Techn.Inf. I Zusammenfassung

Andreas Biri, D-ITET 04.01.14

1. Einleitung

Server

Vernetze Systeme, Parallelverarbeitung, Zuverlässigkeit

Embedded Systems

Informationsverarbeitung, Teil d. übergeordnet. Systems
echtzeitfähig, spezialisiert/optimiert, zuverlässig, effizient

Wegen hoher Leistung/Wärme: geringere Taktraten, dafür mehr Prozessorkerne u. Parallelismus
Prozessor/Memory-Gap: brauche komplexere Architektur

Komponenten eines Rechners

Prozessor: Datenpfad (Operationen/ALU) + Steuerung
Speicher: Cache, Haupt/Arbeitsspeicher, Festplatte/CD
⟹ volatil: RAM, Cache ↔ permanent: Flash, Festplatte
Ein-Ausgabe/ IO: Netzwerkzugriff, Maus, Bildschirm

2. Instruktionssatz

IF: Instruction fetch → Instr aus Hauptspeicher holen
ID: Instruction decode→ dekodieren, Operanden holen
EXE: Execute → Ausführen der Instruktion/ ALU
MEM: Memory → Aus dem Memory laden
WB: Write back → Zurückschreiben & nächste Instr

1 Word = 4 Byte = 4 * 8 Bits

MIPS: byte address, sequential word addresses differ by 4

Speicherinstruktionen **lw, sw**

Basisadressierung:

```
lw $t0, 100($s2) # $t0 = Speicher[100+$s2]
sw $t0, 100($s2) # Speicher[100+$s2]=$t0
```

Arithmetische & logische Instruktionen **addi, slti, and**

Direkte Adressierung:

```
slti $t1 ,$s2, 100 # if($s2<100) then $t1=1
else $t1=0
addi $t1, $s2, 100 # $t1=$s2+100
```

Registeradressierung:

```
add $s1, $s2, $s3 # $s1=$s2+$s3
slt $s1 ,$s2, $s3 # if($s2<$s3) then $s1=1
else $s1=0
and $s1, $s2, $s3 # $s1=$s2 & $s3
```

Sprung- & Verzweigungsinstruktionen **j, jr, jal**

ändern Kontrollfluss; Sprung ändert immer, Verzweigung falls…

Direkte Adressierung (Sprung) :

```
j target # PC=4*target
j 2500 # PC=10000
```

*Pseudodirekte Adressierung (j, jal) -> **zu Marke***

```
j Label1 # go to Label1
jal Label2 # $ra=PC+4 ; go to Label2
```

```
beq $s1, $s2, Label3 # if($s1==$s2) then go
to Label3
bne $s1, $s2, Label4 # if($s1!=$s2) then go
to Label4
```

Registeradressierung (jr, jalr) :

```
jr $ra # set PC=$ra (continue with
instruction in Memory[$ra])
```

PC- relative Adressierung (beq, bne):

```
beq $s1, $s2, imm # if($s1==$s2) then
PC=PC+4*(imm+1)
beq $s1, $s2, 25 # if($s1==$s2) then
PC=PC+4+100
```

Adressierungsarten

Direkte Adressierung: Konstante wird übergeben (Imm.)

Registeradressierung: Adresse aus Register übergeben

Basisadressierung: Registerinhalt + Konstante

PC-relative Adressierung: PC + 4 + Konstante

Pseudodirekte Adressierung: über Label (PC + Konst.)

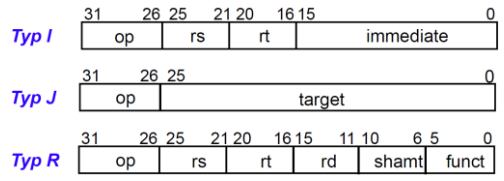
ACHTUNG: PC wird immer 4-fach erhöht !

Register

Name	Register number	Usage	Preserved on call
$zero	0	Constant value 0	n.a.
$v0-$v1	2-3	Values for results and expression evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved	Yes
$t9-$t9	24-25	More temporaries	No
$gp	28	Global pointer	Yes
$sp	29	Stack pointer	Yes
$fp	20	Frame pointer	Yes
$ra	31	Return address	Yes

Instruktionskodierung

MIPS: 32-Bit Kodierungen, 3 Typen



I	immediate (direkt)
J	jump (Sprung)
R	register (Register)
op	6 Bit Kodierung der Operation
rs	5 Bit Kodierung eines Quellenregister
rt	5 Bit Kodierung eines Quellenregisters oder Zielregisters
immediate	16 Bit direkter Wert oder Adressverschiebung
target	26 Bit Sprungadresse
rd	5 Bit Kodierung des Zielregisters
shamt	5 Bit Grösse einer Verschiebung
funct	6 Bit Kodierung der Funktion (Ergänzung des Feldes op)

„Big-endian“ : höstwertiges Byte an niedrigster Adresse
Wort mit höstwertigem Byte adressiert

Zweierkomplement: *signed* oder *unsigned*

$$B = -b_{n-1} * 2^{n-1} + \sum_{i=0}^{n-2} b_i * 2^i$$

Vorzeichenbit erweitern!
Negieren: Invertieren und 1 addieren

Synchronisation

mehrere Programme auf selben Speicher
⟹ *lock :* lock == 1 -> besetzt, sonst frei

1. atomarer Austausch (slt, 1 Takt)
2. Paar spezieller Instruktionen

```
ll $t1, offset($s1) # load linked
sc $t0, offset($s1) # store conditional
```

Assembler

Assemblerprogramm enthält: Kommentare, symbolische Operationscodes, symbol. Registernamen, symbolische Marken (für Jump), Makros (ersetz häufigen Code)

Pseudoinstruktion: wird vom Assembler weiter übersetzt

Latenzen: Operationen brauchen teils mehr als 1 Takt

- Ladeoperationen (Resultat erst nach 2 Takten)
- Sprung- & Verzweigungsinstruktionen
  -> **Branch-Delay-Slot** (nächstes immer ausgeführt)

3. Assembler

.data :	Eintrag wird im Datensegment gespeichert
.align n :	Daten im Speicher auf 2^n Bytegrenzen ausgerichtet
.align 2	-> auf Wortgrenze
.byte b1, … ; .half h1, … ; .word w1,… ; .ascii str :	Datenformat
.text :	Eintrag wird im (Programm-)Textsegment gespeichert
.globl sym :	Marke sym ist global und für andere Files lesbar
# comment :	Kommentar

Beispiel:

```
while (save[i] == k) i = i + 1;
```

- *Annahmen*: `i` und `k` sind in `$s3` und `$s5`. Feld `save` startet bei Adresse `$s6`.
- *Assemblerprogramm*:

```
Loop: sll $t1, $s3, 2      # $t1 = 4 * i
      add $t1, $t1, $s6
      lw $t0, 0($t1)
      bne $t0, $s5, Exit
      addi $s3, $s3, 1
      j Loop
Exit: …
```

Funktionsaufrufe

Bei verschachtelten Aufrufen: Speicherung mittels Stack

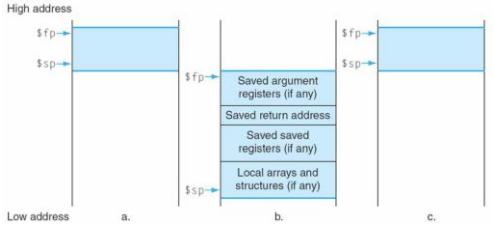
Kontext eines Unterprogramms:
- *Argumente*, die dem Unterprogramm mitgeteilt wurden
- *Registerinhalte*, die darin nicht geändert werden dürfen
- *lokale Variabeln* des Unterprogramms

Beginn d. “activation record”: **Stackpointer ($sp)**
Ende d. “activation record”: **Framepointer ($fp)**

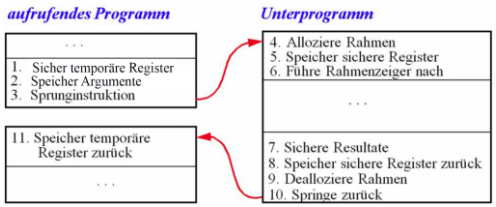
Sichere und temporäre Register

Falls *sichere Register* von Unterprogramm benützt werden, müssen diese zuerst gespeichert werden
Bei *temporären Register* muss das aufrufendes Programm selbst sichern, falls sie danach noch benötigt werden

Preserved	Not preserved
Saved registers: $s0–$s7	Temporary registers: $t0–$t9
Stack pointer register: $sp , $fp	Argument registers: $a0–$a3
Return address register: $ra	Return value registers: $v0–$v1
Stack above the stack pointer	Stack below the stack pointer



Funktionsaufrufe



1. Falls aufrufendes Programm temp. Reg benützt, speichern
2. Erste 4 Argumente werden in $\$a0 - \$a3$ gespeichert, Rest zu Beginn d. Rahmens des Unterprogramms übergeben
3. *jal („jump & link")* : springt und setzt Rücksprungadresse
4. Alloziere Rahmen, indem von *$sp* Grösse abgezogen wird
5. Speichere sichere Register vor eigener Änderung
6. Rahmenzeiger *$fp* wird auf Beginn des Rahmens gesetzt
7. Resultate werden in $\$v0$ *und* $\$v1$ *gespeichert*
8. Wiederherstellung der sicheren Register (siehe Schritt 5)
9. Rahmenzeiger *$sp* wird mit Grösse d. Rahmens addiert
10. Rücksprung zu aufrufendem Programm über *$ra*
11. Gespeicherte temp. Register werden zurückgeschrieben

Kann auch lediglich *$sp* verwenden (kein Framepointer)

Unterbrechungen (Interrupts)

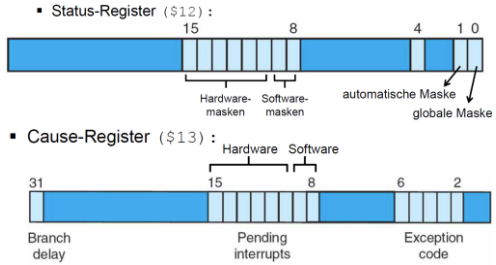
Nicht geplant: Programm wird nach d. laufenden Instruktion verlassen & Unterbrechungsprozedur wird ausgeführt

- Hardwareunterbrechung (Maus, Sensorsignale, Timer)
- arithmetische Ausnahmen (zB. Teilen durch 0)
- falsche Adressierung, Fehler bei Buszugriff
- Softwareunterbrechung (*break, exception*, Breakpoint)

Achtung: das verlassene Programm hat keine Register gesichert, dies wird im Unterbrechungsprogramm getan

Interrupt in Interrupt durch automatische Maske verhindert

Danach wird zur Adresse im *EPC-Register* (*$14*) gesprungen

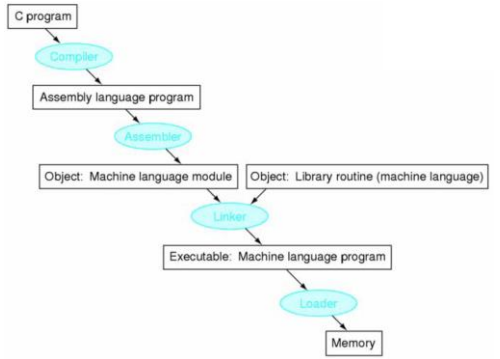


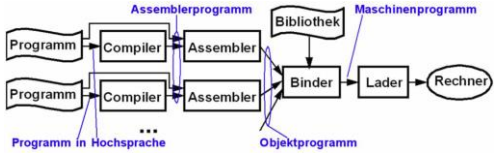
RISC (Reduced Instruction Set)

- einfache Kodierung (alle Instruktionen gleich lang)
- wenige orthogonale Instruktionsklassen (Typ I, J, R)
- viele (32 +) allgemein verwendbare Register

Moderner Instruktionssatz: hohe Parallelität, hohe Taktrate
IA-32 (Complex ISC) : wird zuerst in RISC umgesetzt

4. Vom Programm zur Ausführung





Hochsprache: (C++, Java, Pascal, VHDL, ADA, Lisp)

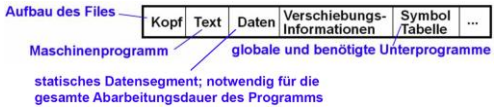
- Problemformulierung unabhängig von Zielrechner
- Prägnante, leicht lesbare und wartbare Formulierung

Assembler: (MIPS-As. , IA-32-Ass.)

- Symbolische Repräsentation eines Maschinenprogramms
- Im Gegensatz zu Hochsprache bereits auf Ziel angepasst
- Enthält (noch) Kommentare, symbolische Marken, Makros
- Bestimmt Beziehung zwischen Marken & Programmzeilen
- Übersetzt jede Assembleranweisung in Maschinencode
- erzeugt Listen nichtaufgelöster Bezüge & globaler Marken

Objektsprache

- enthält Programm in Maschinensprache
- beinhaltet die dazugehörigen binären Daten
- enthält Infos zur Verbindung mehrerer Objektprogramme



Linker

- fasst alle zu einem Programm gehörende Teile zusammen
- sucht in Bibliotheken vordefinierte Teilprogramme
- bestimmt Speicherbereiche für einzelne Programmteile
- löst die Querbezüge zwischen den Programmteilen auf

Loader

- bestimmt Grösse von Text- und Datenteil (aus Kopfteil)
- Freigeben eines Adressbereiches für Programmtext, Daten
- Laden in den Hauptspeicher & Initialisieren der Register

Java

- Java Compiler übersetzt in Java Bytecode für VM
- platformunabhängig durch VM, aber langsamer

5. Rechenleistung

Bewertung eines Rechners: Ausführzeit (*Rechenzeit/Latenz*), Zahl der Programme (*Durchsatz*), Reaktionszeit (*Interrupt*)

Bewertung eines Prozessors: *Taktfrequenz, CPI, MIPS*

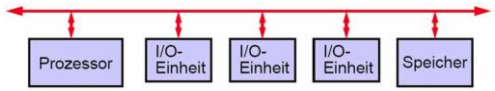
$$CPUZeit = \frac{Rechenzeit}{Programm} = \frac{Instruktionen}{Programm} * \frac{Takte}{Instr.} * \frac{Zeit}{Takt}$$

$$IPS = \frac{Instruktionen}{Zeit} = \frac{Instruktionen}{Programm} * \frac{1}{CPUZeit}$$

6. Eingabe-Ausgabe (I/0)

I/O Datenrate/Bandbreite: Menge an Informationen pro Zeit
I/O Antwortzeit: Gesamtzeit für eine einzelne I/O-Operation

Bus: gemeinsam genutzte Kommunikationsverbindung

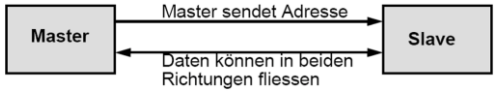


- Einfache Erweiterbarkeit für neue Geräte
- Geringe Kosten, da ein einziges Leitungsbündel genutzt

- Flaschenhals in der Kommunikation durch Bandbreite
- Geschwindigkeit durch physikalische Buslänge begrenzt
- Bus muss viele verschiedene Einheiten unterstützen

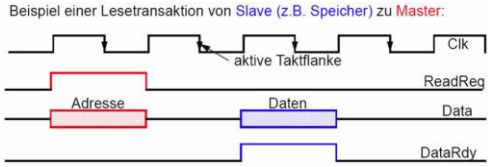
Steuerleitung: Kommunikation, request & acknowledge
Datenleitung: Datentransport v. Daten & Adressen

Master: Startet & beendet Bustransaktion, sendet Adresse
Slave: antwortet auf Anforderung & sendet/empfängt Daten



Synchrone Protokolle

- gemeinsame Taktleitung zur Synchronisation, Protokoll relativ zu diesem Takt
- schnell, falls Taktverschiebung klein
- jede Einheit muss mit gleicher Taktrate arbeiten

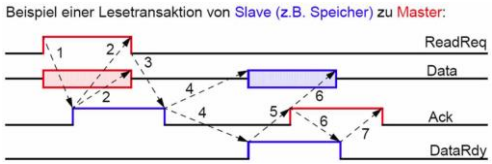


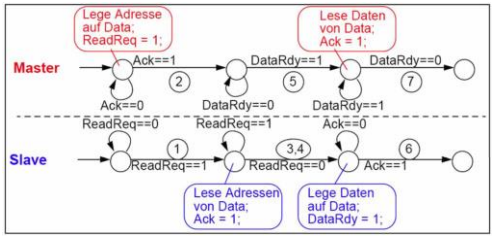
$$ClkToQ + SPD \geq ClkSkew + HoldTime$$
$$ClkPeriod \geq ClkToQ + LPD + SetupTime + ClkSkew$$

ClkToQ: Propagation Delay; ClkSkew: Verzögerung zw. Regis.

Asynchrone Protokolle

- keine Taktleitung, ´handshaking'- Protokoll
- verzögerungsunabhängige Funktion, heterogene Einheiten
- asynchrone ´handshaking'-Logik für das Protokoll





Blocküberträgung: mehrere Worte pro Transaktion/Adresse

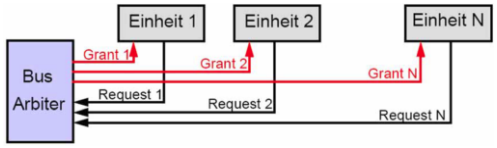
Geteilte Übertragung: kann Bus zwischendurch freigeben
1. Anforderung von Einheit (Master), danach Busfreigabe!
2. Speicher (Master) signalisiert Datenbereitschaft & sendet

Arbitrierungsmechanismen (mehrere Busmaster)

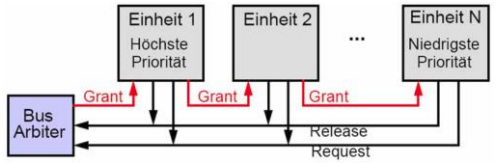
Verteilte Arbitrierung durch Selbstselektion: jedes Gerät legt Identifikation auf Bus & bestimmt eigene Priorität; Gerät mit der höchsten Priorität erkennt dies und sendet

Verteilte Arbitrierung durch Kollisionserkennung: Alle versuchen, den Bus zu reservieren; bei Kollision wiederholt

Zentrale Arbitrierung: Sternförmige Anordnung d. Einheiten



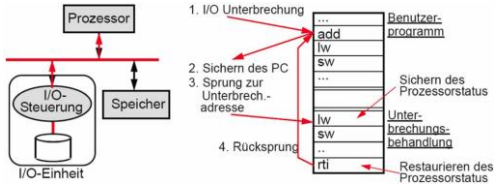
„Daisy Chain"-Arbitrierung: Anordnung nach Priorität
- einfache Implementierung
- keine Fairness bei Buszuteilung (kann ausschliessen)



Betriebssystem: Schnittstelle zw. I/O & Benutzerprogramm
- gibt Befehle an I/O und informiert über Ende & Fehler
- Daten können zw. I/O & Speicher ausgetauscht werden

Speicheradressierte Ein/Ausgabe (memory mapped I/O):
- Einige Adressen sind den I/O-Einheiten zugewiesen
- *Steuerregister:* beinhaltet Befehle an I/O
- *Statusregister:* zeit derzeitige Aktivität/Fehlermeldungen

I/O ⟹ OS : für Fehlermeldung oder Ende d. Transaktion
- *Polling:* Periodisches Abfragen d. Statusregisters durch OS
  (sinnvoll, wenn I/O schnell od. gut vorhersagbar)
- *I/O Unterbrechung*: generiert Interrupt für Meldung
  (Unterbrechungssignal auslösen, erkennen & behandeln)



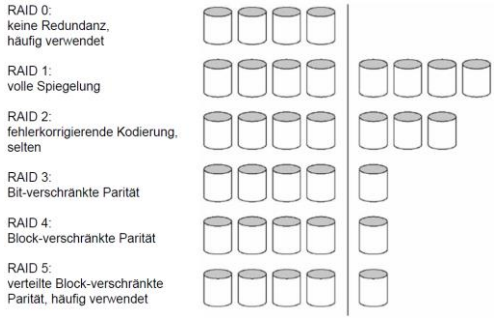
DMA: spezieller I/O-Prozessor (Busmaster), entlastet CPU

7. Plattenspeicher

- lange, nichtflüchtig, grosse Kapazität & billig, LANGSAM

$$Zugriffszeit = Suchzeit + Rotationslatenz + Übertragszeit + Steuerungslatenz + Warteschlangenverzögerung$$

RAID: *Redundant Array of Inexpensive Disks*
- Verwendung vieler kleiner Plattenspeicher
- erhöhter Datendurchsatz durch Parallelität
- Redundanz zur Verbesserung d. Ausfallquote



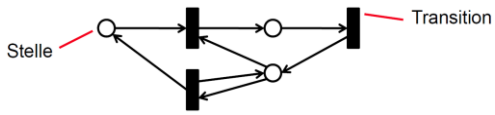
RAID 5: Verteilung d. Parität für paralleles Schreiben

8. Prozessor - Einzeltaktimplementierung

Datenpfad: Verarbeitung und Transport von Instruktionen und Daten
Kontrollpfad: Verarbeitung & Transport von Steuerdaten

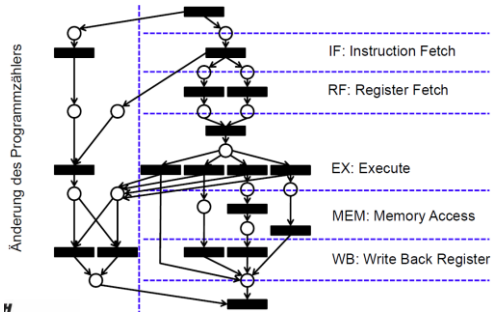
Petri-Netze

Notation zur Darstellung paralleler & verteilter Operationen



Marken (Token): Stellen zugeordnet, beschreiben Zustand

Transitionen: Marken werden folgend „transportiert"
- *„aktiviert"*: 1. Bedingung bei Transition erfüllt
  2. In jeder Eingangsstelle ist mind. eine Marke
- *„feuern"*: aus jeder Eingangsstelle eine Marke entfernt und jeder Ausgangsstelle hinzugefügt + Operationsausführung



Zwischenvariabeln: *IR* (instruction), *A, B, ALUOut, PC, NPC*

Arithmetische Operationen: *ALU(a, b, op)*

Einzeltaktsystem

längster Pfad zw. Register bestimmt minimale Taktperiode

Kombinatorische Schaltungen
- Instruktionsspeicher
- Addierer
- Registerfeld (**Lesen**)
- Hauptspeicher (**Lesen**, falls MemRead==1)

Sequentielle/getaktete Schaltung
- Register
- Registerfeld (**Schreiben**)
- Hauptspeicher (**Schreiben**, falls MemWrite==1)

Kontrollpfad

Instruktion	IR[31...26]	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp
R-Typ	000000	1	0	0	1	0	0	0	10
lw	100011	0	1	1	1	1	0	0	00
sw	101011	X	1	X	0	0	1	0	00
beq	000100	X	0	X	0	0	0	1	01

Eingang | Ausgangssignale

Control | ALU Control

Operation	opcode IR[31...26]	ALUOp	functcode IR[5...0]	ALU Funktion	ALUControl
load word (lw)	100011	00	XXXXXX	'add'	0010
store word (sw)	101011	00	XXXXXX	'add'	0010
branch equal (beq)	000100	01	XXXXXX	'subtract'	0110
add (add)	000000	10	100000	'add'	0010
subtract (sub)			100010	'subtract'	0110
and (and)			100100	'AND'	0000
or (or)			100101	'OR'	0001
set-on-less-than (slt)			101010	'setOnLessThan'	0111

9. Prozessor – Pipelineimplementierung

Pipelining: mehrere Instruktionen überlappend

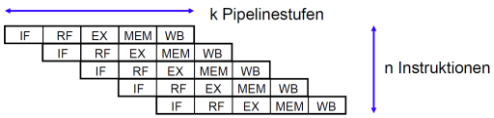
IF (Instruction Fetch) : Lesen der Instruktion

ID/RF (Instruction Decode/Register Fetch) : Register lesen

EX (Execute) : Ausführen d. Instrukt. / Adressberechnung

MEM (Memory) : Zugriff auf den Datenspeicher

WB (Write Back) : Zurückspeichern ins Registerfeld



$$Speedup = \frac{n * k}{k + (n - 1)} \quad ; \quad Effizienz = \frac{Speedup}{k}$$

Hazards

Situation, in der eine Phase der Instruktion nicht direkt im Anschluss an die vorherige Phase ausgeführt werden kann

Strukturelle Hazards: Kombination an Instruktionen unmöglich (zB. falls Daten- u. Instrukt.speicher nicht getrennt)

Ablauf-Hazard: Ergebnis einer Instruktion wird benötigt, um zu entscheiden, welche Instruktion als nächstes kommt

Daten-Hazard: Operand einer Instruktion hängt vom Ergebnis einer vorherigen Instruktion ab

Vermeidung von Daten-Hazards

Reihenfolge: Compiler versucht, Instrukt. Umzugruppieren
Falls keine sinnvolle Instruktion → nop

Forwarding: vorzeitige Weiterleitung zw. Stufen

Mux Ctrl	Source	Explanation
ForwardA = 00	ID/EX	Op 1 from Register
ForwardA = 10	EX/MEM	Op 1 prior ALU Result
ForwardA = 01	MEM/WB	Op 1 from data Mry or earlier ALU Result
ForwardB = 00	ID/EX	Op 2 from Register
ForwardB = 10	EX/MEM	Op 2 prior ALU Result
ForwardB = 01	MEM/WB	Op 2 from data Mry or earlier ALU Result

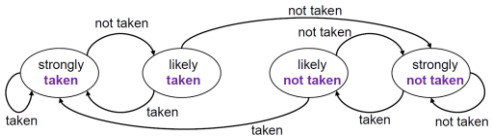
Stalls / Blase: *nop*-Instruktion in späterer Phase einfügen nachfolgende wandern weiter, vorherige bleiben stehen Steuersignale an ID/EX auf „0", PC wird nicht erhöht

Vermeidung von Ablauf-Hazards

Stalls: Nach jeder Entscheidung-Instr. warten bis bekannt

Statische Vorhersage: nehme an, Programm verzweigt nicht

Dynamische Vorhersage: Vorhersage aufgrund vergangener Verzweigungsentscheidungen (*2-Bit Prädiktion*)



Vorverlegen der Berechnung: zusätzliche Vergleichskomponente zur Berechnung in der ID-Stufe

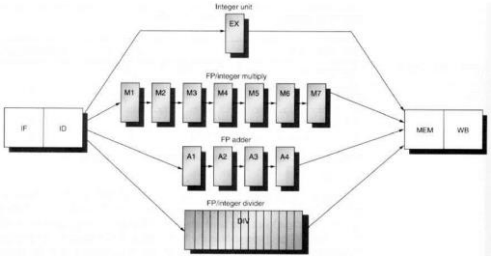
Branch-Delay Slot: Instruktion nach Verzweigung immer ausgeführt, bereits von Compiler/Assembler so eingesetzt

10. Prozessor - Instruktionsparallelität

- mehr Pipelinestufen / tiefere Pipeline -> kürzerer Takt
- Mehrere Instr. pro Takt / mehrere parallele Pipelines

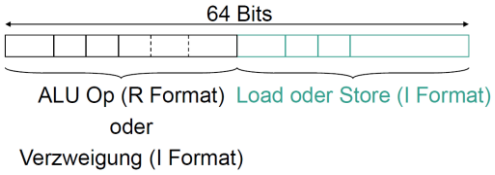
Superpipelining

V.a. bei arithmetischen Einheiten zahlreiche Pipeline-Stufen



- höhere Taktfrequenz durch geringere Laufzeit zw. Stufen
- unterschiedliche Laufzeiten (*„out of order completion"*)
- Einfluss von Hazards auf Ausführungszeit immer grösser

Statische Parallelität: Compiler gruppiert Instruktionen, die gleichzeitig geladen werden (**V**ery **L**ong **I**nstruction **W**ord) *Compiler* detektiert und verhindert Hazards



Dynamische Parallelität: CPU lädt aufeinanderfolgende Instruktionen & bestimmt Abarbeitungsreihenfolge selbst *CPU* löst Hazards durch erweitere Technik zur Laufzeit auf

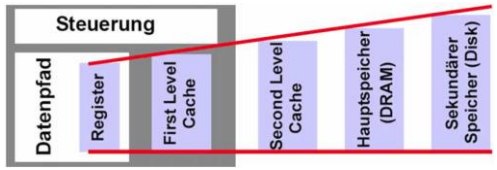
Instruktionen werden nicht in der gegebenen Reihenfolge ausgeführt, aber die Ergebnisse in d. richtigen geschrieben

Compiler-Techniken

- **Umsortieren** d. Instruktionen zur Vermeidung v. Hazards
- **Loop unrolling** zur Optimierung von Schleifen
- **Register Umbenennung:** ordne logischen Register freie physikalische Register zu (superskalare CPU)

- **Spekulation:** Schätzen, was mit einer Instruktion geschieht Prüfe, ob Entscheidung richtig; ansonsten Zustands-Reset Resultate werden zwischengespeichert bis bestätigt

11. Speicherhierarchie



Memory-Gap Problem: Prozessor wartet auf Speicher

Lokalität

Programme greifen auf kleinen Teil d. Adressraums zu

Zeitliche Lokalität: Falls benötigt, whr bald wieder benötigt
- speichere kürzlich benötigte Daten nahe am Prozessor

Örtliche Lokalität: Falls referenziert, werden Daten mit nahegelegenen Adressen bald auch referenziert
- bewege Blöcke aus aufeinanderfolgenden Worten

Speicherzugriff

Daten werden nur zwischen aufeinanderfolgenden Ebenen ausgetauscht, Übertragung in Blöcken als kleinste Einheit

Treffer (Hit): Daten sind in der oberen Ebene vorhanden

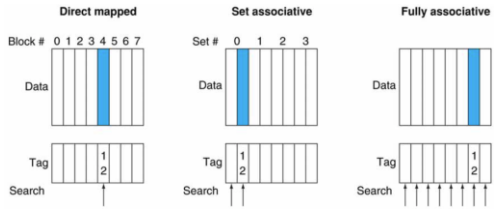
Fehlzugriff (Miss): Daten müssen von weiter geholt werden

$$Hit\_Zeit = Cache\_Zugriffszeit + Zeit\_Bestimmung\_Hit/Miss$$

$$Miss\_Strafe = Finden\_unterer\_Ebene + Übertragung\_hinauf$$

$$Mittlere\_Zugriffszeit = Hit\_Zeit + Miss\_Strafe * Miss\_Rate$$

Platzieren eines Blocks im Cache



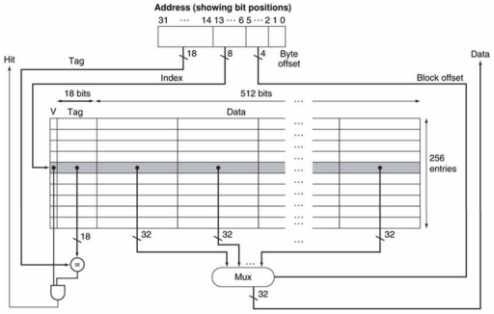
Cacheblock: Daten, die ihren eigenen Tag besitzen
- je grösser, umso besser für örtliche Lokalität & effizientere Speicherung durch kleineren Tag-Verlust
- jedoch höhere Miss-Rate -> Ersetzungszeit grösser

Verbesserung: Assoziativer Cache

Kann bei gleichem Index auswählen, welchen Block ich ersetzen möchte und Miss-Rate dadurch senken
- Zufällige Auswahl des ersetzten Blocks
- längster unbenutzter Block wird ersetzt

Direkte Abbildung

Jeder Block kann nur auf einen Cacheblock abgebildet werden, keine Auswahl möglich



L Bit breite Adresse

2^N Byte Nutzdaten, 2^M Bytes pro Block

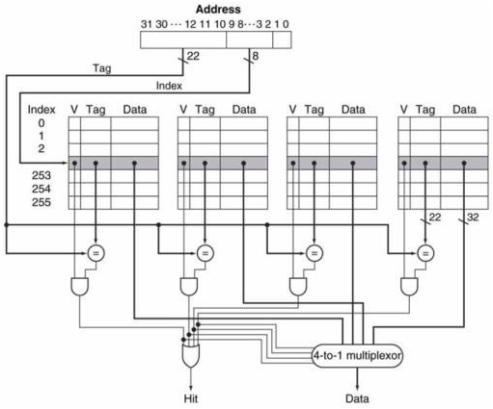
[L-1 - N] : **Tag** zur korrekten Identifizierung
[N-1 - M] : **Index** für das Mapping
[M-1 - 2] : **Block offset** für welches Wort im Block
[1 - 0] : **Byte offset** für die 4 Bytes des Worts

Cachegrösse

$$(1 + (L - N) + 8 * 2^M) * 2^{N-M} Bit$$

Assoziativer Cache

mehrere Einträge pro Index, Speicherblock kann auf K Cacheblöcke abgebildet werden (Auswahl)



Vollassoziativer Cache: kann irgendeinen Cacheblock wählen, gibt keine Index mehr

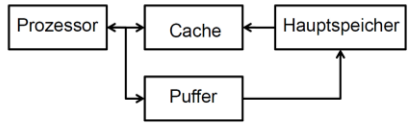
Schreibalternativen

Zurückschreiben (Write back): Prozessor schreibt nur in Cache; falls dieser Block ersetzt werden muss, wird zuerst noch der neue Block in d. Hauptspeicher geladen
- *Dirty Bit* zeig an, ob Block zurückkopiert werden muss
- Cache und HS können über lange Zeit inkonsistent sein

Durchgängiges Schreiben (Write through): bei jedem Schreibvorgang wird auch der Hauptspeicher geändert
- benützt Pufferspeicher, damit Cache noch sinnvoll

Voraussetzung:

mittlere_Speicherrate < 1/Hautpspeicher_Schreibzykluszeit



Leistungsberechnung

$$CPU\_Zeit = (CPU\_Instruktionszyklen + CPU\_Wartezyklen) * Taktperiode$$

$$CPU\_Wartezyklen = \left(\frac{Speicherinstruktionen}{Programm} * Miss\_Rate + \frac{Instruktionen}{Programm} * \frac{Miss}{Instruktion} \right) * \frac{Miss\_Strafe}{Taktperiode}$$

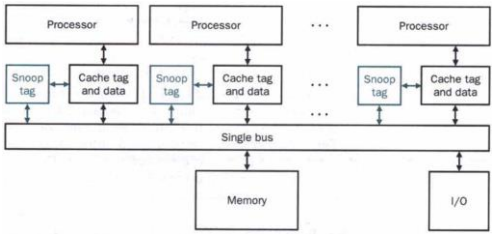
Cache-Ebenen

L1 : klein, schnell, möglichst kleine Hit-Time

L2 : grösser & langsamer, behandelt Misses von L1 möglichst kleine Miss_Rate, damit kein HS-Zugriff

L3 : findet sich vor allem in Multicore-Prozessoren

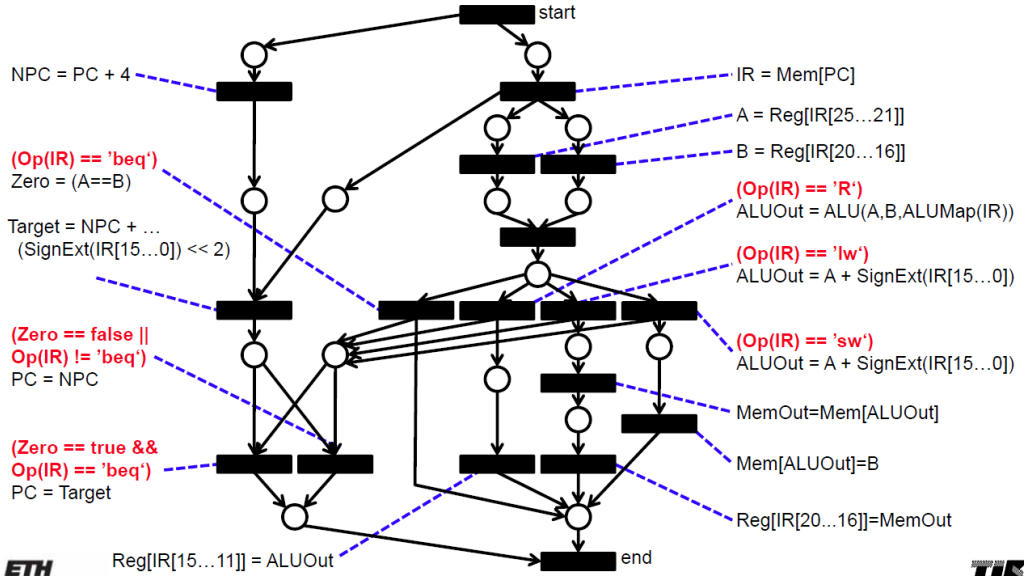
Cache Kohärenz



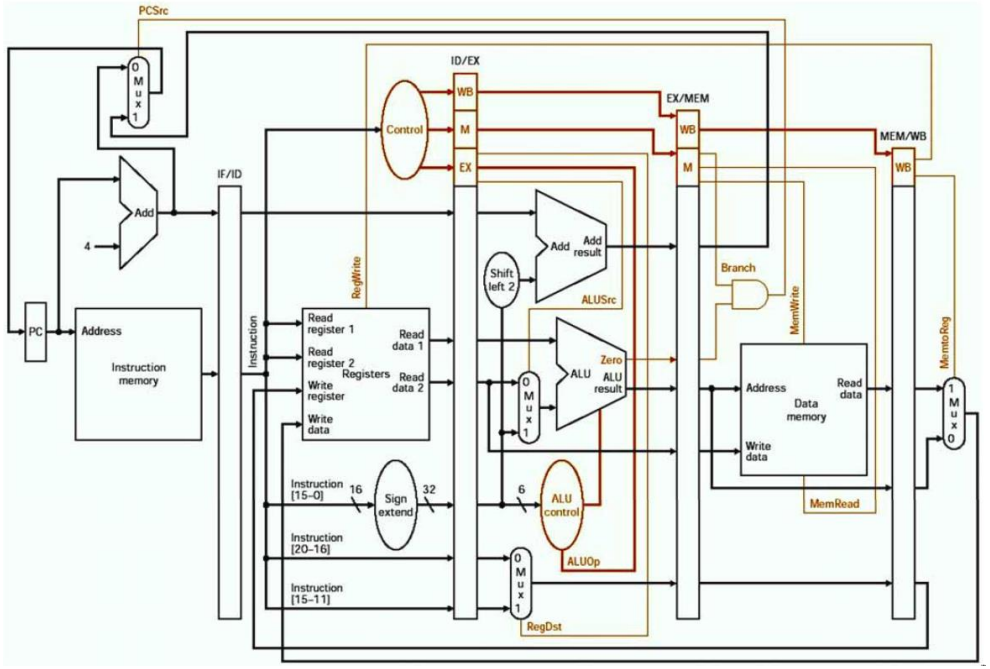
- Erweiterung des Statusbits jeder Zeile
- Zusätzliche Cache-Controller für Cache-Protokoll
- **Snoop tag:** Duplizierte Adressen-Tags und Status-Bits vermeidet Zugriffskonflikte zw. CPU und Controller

Appendix

MIPS-Verfeinerung (ohne j-Instruktion)



Pipelining-Kontrollpfad

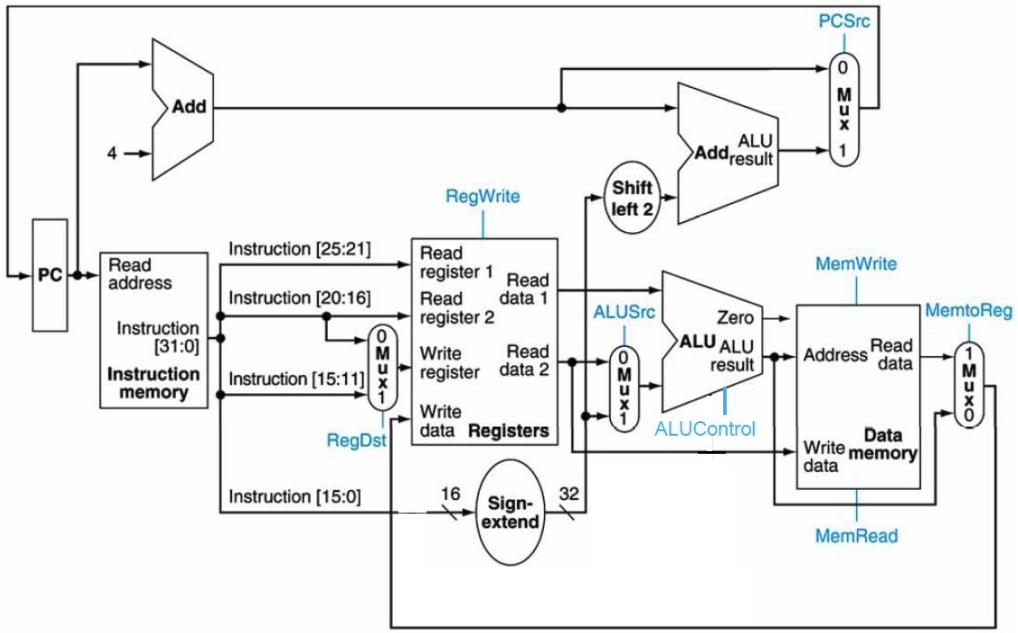


MIPS Instruction Set

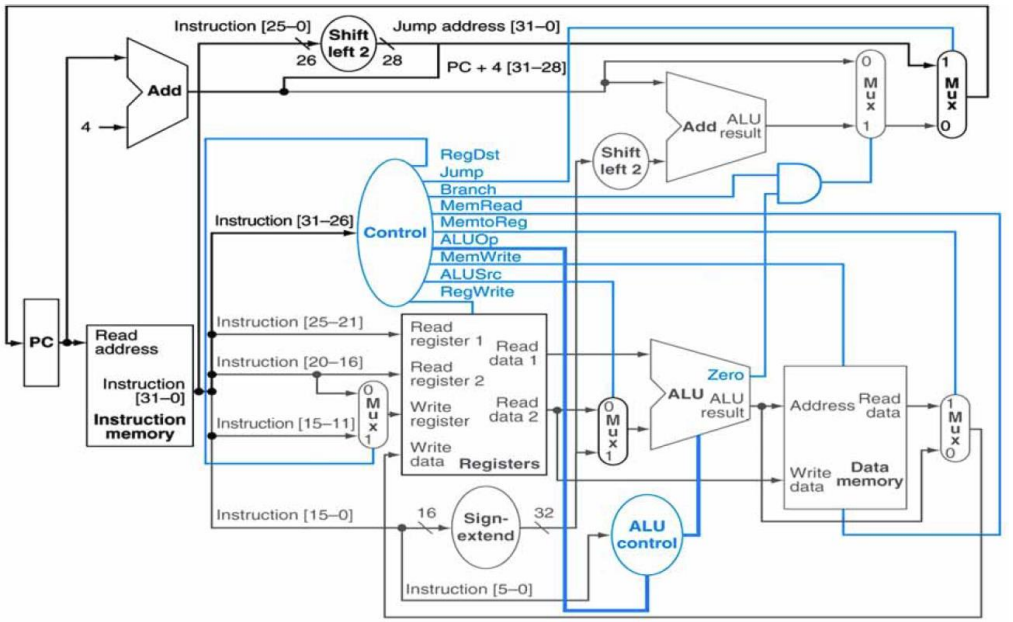
Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for `beq`, `bne`
	set on less than unsigned	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to $ra	For switch, procedure return
	jump and link	`jal 2500`	$ra = PC + 4; go to 10000	For procedure call

Adv. Shift	`sra`	R	Shift Arithm. Right	R[rd] = R[rs] >>> shamt
	`sllv`	R	Shift Logic. Left Var.	R[rd] = R[rs] << R[rt]
	`srlv`	R	Shift Logic. Right Var.	R[rd] = R[rs] >> R[rt]
	`srav`	R	Shift Arith. Right Var.	R[rd] = R[rs] >>> R[rt]
Pseudo-Branching	`blt`	P	Branch Less Than	if (R[rs] < R[rt]) PC = PC + 4 + BranchAddr
	`bgt`	P	Branch Greater Than	if (R[rs] > R[rt]) PC = PC + 4 + BranchAddr
	`ble`	P	Branch Less Than Or Equal	if (R[rs] <= R[rt]) PC = PC + 4 + BranchAddr
	`bge`	P	Branch Greater Than Or Eq.	if (R[rs] >= R[rt]) PC = PC + 4 + BranchAddr
Move	`move`	P	Move / Copy	R[rd] = R[rs]
Adv. Math	`div`	R	Divide	Lo = R[rs] / R[rt]; Hi = R[rs] % R[rt]
	`divu`	R	Divide Unsigned	Lo = R[rs] / R[rt]; Hi = R[rs] % R[rt]
	`mult`	R	Multiply	{Hi, Lo} = R[rs] * R[rt]
	`multu`	R	Multiply Unsigned	{Hi, Lo} = R[rs] * R[rt]
Spez	`mfhi/mflo`	R	Move From Hi / Lo	R[rd] = Hi / R[rd] = Lo
	`mfc0/mtc0`	R	Move From / To Coproc 0	R[rd] = CR[rs] / CR[rs] = R[rd]

Einzeltakt-Datenpfad



Einzeltakt-Kontrollpfad



Kontroll-Pfad Instruktionen

- ***Beschreibung*** der „Forwarding“-Funktion
  - Notation: **`MEM/WB.RegisterRd`**

    Pipeline Register MEM/WB **Feld mit dem Namen** RegisterRd
  - Weiterleiten eines Datums aus dem EX/MEM-Register, das heisst eines vorherigen ALU-Operanden (Beispiel der “Forwarding”-Funktion für R-Instruktionen):

```
if ( EX/MEM.RegWrite = 1      and
     EX/MEM.RegisterRd != 0   and
     EX/MEM.RegisterRd = ID/EX.RegisterRs)
{ForwardA = ’10’;}
if ( EX/MEM.RegWrite = 1      and
     EX/MEM.RegisterRd != 0   and
     EX/MEM.RegisterRd = ID/EX.RegisterRt)
{ForwardB = ’10’;}
```

Beschreibung der „Forwarding“-Funktion

- Weiterleiten eines Datums aus dem MEM/WB-Register, das heisst eines vergangenen ALU-Operanden (Beispiel der “Forwarding”-Funktion für R-Instruktionen):

```
if ( MEM/WB.RegWrite = 1      and
     MEM/WB.RegisterRd != 0   and
     EX/MEM.RegisterRd != ID/EX.RegisterRs and
     MEM/WB.RegisterRd = ID/EX.RegisterRs)
{ForwardA = ’01’;}

if ( MEM/WB.RegWrite = 1      and
     MEM/WB.RegisterRd != 0   and
     EX/MEM.RegisterRd != ID/EX.RegisterRt and
     MEM/WB.RegisterRd = ID/EX.RegisterRt)
{ForwardB = ’01’;}
```

Cache-Kohärenz: Write invalidate/ Write through