Link then C←PC1	

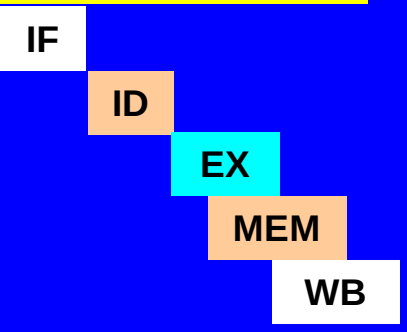
Refining DLX^{seq} MEM, WB stage rules to DLX^{par}

MEM, WB macros	DLX^{seq}	DLX^{par}	Preserve
LOAD	MDR ← mem(MAR)	If opcode(IR2) ∈ Load then LMDR←mem(MAR)	IR3← IR2 C1←C
STORE	mem(MAR) ← MDR	If opcode(IR2) ∈ Store then mem(MAR) ←SMDR	
SUBWORD	C ← opcode(MDR)		
WRITE BACK	dest ← C	If opcode(IR3)∈Alu ∪ Set ∪ Link ∪ {MOVS2I} then dest(IR3) ←C1 If opcode(IR3)∈Load then dest(IR3) ← opcode(IR3)(LMDR)	

Speeding up pipe stages combining SUBWORD with WRITEBACK

separate overlapping load/store accesses splitting MDR into LMDR, SMDR

Pipelining Stages



DLX^{par} Rules

FETCH

- Ideally all rules fire simultaneously
  - one per instruction in its successive stages

OPERAND Preserve

- The problem: how to guarantee that no conflicts arise when an instruction exec uses data which have to be computed by a preceding instruction whose pipelined execution is not yet terminated

ALU MEMADDR PassBtoMDR MOVI2S MOVS2I

BRANCH TRAP JUMP LINK LINK Preserve1

LOAD STORE Preserve2

WRITEBACK

DLX^{par} Correctness Theorem

- Theorem: For each DLX program P, let P^{par} be its transformation obtained by inserting two empty instructions after each occurrence of a jump or branch instruction ("instruction scheduling"). Let C be the DLX^{seq} computation with input P and C^{par} the corresponding DLX^{par} computation with input P^{par}. Then C and C^{par} have the same result if C^{par} is data-hazard-free.
  - Tasks to be accomplished:
    - define "correspondence" of computation
    - define " computation result" and "sameness" for them
    - define "data-hazard" and data-hazard-freeness
    - prove the statement
    - justify the assumption on transforming P to P^{par}
  - Instruction scheduling serves here to separate semantics preserving parallelization from hazard prevention

Defining corresponding instruction cycles

- instruction cycle for instr
  - in DLX^{seq} any subcomputation starting with control state FETCH with loading instr from the code memory until the next occurrence of control state FETCH
  - in DLX^{seq} any subcomputation starting with instr and ending with the first pipe stage of instr at the end of which the values of all result locations of instr are computed and recorded in the appropriate location as defined below
    - this pipe stage is denoted EX(instr), MEM(instr), WB(instr) depending on instr (see table below)
- corresponding instruction cycles in C and C^{par}
  - ever pair (I_i, I_i') of instruction cycles $I_0, I_1, \ldots$ in C and $I_0', I_1', \ldots$ in C^{par} where I_0, I'_0 are corresponding initial states (with corresponding initialization)

Defining result locations and result of an instr

- In DLX every instr has only one result proper (different from <reg,PC>). It is written at the end of the execution of instr.

Result Location of instr	Updated by instr of type	To be collected after the end of pipe stage
<reg,dest(instr)>	Alu ∪Set ∪ Load ∪ Link ∪ { MOVS2I}	WB(instr)
<reg,IAR>	{ TRAP, MOVI2S}	EX(instr)
<reg,PC>	Jump ∪ Branch	EX(instr)
	∉ Jump ∪ Branch	IF(instr)
<mem,arg>	Store	MEM(instr)

result of instr (or instr cycle) = values f(a) assigned to the result locations <f,a> for instr at the end of the execution of instr

result of a computation = sequence of results of its instruction cycles

where arg = value of reg(fstop(instr))+ival(instr) when fetching instr

Defining used locations

Location	Used by instr of type	Critical use in stage
<reg,nthop>	Alu È Set È {MOVI2S} ∪ Branch È ∪ Jump - { TRAP}	EX(instr)
	Load È Store	MEM(instr)
<reg,IAR>	{ MOVS2I}	EX(instr)
<reg,PC>	Jump È Branch	EX(instr)
	Ï Jump È Branch	IF(instr)
<mem,arg>	Load	MEM(instr)

what instr uses:
static info encoded in instr
PC
operands in register file
mem locations
interrupt address reg

where arg = value of reg(fstop(instr))+ival(instr)
when fetching instr

Defining data hazards

- conflicts can arise in two cases:
  - data hazard: I' uses as one of its operands the content of the destination register of a preceding I in the pipe
  - control hazard: I' enters the pipe shortly after a jump or branch instruction
- $I <_{1,2,3} I'$ iff instr cycle for I' starts 1, 2, or 3 steps after an instr cycle for I
- I' data dependent on I iff $I <_{1,2,3} I'$ and one of the following two cases holds:
  - $dest(I) \in \{fstop(I'), scdop(I')\}$ and $I' \notin Jump \cup Branch$
  - $dest(I) = fstop(I')$ and $I' \in Jump \cup Branch$
- DLX^{par} computation data-hazard-free if it contains no occurrence of an I' which is in the pipe together with an occurrence of an I on which it is data dependent

Instruction Scheduling

- Goal: separate analysis of control hazards from correctness proof for parallelization:
  - Problem: when a jump or branch instr is fetched, the two immediately following instruction cycles generate results which could spoil the continuation of the computation after the jump
- Solution: assume for the correctness proof that the compiler produces P^{par} from P placing two empty instructions after each jump or branch instruction.
  - We define empty instructions NOT to start an instruction cycle.
  - We assume these new instructions to be linked by the extension of the next function of P in the obvious way, as formalized by the PC-preservation rule PC1 $\leftarrow$ next(next(PC)).
  - Assume the compiler to guarantee hazard-freeness
    - Both assumptions will be discharged in separate refinement steps

Proof of DLX^{par} Correctness Theorem

- To establish: C and C^{par} have the same result (assuming C^{par} is data-hazard-free)
- Proof by in induction on the given runs, decomposing computations in instruction cycles so that the proof can be done instructionwise with 'local' arguments. This needs a stronger induction hypothesis:

- DLX^{par} Lemma. Let IC_n,IC^{par}_n be the n-th instruction cycle in C, C^{par}. The following holds:
- a) Completeness: If C^{par} is data-hazard-free, then IC_n,IC^{par}_n are instruction cycles for the same instr and start with the same values for the "relevant locations" used by instr
- b) Correctness: Any instr cycle pairs as in a) for any instr which is not data-dependent on any instruction in the pipe compute the same result
  - The lemma implies the theorem.

Defining relevant locations for DLX^{par}

- A location l used by instr is a relevant location except in the following two cases:
  - Irrelev 1. l = <reg,IAR> and instr = MOVS2I enters the pipe 1, 2 or 3 stages after an occurrence of MOVI2S or of TRAP
    - NB. No conflict can arise from using <reg,IAR> because MOVS2I. the only instruction which uses IAR, can never be in conflict with any preceding instruction. If l writes into IAR, then I=TRAP or I=MOVI2S and l writes into IAR in its third pipe stage. Therefore if $I <_{1,2,3} I'$, then I has already written into IAR when I' uses it.
  - Irrelev 2. l = <mem,arg> and instr is a Load instruction which enters the pipe 1, 2, or 3 steps after an occurrence of a Store instruction for the same arg (see justification below)

Rational for Irrelev 2 for DLX^{par}

- Irrelev 2. I = <mem,arg> and instr is a Load instruction which enters the pipe 1, 2, or 3 steps after an occurrence of a Store instruction for the same arg
  - No conflict can arise from using a mem location because load instructions---the only ones which use memory locations---can never be in conflict with preceding store instructions---the only ones which write into mem locations.
    - If I $\in$ Store and I' $\in$ Load, then I updates its result location < mem, reg (fstop(I)) + ival (I)> in its fourth pipe stage and I' reads the value of the location < mem, reg (fstop (I)) + ival (I)> in its fourth pipe stage too. Therefore if I $<_{1,2,3}$ I' and I' loads the value of the result location of I as updated by I, then I has already updated this result location when I' loads from there.
    - NB. DLX does not support self-modifying code

Proof of DLX^{par} Lemma (a)

- ad completeness: IC_n,IC^{par}_n are instruction cycles for the same instr and start with the same values for the "relevant locations" used by instr
  - n=0: by correspondence of runs and initialization
  - n+1: by inductive hypothesis, for each $i \leq n$, $IC^{par,}$ starts with the same values for the relevant locations as does IC_i and they both compute the same result. Therefore IC_{n+1},IC^{par}_{n+1} are instruction cycles for the same instruction, say instr, and start with the same values for the relevant locations to be used by that instruction.

Proof of DLX^{par} Lemma (b)

- ad correctness: any instr cycle pair IC_n,IC^{par}_n as in a) where instr is not data-dependent on any instruction in the pipe compute the same result
  - Due to the absence of stalls, the n+1-th instruction cycle in C^{par} starts
    - after the first step of IC^{par}_n in case $instr_n$ is neither a branch instruction with true branching condition nor a jump
    - otherwise the n+1-th instruction cycle in C^{par} starts after the third step of IC^{par}_n due to the following Jump Lemma
- Jump Lemma. If a jump or branch instruction I is fetched in a DLX^{par}-computation, then the following two fetched instructions are empty and at stage ID(I) the register PC1 is updated by the correct value to be used for the computation of the possible new PC-value in stage EX(I).
  - By a case distinction following the instruction type it then follows from the definition of the rule macros that corresponding updates in C and C^{par} provide the same values for corresponding result locations.

Proof of Jump Lemma

- Jump Lemma. If a jump or branch instruction I is fetched in a DLX^{par}-computation, then the following two fetched instructions are empty and at stage ID(I) the register PC1 is updated by the correct value to be used for the computation of the possible new PC-value in stage EX(I).
- Proof: follows by an induction on the number of jump instructions which are fetched during a run, using the transformation of P to P^{par}.

Resolving data hazards for linear code execution

- Goal: refine DLX^{par} to DLX^{data} where data hazards are resolved for non-jump/branch instructions so that the compiler assumption of hazard-freeness can be restricted to jump/branch instructions.
  - I.e. we have to define DLX^{data} and to prove the following:
- Theorem. Let C, C^{data} be the computations of DLX^{seq} , DLX^{data} started with program P, P^{par} . Assume that in C^{data} no occurrence of a jump or branch instruction is in the pipe together with an occurrence of an instruction on which it is data dependent. Then C and C^{data} compute the same result.

Data hazards for linear code execution

- Data dependence condition DataDep(I,I'):
  - I' $\notin$ Jump $\cup$Branch and
  - I $<_{1,2,3}$ I' and dest(I) $\in$ {fstop(I'), scdop(I') }
  - NB. no write-after-write hazard can occur in DLX
    - writing only in stage WB, but together with any stalled instruction every later instruction in the pipe is also stalled
  - NB. no write-after-read hazard can occur in DLX
    - read stage ID precedes the write stage WB
- Hazard Resolution Method: standard techniques for hazard resolution incorporated into DLX^{par} macros
  - forwarding via
    - new dataflow connections (hardware links)
    - additional decoder logic (multiplexing)
  - stalling

Conservative refinement of DLX^{par} macros to resolve data hazards for linear code execution

- Idea:
  - refine macros only for case DataDep(I,I'):
    - if DataDep(I,I'), then the DLX^{data} macro refinement guarantees that correct arguments are taken to compute the result of instr, namely during EX(instr)-phase
    - otherwise the DLX^{par} macro is executed
  - weakening the compiler hazard-freeness assumption
    - by refining the notion of relevant locations, taking out the hazardous locations since they are taken care of by the refined architecture
- NB. This idea realizes an abstract form of the typical implementation using a scoreboard to keep track of data dependencies

Proof of DLX^{data} Correctness Theorem

  - Proof follows from the following lemma:

- DLX^{data} Lemma. Let $IC_n, IC^{par}{}_n, IC_n{}^{data}$ be the n-th instruction cycle in C, C^{par}, C^{data}. Then:
- Completeness: If C^{data} is free of data hazards for jump/branch instructions, then $IC_n, IC^{par}{}_n, IC_n{}^{data}$ are instruction cycles for the same instruction I' and start with the same values for the relevant locations used by I'.
- For any I'-cycles IC, IC^{par}, IC^{data} as in a):
  - Conservativity: if I' is not data-dependent on any I in the pipe, then IC^{par}, IC^{data} compute the same result
  - Refinement: if $I' \notin$ Jump $\cup$Branch is data-dependent on an $I <_{1,2,3} I'$ in the pipe, then IC, IC^{data} compute the same result

Proof of DLX^{data} Lemma (1)

- By induction on number n of instr cycles:
- ad completeness: IC_n, IC_n^{data} are cycles for the same instr and start with the same values of relevant locs
  - n=0: by correspondence of computations
  - n+1: follows from Jump Lemma as for DLX^{par}
- ad conservativity: to show: if I' is not data-dependent on any I in the pipe, then IC^{par}, IC^{data} compute the same result
  - in case of no data dependence, in every pipe stage rule applied to I', the DLX^{par}-macro is executed. Since by assumption IC^{par} and IC^{data} start with the same values for the relevant locations used by I', the effect of the rule applications to I' is the same in IC^{par} and IC^{data}, including the result locations of I'. Then by the DLX^{par} lemma the result loc values of I' are the same in IC and IC^{data}

Proof of DLX^{data} Lemma (2)

- ad Refinement: Let I'$\notin$ Jump $\cup$ Branch be data-dependent on an I $<_{1,2,3}$ I' in the pipe. To show: IC,IC^{data} compute the same I'-result.
- By definition of dest and assumption on I':
  - I $\in$ Alu$\cup$Set $\cup$Load $\cup$ Link $\cup${ MOVS2I}
  - I' $\in$ Alu$\cup$Set $\cup$Load $\cup$Store $\cup${ MOVI2S} since I' $\notin$ {MOVS2I} $\cup$ Jump $\cup$Branch

function	defined for instr
dest(instr)	Î Alu ÈSet È Load È Link È {MOVS2I}
fstop(instr)	Î Alu ÈSet È Mem È JumpLink È Branch È{ MOVI2S}
sndop(instr)	Î Alu ÈSet ÈStore

Proof of DLX^{data} Lemma: **Case I Ï Mem**

- Then
  - I ∈ Alu∪Set ∪Link ∪{ MOVS2I}, dest(I) is updated to the value used by I' in WB(I), namely to the value of C1, which is copied in MEM(I) from C as
    - computed in EX(I) by an ALU ∪ SET operation
    - or as content of PC1 (Link) or IAR (MOVS2I)
  - Case $I <_{2,3} I'$: ID(I') overlaps with WB(I) or MEM(I) with the expected operand value in C1 or C
  - Case $I <_{1} I'$: expected operand value computed during ID(I') and available not before EX(I')

DLX^{data} Lemma: Case $I \notin Mem$, $I <_{2,3} I'$

- **Case $I <_{2,3} I'$**: Refinement of OPERAND
  - Since ID(I‘) overlaps with WB(I) or MEM(I) with the expected operand value in C1 or C, nthReg=A,B (nth=fst,scd) can be updated with C1 or C (called below C’)
  - if nthop(IR) $\in$ {dest(IR3),dest(IR2)} refinement case

    then nthReg $\leftarrow$ C’

    else nthReg $\leftarrow$ nthop(IR) conservative case

  where C’ = if nthop(IR) = dest(IR3) $\neq$ dest(IR2) then C1

  if nthop(IR) = dest(IR2) then C last update counts

  Irrelev 3. I = <reg,nthop(I’)> where I’$\notin$ Jump $\cup$ Branch for some $I <_{2,3} I'$ such that $I \notin$ Mem, nthop(I’) =dest(I)

  Hw Cost: direct link bw register file exits A,B and C,C1

DLX^{data} Lemma: Case I $\notin$ Mem , I $<_1$ I'

- **Case I $<_1$ I'**: Refining EX-macros by forwarding
  - expected operand value computed during ID(I') is available at the end of EX(I) in C, to be forwarded directly to ALU for EX(I'), shortcutting transfer from C to register file to A,B
- Refine EX-rules for I' $\in$ Alu $\cup$ Set $\cup$Load $\cup$Store $\cup$ { MOVI2S}
  - Method: adding to macros a clause which in the data hazard case provides the argument C instead of A,B
  - Hw Cost: direct link between C and
    - both ALU ports for I' $\in$ Alu $\cup$ Set
    - IAR for I' = MOVI2S
    - MAR and SMDR for I' $\in$ MEM

    together with additional control logic (multiplexers) to select as ALU input the forwarded value rather than the value from the register file

Irrelev 4. <reg,nthop(I')> for possible combinations for I'

DLX^{data} Lemma: Case $I \notin Mem$, $I <_1 I'$: Irrelev locs

- Irrelev 4. <reg,nthop(I')> such that for some $I <_1 I'$ with $I \notin Mem$ one of the following holds:
  - opcode (I') $\in$ Alu $\cup$ Set, iop (opcode (I')), dest (I) = fstop (I'), nth = fst
  - opcode (I') $\in$ Alu $\cup$ Set, not iop (opcode (I')), dest (I) = nthop (I')
  - opcode (I') $\in$ Mem $\cup$ {MOVI2S}, nth = fst, dest (I) = fstop (I')
  - opcode (I') $\in$ Store, nth = scd, dest (I) = scdop (I')

DLX^{data} Lemma: Case $I \notin Mem$, $I <_1 I'$: refined rules

- If opcode(IR1)$\in$ Load $\cup$ Store then MEMADDR
  if fstop(IR1) = dest(IR2) refinement case
  then MAR $\leftarrow$ val_{fst} + ival(IR1)
  else MAR $\leftarrow$ A + ival(IR1) conservative case

where val_{nth} = C if nothop (IR1) = dest(IR2)
= nthReg otherwise with fstReg=A,scdReg=B

- If opcode(IR1)$\in$ Store then PassBtoMDR
  if scdop(IR1) = dest(IR2) refinement case
  then SMDR $\leftarrow$ val_{scd}
  else SMDR $\leftarrow$ B conservative case
- Analogously for ALU and MOVI2S rules

Proof of DLX^{data} Lemma: **Case I Î Mem** and **I' Ï Mem**

- Then I∈ Load, I'∈ Alu∪Set ∪Link∪{ MOVI2S}
  - expected value, loaded by I, available in LMDR at the end of MEM(I)
  - Case $I <_{2,3} I'$: expected operand value available in EX or ID stage in LMDR
    - refining ID-rule OPERAND and EX-rules ALU,MOVI2S
  - Case $I <_{1} I'$: stall pipeline 1 clock cycle, at the latest before EX(I'), to let only MEM and WB stages progress
    - NB. During a pipeline stall a just loaded value in LMDR may be written back; since in the next step it may serve as operand of a stalled instruction, we preserve LMDR

DLX^{data} Lemma: **Case I Î Mem, I' Ï Mem, I $<_{2,3}$ I'**: refined C'

- Case I $<_{2,3}$ I': refining
  - C' in OPERAND by LMDR and
  - $valn_{nth}$ in ALU, MOVI2S by LMDR1
    - using new WB-stage preservation rule LMDR1 ← LMDR
- C' =

LMDR last update in ante-ante-preceding load instr

if nthop(IR) = dest(IR3) ≠ dest(IR2), opcode(IR3) ∈ LOAD

where LMDR = opcode(IR3) (LMDR)

C1 previous case

if nthop(IR) =dest(IR3) ≠ dest(IR2), opcode(IR3) ∉ LOAD

C if nthop(IR) = dest(IR2) previous case

DLX^{data} Lemma: **I Î Mem, I' Ï Mem, I $<_{2,3}$ I'**: refine val_{nth}

- Case I $<_{2,3}$ I': refining
  - $valn_{nth}$ in ALU, MOVI2S by LMDR1
    - new WB-stage preservation rules LMDR1 ← LMDR, IR4 ← IR3 in case an LMDR-value, written back during a pipeline stall, in the next step serves as operand of an instruction stalled after ID
- $valn_{nth}$ =

C if nthop(IR) = dest(IR2) previous case

LMDR if nthop(IR1) = dest(IR3) ≠ dest(IR2), opcode(IR3) ∈ LOAD, opcode(IR4) ∉ LOAD

case of no two successive load instructions

LMDR1 if nthop(IR1) = dest(IR4) ≠ dest(IR2), dest(IR3), opcode(IR3), opcode(IR4) ∈ LOAD

nthReg otherwise DLX^{par} case

DLX^{data} Lemma: **I Î Mem, I' Ï Mem, I $<_{2,3}$ I'**: refined rules

- If opcode(IR1)$\in$ Alu $\cup$ Set then ALU macro

  if iop(opcode) then

  if fstop(IR1) = dest(IR2) or for some m $\in$ {3, 4}

  fstop(IR1) = dest(IRm) and opcode(IRm) $\in$ LOAD

  then C $\leftarrow$ opcode(IR1) (val_{fst} , ival(IR1))

  else C $\leftarrow$ opcode(IR1) (A, ival(IR1))

  else if dest(IR2) $\in$ {fstop(IR1), scdop(IR1) } or

  for some m $\in$ {3, 4} opcode(IRm) $\in$ LOAD and

  dest(IRm) $\in$ {fstop(IR1), scdop(IR1)}

  then C $\leftarrow$ opcode(IR1) (val_{fst} , val_{scd})

  else C $\leftarrow$ opcode(IR1) (A, B)

Analogously for MOVI2S, MEMADDR, PassBtoMDR

DLX^{data} Lemma: **I Î Mem, I' Ï Mem, I $<_{2,3}$ I'**: Irrelev locs

- Irrelev 5. <reg,nthop(I')> with I' $\notin$ Mem $\cup$ Branch $\cup$ Jump such that for some I $\in$ Load with dest (I) = nth op (I'), one of the following holds:
  - I $<_3$ I'
  - I $<_2$ I' , iop (opcode (I')), nth = fst
  - I $<_2$ I' , not iop (opcode (I'))

DLX^{data} Lemma: **Case I Î Mem, I' Ï Mem, I $<_1$ I'**

- Case I $<_1$ I': stop EX(I') together with IF,ID of later instructions for 1 clock cycle, letting only MEM and WB stages progress, to make value loaded by I available to I'
- loadRisk = opcode (IR2) $\in$ LOAD and
  reg (IR1) $\notin$ Mem $\cup$ Jump $\cup$ Branch and
  dest (IR2) $\in$ {fstop (IR1) , scdop (IR1)}
- adding guard not loadRisk to EX,ID,IF stage rules
- adding IF stage rule if loadRisk then IR2 $\leftarrow$ undef
  - whose execution makes loadRisk false so that the full pipelining will be resumed and this case can be handled as the previous one and is resolved by the refined EX-rules
- Irrelev 6. <reg,nthop(I')> for I' $\notin$ Mem$\cup$Branch$\cup$Jump and some I $\in$ Load: I $<_1$ I', dest (I) = nth op (I')

Proof of DLX^{data} Lemma: **Case I,I' Î Mem**

- Case I $<_{2,3}$ I': expected operand value available in EX or ID stage
  - forwarding solution by ID-rule OPERAND and refined EX-rules ALU,MOVI2S
  - hw price: direct links bw LMDR and MAR, SMDR, new LMDR1
- Case I $<_1$ I':
  - Subcase 1. I' uses the loaded val as datum to be stored: dest(I)=scdop(I')
    - forwarding solution
  - Subcase 2. I' uses the loaded val as address for Mem opn: dest(I)=fstop(I')
    - stalling-pipeline solution

DLX^{data} Lemma: **Case I,I' Î Mem,** $I <_{2,3} I'$

- Subcase $I <_3 I'$: resolved by ID-rule OPERAND
- Subcase $I <_2 I'$: resolved by refining MEMADDR and PassBtoMDR as shown above for ALU
- Irrelev 7.
  - <reg,nthop(I')> for $I' \in$ Mem such that for some $I \in$ Load: $I <_3 I'$, dest (I) = nthop (I')
  - <reg,fstop(I')> for $I' \in$ Mem such that for some $I \in$ Load: $I <_2 I'$, dest (I) = fstop (I')
  - <reg,scdop(I')> for $I' \in$ Store such that for some $I \in$ Load: $I <_2 I'$, dest (I) = scdop (I')

DLX^{data} Lemma: **Case I,I' Î Mem,** $I <_1 I'$

- Subcase 1: dest(I)=scdop(I'), i.e.I' uses the loaded val as datum to be stored
  - Then I' ∈ Store, val loaded by I needed in MEM(I') where available in LMDR
    - forwarding solution by refining STORE
    - hw price: link bw LMDR and memory input port
  - Irrelev 8. <reg,scdop(I')> for I' ∈ Store such that for some I∈Load: $I <_1 I'$, dest (I) = scdop (I')
- Subcase 2: dest(I)=fstop(I'), i.e. I' uses the loaded val as address for Mem opn
  - stalling pipeline: refine loadRisk by clause

    or dest(IR2) = fstop(IR1), IR1 ∈ Mem
  - Irrelev 9. <reg,fstop(I')> for I' ∈ Mem such that for some I∈Load: $I <_1 I'$, dest (I) = fstop (I')

Exercises

- Resolve the conflict in case **I** Î **Mem**, **I'** Ï **Mem** above without preserving LMDR.
  - Hint. Anticipate the stalling from the EX phase to the ID phase, recognizing loadRisk already in phase ID. See Hinrichsen 1998.
- Refine DLX^{data} to DLX^{pipe} and prove it to correctly solve also control hazards (those triggered by jump or branch instructions).
  - Hint. See Section 5 in Börger and Mazzanti 1997 (LNCS 1212)

References

- J.L.Hennessy and D.A.Patterson: Computer Architecture: A Quantitative Approach
  - Morgan Kaufmann 1990,1996
- E. Börger and S. Mazzanti: A practical method for rigorously controllable hardware design
  - Springer LNCS 1212 (1997) 151-187
- H. Hinrichsen: Formally correct construction of a pipelined DLX architecture.
  - TR 98-5-1, May 1998, Darmstadt University of Technology, Dept. of Electrical and Computer Engineering
- S.Tahar.Eine Methode zur formalen Verifikation von RISC-Prozessoren.
  - Fortschrittberichte VDI, Reihe 10:Informatik/Kommunikationstechnik Nr. 350, VDI-Verlag, Duesseldorf 1995, pp.XIV+162.
- E. Börger, R. Stärk: Abstract State Machines. A Method for High-Level System Design and Analysis Springer-Verlag 2003, see http://www.di.unipi.it/AsmBook