Pipelined Control

• Use essentially the same control signals as the single cycle implementation.

• As before, no control needed for IF and ID stages of pipe, i.e., control does not affect execution until the EX stage.

• The opcode is available after the IF stage therefore control must be generated in the ID stage.

• No FSM needed since the pipelining keeps the timing of each instruction relative to control signals straight.
● No variable length instructions

● Control signal effects can be associated with a particular stage (the one in which the HW it controls is present)

● control signals must be moved through pipeline registers until the appropriate stage (and no farther)
Insert text figures 6.30 to 6.35 here
Data Hazards and Forwarding

Consider the following code and its dependences:

```
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
```

- All four instructions after the sub require the data in $2.

- Not all dependences are hazards
• Hazards are those that require data before it is available.

• Object is to detect these situations and access the data from some intermediate register in the pipe rather than waiting for it to appear in the register file after the WB stage.
insert text figures 6.36 and 6.37
• Only two hazards

• Note the register file dual timing avoids one.

• Both have results that appear in some intermediate pipeline register before they are used.

• Therefore, both can be removed via forwarding without any stalls.
• the data consumed by the AND in its EX stage is present in the EX/MEM pipeline register.

• the data consumed by the OR in its EX stage is present in the MEM/WB pipeline register.

• so we know where it is, how do we create a forwarding unit that decides combination-ally that forwarding is needed.
• It is the job of the consuming instruction to determine whether or not the data it read from registers (and currently in the ID/EX pipeline register) is valid.

• It is not valid if the instruction currently in the MEM stage has the register as its destination.

• It is not valid if the instruction currently in the WB stage has the register as its destination.

• Both assume that those instructions write to the register file.
• To detect the SUB to AND hazard the EX stage adds:

    if (EX/MEM.RegWrite
        and
        (EX/MEM.RegisterRd not equal 0)
        and
        (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    then ForwardA = 10

    if (EX/MEM.RegWrite
        and
        (EX/MEM.RegisterRd not equal 0)
        and
        (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    then ForwardB = 10

• in this case the firsts succeeds and ForwardA is set to 10 to choose the second ALU input from the data in the EX/MEM pipeline register
To detect the SUB to OR hazard the EX stage adds:

```python
if (MEM/WB.RegWrite
    and
    (MEM/WB.RegisterRd not equal 0)
    and
    (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    then ForwardA = 01

if (MEM/WB.RegWrite
    and
    (MEM/WB.RegisterRd not equal 0)
    and
    (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    then ForwardB = 01
```

● in this case the second succeeds and ForwardB is set to 10 to choose the second ALU input from the data in the EX/MEM pipeline register
Consider the following code:

```
add $1,$1,$2
add $1,$1,$3
add $1,$1,$4
```

The two checks given above will both succeed so we need extra logic to choose the correct one, i.e., the most recent value in the EX/MEM register.
The MEM/WB portion is altered with an additional condition that makes sure the EX/MEM fetch will not be done:

```python
if (MEM/WB.RegWrite
    and
    (MEM/WB.RegisterRd not = 0)
    and
    (EX/MEM.RegisterRd not = ID/EX.RegisterRs)
    and
    (MEM/WB.RegisterRd = ID/EX.RegisterRs))
then ForwardA = 01

if (MEM/WB.RegWrite
    and
    (MEM/WB.RegisterRd not = 0)
    and
    (EX/MEM.RegisterRd not = ID/EX.RegisterRt))
    and
    (MEM/WB.RegisterRd = ID/EX.RegisterRt))
then ForwardB = 01
```